SNMP

Manuel de Programmation

V4r01a

Par Olorin113

Merci à www.utlimate-console.fr et www.snes-fr.com
Contact : snmp@ultimate-console.fr

Sommaire

Première partie

LICENCE DU DOCUMENT

GNU Free Documentation License

Version 1.3, 3 November 2008

¡http://fsf.org/¿

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed

under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML

using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in

an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation

of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Deuxième partie

PREAMBULE

Chapitre 1

Remerciement

Je tient à remercier toutes les personnes qui on contribué (un peu ou beaucoup) à la conception de ce document !

- Snes-fr.com pour m'avoir permis de poster une idée aussi absurde et farfelue ! !
- Hedge qui m'a poussé et motivé à me lancer.
- Remyar qui m'aide et qui heberge sur son site.
- ultimate-console.fr dirigé par Remyar.
- A mon ancien professeur de mathématique Jacques Dumas pour m'avoir expliqué le latex et corriger certaines de mes erreurs. (un des deux meilleurs professeurs de mathématiques que j'ai pût rencontrer dans ma vie).
- A Happexamendios qui rapporté et corrigé les anomalies du SNMP (pour la version V3r01a)
- A Mr Bigonoff qui m'a permis d'apprendre le PIC avec ces cours (2002), ce qui m'a amener à prendre ce projet en main (2008).
- A Mr Bigonoff pour me permettre d'utiliser ses cours sur le langage machine pour apprendre à tout le monde les bases de la programmation.
- A Gatchan pour m'avoir corrigé des fautes
- A Gwendoline pour m'avoir corrigé des fautes aussi :)
- A ceux qui m'ont aidé à faire cette documentation, dont j'ai oublier leur nom, mais qui peuvent me'écrire pour me le rappeler :)

Chapitre 2

Historique des versions de SNMP

Description du choix des versions :

v = version, r = révision.

On change les chiffres de la révision quand on rajoute un chapitre, ou un nombre important de correction

On change les lettres de la révision quand on fait des corrections mineures

On change de version quand un nombre important de changements on été faits (mise en page, style de document, rajout important de chapitres)

Version	**date**	**Changements**
v0r00a Draft 1	07 Avril 2008	Création du document
v1r00a	01 Mai 2008	explication des 94 instructions. Correction sur les exemples, des fautes d'orthographe, et sur les nombres signés
v2r00a Draft1	04 Mai 2008	Changement de la mise en page, utilisation de LaTeX.[1] Revue de TOUTES les instructions, exemples, fautes d'orthographe, véracité des résultats Ajout de tableaux des résultats d'opération Revue complète de l'instruction BIT
v2r00a Draft2	14 Mai 2008	Instructions jusqu'à JMP et LSR
v2r00a Draft3	19 Mai 2008	Ajout de la section "Truc et Astuce" faute dans les tableaux LSR faute dans les images et description de RTI et RTL faute dans les indicateurs affectés par RTS Revue complète de l'instruction SBC ADC TXY TYX TXA TYA Centrer toutes les tables Toutes les instructions principales décrites Enlever le chapitre "Le Programme",et ajout de la section "Programme et compilation" dans le chapitre "Les outils de Programmation"
v2r00a	23 Mai 2008	Ajout de la section "Langage Machine"

TABLE 2.1 – versions SNMP 1/5

[0]c'est un traitement de texte, c'est sur google et vous verrez qu'il n'y a rien de pervers là dedans ;).

Version	date	Changements
v2r01a Draft 1	1 Juin 2008	Traduction ”push ”pull” revue, correction de fautes d'orthographe et de grammaire Ajout de la page ”Remerciement” Ajout d'un indice décrivant si l'instruction est compatible pour le processeur 65816 et/ou 6502 Ajout du chapitre sur le 65816, sur la programmation de la console, et sur le Hardware de la Super Nintendo section : jeu d'instruction, correction des indicateurs du registre P
v2r01a Draft 2	12 juin 2008	rajout de la sous section ”Les Unités Informatique” Ajout des registres PPU de la Super Nintendo ($2100 à $2119) Ajout des Annexes de la Super Nintendo
v2r01a Draft 3	23 juin 2008	Ajout des registres PPU de la Super Nintendo ($211A à $212F)
v2r01a Draft 4	31 juillet 2008	Ajout d'un tutorial pour personnaliser NOTEPAD++
v2r01a Draft 5	12 Août 2008	Ajout du chapitre ”Mapping Mémoire”, modification de l'image 3 dans le tutorial
v2r01a	14 Août 2008	Section ”création asm” supprimé Finalisation et corrections mineur dans les registres PPU
v2r02a	20 Août 2008	correction mineur au registre $211F signe de multiplication changé Correction au, chapitre registre PPU, précision sur les références aux Annexes, registre $2115. Mise en forme du chapitre PPU revue. Nouveau chapitre sur les registres CPU.
v3r01a	15 Octobre 2008	Document sous licence ”GNU FDL 1.2” depuis le 1erOctobre 2008. Correction intégrale du SNMP revue par Happexamendios. Redéfinition du ”forced blank”. Chapitre 7 ”Le programme” supprimé car non commencé. Séparation du tableau ”historique” sur 2 pages.

Table 2.2 – versions SNMP 2/5

Version	date	Changements
v3r02a Draft 1	7 Mars 2009	Registres $2130 et $2131 avaient le même nom Tutorial remplacer par Tutoriel. Chapitre "Tutoriel Notepad++" mis à jour. Ajout du chapitre "LES INFORMATIONS DE LA CARTOUCHE". Ajout d'information sur l'instruction "REP" Correction dans les instructions "TXS","TCD" Section "Activer l'écran" changer en "Initialiser l'écran", changer le mot "Allumer" en "Initialiser", et les valeur $8F en'$0F Ajout du chapitre "Plan de programmation"
v4r00a Draft 1	12 Mars 2009	Pas de version V3r02a officielle car trop de changements, donc V4 Inverser les deux section "WLA" et "La Super Nintendo" Section WLA complète et triée par ordre alphabétique, toutefois non traduite Déplacement du chapitre "mapping mémoire" de la section programmation pour la fusionner avec le chapitre "mapping mémoire" de la section "La Super Nintendo" Correction dans le chapitre "Configuration du Mapping Mémoire". [LateX] Correction de la macro \autoref, ajout d'un espace entre le mot "voir" et les mots "le chapitre", "la section", "la table", "l'équation" et "l'annexe". Visible dans tout le document à chaque fois que l'on fait référence à quelque chose, il manquait l'article défini masculin et féminin devant les noms [LateX] Insertion d'un nouveau style d'affichage du code machine (utilisation du package "listings"). Nouvelle mise en page du titre. Insertion d'une première et d'une quatrième de couverture. Chapitre "Tutoriel Notepad++" modifié, et mis à jour. Ses images ont été supprimées car elle ne reflètent pas la réalité. Toutes les instructions de compilations y ont été ajoutées, d'après le compilateur WLA Le titre de la partie "Le commencement" remplacé par "Préambule" "Le langage Machine" de bigonoff dans la partie "la programmation" est déplacée dans la partie "la programmation". Et ajout d'un commentaire au début du chapitre Nom du chapitre "Programme de compilation" changé en "Tutoriel - Programme de compilation" Optimisation du poids du document

TABLE 2.3 – versions SNMP 3/5

Version	date	Changements
v4r00a Draft 2	22 Mars 2009	Modification de la directive du compilo WLA ".SNESHEADER" Ajout de la section "Paramétrer la compilation" dans le tutoriel "Notepad++" Dans "onglet mot clé" inversion des groupes 3 et 1 Modification dans le tutoriel "Programme de compilation" Titre du chapitre "Initialiser les informations de la Cartouche" modifié par "Initialisation des informations de la Cartouche" Ajout de l'annexe "Le programme" Ajout des sections et chapitres d'initialisation dans la section "Programmation" Faute dans la documentation du compilateur WLA. la directive .SECTION remplace .SUBSECTION [LateX]Correction dans la macro \autoref Ajout des sections "Désactiver les Interruptions" et "Activer les interruptions" Ajout de notes "future versions" pour ne pas oublier que la documentation est incomplète Changement de version de licence. GNU FDL V1.2 remplacée par GNU FDL v1.3 Informations sur le processeur 65816 incomplet
v4r00a Draft 3	3 Mai 2009	Revu de toutes les instructions Insertions du chapitre "un mot sur les unitées" Insertions du chapitre "Vocabulaire" Insertions en annexes, de 3 classements des instructions : par mode d'adressage, par ordre alphabétique, et par opcode redéfinition des registres PPU $213E et $213F Remplacement des mots anglais en français. Tel que "Status Register" en "Registre d'État"
v4r00a	2 Juin 2009	Ajout des chapitres : - "ses premiers programmes" remplace "son premier programme" - "les différents modes d'adressage", manque l'adressage du 65816 Ajout du Programme : Programmation d'un tableau, encore incomplètes
v4r00b	5 Juin 2009	Mise à jour du chapitre "les différents modes d'adressage" avec l'adressage du 65816 référence aux instructions en annexe ajoutée. Images des programmes "tableaux" refaites

TABLE 2.4 – versions SNMP 4/5

Version	date	Changements
v4r01a	23 Juin 2009	Mise à jour du chapitre "Les Instruction", ajout de la section "Classes d'Instructions" Mise à jour de la section "Le jeu d'instruction" dans le même chapitre Section "Effacer la table des Sprites" renommée en "Effacer la table des registres Vidéo" Section "Activer la surveillance du Joystick" renommée en "Activer la surveillance du Joypad" Insertion du chapitre "Technique de programmation", manque une section sur les sous programmes

TABLE 2.5 – versions SNMP 5/5

Troisième partie

LA PROGRAMMATION

C'est à partir d'ici que les choses sérieuses commencent.

Nous allons tout d'abord initialiser la console. Tout ce qui sera écrit dans les parties suivante, seront vus et revue dans cette partie.

Si j'avais mis cette partie "après" la description de la Super Nintendo, ses registres, etc... [1] vous auriez jeté le manuel, par dépit ! J'aurais eu la même réaction...

C'est pourquoi j'ai décidé de passer à la programmation directement, et si vous ne comprenez pas quelque chose, vous irez voir les autres pages de ce manuel qui ne sont pas là pour rien.

Le prochain chapitre est la base de l'assembleur, je l'ai emprunté à Mr Bigonoff qui me l'a gracieusement prêté ! Pour les autres chapitres de programmation, je me suis inspiré de tout ce que je pouvais avoir sur internet, et surtout de wikipedia pour le premier exemple que je vous ai fait !

Enjoy !

Chapitre 3

Le Langage Machine **(sources : M.Bigonoff)**

C'est par ici que l'on va commencez à apprendre le langage machine de base : l'assembleur. Ce cour est tiré de Mr Bigonoff, très explicite et c'est mon mentor sur l'art et la manière de faire un bouquin sur la programmation !

3.1 Les systèmes de Numérotation

Les nombres décimales

Nous sommes habitués, depuis notre enfance à utiliser le système numérique décimal, à tel point que nous ne voyons même plus la manière donc ce système fonctionne, tant c'est devenu un automatisme.

Décimal, pourquoi ? Parce qu'il utilise une numération à 10 chiffres. Nous dirons que c'est un système en BASE 10. Pour la petite histoire, on a utilisé un système base 10 car nos ancêtres ont commencé à compter sur leurs 10 doigts, pas besoin d'aller chercher plus loin.

Mais la position des chiffres a également une grande importance. Les chiffres les moins significatifs se situent à droite du nombre, et leur importance augmente au fur et à mesure du déplacement vers la gauche. En effet, dans le nombre 502, le 5 a une plus grande importance que le 2. En réalité, chaque chiffre, que l'on peut appeler DIGIT, a une valeur qui dépend de son RANG . Quel est le multiplicateur à appliquer à un chiffre en fonction de sa position (rang) ? Il s'agit tout simplement de l'élévation de la BASE utilisée à la puissance de son RANG .

Cela a l'air complexe à écrire, mais est très simple à comprendre. Lorsque vous avez compris ceci, vous comprenez automatiquement n'importe quel système de numération.

Reprenons, par exemple notre nombre 502. Que signifie le 2 ? Et bien, tout simplement que sa valeur est égale à 2 multiplié par la base (10) élevée à la puissance du rang du chiffre, c'est à dire 0.

Remarquez ici une chose très importante : le comptage du rang s'effectue toujours de droite à gauche et en commençant par 0. Pour notre nombre 502, sa valeur est donc en réalité :

$502 = 2 \times 10^0 + 0 \times 10^1 + 5 \times 10^2$. Et rappelez-vous que $10^0 = (10/10) = 1$, que $10^1 = 10$, et que $10^2 = 10 \times 10 = 100$, etc...

(sources : M.Bigonoff)

Le système binaire

Vous avez compris ce qui précède ? Alors la suite va vous paraître simple. Cela ne pose aucun problème pour vous de compter sur vos 10 doigts, mais pour les ordinateurs, cela n'est pas si simple. Ils ne savent faire la distinction qu'entre 2 niveaux[1]. Le système de numération décimal est donc inadapté.

On comprendra immédiatement que le seul système adapté est donc un système en base 2, appelé système binaire. Ce système ne comporte donc que 2 chiffres, à savoir 0 et 1. Comme, de plus, les premiers ordinateurs travaillent avec des nombres de 8 chiffres binaires, on a donc appelé ces nombres des octets (ou bytes en anglais). Le chiffre 0 ou 1 est appelé un BIT(BInary uniT) .

Pour nous y retrouver dans la suite de ce petit ouvrage, on adoptera les conventions suivantes : tout nombre décimal est écrit tel quel, ou en utilisant la notation D'xxx' ; tout nombre binaire est écrit suivant la forme B'xxxxxxxx' dans lesquels les 'x' valent ?. . . 0 ou 1 effectivement, vous avez bien suivi.

Analysons maintenant un nombre binaire, soit l'octet : B'10010101'. Quelle est donc sa valeur en décimal ?

Et bien, c'est très simple, on applique le même algorithme que pour le décimal. Partons de la droite vers la gauche, on trouve donc :

B'10010101' $= 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 0 \times 2^5 + 0 \times 2^6 + 1 \times 2^7$

Comme, évidemment 0 multiplié par quelque chose = 0 et que 1 multiplié par un chiffre = le chiffre en question, on peut ramener le calcul précédent à :

B '10010101' = 1+4+16+128 = 149

Vous voyez donc qu'il est très facile de convertir n'importe quel chiffre de binaire en décimal. Et l'inverse me direz-vous ? Et bien, c'est également très simple. Il faut juste connaître votre table des exposants de 2. Cela s'apprend très vite lorsqu'on s'en sert.

On procède simplement par exemple de la manière suivante (il y en a d'autres) :

Quel est le plus grand exposant de 2 contenu dans 149 ? Réponse 7 (27 = 128)

On sait donc que le bit 7 vaudra 1. Une fois fait, il reste 149-128 = 21

Le bit 6 représente 64, c'est plus grand que 21, donc b6 = 0

Le bit 5 représente 32, c'est plus grand que 21, donc b5 = 0
Le bit 4 représente 16, donc ça passe, b4 = 1, il reste 21-16 = 5
Le bit 3 représente 8, c'est plus grand que 5, donc b3 = 0
Le bit 2 représente 4, donc b2 = 1, reste 5-4 = 1
Le bit 1 représente 2, c'est plus grand que 1, donc b1 = 0
Le bit 0 représente 1, c'est ce qu'il reste, donc b0=1, reste 0

Le nombre binaire obtenu est donc B'10010101', qui est bien notre octet de départ. Notez que si on avait trouvé un nombre de moins de 8 chiffres, on aurait complété avec des 0 placés à gauche du nombre. En effet, B'00011111' = B '11111', de même que 0502 = 502.

Pensez à toujours compléter les octets de façon à obtenir 8 bits, car c'est imposé par la

[1] présence ou absence de tension

plupart des assembleurs[1].

Notez que la plus grande valeur pouvant être représentée par un octet est donc : B'11111111'. Si vous faites la conversion [2], vous obtiendrez 255. Tout nombre supérieur à 255 nécessite donc plus d'un octet pour être représenté.

(sources : M.Bigonoff)

[1] nous verrons ce que c'est dans la suite de ces leçons

[2] ou en utilisant la calculette de Windows en mode scientifique

Le système hexadécimal

La représentation de nombres binaires n'est pas évidente à gérer, et écrire une succession de 1 et de 0 représente une grande source d'erreurs. Il fallait donc trouver une solution plus pratique pour représenter les nombres binaires. On a donc décidé de couper chaque octet en 2 (QUARTET) et de représenter chaque partie par un chiffre.

Comme un quartet peut varier de b'0000' à b'1111', on constate que l'on obtient une valeur comprise entre 0 et 15. Cela fait 16 combinaisons. Les 10 chiffres du système décimal ne suffisaient donc pas pour coder ces valeurs.

Plutôt que d'inventer 6 nouveaux symboles, il a été décidé d'utiliser les 6 premières lettres de l'alphabet comme CHIFFRES. Ce système de numération en base 16 a donc été logiquement appelé système hexadécimal.

Notez que ce système est simplement une représentation plus efficace des nombres binaires, et donc que la conversion de l'un à l'autre est instantanée. Dans la suite de ces leçons, nous noterons un nombre hexadécimal en le faisant précéder de $. Voyons si vous avez bien compris :

Tableau de conversion des différents quartets (un demi-octet)

Binaire	Hexadécimal	Décimal
B'0000'	$0	0
B'0001'	$1	1
B'0010'	$2	2
B'0011'	$3	3
B'0100'	$4	4
B'0101'	$5	5
B'0110'	$6	6
B'0111'	$7	7
B'1000'	$8	8
B'1001'	$9	9
B'1010'	$A	10
B'1011'	$B	11
B'1100'	$C	12
B'1101'	$D	13
B'1110'	$E	14
B'1111'	$F	15

Pour représenter un octet il faut donc 2 digits hexadécimaux. Par exemple, notre nombre B'10010101' est représenté en hexadécimal par $95. Si vous faites la conversion de l'hexadécimal vers le décimal, vous utilisez le même principe que précédemment, et vous obtenez $\$95 = 9 \times 161 + 5 \times 16^0 = 149$, ce qui est heureux.

Pour preuve, quel est le plus grand nombre hexadécimal de 2 digits pouvant être représenté ? Réponse : $FF, soit $15 \times 16 + 15 = 255$.

Si vous avez bien tout compris, vous êtes maintenant capable de convertir n'importe quel nombre de n'importe quelle base vers n'importe quelle autre. Vous trouverez également dans certaines revues, des allusions au système octal, qui est un système en base 8 qui a été largement utilisé.

(sources : M.Bigonoff)

Les Unités Informatique

C'est très simple :

Anglais	Français	désignation	Nombre de Valeurs possible	s'écrit sous la forme
1 bit	1 bit	= unité de base	0 ou 1	%0
1 byte	1 Octet	= 8 bits	0 à 255	$12 ou %00010010
1 word	1 mot	= 2 octets	0 à 65535	$4512
1 long	1 long	= 3 octets	0 à 8388607	$45 :7891

Dans le manuel, on prendra les unités bit, octet, word et Long. Je ne prendrais pas "mot" car il y a trop de confusions , un "word" est plus explicite, et cela vous évitera de faire la conversion lors d'une lecture d'un document Anglais. Le Long sera rarement utilisé, car nous avons très peu de commande la dessus. Par contre, ce long de 3 octets est spécifique au 65816, car notre Long fait 23 bits, 2^23 = 8Mo.

Voici les conversions des unités :
bit = bit = bit
octets = octets = 8 bits
Ko = Kilo octets = 1024 octets
Mo = Mega octets = 1024 Kilo octets

(sources : Moi)

Les opérations

Après avoir converti les nombres dans différents formats, vous allez voir qu'il est également très simple de réaliser des opérations sur ces nombres dans n'importe quel format. Il suffit pour cela d'effectuer les mêmes procédures qu'en décimal.

Petit exemple :

Que vaut B'1011' + B '1110' ?

Et bien, on procède exactement de la même façon que pour une opération en décimal.

```
    B'1011'
+   B'1110'
-----------
```

- On additionne les chiffres de droite, et on obtient 1+0 = 1
- On écrit 1
- On additionne 1 + 1 , et on obtient 10 (2 n'existe pas en binaire). On écrit 0 et on reporte 1
- On additionne 0 + 1 + le report, et on obtient 10. On écrit 0 et on reporte 1
- On additionne 1 + 1 + le report, et on obtient 11. On écrit 1 et on reporte 1
- Reste le report que l'on écrit, soit 1.

La réponse est donc B'11001', soit 25.

Les 2 nombres de départ étant B'1011', soit 11, et B'1110', soit 14. Vous procèderez de la même manière pour les nombres hexadécimaux, en sachant que $F + $1 = $10, soit 15+1 = 16.

(sources : M.Bigonoff)

Les nombres signés

Dans certaines applications, il est nécessaire de pouvoir utiliser des nombres négatifs. Comme les processeurs ne comprennent pas le signe "-"[1] , et comme il fallait limiter la taille des mots à 8 bits, la seule méthode trouvée a été d'introduire le signe dans le nombre.

On a donc choisi[2] le bit 7 pour représenter le signe. Dans les nombres signés, un bit 7 à "1" signifie nombre négatif. Si on s'était contenté de cela, on aurait perdu une valeur possible. En effet, B'10000000' (-0) serait alors égal à B'00000000' (0). De plus, pour des raisons de facilité de calcul, il a été décidé d'utiliser une notation légèrement différente.

Pour rendre un nombre négatif, il faut procéder en 2 étapes.

- On inverse la totalité du nombre.
- On ajoute 1

On obtient alors ce qu'on appelle le COMPLEMENT A DEUX du nombre.

Exemple :

soit le nombre 5 : B'00000101' Comment écrire -5 ?

on inverse tous les bits (complément à 1)	B'11111010'
on ajoute 1 (complément à 2) -5 =	B'11111011'

Pour faire la conversion inverse, on procède de façon identique.

On inverse tous les bits	B'00000100'
On ajoute 1	B'00000101'

Et on retrouve notre 5 de départ, ce qui est logique, vu que -(-5) = 5.

Dans le cas des nombres signés, on obtient donc les nouvelles limites suivantes :

- La plus grande valeur est B'01111111', soit +127
- La plus petite valeur devient B'10000000', soit -128.

Remarquez que les opérations continuent de fonctionner. Prenons -3 + 5

	B	'11111101'	(-3)
+	B	'00000101'	(5)
=	B	'100000010'	(2)

Et là, me direz vous, ça ne fait pas 2 ?

Et bien si, regardez bien, il y a 9 bits , or le processeur n'en gère que 8. Le 9ème est donc tombé dans un bit spécial que nous verrons plus tard. Dans le registre du processeur, il reste donc les 8 bits de droite, soit 2, qui est bien égal à (-3)+ 5.

[1] moins
[2] pas au hasard

Maintenant, si vous avez bien suivi, vous êtes en train de vous poser la question suivante : Quand je vois B'11111101', est-ce que c'est -3 ou est-ce que c'est 253 ? Et bien vous ne pouvez pas le savoir sans connaître le contexte.

Sachez que les nombres signifient uniquement ce que le concepteur du programme a décidé qu'ils représentent. S'il travaille avec des nombres signés ou non, ou si cet octet représente tout autre chose. La seule chose qui importe c'est de respecter les conventions que vous vous êtes fixées lors de la création de cet octet. C'est donc à vous de décider ce dont vous avez besoin pour tel type de données.

(sources : M.Bigonoff)

3.2 Les opérations booléennes

Qu'est-ce que c'est que ça, me direz-vous ? Et bien, pour faire simple, disons que se sont des opérations qui s'effectuent bit par bit sur un octet donné. Plutôt qu'une grosse théorie sur l'algèbre de Boole (j'en vois qui respirent), je vais donner dans le concret en présentant les opérations indispensables à connaître dans la programmation des processeurs.

(sources : M.Bigonoff)

Le complément

Que vous trouverez également sous les formes ”inversion” ou ”NOT” ou encore complément à 1. Elle est souvent notée ”!” Son fonctionnement tout simple consiste à inverser tous les bits de l'octet (0 devient 1 et 1 devient 0).

Exemple : NOT B'10001111' donne B'01110000'.

Vous voyez ici que pour les opérations booléennes, il est plus facile de travailler en binaire. Traduisez l'exemple ci-dessus successivement en hexadécimal (on dira maintenant ”hexa”), puis en décimal, et essayez de complémenter directement. Bonjour les neurones.

A quoi sert cette opération ? Par exemple à lire une valeur dont les niveaux actifs ont été inversés, à réaliser des nombres négatifs, ou autres que nous verrons par la suite.

(sources : M.Bigonoff)

La fonction ”ET” ou ”AND”

Appelée également multiplication bit à bit, ou ”AND”, et souvent notée ”&”.

Elle consiste à appliquer un mot sur un autre mot et à multiplier chaque bit par le bit de même rang. Pour faire une opération ”ET”, il faut donc toujours 2 octets.

Les différentes possibilités sont données ci-dessous (le tableau se lit horizontalement).

Première ligne : 0 AND 0 = 0. Ce type de tableau s'appelle ”table de vérité”

bit1	bit2	AND
0	0	0
0	1	0
1	0	0
1	1	1

On voit donc que la seule possibilité pour obtenir un ”1” est que le Bit1 ET le Bit2 soient à ”1”. Ceci correspond à une multiplication. $1 \times 1 = 1, 0 \times 1 = 0, 1 \times 0 = 0$.

Exemple :

Soit B'11001100' AND B'11110000' donne B'11000000'

A quoi sert cette instruction ? Et bien, elle est utilisée pour MASQUER des bits qui ne nous intéressent pas.

Prenez l'exemple ci-dessus : Le 2ème octet contient 4 bits à 1 et 4 bits à 0. Regardez le résultat obtenu : Les 4 premiers bits de l'octet 1 sont conservés (1100), à l'emplacement des 4 autres nous trouvons des 0.

On peut donc à l'aide de cette instruction positionner un ou plusieurs bits dans un mot à 0 sans connaître son contenu précédent.

(sources : M.Bigonoff)

La fonction ”OU” ou ”OR”

Encore appelée OR, souvent notée ”|” elle permet, comme son nom l'indique, de positionner un bit à 1 si le Bit1 OU le Bit2 est à 1.

La table de vérité suivante explique le fonctionnement de cette fonction.

bit1	bit2	OR
0	0	0
0	1	1
1	0	1
1	1	1

Petit exemple B'10001000' OR B'11000000' donne B'11001000'

A quoi sert cette instruction ? Et bien, tout simplement elle permet de forcer n'importe quel bit d'un mot à 1 sans connaître son contenu précédent.

Vous voyez que dans l'exemple précédent, les 2 premiers bits ont été forcés au niveau 1, indépendamment de leur niveau précédent.

(sources : M.Bigonoff)

La fonction ”OU EXCLUSIF” ou ”Exclusif OR” ou ”XOR”

Voici la dernière fonction que nous allons aborder dans cette mise à niveau. Elle est souvent appelée XOR[1]. Elle se comporte comme la fonction OR, à un détail près.

Pour obtenir 1, il faut que le Bit1 soit à 1 OU que le Bit2 soit à 1 à l'EXCLUSION des deux bits ensemble. Si les 2 bits sont à 1, alors le résultat sera 0.

Voici donc la table de vérité.

bit1	bit2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Petit exemple : B'10001000' XOR B'11000000' donne B '01001000'

A quoi sert cette instruction ? Et bien tout simplement à inverser un ou plusieurs bits dans un mot sans toucher aux autres. Dans l'exemple précédent, vous voyez qu'à l'emplacement des 2 bits à 1 du 2ème octet, les bits correspondants du 1^er octet ont été inversés.

Voilà, ainsi se termine le premier chapitre consacré au WDC65816. Je sais qu'il était particulièrement rébarbatif, mais, si vous ne maîtrisez pas parfaitement ce qui vient d'être expliqué, vous ne pourrez pas réaliser correctement vos propres programmes.[2]

(sources : M.Bigonoff)

[1] eXlusif OR

[2] Expérience de mon propre vécu !

3.3 Les Notations

Dans la suite de ce manuel vous serez confronté aux notations que j'effectue. C'est pour cela que je vais vous expliquer à quoi elles correspondent.

Dans le Manuel

Quand je parle en hexadécimale :	je place un $ devant le nombre => $50
Quand je parle en binaire :	c'est rare, mais je précise avant, et puis je place % devant le nombre => %0101
Quand je parle en décimale :	je précise avant, et dans le contexte vous vous en apercevrez, il n'y aura ni $ ni %

En programmation

Quand on travail avec	Type de valeur
une valeur en hexadécimale :	#$valeur
une valeur en binaire :	#

(sources : Moi)

3.4 Un mot sur les unités(sources : M.Bigonoff)

Nous avons parlé d'octets et de bits. En informatique, on utilise couramment les termes de Byte (octet) et bit (binary unit).

Les symboles courants utilisés sont :

Octet : « B » (pour byte) ou « O » (pour octet), ou encore « o »
Bit : « b »

Les anglais sont obligés d'utiliser le « B » Majuscule pour l'octet, pour éviter qu'on ne confonde avec bit. En français cette confusion n'existe pas, le terme bit ayant échappé à la traductomania française. Donc, « B » pour tous les pays, sauf pour la France où c'est « O » ou « o ».

Il nous reste cependant à parles des multiples (kilo, mega etc), et là la situation se gâte rapidement.

En effet, en base 10, notre base "de tous les jours", on a décidé d'adopter des multiples de 3 de la puissance 10. Ainsi par exemple,

kilo $= 10^3$
Mega $= 10^6$
Giga $= 10^9$
Tera $= 10^{12}$

Mais en travaillant en base 2, ces valeurs ne "tombent pas juste", et 10^3 ne représente en réalité qu'un nombre parmi d'autres (je vous laisse le convertir en binaire). Il fallait donc pouvoir représenter des "multiples" qui soient particuliers. On a donc procédé dans un premier temps à la récupération pure et simple des termes utilisés en base 10 (Kilo, Mega etc.), exactement comme on avait récupéré des lettres pour en faire des chiffres en hexadécimal. On leur a cependant réaffecté des valeurs en exposant de 2 par multiple de 10.

Ainsi, on a défini :

kilo $= 2^{10}$
Mega $= 2^{20}$
Giga $= 2^{30}$
Tera $= 2^{40}$

Or donc, un kilooctet en informatique valait 2^{10} octets, et donc 1024 octets, alors qu'un kilogramme valait, lui, 1000 grammes. La situation empirait pour le Mega, puisque si un Megaoctet valait 2^{20} octets, et donc 1048576 octets, un Megagramme, lui, valait 1000000 grammes.

Les vendeurs de matériel de stockage de masse ont sauté sur l'occasion. Ainsi, eux ont conservé l'appellation originale (légale) en puissance de 10, alors que tous les informaticiens raisonnaient en puissance de 2. Ainsi, lorsqu'on achetait un disque dur de 100Mo (100 Megaoctets), il faisait bel et bien 100.000.000 octets. Les OS (Windows) renseignant le plus souvent la taille en « Mega informatique », ils traduisaient dès lors cette capacité en (100.000.0000 / 2^{20}) = 95,4Megaoctets. Bref, 100 Megaoctets de constructeur de disque dur = 95,4 Megaoctets d'informaticien.

Il est clair que la situation ne pouvait perdurer. Début des années 2000, on décide donc de règlementer tout ça. Il était clair dès lors qu'on serait contraint d'inventer de nouveaux termes, ce qui fut fait, le premier étant le « kibi » qui représente le multiple de base : 2^{10}. On en arriva

donc aux conventions suivantes :

Kibi (Ki)	2^{10}	1024	1,024k
Mébi(Mi)	2^{20}	1048576	1,048586 M
Gibi (Gi)	2^{30}	1073741824	1,073741824 G
Tébi (Ti)	2^{40}	1099511627776	1,099511627776 T

Ainsi posées les nouvelles conventions, il n'existait dorénavant plus d'ambiguïté sur les termes utilisés. On devrait donc retrouver en informatique les termes Ki, Mi etc, au lieu de K, M... La situation réelle est loin de ressembler à ça, et la grande majorité des ouvrages informatiques (et la documentation SNMP) et des logiciels continuent à utiliser les kilo pour des puissances de 10. A vous d'être attentif à cette situation, qui mettra probablement des années à se régulariser.

Essayez de ne pas confondre le Gibi avec une marque de Wisky célèbre, ce n'est qu'une coïncidence, et se serait mauvais pour vos neurones.

Le Manuel de Programmation utilise les termes Kilo, Mega etc... car cette console date des années 90 alors on utilisera le vocabulaires de ses années là.
(sources : M.Bigonoff)

3.5 Vocabulaire

Il y a certains mots qui vont être cité dans la suite de la documentation, et il est indispensable que vous les connaissiez.

Commentaire

Les commentaires s'insèrent après le symbole ";". Vous pouvez mettre tous ce que vous voulez après ce symbole. Les commentaires se finissent à la fin de la ligne, pour écrire un commentaire sur la ligne du dessous, ou dessus, il faut ajouter ";" avant vos commentaires.

Prenez l'habitude de toujours commenter vos programmes. Soyez sûr que dans 6 mois, vous ne vous rappellerez plus ce que vous avez voulu faire, les commentaires vous seront alors d'une grande utilité si vous décidez de modifier votre programme.

Mnémonique-Mnémonic

En programmation informatique le mot mnémonique ,ou mnémonic en anglais, est lié à un concept bien précis. Il s'agit en fait d'une abréviation d'une instruction. Cela permet de faciliter la mémorisation, mais aussi et surtout de dominer le temps nécessaire à la saisie du code, à sa taille prise en mémoire et ainsi de suites. On parle aussi de "diminutif".

Instruction

Les instructions permettent de donner des ordres au processeur afin qu'il effectue des opérations. Tel qu'une addition, soustraction, déplacement de données d'un endroit à un autre, etc...

Chaque instruction est caractérisée par un numéro appelé **opcode** ou **code opération** pour que le processeur puisse comprendre ce qu'il doit faire.

Dans le cour, elles sont caractérisées en mots de 3 lettres.

Par exemple on a l'instruction : "Charger dans l'accumulateur", la mnémonique de cette instruction s'écrira "LDA"(Load to Accumulator). Par abus de langage, je dirais ; "l'instruction LDA" au qui sous entendra ; "la mnémonique LDA de l'instruction Load To accumulator". Pour ainsi aller plus vite dans le cour.

Opcode

L'opcode est un octet (8bits) qui permet au processeur d'effectuer une opération (addition soustraction etc...) Vous verrez avec les instructions, une colonne Opcode (voir l'annexe 24.2 "Classements par Instruction" à la page 507). Ce dernier sera en fait la traduction directe de l'instruction à la machine.

Par exemple nous on dira "Mettre la valeur 2 dans l'accumulateur", on écrira "LDA #$02", et le compilateur traduira pour le processeur "$A9 $02".

Attention, ceci est un exemple , car l'instruction LDA équivaut à $A9 dans un mode d'adressage immédiat. Si ce mode change, il aura une autre valeur. Mais on verra les modes d'adressages plus tard.

Opérande

L'opérande est ce qui suit l'opcode. Dans l'exemple de ce dernier on avait demandé à charger la valeur 2 dans l'accumulateur. L' instruction était "LDA", et la valeur "#$02". L'opérande est "$02", elle peut être sur 8 ou 16 bit (selon la configuration du 65816).

en 8 bits : Opcode - Operande

en 16 bits : Opcode - Opérande basse - Opérande haute

Par exemple, l'accumulateur n'est plus en 8 bits, mais en 16 bits, et l'on veut charger la valeur "$1234" dedans. On écrira "$LDA #$1234", ce qui va être traduit en "$A9 $34 $12". Les deux octets de l'opérande sont inversée, normal puisque le processeur va lire d'abords le premier octet, et ensuite le second.

L'opérande peut être, ou ne peut être, requise, selon le type d'opcode.

Directive

Elles ne font pas partie du programme, elles ne sont pas traduites en opcode, elles servent à indiquer à l'assembleur de quel manière il doit travailler. Ces directives sont visibles dans la section Directives du compilateur WLA à la page 335. Se sont des commandes destinées à l'assembleur lui-même.

Au contraire des instructions, qui elles seront traduites en opcode et chargées dans le 65816. Il est donc impératif de bien faire la distinction.

Flag/Indicateur

Un ”flag” est un drapeau en traduction littéral, mais dans le contexte où nous somme nous allons le traduire par ”indicateur”. Par exemple, le flag ”m” se traduit par l'indicateur ”m”, car ”m” indique un changement dans le comportement du processeur.

3.6 Parler le 65816

Comment parler le 65816 ? en assembleur et en language C. Ce dernier est très confortable d'utilisation, mais ne permet pas d'optimiser au maximum le code. J'avouerais que pour le moment il nous serait utile, mais on préfère le bon vieux assembleur.

Donc parler le 65816, c'est donner des instructions au processeur à exécuter , pour cela il utilise un vocabulaire (ses instructions plus les modes d'adressages) bien définis. Nous on va essayé de lui faire exécuter un ordre, et selon ses capacité, il exécutera dans un temps bien déterminé.

Par contre n'essayez pas tout de suite de dire "affiche la phrase ...", il ne comprendra pas. En fait se sera a nous de créer cette phrases avec le vocabulaire du microprocesseur.

Chapitre 4

Technique de programmation

Nous allons voir ici les techniques de bases de la programmation du 65816

4.1 Programmes arithmétiques

Nous allons voir l'addition, la soustraction, la multiplication, la division, et la soustraction. Chacun utilise un seul registre.

Additions sur 8 bits

Voici un programme exécutant un opération sur 8 bits :

```
1    LDA Adr1     ;Charger OP1 dans A
2    ADC Adr2     ;Additionner OP2 à OP1
3    STA Adr3     ;Sauver le résultat RES dans ADR3
```

Dans ce programme, deux opérandes de 8 bits, OP1 et OP2, stockés dans les adresses mémoire ADR1 et ADR2, ont été additionnées dans RES et stockés à l'adresse ADR3.

Particularité du 65816

le programme précédent serait complet pour la plupart des processeur, mais pour celui-ci, il faut rajouter deux instructions.

Premièrement, l'instruction ADC signifie réellement " Additionner avec Report" plutôt que "additionner" ; la différence se situe dans le fait qu'une instruction normale d'addition ajoute deux nombres, alors qu'une addition avec report additionne deux nombre plus la valeur du bit de report. Puisque nous additionnons des nombres sur 8bits, aucun bit de report n'est possible et, au moment où l'addition commence, on ne connait pas les conditions du bit de report (il a pût être sélectionné par une instruction précédente) ; il faut donc le remettre à 0.

L'instruction CLC "efface le report" le permet. Malheureusement, le 65816 n'autorise pas les deux types d'opérations. Seule l'opération ADC existe ; il faut donc prévoir l'effacement du bit de report avant toute addition sur 8 bits.

La seconde particularité de ce processeur est liée à son jeu de puissantes instructions décimales. Le processeur opère toujours dans l'un des deux modes, binaire ou décimal ; l'état dans lequel se trouve le 65816 dépend du bit D, bit d'état du registre P. Comme dans cet exemple nous opérons en binaire, il est nécessaire de veiller à la mise en fonction du bit D. L'instruction CLD le permet, en effaçant le bit D.

Naturellement, si tous les calculs sont effectués en binaire, le bit D est remis à 0 une fois pour toutes, en début de programme.

Le programme complet sur 8 bits est :

```
1    CLC          ;Remise à zéro du bit de report
2    CLD          ;Remise à zéro du bit décimal
3    LDA Adr1     ;Charger OP1 dans A
4    ADC Adr2     ;Additionner OP2 à OP1
5    STA Adr3     ;Sauver le résultat RES dans ADR3
```

On peut vouloir utiliser les adresses numériques réelles au lieu de Adr1, Adr2, et Adr3. Pour garder les adresses symboliques, on doit utiliser des pseudo-instructions qui spécifient la valeur des adresses symboliques de façon que, pendant la traduction, le programme d'assemblage puisse substituer aux adresses symboliques les adresses physiques effectives. En voici quelques exemples :

```
1 Adr1 EQU $100
2 Adr2 EQU $100
3 Adr3 EQU $100
```

En conclusion, une addition sur 8 bits ne permet l'addition que de nombres de 8 bits (entre 0 et 255), si le binaire naturel est employé. Pour la plupart des applications pratiques, il est nécessaires d'additionner des nombres de 16 bits ou plus, il faut donc utiliser la précision multiple. Voyons maintenant des exemples de calcul sur des nombres de 16 bits.

Addition sur 16 bits

Dans cet exemple, on présume que le premier opérande est stocké aux adresses mémoire Adr1 et Adr1 - 1. Puisque OP1 est maintenant un nombre de 16 bits, il nécessite deux emplacements mémoire de 8 bits. OP2 est stocké dans Adr2 et Adr2-1 ; le résultat est déposé aux adresses mémoire Adr3 et Adr3-1.

Notons que H(high) indique la moitié supérieur (bits 8 à 15) et L(low) indique la moitié inférieur (bits 0 à 7).

La logique de ce programme est la même que la précédente. Les moitiés inférieur des deux opérandes sont tout d'abord ajoutées, tout report occasionné par cette addition est stocké immédiatement dans le bit de report (C), puis les moitié de poids fort des deux opérandes sont additionnées à l'éventuel report et le résultat est rangé dans la mémoire. Voici ce programme :

```
1     CLC
2     CLD
3     LDA Adr1          ;Charger moitié inférieur OP1
4     ADC Adr2          ;Additionner les parties basses de OP1 et OP2
5     STA Adr3          ;Stocker la partie basse du résultat
6     LDA Adr1-1        ;Charger moitié supérieur OP1
7     ADC Adr2-1        ;(OP1 + OP2) Partie supérieur + report
8     STA Adr3-1        ;Stocker partie haute du résultat
```

Les deux premières instructions assurent que le processeur fonctionne pour le type de calcul voulu. Les trois instructions suivantes sont identiques à celles utilisées pour l'addition sur 8 bits ; elles additionnent la moitiés de poids faible (bit 0 à 7) de OP1 et de OP2. La somme, RES, est stockée dans l'emplacement mémoire d'adresse Adr3. (voir la figure 4.1 "Les opérandes pour addition sur 16 bits " à la page 61)

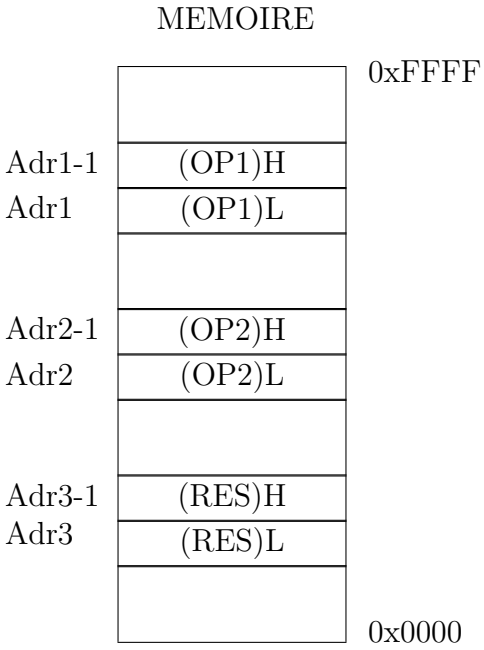


FIGURE 4.1 – Les opérandes pour addition sur 16 bits

Lorsqu'une addition est exécutée, tout report (0 ou 1) est sauvegardé dans le bit de report (C) du registre d'état (P). Si les deux nombres de 8 bits génèrent un report, alors ce dernier est équivalent à 1. Dans le cas contraire, la valeur du bit de report est de 0.

Les trois instructions suivantes sont identiques à celles utilisées dans le précédent programme. Cependant, elles additionnent, cette foie-ci, en plus du report, les parties de poids fort OP1 et OP2 et rangent le résultat d'adresse ADR3-1. Enfin, le résultat de 16 bits est stocké aux emplacements mémoire ADR3 et ADR3-1.

Mais qu'arrive-t-il lorsque l'addition des deux moitiés supérieures des opérations génèrent un report ? Cette situation peut être résolue de deux manières. D'abord, on peut penser que cela n'arrivera pas, car le programme fonctionne pour des résultat n'excédant pas 16 bits, donc pas 17, et qu'il s'arrêtera si le report est mis à 1. On peut aussi inclure des instructions mettant le bit supplémentaire dans un autre mot en mémoire, formant ainsi un total de 24 bits.

On remarque que, dans ce dernier programme, la partie supérieur de l'opérande est rangé "au dessus" de la partie basse, c'est à dire à l'adresse mémoire inférieur ; ce n'est pas une nécessité. En fait, le 65816 stocke les adresses à l'inverse : la partie basse en premier, la haute en second à l'adresse suivant. Néanmoins, la convention généralement adoptée est de mettre toutes les adresses et les données, partie haute au-dessus, comme le montre la figure voir la figure 4.2 "Stockage d'opérandes de 16 bits dans le 65816 " à la page 62

Lorsque l'on travaille sur des opérandes de plusieurs octets, il est important de rappeler les informations suivantes :

1. L'ordre dans lequel les données sont rangées en mémoire.
2. L'emplacement désigné par les pointeurs de données : octet inférieur ou octet supérieur.

Ainsi, il est important de savoir comment stocker les nombres 16 bits (partie supérieur ou inférieur en premier), et si les références d'adresses doivent indiquer la partie supérieur ou inférieur de ces nombres.

Les programmes présenté jusqu'à maintenant sont classiques, ils utilisent un accumulateur 8bits. voyons maintenant un autre programme pour l'additions sur 16 bits, qui n'emploie pas un simple accumulateur 8 bits, mais l'accumulateur 16 bits. Les opérandes seront stockés comme le montre la figure 4.2 : Stockage d'opérandes de 16 bits dans le 65816 .

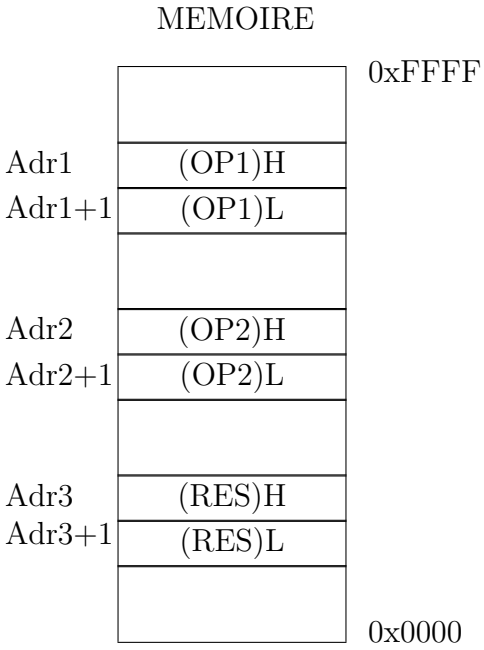


FIGURE 4.2 – Stockage d'opérandes de 16 bits dans le 65816

Le programme est :

```
1    REP #$20          ; initilialiser bit M de P
2    CLC
3    CLD
4    LDA Adr1          ;Charger l'accumulateur avec OP1
5    ADC Adr2          ;Additionner OP2 et OP1 (sur 16 bits)
6    STA Adr3          ;Stocker résultat dans Adr3
```

Ce programme est plus court que la version précédente ; la première instruction, Reset P, place le 65816 en mode accumulateur 16 bits, en mettant le bit M à 0.

Les nombres de 16 bits peuvent êtres facilement étendus à 24, 32 bits ou plus (toujours multiples de 8bit). En utilisant les instructions pour 16 bits que nous venons d'employer, nous allons maintenant écrire un programme d'addition pour des opérandes de 32 bits, en supposant que les opérandes soient stockés comme indiqué dans la figure 4.3 :

```
1    REP #$20          ; initilialiser bit M de P
2    CLC
3    CLD
4    LDA Adr1          ;Charger moitié inférieur OP1
5    ADC Adr2          ;Additionner moitié inférieur de OP2
6    STA Adr3          ;Stocker la moitié inférieur du résultat
7    LDA Adr1+1        ;Charger moitié supérieur de OP1
8    ADC Adr2+1        ;Additionner moitié supérieur de OP2
9    STA Adr3+1        ;Stocker moitié supérieur du résultat
```

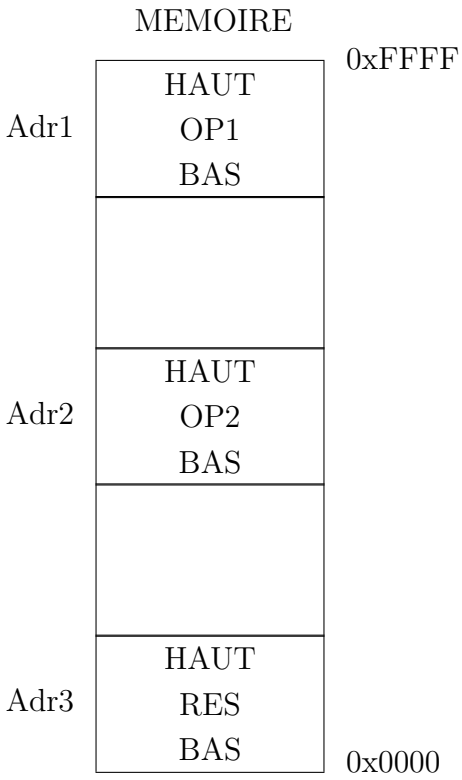


FIGURE 4.3 – Stockage d'opérandes de 32 bits

Soustraction

Comme à l'accoutumé, les deux nombre OP1 et OP2 sont stockés aux adresses Adr1 et Adr2. Pour soustraire, il faut l'opération de soustraction (SBC) à la place de l'opération d'addition (ADC). La seule modification, comparée au cas de l'addition, sera l'emploi de l'instruction SEC en début de programme à la place de CLE ; SEC signifie : "Mettre le report à 1", cela indique une condition de non-emprunt. La suite du programme est identique :

```
1     REP #$20          ; initilialiser bit M de P
2     CLD
3     SEC
4     LDA Adr1          ;OP1 dans A
5     SBC Adr2          ;OP1 - OP2
6     STA Adr3          ;Résultat dans Adr3
```

Ce programme est essentiellement le même que dans le cas de l'addition sur 16 bits. En arithmétique en complément à 2, la valeur finale du report indique un emprunt ; si une condition d'emprunt apparaît comme résultat de la soustraction, le bit de report de registre d'état du processeur est à 0 et peut être testé. Les exemples présentés jusqu'à maintenant étaient des additions et soustractions en binaire pur. Cependant, l'utilisation d'un autre type d'arithmétique, le BCD, peut être intéressante.

4.2 L'arithmétique BCD

L'addition BCD sur 8 bits

Le concept de calcul BCD a été développé au cours du Chapitre 1. On se souvient que celui-ci est généralement employé pour des applications de gestion, où il est impératif de garder chaque chiffre significatif dans les résultats.

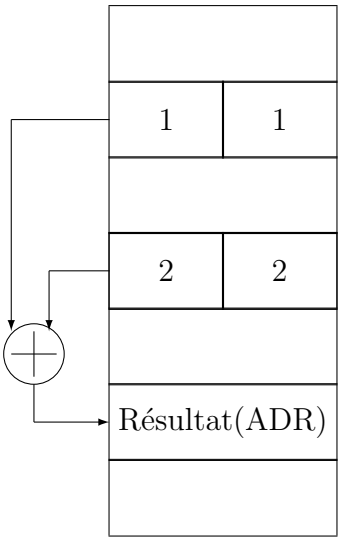


FIGURE 4.4 – Stockage de chiffres BCD

Dans la notation BCD, un quarter sert à représenter un chiffre décimal (0 à 9) ; chaque octet peut donc contenir deux chiffres BCD (BCD compacté). Additionnons d'abord deux octets, chacun contenant deux chiffres BCD (voir la figure 4.4 "Stockage de chiffres BCD " à la page 64.

Addition 01 et 02 :

01 est représenté par 0000 0001
02 est représenté par 0000 0010
Le résultat est : 0000 0011

Ce résultat est la représentation BCD de 03. Voyons un autre exemple :
08 est représenté par 0000 1000
02 est représenté par 0000 0010
Le résultat est : 0000 1010

Si vous obtenez 0000 1011, c'est que vous avez calculé la somme binaire de 8 et de 3. Vous avez obtenue 11 en binaire ; malheureusement, 1011 est illégal en BSD. La représentation BCD de 11 est 0001 0001. La différence provient du fait que la représentation BCD n'utilise que les 10 premières combinaisons de quatre chiffres pour coder les symboles décimaux de 0 à 9. Ainsi, les six combinaisons restantes de quatre chiffres sont inutilisées en BCD, la valeur 1011 est une de ces combinaisons illégales. En d'autres termes, lorsque la somme de deux chiffres BCD est supérieure à 9, on doit ajouter 6 au résultat pour éviter les six combinaisons non utilisées. Voyons un autre exemple où l'on additionne la représentation binaire de 6 à 1011 :

1011 (résultat illégal)
+ 0110 (+6)
le résultat est 0001 0001
Le résultat est bien 11 en notation BCD, nous obtenons le résultat correct.

Cet exemple illustre une des grandes difficultés du mode BCD ; on doit compenser la perte des six codes. Une instruction spéciale d'ajustement d'addition décimale (DAA) permet d'ajuster le résultat de l'addition binaire à de nombreux processeur. LE 65816 possède déjà se mode dans l'instruction ADC.

Le même exemple va servir maintenant à illustrer une autre particularité. Dans celui-ci, le report est généré entre le chiffre BCD le moins significatif(le + à droite) et le chiffre de gauche ; ce report doit être pris en considération et ajouté au second chiffre BCD. L'instruction d'addition le prend en compte automatiquement.

Comme dans l'addition binaire, CLC et SED placent le processeur en mode BCD. Voici un programme d'addition des nombres BCD 11 et 22 :

```
1     CLC           ;Effacer le report
2     SED           ;Initialiser le mode décimal
3     LDA #$11      ;charger le litteral BCD 11
4     ADC #$22      ;Additionner le littéral BCD 22
5     STA Adr3      ;Stocker le résultat
```

Dans ce programme, deux nouveaux symboles sont employés : # et $. Le symbole # signifie qu'un littéral (ou constante) suit. Le signe $, dans le champ opérande de l'instruction, spécifie que la donnée qui le suite est exprimée en hexadécimal. Les représentations hexadécimales et BCD pour les chiffres 0 à 9 sont identiques. La dernière ligne du programme stocke le résultat à l'adresse ADR.

La soustraction BCD

Elle paraît plus complexe ; en effet pour soustraire en BCD, il faut ajouter le complément à 10 du nombre. Le complément à 10 est obtenu en calculant le complément à 9 auquel on ajoute 1 ; cela implique, pour un microprocesseur standard, trois ou quatre opérations. Le 65816 est équipé d'uns instruction spéciale de soustraction BCD exécutant ce calcul automatiquement. le programme débute par les instructions SED (mode décimal) et SE (mise à 1 du report). Le programme permettant de soustraire la valeur BCD 25 et la valeur BCD 26 est le suivant :

```
1     SED           ;Initialiser le mode décimal
2     SEC           ;mettre à 1 le report
3     LDA #$26      ;charger la valeur BCD 26
4     SBC #$25      ;Soustraire la valeur BCD 25
5     STA Adr3      ;Stocker le résultat
```

L'addition BCD sur 16 bits

L'addition BCD sur 16 bits s'effectue aussi simplement que l'addition en binaire.

```
1     REQ           ;Initialisation du mode 16 bits
2     CLC
3     SED
4     LDA Adr1
5     ADC Adr2
6     STA Adr3
```

Indicateur (flags) BCD

En mode BCD, l'indicateur de report indique, pour l'addition, que le résultat est supérieur à 99. La situation est différente du complément à 2, car les chiffres BCD sont représentés en réel binaire. Inversement, l'absence d'indicateur de report pendant la soustraction ainsi qu'une retenue..

Addition en BCD compacté

En pratique, les nombres BCD comportent un nombre quelconque d'octet.

Nous présumons ici que les deux nombre, N1 et N2, incluent le même nombre d'octets BCD, et appelons ce nombre COUNT. La figure 4.5montre les registres et les allocations mémoire.

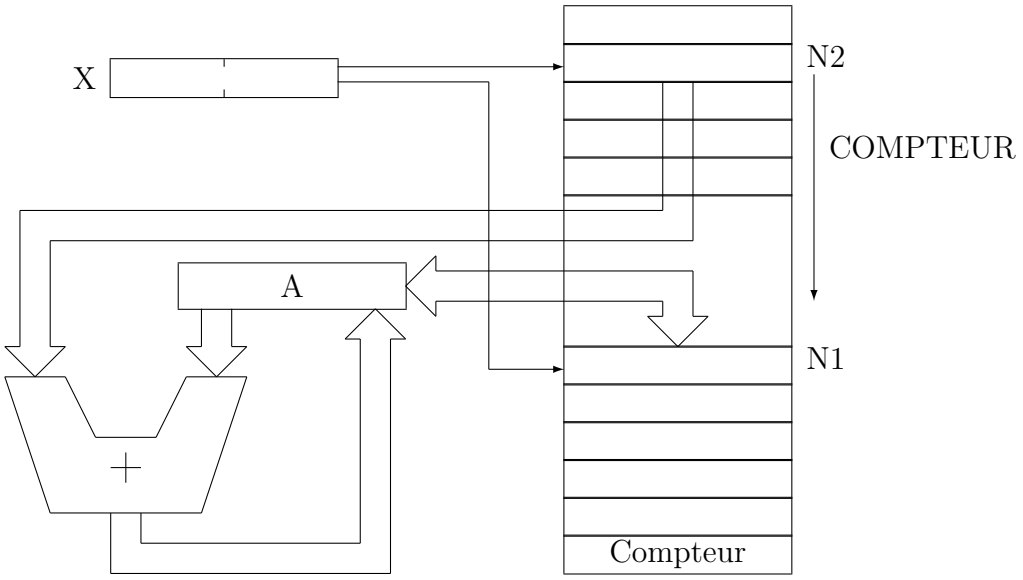


FIGURE 4.5 – Addition en BCD compacté N1 <- N2 + N1

Voici le programme :

```
1  BCDPACK:
2       LDA #COUNT
3       STA COUNTER
4       LDX #0   ; Mettre le registre X à zéro
5       CLC      ;Mettre le report à zéro
6       SED      ;Initialisation en mode décimal
7  PLUS:
8       LDA N2,X     ; Charger octet N2
9       ADC N1,X     ; Additionner octet N1
10      STA N1,X     ; Stocker résultat en N1
11      INX          ; Incrémenter X
12      DEC COUNTER  ; Compteur - 1
13      BNE PLUS     ; Boucler jusqu'à ce que compteur = 0
```

N1 et N2 représentent les adresses où les nombres BCD sont stockés. la valeur COUNT est mise dans le compteur d'adresses de mémoire et le registre d'index X est remis à 0 :

```
1  BCDPACK:
2       LDA #COUNT
3       STA COUNTER
4       LDX #0   ; Mettre le registre X à zéro
```

En prévision de la première addition, le bit de report doit être remis à 0 et le processeur initialisé en mode décimal :

```
1       CLC      ;Mettre le report à zéro
2       SED      ;Initialisation en mode décimal
```

Le premier octet de N2 est chargé dans l'accumulateur, puis le premier octet de N1 lui est ajouté ; le résultat est stocké en N1 :

```
1 PLUS:
2     LDA N2,X      ; Charger octet N2
3     ADC N1,X      ; Additionner octet N1
4     STA N1,X      ; Stocker résultat en N1
```

La forme N2,X indique l'utilisation de l'indexation absolue. l'adresse de l'opérande est formé en ajoutant N2 à X. Les registres d'index sont incrémentés, le compteur est décrémenté et la boucle d'addition est exécutée jusqu'à ce que le compteur atteigne la valeur 0 :

```
1     INX           ; Incrémenter X
2     DEC COUNTER   ; Compteur - 1
3     BNE PLUS      ; Boucler jusqu'à ce que compteur = 0
```

En utilisant le registre d'index, on gagne en rapidité et on simplifie le programme. Dans ce mode, l'instruction emploie la somme des contenus du registre d'index et l'opérande immédiat pour former l'adresse de la donnée. Pour des informations complémentaires sur le mode d'adressage, voir le chapitre 10 "Les différents modes d'adressage" à la page 117.

4.3 Multiplication

Un problème complexe d'arithmétique : la multiplication des nombres binaires. On commence par un exemple de multiplication décimale :

12 (multiplicande) x 23 (multiplicateur) 36 (produit partiel) + 24 = 276 (résultat final)

La multiplication s'effectue en multipliant le chiffre le plus à droite de multiplicateur par le multiplicande : 3 x 12 (produit partiel : 36) ; puis, en multipliant le chiffre suivant du multiplicateur (2) par 12, ce qui donne 24 qui sera ajouté au produit partiel.

Nous devons exécuter une opération supplémentaire, déplacer 24 d'un chiffre vers la gauche (le chiffre additionné à 36 est , en réalité, 240). Les deux nombres, correctement décalés, sont alors additionnés et la somme donne 276. Voyons maintenant un exemple de multiplicateur binaire dont la procédure reste la même. Multiplions 5 par 3

101 (multiplicande)(5) x 011 (multiplicateur)(3) 101 (produit partiel) 101 + 000 01111 (résultat final)(15)

Pour résoudre cette opération, nous avons procédé comme précédemment. La représentation formelle de cet algorithme est sur la figure 4.6

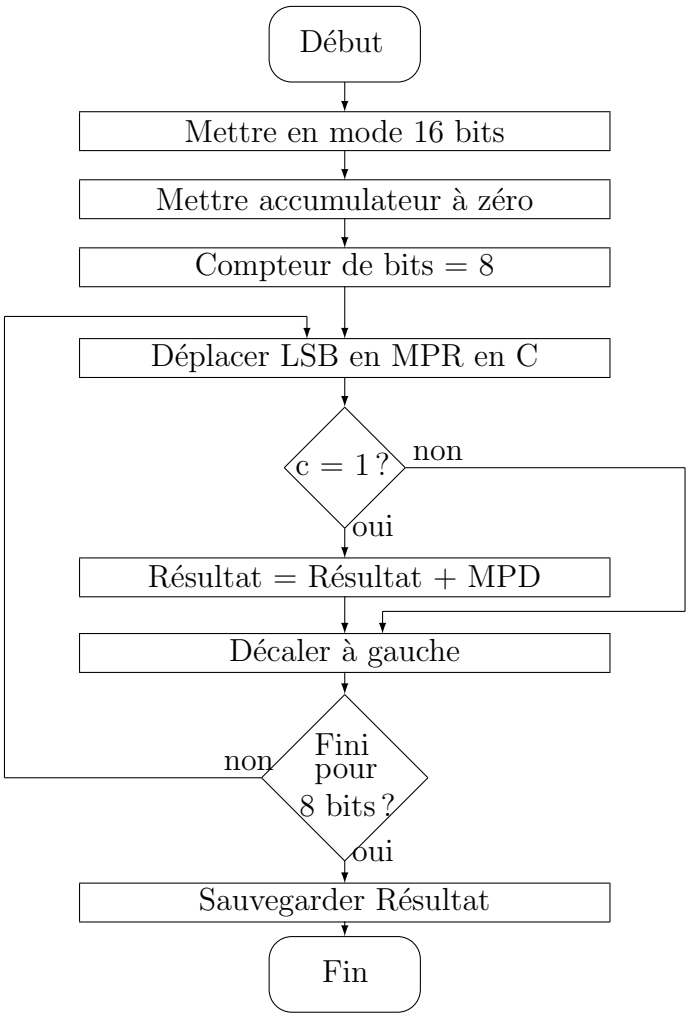


FIGURE 4.6 – Ordinogramme de l'algorithme de la multiplication

Multiplication 8 par 8

Nous allons maintenant traduire l'organigramme de la figure 4.6 en programme 65816. Remarquez que chaque boîte dans l'organigramme est traduite par une ou plusieurs instructions (dans se programme MPR et MPD ont déjà une valeur)

```
1   MULT88:
2       REP #$30      ; Initialiser les registres en 16 bits
3       LDA #0        ;Effacer le contenue de l'accumulateur
4       LDX #8        ; Mettre le compteur à 8
5   MULT:
6       LSR MPRAD     ;Deplacer MPR à droite
7       BCC NOADD     ;Tester Bit report
8       CLC           ;Preparer à additionner
9       ADC MPDAD     ;Additionner MPD à A
10  NOADD:
11      ASL MLPAD     ; Deplacer MPDAD à gauche
12      DEX           ; Décrémenter compteur
13      BNE MULT      ; Recommencer jusqu'à ce qeu compteur = 0
14      STA RESAD     ; Sauver le résultat
```

Multiplication des nombres 16 bits

La multiplication 16 par 16 a un produit de 32 bits qui nécessite deux mots de mémoire. Le multiplicande requiert également un mot de plus, car le bit le plus à gauche du multiplicande doit être sauvé à chaque fois qu'il est déplacé à gauche. L'adresse mémoire appelée TEMP est utilisé pour stocker les bits du multiplicande (voir figure 4.6). Voici le programme pour la multiplication 16 x 16 :

```
1   MULT88:
2       REP #$30      ; Initialiser les registres en 16 bits
3       LDA #0        ; Effacer le contenue de l'accumulateur
4       STA TEMP      ; Mettre TEMP à zéro
5       STA RESAD     ; Mettre Résultat bas à zéro
6       STA RESAD+2   ; Mettre résultat haut à zéro
7       LDX #16       ; Charger 16 Dans X
8   MULT:
9       LSR MPRAD     ;Deplacer MPR LSB à C
10      BCC NOADD     ;Tester report C
11      CLC           ;Preparer à additionner
12      LDA RESAD     ;Prendre résulat Bas
13      ADC MPDAD     ;Additionner MPD à A
14      STA RESAD     ; Sauver résultat Bas
15      LDA RESAD+2   ; Charger résultat haut
16      ADC TEMP      ;Additionner bit haut de MPD
17      STA RESAD+2   ;Sauver Résultat gauche
18  NOADD:
19      ASL MPDAD     ; Deplacer MPD à gauche
20      DEX           ; Décrémenter compteur de bit
21      BNE MULT      ; Recommencer jusqu'à ce qeu compteur = 0
```

Lorsque le multiplicande est déplacé à gauche, le bit de report doit être transféré dans le mot TEMP ; ce transfert s'effectue pas l'instruction ROL signifiant "rotation à gauche". Dans uns opération de permutation circulaire, par opposition à une opération de décalage, le bit arrivant dans le mot correspond au contenu du bit de report C (voir figure décalage et rotation, dans le l'instruction ROL). Le contenue de C est chargé dans la partie la plus à gauche de TEMP, et ainsi, le bit le plus à gauche de MPD se trouve transféré.

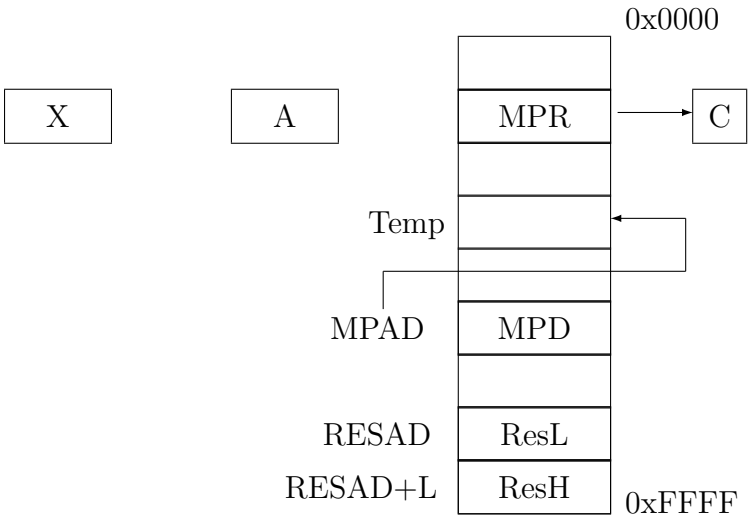


FIGURE 4.7 – Registre de Multiplication 16x16

4.4 Division Binaire

La division est un problème complexe, car il n'existe pas, dans le 65816, d'instruction de division ; il faut donc développer un algorithme pour écrire un programme de division. Voyons, pour commencer, une simple division décimale, 254 divisé par 12 :

```
        21   Quotient
/   12 254   dividende
        24
        14
        12
         2   reste
```

La division s'effectue en soustrayant le multiple le plus grand possible du diviseur des chiffres les plus à gauche du dividende ; le nouveau dividende est 14, le multiplicateur du diviseur devient le second chiffre du quotient, le reste est le résultat de la dernière soustraction.

Des essais de soustractions ou comparaisons sont nécessaires pour trouver le plus grand multiple du diviseur pouvant être soustrait du dividende. Notons qu'en déterminant le premier chiffre du quotient, le nombre est 20 et non pas 2, et que le nombre soustrait du dividende est 240, et non pas 24.

La division binaire est exécutée de la même manière qu'une division décimale. Voyons, par exemple, 10 divisé par 3 :

```
        0011   quotient
/   11 1010   dividende
          11
         100
          11
           1   reste
```

La représentation de cet algorithme apparaît dans la figure 4.8 présente une division 16 par 16, l'implantation en mémoire et le registre.

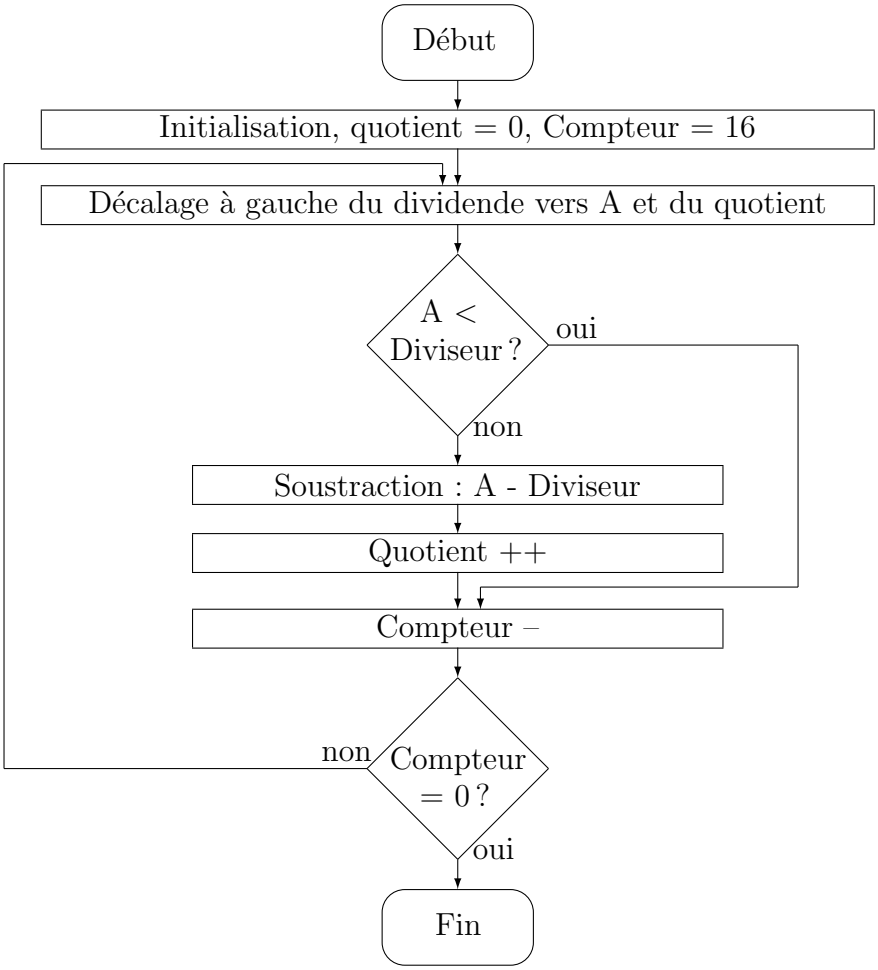


FIGURE 4.8 – Ordinogramme de la division binaire 16 bits

Voici ce programme

```
1   DIV16:
2       REP #$30      ; Initialiser les registres en 16 bits
3       LDX #16       ;Charger le compteur de Bits à 16
4       LDA #0        ; Effacer le contenue de l'accumulateur
5       STA QUOTAD    ; Mettre quotient à zéro
6   DIVD:
7       ASL QUOTAD    ; Déplacer quotient à gauche
8       ASL DVDAD     ; Déplacer dividande à gauche
9       ROL A         ;Deplacer dividende en A
10      CMP DVSAD     ; Comparer A avec Diviseur
11      BCC NOSUB     ; Si A<DVS, Sauter soustraction
12      SBC DVSAD     ; Soustraire DVS de A
13      INC QUOTAD    ; Additionner 1 au quotient
14  NOSUB:
15      DEX           ; Decrementer compteur
16      BNE DIVD      ; Boucler jusqu'à fin 16 bits
17      STA REMAD     ; Stocker reste en A
```

Ce programme introduit une nouvelle instruction, CMP, qui est une opération comparaison signifiant "comparer le contenu de l'accumulateur à celui de DVSAD de A". Elle soustrait le diviseur du dividende qui est décalé en A, mais ce n'est pas une soustraction normale puisque le contenu de A reste inchangé ; seul les bits du registre d'état sont affectés. Par exemple, si A égale DVS, le bit Z dans le registre d'état est mis à A. L'opération de comparaison fait une soustraction interne des deux opérandes, une adresse mémoire est ôtée de l'accumulateur et le registre d'état est établi selon le résultat de la soustraction. le opérandes sont inchangés, les indicateurs

d'état sont prêts à êtres utilisés pour une instruction de branchement.

Les programmes de division présentés jusqu'à maintenant présentent deux défauts possibles. Le premier est le manque de contrôle, dans le cas d'une division par zéro ; celle-ci, n'étant pas définie, constitue ainsi une condition d'erreur. (Le programme devrait vérifier la division au début. Si le diviseur est zéro, un branchement vers un code gérant l'erreur devrait être possible). L'autre problème est l'estimation des nombres sans leur signe. Ce problème est résolu en déterminant le signe du résultat à partir des signes du dividende et du diviseur avant que la division ne soit effectuée. Il suffit ensuite de convertir le dividende et le diviseur en nombres positifs et d'exécuter le programme de division. La phrase finale doit alors ajuster le signe du résultat au signe détermine avant la division.

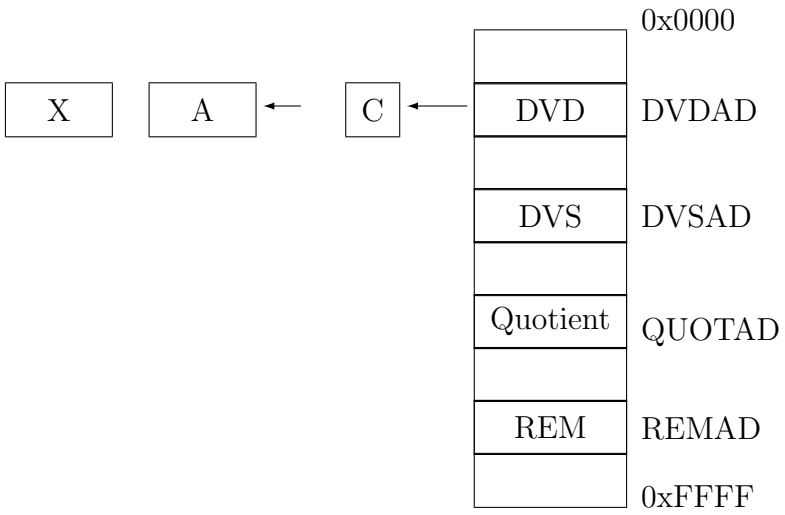


FIGURE 4.9 – Registre de la division 16 par 16

4.5 Opération Logique

D'autres instructions peuvent êtres exécutées par l'ALU[1] : ce sont les instructions logiques :

AND
OR
EOR

En outre, on peut également inclure les opérations de décalage et de permutation circulaire déjà utilisées, ainsi que l'instruction de comparaison (CMP). Le programme suivant permet vérfier si l'emplacement mémoire, appelé LOC, contient la valeur 0, la valeur 1, ou autre chose. Ce programme utilise l'instruction de comparaison et exécute une série de tests logiques ; selon le résultat de la comparaison, une portion de programme est alors exécutée.

```
1     LDA LOC ;Lire le caractère donc LOC
2     CMP #$00 ;Comprarer à zéro
3     BEQ ZERO ; est-ce zéro?
4     CMP #$01 ; Comparer à un
5     BEQ UN   ;est-ce un?
6  PAS-TROUVE:
7        ...
8        ...
9  ZERO:
10       ...
11       ...
12 UN:
13       ...
```

La première instruction,

```
1     LDA LOC ;Lire le caractère donc LOC
```

lit le contenu de l'emplacement mémoire LOC et le charge dans l'accumulateur A ; la donnée dans LOC est le caractère que l'on veut tester. L'instruction :

```
1     CMP #$00 ;Comprarer à zéro
```

Compare le contenue de A à la valeur hexadécimale 00 (la configuration de bits 0000 0000. Si cette instruction de comparaison est satisfaite, le bit Z du registre d'état est mis à 1. Ce bit est alors testé par l'instruction de branchement :

```
1     BEQ ZERO
```

Si la comparaison est satisfaite (si le bit Z a la valeur A), le branchement s'effectue, le programme pass alors à l'adresse ZERO. Si le test échoue, les instructions séquentielles suivantes sont exécutées :

```
1     CMP #$01
2     BEQ UN
```

De même, l'instruction de branchement suivante s'effectue un branchement à l'adresse UN, si la comparaison est satisfaite. Si aucune des comparaisons n'est positive, alors l'instruction à l'adresse PAS-TROUVE sera exécutée.

Ce programme illustre l'instruction de comparaison suivie d'un branchement, combinaison utilisé fréquemment dans les prochains programmes.

[1] ALU = Unitée Arithmétique et Logique, élément du micro-contrôleur permettant d'exécuter des opérations logiques

Chapitre 5

Les Outils de Programmation

Avant de commencer, nous allons prendre des outils de programmation qui nous servirons par la suite. J'en donne quelques uns qui me paraissent d'être le plus efficace pour programmer. Par contre si vous en avez qui vous paraissent bien plus efficaces, prévenez moi, je serais intéressé !

L'éditeur

En assembleur, généralement, l'extension des programmes se nomment ".asm", et les fichiers qui comportent des constantes à inclure ".inc".

Pour lire ceci, il existe le redoutable "notepad" de windows, et... franchement c'est pas sexy quand tout est de la même couleur. C'est pourquoi j'ai trouvé le "notepad++", qui permet de lire votre fichier avec des couleurs (que vous aurez au préalable prédéfini).

Le compilateur

Le compilateur utilisé est WLA. D'autres existent (dont un encore en développement, xkas par byuu), dont le code source est disponible.

Vous pouvez l'adapter à vos besoin, mais cela sera plus long.

Vous trouverez la dernière version de WLA ici :

http ://www.villehelin.com/wla.html.

Bon maintenant vous avez la base pour programmer. Veuillez suivre les tutoriels pour bien comprendre comment sa fonctionne.

5.1 Tutoriel : Notpad++

Pour commencer vous téléchargez le programme sur
http ://notepad-plus.sourceforge.net/fr/site.htm

Après avoir installé le programme, lancez le !

Maintenant, il faut définir l'environnement Super Nintendo dans lequel nous allons programmer.

Cliquer sur "Langage", là vous avez une liste de beaucoup de langage, excepté celui du 65816, donc on va ce le fabriquer nous même en cliquant sur **"user defined"** (tout en bas).

Pour le paramétrer, on vas cliquer sur **"affichage"** et **"panneau de langage défini par l'utilisateur"**.

On va déjà enregistrer le nom de notre langage définit en cliquant sur **"Enregistrer sous"** et mettre le nom de "Snes" ou autre c'est à votre choix ça ne me regarde pas !

en haut à droit dans le label **"Ext :"** on vas mettre les extensions de la programmation, "asm inc". La prochaine fois que vous ouvrez un .asm ou un .inc, ça va automatiquement sélectionné le langage **"Snes"**, bien que parfois il faut le faire manuellement...

Profitez en pour cocher **Ignorer la casse**, par exemple vous aller écrire "XCE" et plus loin "xce", ces deux là vont être pris en compte, sinon il faudra définir "xce" deux fois, une en majuscule deuxième en minuscule.

Onglet "Bloc & Défaut"

"Style par défaut",on ne change rien.

"Définition de bloc Ouvrant". Ici se trouve les instructions de compilations qui ouvrent un bloc, mettez : ".ASCTABLE .ASCIITABLE .ASM .BLOCK .DSTRUCT .IF .IFDEF .IFDEFM .IFEQ .IFEXISTS .IFGR .IFGREQ .IFLE .IFLEEQ .IFNDEF .IFNDEFM .IFNEQ .ENUM .SNESEMUVECTOR .SNESHEADER .SNESNATIVEVECTOR .MACRO .MEMORYMAP .REPEAT .REPT .ROMBANKMAP .SECTION .STRUCT"

"Définition de bloc Fermant". Ici se trouve les instructions de compilations qui ferment un bloc, mettez : ".ENDASM .ENDB .ENDE .ENDIF .ENDM .ENDME .ENDR .ENDRO .ENDS .ENDST .ENDEMUVECTOR .ENDNATIVEVECTOR .ENDSNES"

Onglet "Mots clé"

Ici le premier mot clé passe avant les autres, si un mot est définit dans le 1er et 2ème, il sera de la couleur du 1er.

Configurez-le comme vous voulez (police, gras, Italique, couleur, etc...)

"1er Groupe"

Pour les autres instructions de compilation qui ne forment pas un bloc mettez : ".EMPTYFILL EXPORT .FASTROM .HIROM .LOROM .OUTNAME .SLOWROM .SMC .8BIT .16BIT .24BIT .ACCU .ASC .BACKGROUND .BANK .BASE . BR .BREAKPOINT .BYT .DB .DBCOS .DBM .DBRND .DBSIN .DEFINE .DEF .DS .DSB .DSTRUCT .DSW .DW .DWCOS .DWM .DWRND .DWSIN .EQU .FAIL .FCLOSE .FOPEN .FREAD .FSIZE .INCBIN .INCDIR .INCLUDE .INDEX .INPUT .ORG .ORGA .PRINTT .PRINTV .RAMSECTION .REDEF .REDEFINE .ROMBANKS .ROMBANKSIZE .SEED .SHIFT SLOT .SYM .SYMBOL .UNBACKGROUND .UNDEFINE .UNDEF .WORD SLOTSIZE DEFAULTSLOT VERSION LICENSEECODE COUNTRY SRAMSIZE ROMSIZE CARTRIDGETYPE FASTROM SLOWROM HIROM LOWROM NAME ID COP UNUSED ABORT NMI RESET IRQBRK BRK IRQ"

"2^{e}groupe"

Servira pour afficher les chiffres (ou lettre) qui suivent les caractères, mettez : "# $ %" et cocher **"mode préfixe"**. Le mode "Préfixe" servira pour afficher les chiffres, ou lettres qui suivent le caractère sans espaces, de la couleur des caractères définis.

exemple :
- Si le mode préfixe est coché : $25
- Si le mode préfixe est coché, et qu'il y a un espace : $ 25
- Si il n'est pas coché :$25

"3^{e}groupe" Sera pour les instructions du 65816, mettez : "ADC AND ASL BIT CMP CPX CPY DEC EOR INC LDA LDX LDY LSR ORA ROL ROR SBC STA STX STY STZ TRB TSB BCC BCS BEQ BMI BNE BPL BRA BVC BVS CLC CLD CLI CLV DEX DEY INX INY NOP PEA PEI PER PHA PHB PHD PHK PHP PHX PHY PLA PLB PLD PLP PLX PLY SEC SED SEI TAX TAY TCD TCS TDC TSC TSX TXA TXS TXY TYA TYX XCE BRK BRL COP JML JMP JSL JSR MVN MVP REP RTI RTL RTS SEP STP WAI XBA" Se sont toutes les instruction du 65816, si il y a une erreur un email est sur la page de garde.

"4^{e}groupe" Pour la seule instruction de compilation qui a la couleur des instructions de bloc, mais qui ne forme pas un bloc : ".ELSE"

Onglet "Commentaire & Nombre"

On configurera la couleur des commentaires avec le mot qui mettre la couleur.

"Commentaire sur une Ligne" Choisissez la couleur la police etc... Personnellement j'ai mis en vert ! Ensuite cocher la case **"traiter comme un symbole"** ce qui signifie que tous ce que vous taperez après le(s) symbole(s) défini(s) sera de la couleur et de la police définit. D'ailleurs dans la case vous ajouterez " ;".

"Commentaire sur plusieurs Lignes" Rien à faire ici !

"Nombre" Configurez les nombres que vous allez écrire dans le programme, personnellement je les ais mis en rouge.

Onglet "Opérateurs"

Dans la section "Délimiteur 1", mettez comme borne ouvrante les guillemets (") et les mêmes en borne fermante.

Pour la couleur, j'ai mis du gris foncé, vous pouvez changer selon vos envies.

Au final à chaque string [1] vous aurez une ligne grise !

Voilà vous avez personnalisé l'affichage des documents ASM et INC pour visualisez correctement votre code ! !

[1] string = chaine de caractère en anglais, pour confirmer votre doute...

Paramètrer de la compilation

Si vous avez déjà fait un programme, en C C++ ou autre language, vous devez toujours compiler votre programme.

Ici nous avons WLA-65816 et WLALINK pour faire un programme sur la Super Nintendo. Or, ce n'est pas toujours évident de taper des lignes de codes pour compiler un projet ! On pourrait passez plus de temps à le compiler qu'à le programmer.

Premièrement vous aller suivre le tutoriel suivant (voir la section 5.2 "Tutoriel : Programme de compilation" à la page 79) pour créer votre programme de compilation automatique. Ce programme, pour la suite de ce tutoriel, nous allons l'appeler "WLA.bat".

Vous allez dans l'onglet "Exécution" (ou appuyez sur "F5"), et "Exécuter".

Le programme vous demande le fichier à exécuter, vous allez donc chercher "WLA.bat" qui se trouve normalement dans votre répertoire de travail.

(moi c'est "c :\Projets_SNES\Projet01\WLA.bat").

Là vous cliquez sur "Enregistrer...", et Notepad++ vous demandera le nom du "shortcut" (racourcis) et la série de touche à composer pour exécuter se racourcie !

Personnellement, j'ai mis "Name : WLA" et "F10" sans avoir coché les autres touches "CTRL", "ALT", ou "SHIFT", vous pouvez faire n'importe quelle combinaison, mais j'ai pris la plus simple !!! Au final vous enregistrez votre commande

Maintenant, vous avez un raccourci clavier pour lancer votre compilateur, au lieu de taper des lignes de commandes. Plus tard, vous pourrez rajouter d'autres outils utilisables de la même façon.

5.2 Tutoriel : Programme de compilation

(Le programmes est présent en annexes page 492)

Vous avez préalablement téléchargé WLADX de Ville Helin ? non ? Vous n'avez donc pas lu le début du chapitre ! Bien... rendez vous ici => http ://www.villehelin.com/wla.html pour télécharger ses derniers compilateur "WLA-65816" et linker "WLALink" (WLADX).

(commentaire) : plutôt que de polluer les répertoires Windows, il suffit de rajouter au "PATH" les chemins vers les exécutables de compilation. Le mieux serai d'avoir un répertoire avec les binaires utilisés pour la programmation Super Nintendo (compilateurs, mais aussi convertisseurs d'image, emulateurs, etc...).

Création d'un programme batch

Une fois dans le répertoire, créez un fichier "compilation.bat" ou un autre nom, mais d'extension ".bat". Ce dernier doit être dans le répertoire de vos sources de votre projet Super Nintendo.

Mettez les lignes suivantes :

```
1  @echo off
2  c:
3  cd \Projets_SNES\Projet01
4  echo [objects] > temp.prj
5  echo snos.obj >> temp.prj
6
7  echo on
8  wla-65816 -vo snos.asm snos.obj
9  wlalink -vr temp.prj snes.smc
10 @echo off
11
12 del snos.obj
13 del temp.prj
14
15 PAUSE
```

Explication du programme

Une explication s'impose, ceci est du BATCH, utilisé quand on voulait faire de petit programme sous DOS, (oui DOS, un OS de Microsoft dans les années 80-90). Sur internet vous trouvez plein de petit truc pour faire des petits programmes de ce style.

Déjà on cache les futures résultats des exécutions avec echo off.

Ensuite on se met dans le répertoire du projet, celui qui comporte les sources, ici j'ai mis "c :\Projets_SNES\Projet01", donc je vais dans le disque dur "c :" et dans le repertoire "cd \Projets_SNES\Projet01".

Après on met "[OBJECTS]" dans le fichier temp.prj, si ce fichier n'est pas créé il le sera automatiquement car ">" !

C'est le tour de "snos.obj", le nom du fichier aprés compilation, à être mit à la suite , avec ">>" dans le fichier temp.prj.

On affiche les futures résultats des commandes.

On exécute WLA-65816 avec comme fichier d'entrée "snos.asm" et objet de sortie "snes.obj". Vue que l'on compile en "objet", l'argument passé au WLA-65816 sera "-o", en plus on ajoute du "verbose", des traces de compilation, avec "-v", pour connaitre l'état de la mémoire, ce qui nous fera "-vo"

Maintenant que WLAlink sait quels objet sont disponibles, avec l'aide du fichier "temp.prj", on exécute WLAlink avec comme fichier d'entrée "temp.prj" et fichier de sortie "snes.smc". Pareil que pour WLA-65816, on ajoute un "verbose" pour avoir le résultat final de la mémoire.

Au final on cache encore le résultat de nos future commandes, et on efface les deux fichiers "temp.prj" et "snos.obj" car ils ont été compiler et linker.

Truc & Astuce

Si vous voulez créer plusieurs programmes avec le même fichier batch, vous devez rajouter un menu dans ce batch, et ajouter vos projets au fur et à mesure. (commentaire) section à supprimer. En effet, il est quand même rare d'avoir besoin de compiler plusieurs projets en même temps (surtout pour la Super Nintendo, où au final on cherche à obtenir un seul fichier smc).

Chapitre 6

Plan de programmation

Pour faire un programme ”basique”, voici les étapes à exécuter afin de pouvoir programmer sur la console.

1. Configuration du Mapping mémoire utilisé
2. Initialisation des informations de la Cartouche
3. Initialisation des tables d'interruptions
4. Initialisation des routines d'interruptions
5. Initialisation des Registres CPU et PPU
6. Allumer l'écran

Les sources du programme développé seront visible dans les annexes.

6.1 Configuration du Mapping mémoire utilisé

D'après le compilateur WLA-65816, on doit configurer correctement le mapping mémoire de la cartouche, après celui de la ROM, qui est dans la cartouche, et ensuite l'initialisation des registres de la Super Nintendo. Pour toutes informations sur le mapping mémoire aller voir le chapitre 15 "Mapping mémoire" à la page 253

(Programme en Annexes, voir page 493)

Avec les exemples, on va voir comment configurer le maping mméoire de la cartouche en mode 20 21 ou 25, ce qui correspond aux modes 3 principaux, mais il y a aussi d'autres mapping mémoire spéciaux ; TOP par exemple. Ce dernier ne sera pas vue dans ce document . Pour le moment on va faire simple avec deux directive d'assembleur :

```
1 .MEMORYMAP
2 .ENDME
```

Entre ces deux directives, vous devez indiquer la taille de vos banque et son nombre. Pour les exemples qui suivent, munissez vous du mapping mémoire de la Super Nintendo.

Deux autres viennes s'ajouter :

```
1 .ROMBANKSIZE
2 .ROMBANKS
```

La première directive (.ROMBANKSIZE) doit être informé de la taille d'une banque. Et la seconde (.ROMBANK), le nombre de banque présente de la taille indiqué dans .ROMBANKSIZE .

exemple 1 : Mode 20 LoRom

Toutes les banques ont une taille de 0x8000 (soit 32768 octets) et il y en a 0x7D (125).

```
1 .MEMORYMAP
2 DEFAULTSLOT 0
3 SLOTSIZE $8000
4 SLOT 0 $8000
5 .ENDME
6
7 .ROMBANKSIZE $8000
8 .ROMBANKS 8
```

Le slot cartouche par défaut commence à 0, si on en a plusieurs, on aurais rajouté SLOT 1, SLOT 2 etc... Ici nous n'en avons qu'un.

La taille de la cartouche (SLOTSIZE) est la taille d'une banque sur la cartouche, soit 0x8000.

Le slot sur lequel on va travailler doit avoir une adresse de départ (sans s'occuper des banques). D'après le mapping mémoire, l'adresse de commencement est à 0x8000.

ROMBANKSIZE prend la valeur de SLOTSIZE, et RAMBANKS annonce le nombres de banque de taille $8000 disponnible, soit 8.

La taille de la cartouche fera $8000 x 8 = $40000 soit 32Kbytes x 8 = 256KBytes

exemple 2 : Mode 21

Toutes les banques ont une taille de 0x10000 (soit 65536 octets par banque) et il y en a 0xFF-0xCO+1 = 0x40 (64) (alors pour traduire ce petit calcul, voir le chapitre 15 "Mapping mémoire" à la page 253) .

```
1 .MEMORYMAP
2 DEFAULTSLOT 0
3 SLOTSIZE $10000
4 SLOT 0 $0000
5 .ENDME
6
7 .ROMBANKSIZE $10000
8 .ROMBANKS 32
```

6.2 Initialisation des informations de la Cartouche

(Programme en Annexes, voir page 493)

Ici on informe donne les informations de notre ROM qui vont êtres prises en compte par la suite.

Les informations de la ROM sont à l'origine utilisées par Nintendo, qui forçait les développeurs à les informer sur les caractéristiques de leurs jeux. Cette partie est essentielle, car sinon, comment savoir qu'un jeu nécessite de la SRAM, ou encore quelle taille il fait. Idem pour les informations sur le mapping mémoire.

De plus, cette partie peut-être utilisée pour une cartouche qui lirait la rom à partir d'une carte MMC (au hasard... :D)

voir le chapitre 16 "Les informations de la Cartouche " à la page 257

Valeurs :

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0xFFB0	DE	FD	00	00	00	00	00	00	00	00	00	00	00	07	00	00
0xFFC0	"SNOS by Olorin11"															
0xFFD0	"3 cpy"					20	F7	0D	07	06	33	00	00	FF	00	00

Alors commençons sans plus attendre l'initialisation des informations de la cartouche [1] :

```
1  .SNESHEADER
2    ID "SNES"                          ; Identification en 4 lettres
3
4    NAME "SNOS by Olorin113V0r00"  ; Titre du programme en 21 bits/lettres (
         utiliser des espaces pour les bits non utilisés)
5
6    SLOWROM
7    LOROM                          ; Ici on c'est mis en lowrom et slowrom
8
9    CARTRIDGETYPE $00                  ; Type de la cartouche / $00 = ROM seulement
10   ROMSIZE $08                        ; Taille de la rom / $08 = 2 Mbits
11   SRAMSIZE $00                       ; Pas de SRAM
12   COUNTRY $01                        ; Localisation, U.S, Japon, europe etc...
13   LICENSEECODE $00                   ; Utiliser $00 tout le temps ;)
14   VERSION $00                        ; La version ($00=1.0, $01=1.1 etc...)
15 .ENDSNES
```

[1] pour plus d'infos voir la sous section ".SNESHEADER" à la page 439

6.3 Initialisation des tables d'interruptions

(Programme en Annexes, voir page 493)

La table d'interruption sert à référencer les programmes qui vont être exécutés lorsqu'une interruptions interviendra.

En mode émumlation, (utilisation du processeur 6502), selon le type d'interruption, on rentrera dans un certain programme.

```
1 .SNESEMUVECTOR
2 COP       $0000
3 UNUSED    $0000
4 ABORT     $0000
5 NMI       VBlank
6 RESET     Main
7 IRQBRK    $0000
8 .ENDEMUVECTOR
```

De même pour le mode natif, (utilisation du processeur 65816).

```
1 .SNESNATIVEVECTOR
2   COP       $0000
3   BRK       $0000
4   ABORT     $0000
5   NMI       VBlank
6   UNUSED    $0000
7   IRQ       $0000
8 .ENDNATIVEVECTOR
```

Les "$0000" signifie que l'on ne rentrera dans aucun programme.

Pour le moment, on initialisera l'interruption "VBlank". Le programme "Main" n'est que le retour au début du programme.

6.4 Initialisation des routines d'interruptions

(Programme en Annexes, voir page 494)

C'est ici que l'on initialisera les routines d'interruptions. Dans ces routines ont peut mettre ce que bon nous semble. Ici, nous allons placer les fonctions qui géreront chaque interruption. Plus tard, si nécessaires, ces fonctions pourront être enrichies.

```
1  .SECTION "INTERRUPTIONS" SEMIFREE
2  COPHandler:
3      REP #$30          ;A/Mem=16bits, X/Y=16bits
4      PHB               ;Sauvegarde du registre de banque de donnée (DBR) dans la
           pile
5      PHA               ;Sauvegarde de l'accumulateur dans la pile
6      PHX               ;Sauvegarde l'index X dans la pile
7      PHY               ;Sauvegarde l'index Y dans la pile
8      PHD               ;Sauvegarde du Registre de Page Zéro (D) dans la pile
9  ;METTRE LE CODE ICI
10     PLD               ;Remet le Registre de Page Zéro (D) sauvegardé dans la pile
11     PLY               ;Remet l'index Y sauvegardé dans la pile
12     PLX               ;Remet l'index X sauvegardé dans la pile
13     PLA               ;Remet l'accumulateur sauvegardé dans la pile
14     PLB               ;Remet le DBR sauvegardé dans la pile
15     RTI               ;Retour d'interruption
16
17 BRKHandler:
18     (...)
19     RTI               ;Retour d'interruption
20
21 VBlank:
22     (...)
23     RTI               ;Retour d'interruption
24
25 IRQBRKHandler:
26     (...)
27     RTI
28
29 .ENDS
```

6.5 Initialisation des Registres CPU et PPU

(Voir à l'annexe "Le Programme" à la page 495 pour avoir le programme)

Voici les registres de la Super Nintendo à Initialiser d'après la documentation :

Adresse	Données (en Hexa)		Adresse	Données (en Hexa)	
2100	8F		2120	00	00
2101	00		2121	00	
2102	00		2122	(donnée CG)	
2103	00		2123	00	
2104			2124	00	
2105	00		2125	00	
2106	00		2126	00	
2107	00		2127	00	
2108	00		2128	00	
2109	00		2129	00	
210A	00		212A	00	
210B	00		212B	00	
210C	00		212C	00	
	(Bas)	(Haut)	212D	00	
210D	00	00	212E	00	
210E	00	00	2130	30	
210F	00	00	2131	00	
2110	00	00	2132	E0	
2111	00	00	2133	00	
2112	00	00	4200	00	
2113	00	00	4201	FF	
2114	00	00	4202	00	
2115	80		4203	00	
2116	00		4204	00	
2117	00		4205	00	
2118	(Données Vram)		4206	00	
2119	(Données Vram)		4207	00	
211A	00		4208	00	
211B	00	01	4209	00	
211C	00	00	420A	00	
211D	00	00	420B	00	
211E	00	01	420C	00	
211F	00	00	420D	00	

Désactiver les interruptions

Pendant l'initialisation de la console, il faut désactiver les interruptions, afin de ne pas se retrouver dans un programme d'interruptions et d'aller taper dans un registre non initialisé !

Une seule ligne de code :

```
1 SEI   ; Désactivation des interruptions
```

Processeur en mode "Natif"

Qu'est que le mode natif ? Sur le processeur 65816 que possède la Super Nintendo, il y a deux modes, émulation (8 bits) et Natif (16 bits) !

D'après la datasheet du 65816, le mode "Emulation" emule un processeur 6502, donc nous avons tout d'un 6502, et pour basculer en mode "Natif" il y a une simple manip à faire.

Le Registre d'État du Processeur (P), à un bit qui s'appel "Emulation", écrit "e" par la suite.
e=1 (mode émulation activé, on travail avec un 6502)
e=0 (mode natif activé, on travail avec un 65816)

Pour mettre le "e" à 0, il faut y accéder par la carry.

Il faut utiliser l'instruction xce (voir l'instruction "XCE - Exchange Carry and Emulation Bits" à la page 238), pour échanger la carry avec le "Emulation Mode".

On veut du mode 16bits donc E=0
Resultat :
1-On charge 0 dans la carry
2-on échange la carry avec E

D'après le chapitre des instructions , on efface la carry avec l'opération clc (voir l'instruction "CLC - Clear Carry Flag" à la page 152), et on fait un xce pour charger 0 dans "Emulation Mode"

```
1 CLC
2 XCE ;Passage en Mode Natif
```

Initialisation des variables system, et du pointeur de pile

Il faut d'abord passer en mode binaire.

Dans le Registre du processeur :

- Index Register Select (bit x) à 8 bits
- Decimal Mode (bit d) à off (on passe en mode binaire)

Pour effacer un bit d'un registre, on peut le faire avec l'instruction rep(voir l'instruction "REP - Reset Status Bits" à la page 202) et sep(voir l'instruction "SEP - Set Status Bits" à la page 215) pour mettre à un. Ici on met à zéro le registre "Index Register Select" (x) et "Decimal mode" (d)(oui je répète...c'est pour que sa rentre !). Le bit 5 pour "X" et le bit 4 pour "D", 5éme bit => $10 AND 4éme bit => $08, ce qui nous donne un masque de $18 !!! On applique donc ce masque dans le Registre

```
1 REP #$18   ; Passage en mode binaire, X/Y 16 bit
```

Ensuite on doit initialiser le pointeur de pile (S), on à l'instruction txs(voir l'instruction "TXS - Transfer Index Register to Stack Pointer" à la page 231) pour faire ceci.

Maintenant on doit stocker la valeur $1FFF dans le S(Pointeur de pile).

```
1 LDX  #$1FFF ; paramétré le pointeur de pile à $1FFF
2 TXS         ; index X dans le pointer de pile
```

Initialiser l'écran

Initialisez l'écran et changez la luminosité de manière à avoir un résultat visible.

Le registre associé à l'écran est $2100

Il est expliqué que le bit 8 à 0 allume l'écran, et les bit 4 au bit 1, règlent la brillance. Nintendo dit qu'il faut activer l'écran, et mettre la brillance à fond ! la valeur est $0F. On charge $0F dans l'adresse $2100

Nous allons donc charger 0x0F dans l'adresse $2100. Pour cela, on charge d'abord la valeur 0x0F dans l'accumulateur (A), puis ce dernier est enregistré dans l'adresse $2100, tout cela grâce aux instructions LDA(voir l'instruction "LDA - Load Accumulator from Memory" à la page 174) et STA(voir l'instruction "STA - Sore Accumulator to Memory" à la page 216)

```
1 LDA #$0F
2 STA $2100
```

Initialiser les Registres Hardware

Initialiser les registre hardware à leur valeur par défaut

D'après la documentation de Nintendo, tout les registres de $2101 à $210C doivent être à zéro.

```
1  STZ $2101 ; OBJSEL Object Size & Object Data Area Designation
2  STZ $2102 ; OAMADDL Address For Accessing OAM (Object Attribute memory) le L est
       pour la partie basse (bit0->bit7)
3  STZ $2103 ; OAMADDH (la même que précédent) le H pour l'octet le plus Haut (Bit8
     ->Bit15)
4  STZ $2104 ; OAM DATA Data for Oam Write
5  STZ $2105 ; BG Mode Bg Mode & Character Size Settings
6  STZ $2106 ; MOSAIC Size & Screen Designation For Mosaic Display
7  STZ $2107 ; BSG1SC Adress for Storing SC-Data Of Each BG & SC Size Designation (
     Mode 0 à 6)
8  STZ $2108 ; BSG2SC cf registre $2107
9  STZ $2109 ; BSG3SC cf registre $2107
10 STZ $210A ; BSG4SC cf registre $2107
11 STZ $210B ; BG12NBA BG Character Data Area Designation
12 STZ $210C ; BG34NBA cf registre $210B
```

(commentaire) : Dans votre programme, il serait bien d'avoir des EQUS pour les registres, plutôt que d'utiliser les adresses. Exemple à mettre au début du programme :

```
1  EQUS OBJSEL $2101
```

Mettre les registres ”Scroll” à zéro

Les registres se trouvent à $210D jusqu'à $2114.

Il faut savoir que ces registres sont en 16 bits ! Et souvenez vous, on a configuré notre processeur en 8 bits. voir la sous section ”Initialisation des variables system, et du pointeur de pile” à la page 88).

Quand on fait deux accès consécutifs dans un registre 16 bits lorsque l'on est configuré en 8, le premier accès écrit dans la partie basse (Low) et le second accès écrits dans la partie haute (High).

Deux STZ consécutif pour écrire dans un registre 16 bits lorsque l'on est configuré en 8 bits.

```
1  STZ $210D ; BG1H0FS bit Low Horizontal Scroll Value Designation For BG-1
2  STZ $210D ; BG1H0FS bit High Horizontal Scroll Value Designation For BG-1
3  STZ $210E ; BG1V0FS bit Low Vertical Scroll Value Designation For BG-1
4  STZ $210E ; BG1V0FS bit High Vertical Scroll Value Designation For BG-1
5  STZ $210F ; BG2H0FS bit Low Horizontal Scroll Value Designation For BG-2
6  STZ $210F ; BG2H0FS bit High Horizontal Scroll Value Designation For BG-2
7  STZ $2110 ; BG2V0FS bit Low Vertical Scroll Value Designation For BG-2
8  STZ $2110 ; BG2V0FS bit High Vertical Scroll Value Designation For BG-2
9  STZ $2111 ; BG3H0FS bit Low Horizontal Scroll Value Designation For BG-3
10 STZ $2111 ; BG3H0FS bit High Horizontal Scroll Value Designation For BG-3
11 STZ $2112 ; BG3V0FS bit Low Vertical Scroll Value Designation For BG-3
12 STZ $2112 ; BG3V0FS bit High Vertical Scroll Value Designation For BG-3
13 STZ $2113 ; BG4H0FS bit Low Horizontal Scroll Value Designation For BG-4
14 STZ $2113 ; BG4H0FS bit High Horizontal Scroll Value Designation For BG-4
15 STZ $2114 ; BG4V0FS bit Low Vertical Scroll Value Designation For BG-4
16 STZ $2114 ; BG4V0FS bit High Vertical Scroll Value Designation For BG-4
```

Effacer la table des registres Vidéo

Les sprites sont des éléments graphiques. Une image peut-être composée de sprites. Pour faire simple, ils sont utilisés pour afficher une image sur votre télé.

Les adresses pour ces registres sont de $2115 jusqu'à $2133.

```
1  LDA #$80
2  STA $2115 ; VMAINC  VRAM Address Increment Value designation
3  STZ $2116 ; VMADDL  Adress for Vram Read and Write
4  STZ $2117 ; VMADDH  Adress for Vram Read and Write
5  STZ $2118 ; VMDATAL  byte Low Data for Vram Write
6  STZ $2119 ; VMDATAH  byte high Data for Vram Write
7  STZ $211A ; M7SEL  Initial Setting in Sreen Mode-7
8  STZ $211B ; M7A  Byte Low - Rotation/Enlargement/Reduction in mode-7, Center
       Coordinate Settings& Multiplicand/Multiplier Settings of Complementary
       Multiplication
9  LDA #$01
10 STA $211B ; M7A  Byte High & cf $211B
11 STZ $211C ; M7B  Byte Low & cf $2118
12 STZ $211C ; M7B  Byte High & cf $2118
13 STZ $211D ; M7C  Byte Low & cf $2118
14 STZ $211D ; M7C  Byte High & cf $2118
15 LDA #$01
16 STZ $211E ; M7D  Byte Low & cf $2118
17 STA $211E ; M7D  Byte High & cf $2118
18 STZ $211F ; M7X  Byte Low & cf $2118
19 STZ $211F ; M7X  Byte High & cf $2118
20 STZ $2120 ; M7Y  Byte Low & cf $2118
21 STZ $2120 ; M7Y  Byte High & cf $2118
22 STZ $2121 ; CGADD  Address for CG-RAM Read and Write
23 STZ $2122 ; CGDATA  Data For CG-RAM Write
24 STZ $2123 ; W12SEL  Window Mask Settings (BG1 BG2)
25 STZ $2124 ; W34SEL  Window Mask Settings (BG3 BG4)
26 STZ $2125 ; WOBJSEL  Window Mask Settings (OBJ Color)
27 STZ $2126 ; WH0  Window Position Designation
28 STZ $2127 ; WH1  Window Position Designation
29 STZ $2128 ; WH2  Window Position Designation
30 STZ $2129 ; WH3  Window Position Designation
31 STZ $212A ; WBGLOG  Mask logic Setting For Window 1 & 2 on each Sreen
```

```
STZ $212B ; WOBJLOG  Mask logic Setting For Window 1 & 2 on each Sreen
STZ $212C ; TM  Main Sreen Designation
STZ $212D ; TS  Sub Screen Designation
STZ $212E ; TMW  Window Mask Designation For Main Screen
STZ $212F ; TSW  Window Mask Designation For Main Screen
LDA #$30
STA $2130 ; CGSWSEL  Initial Setting For Fixed Color Addition Or Sreen Addition
STZ $2131 ; CGADSUB  Addition/Substrraction & Substraction Designation for each
    BG Sreen OBJ & Background Color
LDA #$E0
STA $2132 ; COLDATA  Fixed Color Data for Fixed Color Addition/Substraction
STZ $2133 ; SETINI  Screen initial Setting
```

Activer la surveillance du Joypad

On "Surveille" le joystick pour savoir s'il a bougé, s'il a dit quelque chose, s'il veut un café, un thé...

Petite information, le joystick fonctionne par "interruption", en fait quand vous appuyez sur une touche de la manette, celle-ci envoie un signal au processeur, sous forme d'interruption !

Les registres à initialiser sont de $4200 à $4209

```
STZ $4200 ; NMITIMEN  Enable Flag for V-Blank, Timer Interrupt & Standard
    Controller Read
LDA #$FF
STA $4201 ; WRIO  Programmable I/O Port (Out-Port)
STZ $4202 ; WRMPYA  Multiplicand By Multiplication
STZ $4203 ; WRMPYB  Multiplier By Multiplication
STZ $4204 ; WRDIVL  Byte Low Dividend by Divide
STZ $4205 ; WRDIVH  Byte High Dividend by Divide
STZ $4206 ; WRDIVB  Divisor by Divide
STZ $4207 ; HTIMEL  Byte Low H-Count Timer Settings
STZ $4208 ; HTIMEL  Byte High H-Count Timer Settings
STZ $4209 ; VTIMEL  Byte Low V-Count Timer Settings
STZ $420A ; VTIMEH  Byte High V-Count Timer Settings
STZ $420B ; MDMAEN  Channel Designation for General purpose DMA & Trigger (Start
    )
STZ $420C ; HDMAEN  Channel designation for H-DMA
STZ $420D ; MEMSEL  Access Cycle Designation In Memory (2) Area (Refer to "
    memory map")
```

Activer le pad

Préparer Le pad à être lu.

Registre $4212

```
LDA #$81
STA $4212 ; HVBJOY H/V Blank Flag & Standard Controller Enable Flag
```

Activer les interruptions

Après l'initialisation de la console, il faut activer les interruptions pour la bonne marche du programme.

Une seule ligne de code :

```
CLI  ; Activation des interruptions
```

Chapitre 7

Ses Premiers Programmes

7.1 Programme Principal

Une fois le tout initialisé, on va créer le programme principal.

Mettez les fichiers suivants dans le même répertoire : "main.inc", "interruptions.inc", "initSnes.asm", et "main.asm".

D'ailleur créez le fichier "main.asm", et mettez le programme qui se trouve en annexe page 497.

Les explications sont les suivantes :

On ajoute le fichier d'en-tête, main.inc en premier, c'est celui qui a la configuration du mapping mémoire.

```
1 ;=== Init Mapping Mémoire =====
2 .INCLUDE "main.inc"
```

On inclut les routines d'interruptions

```
1 ;=== Init routines d'interruptions =====
2 .INCLUDE "interruptions.inc"
```

Et on finis par inclure le sous-programme pour initialiser la console.

```
1 ;=== Init Registres CPU et PPU =====
2 .INCLUDE "initSnes.asm"
```

Après ces insertions, vous pouvez inclure d'autres fichiers qui contiennent d'autres routines, macro, sous-programmes, etc...

Maintenant on va mettre le programme principal. On défini la Banque dans laquelle on va mettre le programme principal, et le slot (pas obligé vue qu'il n'y en a qu'un).

La directive "org" défini l'adresse d'origine du programme dans la banque. C'est sur 4 octets, de $0000 à $FFFF selon le mode de ROM que l'on a choisie. Ici on a initialisé la console en mode 20, Lowrom, donc de $0000 à $8000 max. Voir le mapping mémoire pour plus d'information à l'Annexe MM page 4.

```
1 ;*******************************************************
2 ;                       MAIN PROGRAM
3 ;*******************************************************
4 .BANK 0 SLOT 0
5 .ORG 0
```

Après avoir mis notre programme à une certaine adresse, on commence par faire une SECTION, voir la sous section ".SECTION "Init" FORCE" à la page 431 pour plus d'information sur cette directive de compilation. Bien sur on doit fermer celle-ci par la directive "ENDS".

```
1 .SECTION "MainCode" FORCE
```

On met le label "Main", pour définir le programme principal, et ce label doit être au préalable, référencé dans la table des vecteurs d'interruption.

```
Main:
```

On appel le sous-programme (ou routine, c'est comme vous voulez) d'initialisation de la Super Nintendo.

```
    ;Initialisation de la Super Nintendo
    JSR InitSnes
```

Bon maintenant on rentre réelement dans le programme principal. On va commencer par un exercice tout bête, mettre l'écran en blanc (oui ça viens de Wikipédia, je sais j'ai honte :))

Le registre $2122 sert à changer le fond d'écran, voir le registre "$2122 CGDATA" à la page 283 pour plus d'informations.

Mais avant de changer ce registre, il faut d'abord forcer le VBLANK. Pour cela, il suffit juste d'éteindre l'écran...

Cela nous donne la séquence suivante :

- Se mettre en Accumulateur 8 bits (afin de paramétrer les registres PPU)
- Forcer VBLANK (eteindre l'écran)
- Paramétrer le registre $2122
- Allumer l'écran

Donc on commence à mettre l'Accumulateur en 8 bits.

```
    SEP     #$20
```

On force le VBLANK en éteignant l'écran.

```
    LDA     #%10000000
    STA     $2100
```

Ce registre est un peu spécial, il faut y accèder 2 fois pour paramétrer ses 15 bits.

On charge les bit 0 à 8 dans le registre $2122. Et ensuite les bits 9 à 15 (oui le 16 n'existe pas) dans le registre $2122.

```
    LDA     #%00000000  ; Charger l'octet bas de la couleur blanche.
    STA     $2122
    LDA     #%00000000  ; Charger l'octet haut de la couleur blanche
    STA     $2122
```

Et voilà, maintenant on allume l'écran, et on boucle à l'infinie.

```
    LDA     #%00001111  ;fin du VBlank, mettre la luminosité à 15 (100%).
    STA     $2100

    ; boucler à l'infini.
Forever:
    JMP Forever
```

Faite bien attention de fermer la section avec la directive de compilation ".ENDS"

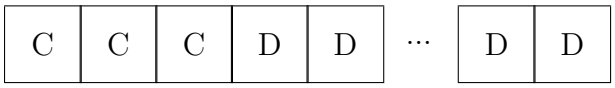
```
.ENDS
```

Vous avez fini votre premier programme.

7.2 Tableau - définition

Avant de voir les programmes sur les tableaux, nous allons regarder la base d'un tableau.

Un tableau simple ou alphabétique utilisent toujours la même base pour chaque élément, comme le montre la figure 7.1.



C = Etiquette à trois octets

D = Données

FIGURE 7.1 – Élément d'un tableau

Chaque élément possèdent 3 octets et un bloc de données de n octets, n étant compris entre 1 et 253. Chaque élément emploie au maximum une page (soit 256 octets). Dans chaque tableaux, tous les éléments ont la même longueur.

Nous allons utiliser les étiquettes suivantes, afin de nous savoir de quoi on parle :

- ENTLEN est la longueur d'un élément. Si par exemple, chaque élément a 9 octets de données, ENTLEN = 3 + 9 = 12.
- TABASE est la base du tableau en mémoire
- POINTR est le pointeur courant vers l'élément courant.
- OBJECT est l'élément courant à placer, insérer ou supprimer
- TABLEN est le nombre d'élément

7.3 Tableau Simple

Maintenant nous allons voir le tableau simple à n éléments.(voir la figure 7.2 "Tableau Simple " à la page 96).

Lors d'une recherche, le tableau est parcouru jusqu'à ce que la fin du tableau soit atteinte.

Pour une insertion, le nouvel élément est ajouté aux éléments existants.

Pour une suppression, les éléments éventuels ayant les adresses les plus élevées sont décalés vers le bas pour conserver la continuité du tableau.

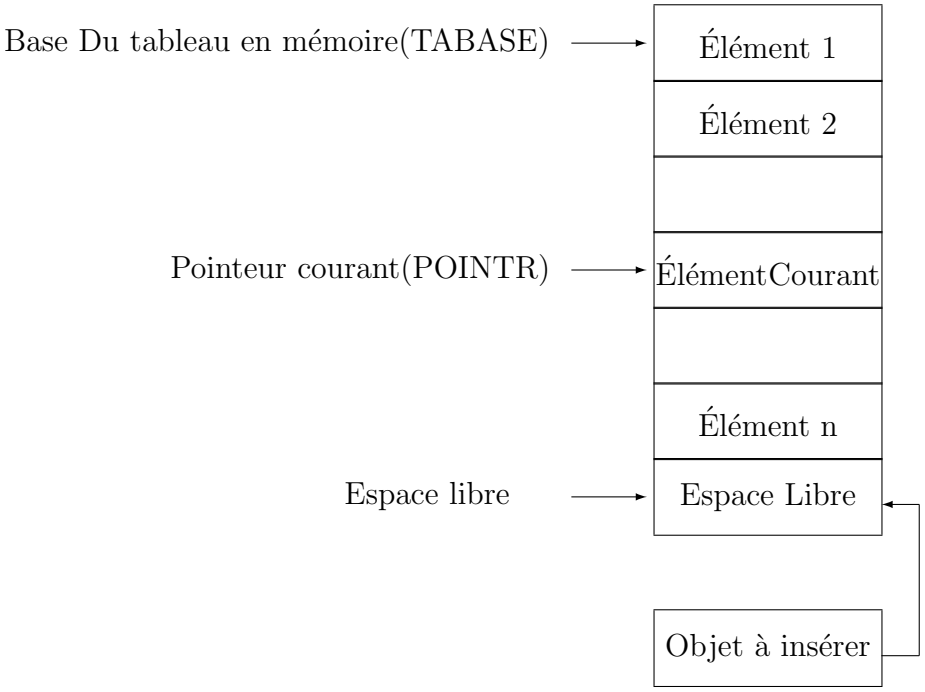


FIGURE 7.2 – Tableau Simple

Recherche dans un tableau

Voici maintenant un exemple utilisant une technique de recherche linéaire, où le champ étiquette (label) de chaque élément est comparé tour à tour à l'étiquette OBJECT, lettre par lettre.

Si le mode 16 bits est utilisé, deux comparaisons seulement sont nécessaires pour vérifier les trois octets de l'étiquette, mais la longueur des éléments doit correspondre à un nombre pair.

On utilise, dans ce programme, des adressages indexés absolus et indexés indirects directs. Vous trouverez l'ordinogramme à la figure suivante.

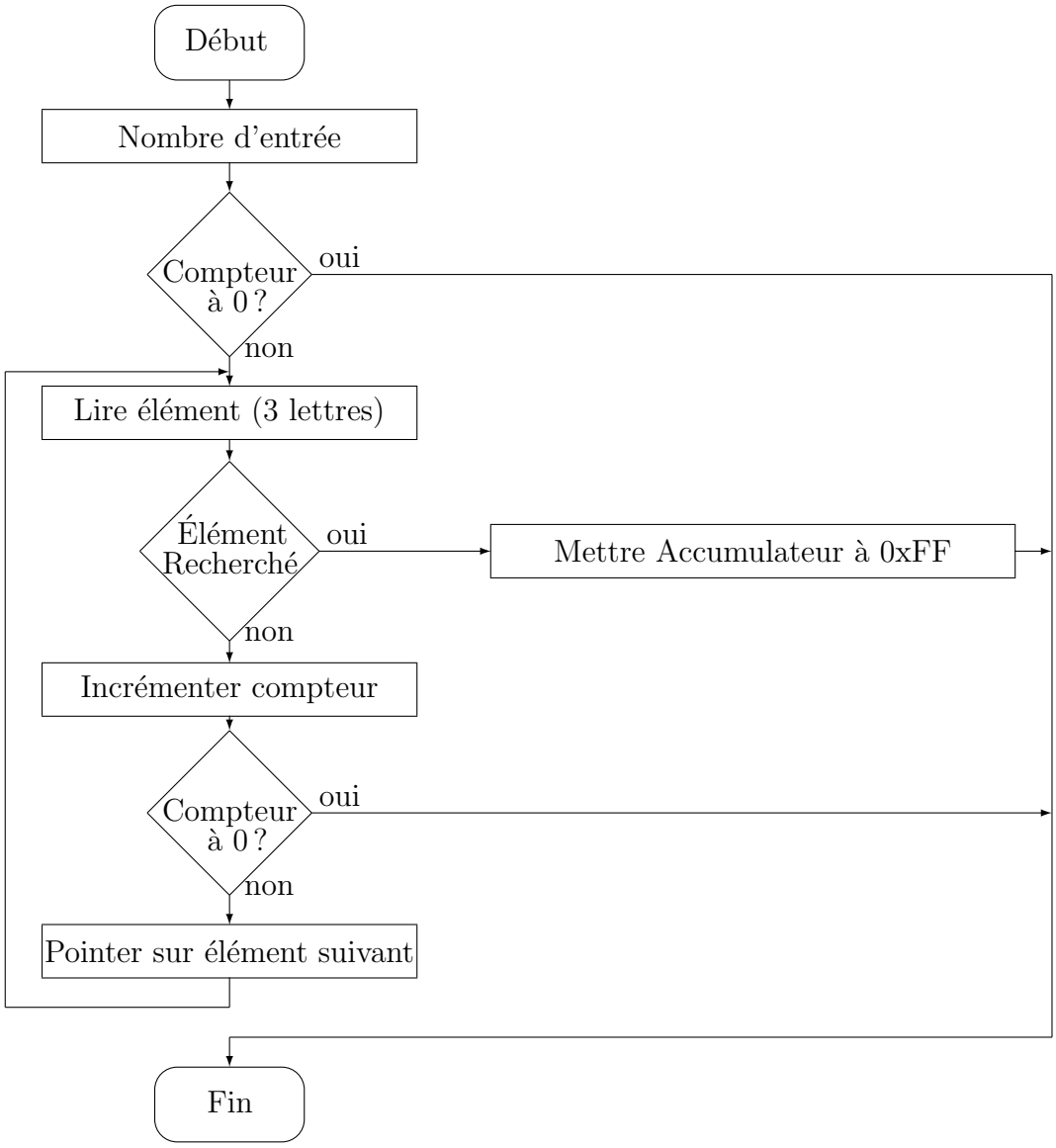


FIGURE 7.3 – Recherche d'un élément dans un tableau simple

Premièrement, on va faire pointer notre pointeur vers le début de notre tableau.

```
1 SEARCH:
2     LDA #TABASE ;mettre le pointeur vers le début du tableau
3     STA PONTR
```

Ensuite on va lire la longueur du tableau, et si elle est nulle on sort de se programme

```
1     LDX #TABLEN ;Lire la longueur du tableau
2     BEQ OUT
```

On initialise l'indexe Y à 0 et on compare les deux premiers octets à notre élément OBJECT.

```
1 ENTRY:
2     LDY #0
3     LDA OBJECT,Y
4     CMP (POINTR),Y ; On compare les deux premiers octets du tableau (case 0 et
          case 1)
5     BNE NOGOOD ; Si la comparaison n'est pas égal à 0, alors on va à l'étiquette
          NOGOOD
```

On incrémente Y pour aller pointer vers les 2^{e}et 3^{e}octets

```
1 ENTRY:
2     INY             ; pointer vers les 2\ieme et 3\ieme octets (case 2 et case
          3)
3     LDA OBJECT,Y
4     CMP (POINTR),Y ; On compare les deux premiers octets du tableau
5     BEQ FOUND ; Si la comparaison est égal à 0, alors on va à l'étiquette FOUND
```

Si ce n'est pas égal, on trouve l'étiquette NOGOOD. Ici on va vérifier si toutes les entrées ont été vérifiées.

```
1 NOGOOD:
2     DEX       ; On décrémente la longueur du tableau
3     BEQ OUT ; Si la longueur du tableau est à 0 alors on sort du programme
4     LDA #ENTLEN ; On va aller pointer vers l'élément suivant
```

Si il reste encore des éléments à tester, alors on vas réinitialiser le pointeur. Premièrement on va initialisé le bit de report (l'indicateur carry du registre P), ensuite on va additionner la valeur du précédent pointeur à la valeur de nouveau pointeur. En gros, on initialise le pointeur pour qu'il aille pointer sur l'élément suivant !

```
1     CLC           ; on efface l'indicateur de report : "Carry"
2     ADC POINTR   ; On additionne l'adresse du pointeur avec le bit de report avec
          précédent le pointeur stocké dans A. Le résultat se trouve dans l'acc
          umulateur.
3     STA POINTR   ; On stock le résultat précédent dans le pointeur
4     BRA ENTRY    ; On repart au début des tests.
```

Bon à la fin si on trouve notre élément dans le tableau, alors on va mettre l'accumulateur à 0xFFFF pour dire que l'on a trouvé l'élément, pour éviter tout problèmes et on sort du programme.

```
1 FOUND:
2     LDA #$FFFF
3 OUT:
4     RTS
```

Insertion dans un tableau

Quand on insère un nouvel élément, on utilise le premier bloc en mémoire, de ENTLEN octets, disponible à la fin du tableau (voir la figure 7.2 "Tableau Simple " à la page 96

Le programme vérifie d'abord que le nouvel élément n'est pas déjà dans le tableau ; on suppose, dans cet exemple, que les étiquettes sont toutes différentes. Si l'élément n'est pas trouvé, le programme incrémente la longueur du tableau. Le figure 7.4 correspondant au programme appelé NEW.

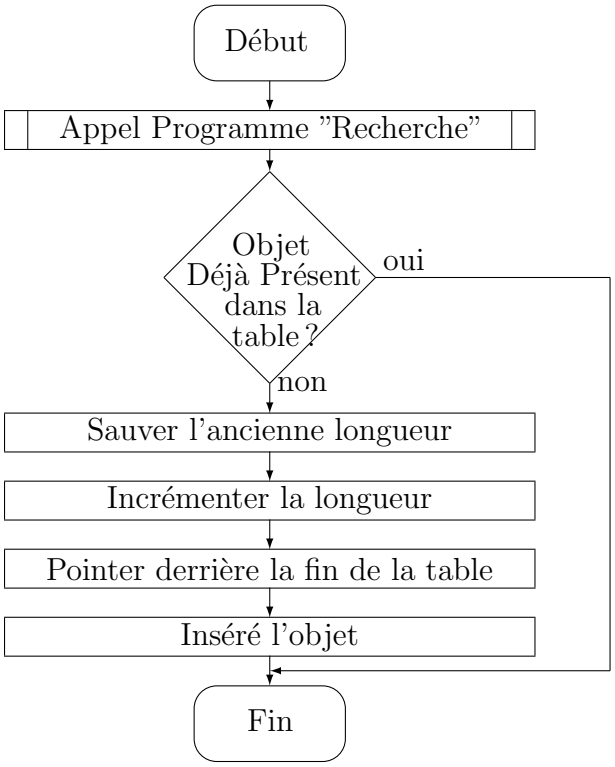


FIGURE 7.4 – Ajout d'un élément dans un tableau simple

On emploie le mode 16 bits seul ; la moitié des transferts correspondant au nombre d'octets de la liste est nécessaire.

Premièrement on se met en mode natif, car plus rapide, et surtout nous permet la manipulation des données sur 16 bits. D'abord on recherche si l'élément à ajouter est déjà présent dans le tableau

```
1 NEW:
2     ... ;procédure d'entrée
3     REP #$30 ;Mode 16 bits
4     JSR SEARCH ; on va chercher si l'élément est présent dans le tableau.
```

Une fois que le programme "Search" effectué, on teste à savoir en sortie l'élément cherché est présent où non

```
1     BNE OUTN ; Si le programme Search a trouvé l'élément (0 dans le bit d'état N
          ), alors on sort du programme
```

On insère la longueur du tableau dans l'accumulateur, si elle egale à zéro, on va pour insérer le premier élément. Sinon on va pointer à la fin du tableau pour mettre l'élément.

```
1     LDA TABLEN   ; Mettre la longueur du tableau dans l'accumulateur
2     BEQ INSERT   ; Tester si la longueur est à 0 ou non. Si 0, alors on va à l'
          étiquette "Insert"
3     LDA POINTR   ; Si la longueur est différente de 0, alors on va pointer à la f
          in du tableau. On insère le pointeur actuel dans l'accumulateur
4     CLC          ; on efface l'indicateur de report : "Carry"
```

```
5       ADC #ENTLEN ; On ajoute la valeur du pointeur avec la carry et la longueur d
            u tableau, pour ensuite pointer à la fin
6       STA POINTR  ; On stock le résultat précédent dans le pointeur courant pour
            pointer à la fin du tableau
```

Maintenant que l'on arrive à la fin du tableau, ou au début si il est vide, on ajoute alors l'élément. Pour cela la longueur du tableau se verra augmenter de 1.

Petite subtilitée dans se passage : on divise par deux la longueur d'un élément, car vue que l'on est sur 16 bits, on déplace donc 16 bits de données à chaque fois, est la longueur est exprimer le nombre d'octets total, 1 octet = 8 bits.

```
1   INSERT:
2       INC TABLEN  ; On incrémente le tableau de 1, car on ajoute un élément
3       LDY #0      ; On initialise l'index Y
4       LDA #ENTLEN ; charger la longueur d'un élément dans l'accumulateur
5       LSR A       ; On fait un décalage à droite de l'accumulateur. Ce qui revient
                      à le diviser par 2
6       TAX         ; Contenue de l'accumulateur dans X
```

Maintenant on prend l'objet et on le stocke à la fin du tableau

```
1   LOOPN:
2       LDA OBECT,Y ; On charge l'objet dans l'accumulateur
3       STA (POINTR),Y ; On stocke le contenue l'accumulateur dans le pointeur co
            urant, ajouter de Y
4       INY         ; On incrémente Y 2 fois
5       INY
6       DEX         ; On décrémente X, qui a le nombre d'élément sur 16 bits.
7       BNE LOOPN   ; On boucle jusqu'à ce que tous l'élément soient à la fin du
            tableau
```

```
1   OUTN:
2       ... ; procédure de sortie
```

Suppression d'un élément dans un tableau

Pour supprimer un élément d'un tableau, il suffit de décaler d'un cran vers le haut les éléments qui le suivant. il faut aussi diminuer la longueur du tableau (voir la figure 7.5 "Suppression d'un élément dans un tableau simple " à la page 101). Le programme correspondant est nommé DELETE.

(Momentanément indisponnible, faite travailler votre matière grise en attendant ;)))

FIGURE 7.5 – Suppression d'un élément dans un tableau simple

```
1  NEW:
2       ... ;procédure d'entrée mode natif
3       REP #$30 ;Mode 16 bits
4       JSR SEARCH ; on va chercher si l'élément est présent dans le tableau.
```

Une fois que le programme "Search" effectué, on teste à savoir en sortie l'élément cherché est présent où non

```
1       BEQ OUTD ; Si le programme Search n'a pas trouvé l'élément (0 dans le bit d'
            état N), alors on sort du programme
```

On décrémente la longueur du tableau, et on test si elle est à 0. Si c'est à zéro, vous êtes d'accord que cela ne sert à rien de chercher un élément dans un tableau vide.

```
1       DEC TABLEN ; on décrémente la longueur du tableau
2       BEQ OUTD   ; Si la longueur est à 0 alors on sort du tableau.
```

On pointe sur l'entrée après effacement de cette dernière.

```
1       LDA POINTR
2       CLC
3       ADC #ENTLEN ; pointe vers l'entrée après effacement de l'entrée
4       STA TEMPTR
5       LDY #0
6  LOOPD:
7       LDA #ENTLEN
8       LSR A
9       STA LENGTH
10      BNE MOVENT
11      DEX           ;Jusqu'à ce que toutes les entrées soient déplacées
12      BNE LOOPD
13 OUTD
14      ...       ; Procédure de sortie
```

Quatrième partie

LE PROCESSEUR 65816

Chapitre 8

Architecture du 65816

Le processeur 65816 est à la fois un processeur 8 et 16 bits. Il peut émuler un le processeur 6502, celui utilisé par la grande soeur de la Super Nintendo : La Nintendo !

Dans un premier temps il aura l'explication du 65816, le mode natif du 65816, et dans un second temp se sera au tour du 6502 émulé, le mode émulation du 65816 ! Au final vous aurez un tableau de comparaison entre les deux modes.

8.1 Mise ne route : 6502 Mode Émulation

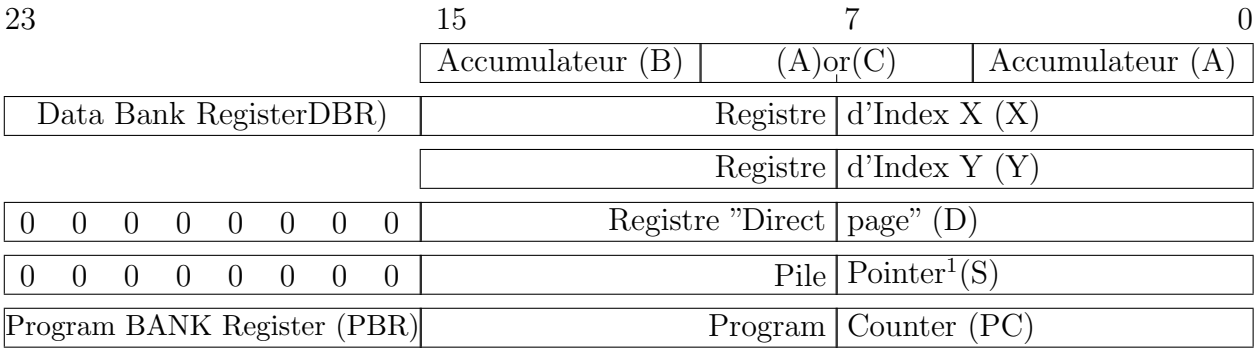
Quand le 65816 démarre, il s'initialise en 6502 mode émulation, qui émule le processeur 6502. La pile est confinée à la page une, comme le pointer de pile du 6502. Les registres sont configurés sur 8 bits, comme les registres du 6502. Tous les registres du 6502 sont implémenté à l'identique. Le timing de chaque instruction est exactement le même que le 6502 original. Le "Direct Page Register" (D) ; Registre de Page Zéro, du 65816, qui comme vous allez l'apprendre peut être reloué en utilisant son registre 16bits, est initialisé) la page zero, faisant un adressage du registre D, exactement comme l'adressage de la page zero du 6502.

Le programme et le registre de la banque de donnée (DBR), qui comme vous allez l'apprendre, fournie un accès efficace dans le 65816 à une ou deux banques quelconques de 64K de mémoire à la fois. Le programme et le DBR sont initialisés à la banque zero.

8.2 Le 65816

Ci dessous la représentation des registres de base du 65816

Accumulateur 16 bits & mode de Registre d'index : m = 0 & x = 0



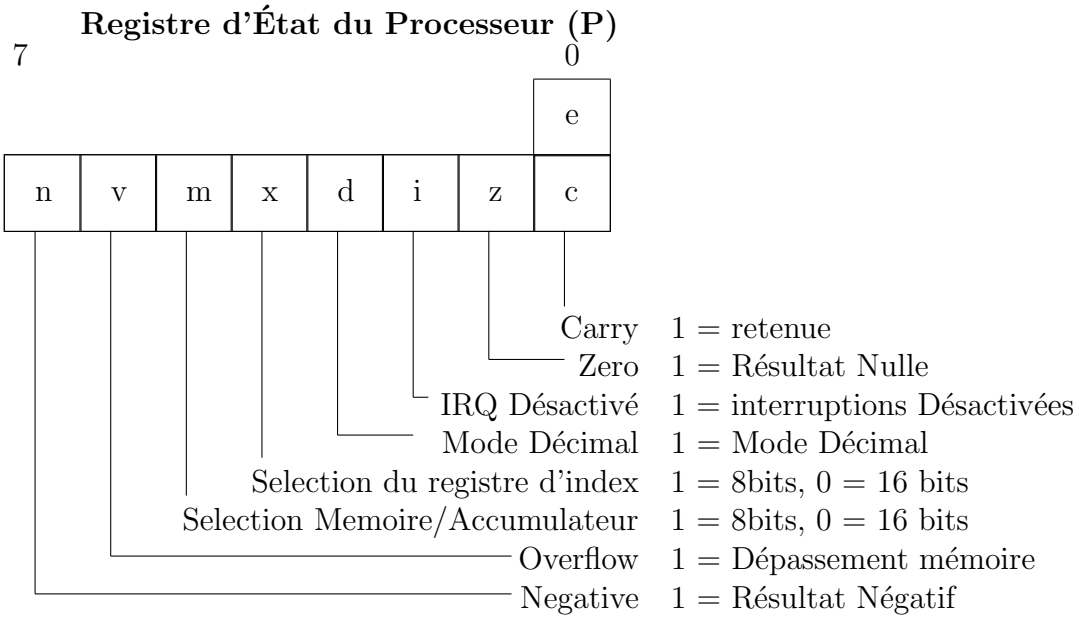


FIGURE 8.1 – Registres du 65816 mode Native

La figure 8.1 montre le modèle de la programmation du mode natif du 65816.

L'accumulateur est représenté sur un registre de 16 bits, il pourrait être paramétré pour être un accumulateur 16 bits (A ou C) ou un accumulateur 8 bits, un accessible (A) et l'autre moitié caché mais échangeable (B).

L'index de registre est représenté sur un registre de 16 bits, il peut être paramétré, comme paire, sur 16 ou 8 bits. Les 8 bit haut [1] sont à 0 quand il est paramétré sur 8 bits [2].

Il y a un avantage évidence de commuté depuis un processeur de registres 8 bits vers un 16 bits. C'est la capacité à écrire des programmes qui sont 25 à 50 fois plus petit, et qui fonctionne 25% à 50% plus vite, due à la facilité avec laquelle les données de 16 bits sont manipulé.

Il y a deux nouveaux registres de huit bits appelés : Registre de Banque (Bank Register). Un, appelé le "Data Bank Register" (DBR) ; registre de banque de données, représenté au-dessus du registres d'index, et l'autre, appelé le "Program Bank Register" (PBR) ; Registre de Banque de Programme, est apposé au "Programme Counter" (PC) ; Compteur de Programme. Le 65816 utilise les deux Registres de Banque afin de fournir un adressage de 24 bits.

[0] Pointer de pile

[1] bits 8 à 15

[2] seul les bit 0 à 7 sont paramétrables

Une Banque(bloc) mémoire c'est comme une page, la mémoire peut être définie par 8 bits (256 octets), ou par 16 bits (64 Koctets). Pour les processeurs comme les 6502, qui ont seulement un adressage de 16 bit, une banque de 64K n'est pas un concept approprié, puisque la seule banque est celle étant actuellement adressée. Les 65816, d'autre part, partitionne sa gamme de mémoire dans une banque de 64K de sorte que les registres de 16bits et des modes d'adressage puisse être utilisés pour adresser toute la mémoire.

8.3 Registre d'Etat (Status Register)(P)

Appelé Registre d'État du Processeur (P) ou encore "Status Register", et même ; registre "P". Ce registre 8 bits étant primordial pour un processeur, il indique l'état de ce dernier. Il est "sa conscience". Après l'exécutions de la plupart des instructions, ses états (appelé "flags" ou "indicateurs" dans le reste du document) changent ou pas de valeurs. Il configure les registres, active ou non les interruptions, etc...

bit	Nom	abréviation	Significations
7	négative	n	Se base sur le MSB du résultat [3] 0 : Résultat négatif 1 : Résultat positif
6	overflow	v	1 : Carry non valide, Résultat <-128 ou >+127 [4] 0 : Carry valide Résultat >= -128 ou <= +127 [5]
5	Memoire / Accumulateur	m	Indique si le registre 0 : Memoire et Accumulateur en 8bits 1 : Memoire et Accumulateur en 8bits
4	Registre d'Index	x	Indique si X et Y sont en 16bits ou 8 bits 1 : Registre 8 bits 0 : Registre 16 bits
3	Décimale	d	Indique le mode pour les opérations arithmétiques 1 : mode décimale, ex : 16 = $16 0 : mode binaire, ex : 16 = $10
2	Interruptions	i	Détermine l'activation des interruptions ou non 1 : Désactivées 0 : Activées
1	zéro	z	Indique si le résultat est nulle 1 : Résultat nulle 0 : Résultat positif ou négatif
0	carry	c	Indique si il y a une retenue au résultat 1 : Retenue 0 : Résultat positif
0	Émulation	e	Bit Fantôme, accessible depuis la carry [6] 1 : Mode Émulation 0 : Mode Native

[0]bit 7 en mode 8 bits, m=1, bit 15 en mode 16 bits, m=0
[0]en mode 16 bits, si résultat < à -32768 ou > 32767
[0]en mode 16 bits, si résultat >= à -32768 ou <= 32767
[0]voir l'instruction "XCE - Exchange Carry and Emulation Bits" à la page 238

8.4 Compteur de Programme (Program Counter)(PC)

(source M.Bigonoff)

Un processeur, quel qu'il soit est un composant qui exécute SEQUENTIELLEMENT une série d'INSTRUCTIONS organisées selon un ensemble appelé PROGRAMME. Il existe donc dans le processeur un SEQUENCEUR, c'est à dire un compteur qui permet de pointer sur la PROCHAINE instruction à exécuter. Ce séquenceur est appelé suivant les processeurs "compteur ordinal", "Pointeur de programme" etc. Dans le cas des 65816, il s'appelle PC, pour Program Counter.

Le principe de base est toujours le même. Dans le 65816, les registres ne font que 16 bits, on ne peut donc stocker qu'une adresse maximale de 65535. Il faudra donc 2 registres pour accéder à une adresse.

8.5 Registre de Banque de Programme (Program Bank Register)(PBR)

(source M.Bigonoff)

Nous trouvons tout d'abord un registre qui complète l'adresse du PC, se sont les 8 bits de poids fort. Il est appelé PBR (Program Bank Register) (voir la figure 8.1 "Registres du 65816 mode Native" à la page 106). Retenez bien que ce registre concaténé au PC, forment une adresse "Réel".

Si le PC a comme valeur "$2530" avec un PBR à "$45". La prochaine instruction à exécuter est à "$45 :2530".

Autre exemple. A l'annexe MM[1] page 1 vous avez d'un coté le PC sur 16bits, et en haut le PBR sur 8bits. Si vous voulez exécuter un bout de code à l'adresse $00 :4567 et ensuite aller à $12 :4458, vous devez changer de banque (bloc mémoire). Si vous mettez dans le PC $4458 sans toucher au PBR vous vous retrouverez à exécuter du code à l'adresse réel $00 :4458.**(source de Moi)**.

Il est très important de se rappeler que le PC pointe toujours sur l'instruction suivante, donc l'instruction qui n'est pas encore exécutée. C'est indispensable de bien comprendre ceci pour analyser les programmes en cours de debuggage.

8.6 Registre de banque de donnée (Data Bank Register)(DBR)

Ce registre permet de contrôler les données. Comme le PBR, ce registre gère les banques, et l'adresse réel d'une donnée c'est la concaténation de DBR avec celle-ci (qui est bien souvent exprimé en 16 bits).

Exemple : Si l'on charge la valeur $5000 et que le DBR vaut $47, votre adresse que vous voudrez charger se trouve à l'emplacement $47 :5000 de votre mémoire [2].

8.7 Registre de Page Zéro(Direct Page Register)(D)

Ce Registre de Page Zéro (D), pointe sur la première page, les premier 64k octets de la mémoire. Il est utilisé le plus souvent quand on exécute une instruction suivit d'une adresse de 8 bits ! Et oui le processeur voit les 8 bits, voie le DBR et se dit "bon j'ai l'adresse, mais j'ai seulement les bits de 0 à 7, et de 16 à 23 ! où sont passé les bits de 8 à 15 ?" et plantage, écran bleu...

Pour palier à ce problème, ce registre nous permet de "compléter" ces bits manquants.
Par exemple, vous voulez stocker l'adresse "$10" dans l'Accumulateur.

```
1 LDA $10 ; charger l'adresse $10 dans l'Accumulateur
```

Si D à été correctement initialisé, admettons D = $4700, vous chargerez la valeur qu'il y a à l'adresse $4710 [3] dans l'Accumulateur.

[1] Mapping Mémoire
[2] le " :" est là pour faire la séparation entre l'adresse de 16 bits et le DRB
[3] j'ai pas mit le DRB car j'ai voulu simplifier les choses

8.8 Accumulateur (A B ou C)

L'Accumulateur a plusieurs abréviations, cela dépend du contexte. Quand vous êtes paramétré en mode 8 bits, il est nommé "A" pour les bits de poids faible et "B" pour les bits de poids fort. Il sera nommé "C" quand vous serez en mode 16 bits.

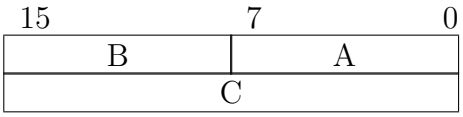


FIGURE 8.2 – Accumulateur

C'est un registre "primordiale" du processeur, il sert dans la majorité des opérations, c'est un registre mathématique tout usage. Plus concrètement, nous pouvons stocker grâce à lui toutes sortes de valeurs et effectuer des calculs dessus. Exemple : Vous souhaitez stocker la valeur "20" en hexadécimal :

```
1 LDA #$20
```

voir l'instruction "LDA - Load Accumulator from Memory" à la page 174

Ça place la valeur "20" en hexadécimal dans l'accumulateur.

Quand ce registre est sur 8 bits les valeurs pourront être comprises entre 0 et 255, et sur 16 bits les valeurs pourront être comprises entre 0 et 65535. Il est pratiquement impossible de savoir quand ce registre est en 8 ou 16 bits à moins que vous vérifier par vous même l'indicateur "m" (bit 5) du Registre d'État (P) (nous verrons ça un peu plus tard). En gros si cet indicateur est à un, ce registre est de 8 bits, sinon il est de 16 bits.

8.9 Registres d'Index (Index Register)(X et Y)

C'est quasi la même chose que l'Accumulateur, à la différence qu'ils sont beaucoup moins exploitables que ce dernier.

Ces registres sont généralement utilisés pour la composition d'une adresse, mode d'adressage indexé, ou compteur de boucles.

Ils s'incrémentent et se décrémentent facilement, par une seul instruction (chacune). Ils peuvent êtres utilisés pour accéder à une case dans un tableau, bouger la mémoire de place, et boucler à répétition. Contrairement à l'accumulateur, aucune opération logique ou arithmétique (autre qu'incrémenter, décrémenter ou comparer) ne peut s'exécuter sur eux. Mais on peut stocker des valeurs ou adresses comme l'accumulateur.

C'est le bit "x" de P qui les configurent en mode 8(x=1) ou 16 bits (x=0).

A retenire, deux registres X et Y.

8.10 Pointeur de pile (Stack Pointer)(S)

Le pointeur de Pile est ici pour sauvegarder des variables à court thermes. Une valeur que vous réutiliserez un peu plus tard par exemple. Ca fonctionne comme une LIFO. Dernière valeur entré, première sortie. C'est comme vos assiettes de votre cuisine, vous prenez celles d'au-dessus pour manger, et vous les reposer une à une au dessus [1].

Par contre il faut préciser que ce registre est sur 16 bits, et que d'après la figure (voir la figure 8.1 "Registres du 65816 mode Native" à la page 106) vous n'avez rien d'autre pour préciser en quelle banque vous travaillez. Si il n'y a pas de précision, vous êtes, par défaut, en banque $00.

La longueur de la pile est limitée à 256 octets, et son emplacement se trouve entre $100 à $1FF. Le pointer est initialisé à $1FF et se décrémente jusqu'à $100 .

Le pointeur de Pile, va dire au début "Je pointe sur la prochaine case vide de la pile". Ensuite, dés que vous stockez une valeur, le pointeur vous dit "je vais à la case libre suivante". Les valeurs mises dans la Stack ont une taille de 16bits (2 octets), si cela dépasse, la suite de la valeur sera stocké dans la case libre suivante.

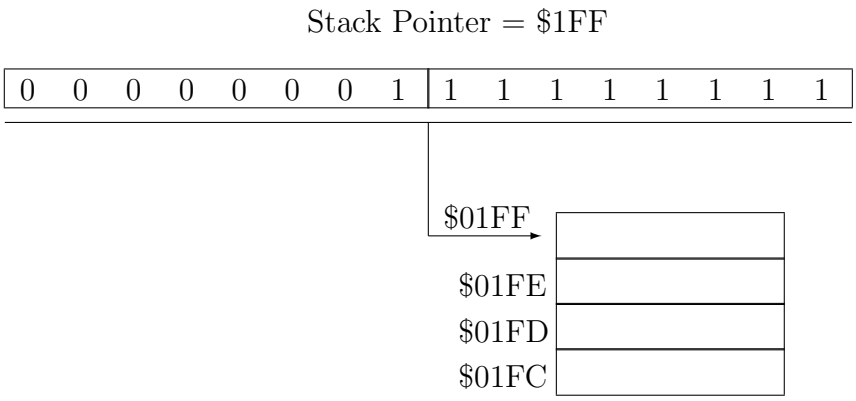


FIGURE 8.3 – Stack Pointer vide

D'après la figure ci-dessus, vous avez le pointer qui se trouve à $1FF, qui pointe la prochaine case vide. Quand une valeur sort de la pile, le pointeur s'incrémente de un (de $1FE à $1FF). Quand on rentre une valeur il se décrémente de 1 (de $1FF à $1FE par exemple).

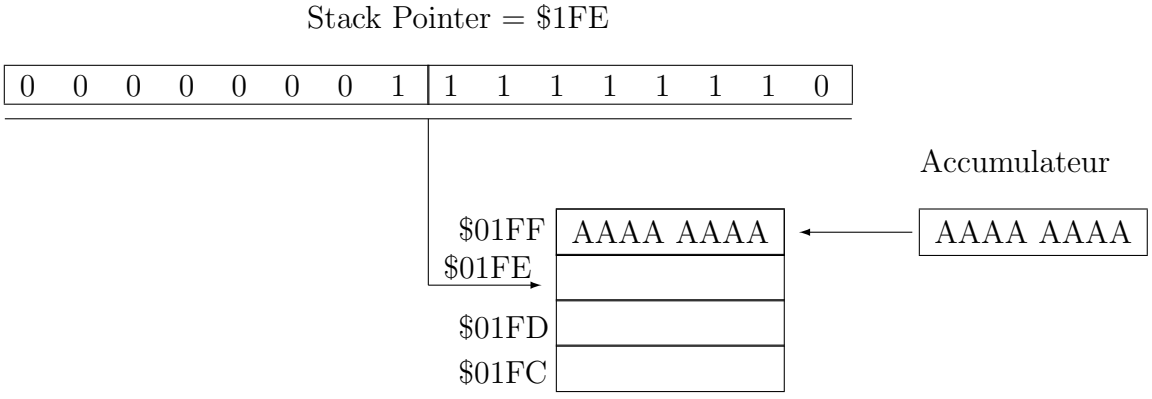


FIGURE 8.4 – Pointeur de pile après chargement

[1] une fois lavé bien-sûr

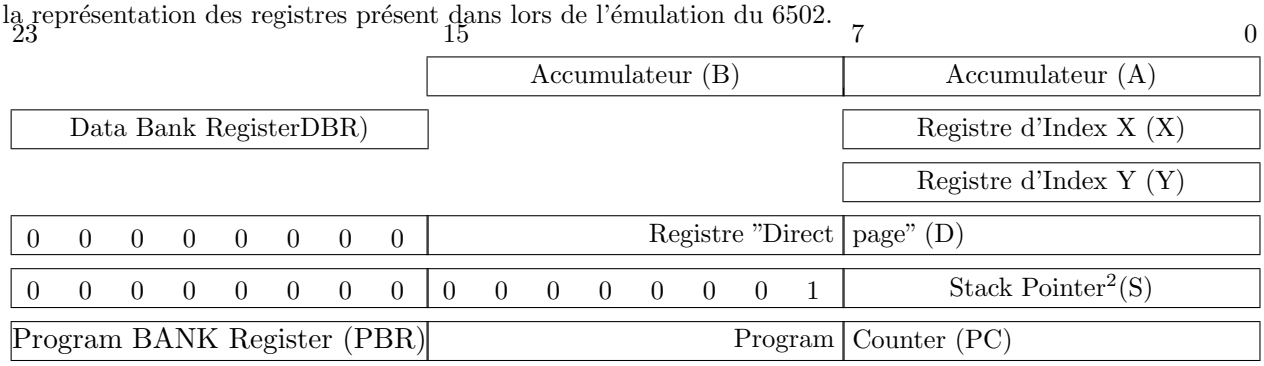
8.11 Le 65816 : Mode Emulation

Dans le mode émulation du 65816, nous simulons un processeur : le 6502, qui est sur 8 bits. La suite vous explique ce que les caractéristique d'un 6502 "émulé" car il y a quelques différences par rapport à un vrai 6502.

Les registres du 6502 sont :

- L'Accumulateur, ou registre "**A**", est le registre le plus utilisé, et manipulé, ils contiennent le plus souvent les résultats d'une opération. Vous le verrez beaucoup par la suite
- Les registres d'**index X** et **Y** sont utilisés principalement en formant des adresses réelles, pour des accès mémoire, et comme compteurs de boucle.
- Le **Registre d'État du Processeur**, ou registre "**P**", registre contenant les huit bits pour indiquer les différents changements après les instructions
- Le **Pointeur de pile**, ou registre "**S**". C'est un pointeur qui montre le prochain emplacement disponible dans la pile, un endroits spécial de la mémoire lui est alloué, pour stocker des données provisoires. En plus d'être à la disposition de l'utilisateur, le pointeur de pile et la pile sont également employés automatiquement chaque fois qu'une sous-routine l'appelle ou une interruption se produit pour stocker l'information de retour.
- Le **Program counter**, ou "**PC**", est un pointeur mémoire où se trouve la prochaine instruction à exécuter.

Voici la représentation des registres présent dans lors de l'émulation du 6502.



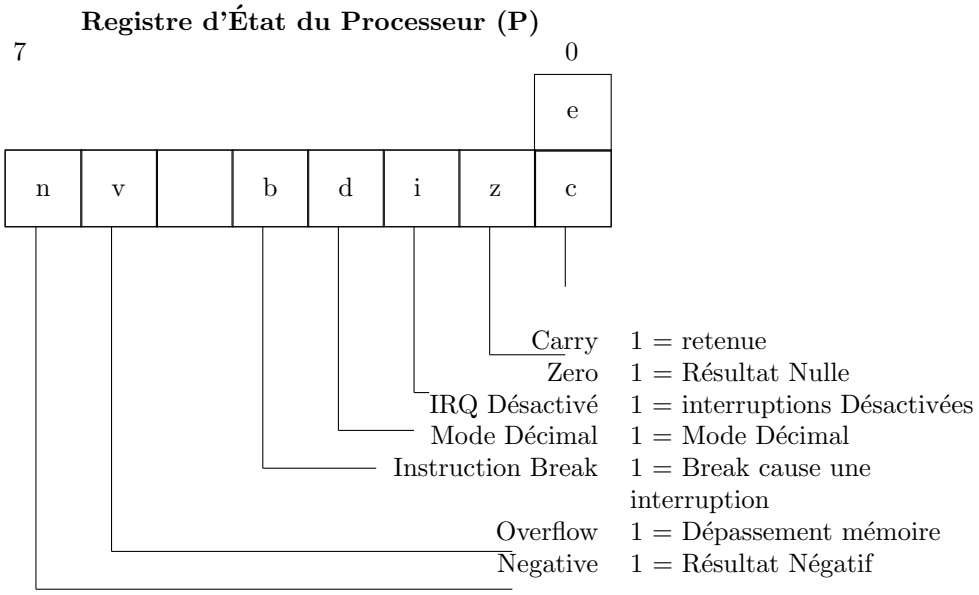


FIGURE 8.5 – Registres du 65816 mode Emulation

Chapitre 9

Différences entre mode Native et Emulation

Table 9.1 – Table des différence entre e=1& e=0

	65816 Native	65816 Emulation
signal d'annulation	oui	oui
accumulateur	16 or 8/8bits	8/8bits
mode d'addressage	25	25
espace addressable	16M	16M
registre de banque	oui	oui
déplacement en block	oui	of little use
indicateur "break"	non	oui
indicateur "mode décimale"	N,V,Z valid	N,V,Z valid
page zéro indexé	crosses page	wraps
indicateur après interruption	D = 0	D non modified
indicateur après reset	D = 0	D non modified
registre d'index	8 ou 16 bits	8 bits
instructions	256	256
interruptions	FFE4,FFEF	FFF4,FFFF
mnemonics	92	92
page spéciale	page 0(DP)	page 0 (DP)
pile	banque 0	page 1
opcode non utilisé	aucun	aucun

Chapitre 10

Les différents modes d'adressage

Le terme "Mode d'adressage" réfères à la méthode par laquelle le processeur va déterminé où il est pour avoir la donnée nécessaire, afin d'exécuter une opération donnée.

Chaque instruction possède un ou plusieurs types d'adressages. En effet quand vous vous servez d'une instructions, vous pouvez vous en servir d'une ou plusieurs manières différentes.

On a 23 modes d'adressage différents... Oui cela fait beaucoup, et l'on va les voir un par un.

Nous allons maintenant aborder la théorie générale de l'adressage, puis nous examinerons les différentes techniques permettant l'accès aux données. Nous verrons enfin les grandes possibilités d'adressage dont les modes les plus importants sont les adressages indexé, direct et indirect.

Le 65816 est particulièrement performant dans la mesure où ses registres spéciaux et les modes d'adressage indexés permettent d'écrire des programmes et des sous-programmes capables de manipuler des structures de données complexes ; ses modes d'adressage autorisent également l'écriture de code indépendant de la position (important dans le cas des applications qui utilisent la mémoire en lecture seule).

Alors que les méthodes complexes d'accès aux données ne sont pas obligatoires pour l'apprentissage de la programmation, il est cependant primordial de bien comprendre les modes d'adressage si l'on veut exploiter toute la puissance du processeur ; ces techniques, une fois maîtrisées, l'écriture de programmes manipulant efficacement les données paraîtra plus simple.

L'adressage, dans une instruction, spécifie l'emplacement de l'opérande avec lequel l'instruction doit être exécutée.

Dans un premier temps, nous allons voir les six principaux modes d'adressages.

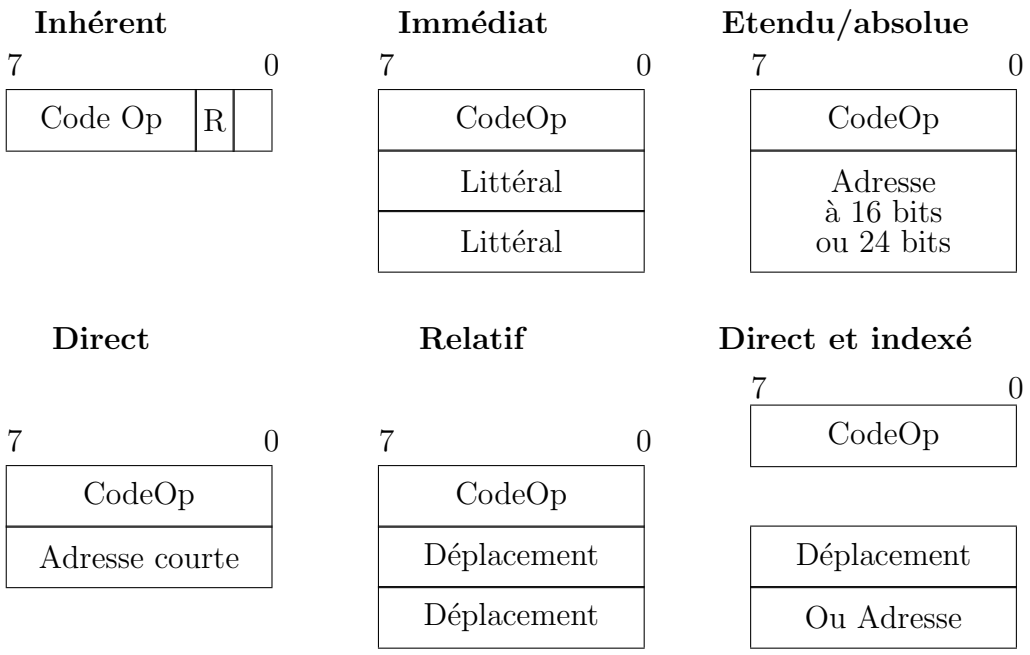


FIGURE 10.1 – Mode d'adressage de base

10.1 Adressage implicite (inhérent) ou adressage de registre

Les instructions qui opèrent uniquement sur des registres, utilisent normalement l'adressage implicite (voir figure 10.1).

Une instruction implicite ne contient pas l'adresse de l'opérande sur lequel elle agit ; c'est son code opération (opcode) qui spécifie un ou plusieurs registres. Puisque les registres internes sont peu nombreux (généralement huit), le code opération ne nécessite qu'un petit nombre de bits pour désigner un registre particulier. Voici un exemple d'instruction d'adressage implicite :

DEC A

Cette instruction signifie : "Décrémenter de 1 le contenu du registre A".

CE mode d'adressage est utilisé, sur le 65816, principalement par les instructions à un seul octet opérant sur des registres internes. La plupart de ces instructions nécessitent seulement deux cycles pour leur exécution. Le Tableau 10.1 montre les instructions utilisant l'adressage implicite. ce¡taines instructions, telles que XBA, requièrent plus de deux cycles d'horloge pour leur exécution. L'adressage implicite se nomme également adressage de registre.

ASL	CLC	DEC	INX	ROL	SEI	TSC	TXY	XBA
CLC	CLD	DEX	INY	ROR	STP	TSX	TYA	XCE
CLD	CLI	DEY	LSR	SEC	TAX	TXA	TYX	
CLI	CLV	INC	NOP	SED	TCD	TXS	WAI	

TABLE 10.1 – Instructions utilisant l'adressage inhérent

voir l'annexe 24.3 "Classements par Mode d'adressage" à la page 520

10.2 Adressage Immédiat

Dans ce mode, un nombre (appelé aussi :"littéraux") de 8 ou 16 bits suit le code opération de 8 bits (voir figure 10.1). Puisque le microprocesseur est équipé de registres 16 bits, il peut être nécessaire de charger des nombres

de 8 ou 16 bits. Voici un exemple de ce type d'instruction :

ADC #$5

Le second mot de cette instruction contient le nombre 5, qui est additionné à l'accumulateur A.

Donc le 65816 permet deux types d'adressage immédiat avec es littéraux de 8 ou 16 bits, car il possède à la fois des registres à simple longueur (8 bits) et à double longueur (16 bits) ; les instructions auront alors une longueur de deux ou trois octets.

Voici un exemples d'instructions utilisant le mode d'adressage immédiat :
LDA #N (un octet)
LDX #NN (un octet)
ADC #NN (un octet)

Le nombre d'octets dépend du mode en cours, 8 bits ou 16 bits pour l'accumulateur ou les registres d'index. Les instructions suivantes utilisent l'adressage immédiat :

- Avec des opérations de 8 ou 16 bits : ADC, AND,BIT, CMP, CPX, CPY, DEC, EOR, LDX, LDY, ORA, SBC.
- Avec des opérandes de 8 bits uniquement : REP, SEP

voir l'annexe 24.3 "Classements par Mode d'adressage" à la page 519

10.3 Adressage absolu (ou étendu)

Par définition, l'adressage absolu nécessite trois ou quatre octets ; le premier est le code opération et les deux octets suivants : l'adresse de 16 bits spécifiant l'emplacement mémoire (l'adresse absolue). Si le mode absolu long est employé, l'instruction nécessite alors quatre octets, le premier est le code opération et les trois suivants : l'adresse de 24 bits.

L'adressage absolu spécifie toujours une adresse particulière ne changeant pas pendant l'exécution du programme. Les programmes de sortie et d'entrée utilisent fréquemment ce type d'adressage. En voici un exemple :
LDA $10
JMP $1234

où les deux nombres hexadécimaux représentent les adresses de 16 bits des données ou des instructions. Les instructions pouvant utiliser l'adressage absolu sont :

ADC	BIT	CPY	INC	LDY	ROL	STA	STZ
AND	CMP	DEC	LDA	LSR	ROR	STX	TRB
ASL	CPX	EOR	LDX	ORA	SBC	STY	TSB

L'adressage absolu présente un inconvénient : il demande une instruction à trois ou quatre octets. Pour améliorer le rendement du microprocesseur, on peut utiliser un autre mode d'adressage employant un seul mot : l'adressage direct.

voir l'annexe 24.3 "Classements par Mode d'adressage" à la page 516

10.4 Adressage Direct

Dans ce mode, le code opération est suivi d'une adresse de 8 bits (voir figure 10.1). Ce procédé présente un avantage : il ne nécessite que deux octets (au lieu de trois dans le cas de l'adressage absolu).

L'inconvénient est la limitation de l'adressages aux adresses comprises entre 0 et 255. Mais le 65816 ne possède pas cette limite.

Lorsque ces adresses (0 à 255) sont utilisées, ce type d'adressage est également appelé "adressage court" ou "adressage en page zéro" ou encore "Direct Page (D)".

Voici un exemple de l'adressage direct :

ADC <$10

Le symbole < indique l'adresage direct.

Lorsque l'adressage direct est utilisé, l'octet suivant le code opération est additionné au registre D pour former l'adresse de l'opérande. Toute page en mémoire peut être adressée à condition de changer le registre D de manière appropriée. Lorsque le registre D contient 0, le mode d'adressage direct 65816 opère de la même manière que le microprocesseur 6502. Si l'octet inférieur du registre 0 n'est pas 0, un cycle d'instructions supplémentaire est alors nécessaire.

Les instructions suivantes utilisent l'adressage direct :

ADC	BlT	CPY	INC	LDY	ROL	STA	STZ
AND	CMP	DEC	LDA	LSR	ROR	STX	TRB
ASL	CPX	EOR	LDX	ORA	SBC	STY	TSB

voir l'annexe 24.3 "Classements par Mode d'adressage" à la page 517

10.5 Adressage Relatif

L'adressage relatif est utilisé avec les instructions de branchement. Si la position des indicateurs ou flags) d'état indique que le test réalisé par l'instruction de branchement est satisfait, l'instruction charge alors une nouvelle adresse dans le compteur de programme (PC) d'adresse. L'octet suivant, le code opération, appelé déplacement, est additionné au contenu du PC pour former le nouveau contenu ; cette nouvelle valeur constitue l'adresse de branchement. La figure 10.1 montre la structure du mode d'adressage relatif.

Puisque le déplacement est un nombre positif ou négatif, une instruction de branchement relatif permet un branchement en avant de 127 octets ou en arrière de 128 octets (habituellement +129 ou - 126, puisque le PC aura été incrémenté de 2). Les instructions de branchements sont employées dans les boucles de programmes puisque la plupart de ces boucles sont courtes, le branchement est le plus employé. Le branchement relatif permet souvent une amélioration des performances des programmes courts.

Si un déplacement plus important est nécessaire, on utilise alors l'instruction de branchement long (avec un déplacement de 16 bits). Cette instruction comporte trois octets (voir Figure 10.1) ; elle permet de se brancher à toute adresse mémoire puisque le déplacement s'étend de -32768 à +32767. Bien sûr, le temps d'exécution de ces instructions est plus long, aussi ne sont-elles employées que si le branchement court ne convient pas. L'adressage relatif employé, les instructions de branchement permet une amélioration des performances au niveau de la rapidité. Un programme employant l'adressage relatif peut être aisément se déplacer sur différentes zones mémoire. En outre, si l'adressage absolu n'est pas utilisé, on peut également transférer le programme sur d'autres zones mémoire. L,instruction de saut (JMP) utilise l'adressage absolu. En général, on emploie de préférence l'adressage relatif plutôt que l'adressage absolu.

Par définition, l'adressage relatif nécessite deux octets ; le premier est le code opération de branchement relatif, le second indique le déplacement et son signe. Un branchement long requiert un octet supplémentaire pour le déplacement, ce qui fait un total de trois octets. Les instructions utilisées pour l'adressage relatif sont : BCC, BCS, BEQ, BMI, BNE, BPL, BRA, BRL, BVC et BVS.

Lorsque le temps d'exécution est important, ces instructions sont à considérer avec attention. Qu'un test soit satisfait ou non (présence ou non d'un branchement), toute instruction de branchement court nécessite deux ou trois cycles. Si un branchement, s'effectue et que la limite d'une page est dépassée, l'instruction s'exéctue en quatre cycles.

Si vous n'êtes pas sûr que le branchement s'effectue, vous devez vous souvenir que, parfois, l'instruction nécessite deux cycles (si la condition n'est pas remplie) et parfois trois ou quatres cycles (si la condition est remplie). Une valeur moyenne est souvent utilisée pour évaluer la durée d'un branchement.

Ce problème de temps d'exécution n'apparaît pas avec les instructions (BRA) de branchement ni avec les instructions de branchement long (BRL) qui ne testent aucune condition.

(Note : pour différencier l'instruction de saut absolu de l'instruction de branchement relatif, l'instruction de saut est référencée JMP)

ll existe dans le 65816, un autre type d'adressage relatif utilisant le pointeur de pile (S). Le mode d'adressage de pile relative additionne l'octet qui suit le code opération au pointeur de pile pour former l'adresse de l'opérande. La valeur non signée de l'octet de déplacement est utilisée, donc seules peuvent êtres adressées les valeurs 0 à 255 au-dessus de la valeur de S. L'adresse doit être dans la banque 0. Voici un exemple d'adressage de pile relative ;

ADC $50,S

Les instructions qui peuvent utiliser l'adressage de pile relative sont : ADC, AND, CMP, EOR, LDA, ORA, SBC et STA.

voir l'annexe 24.3 "Classements par Mode d'adressage" à la page 520

10.6 Adressage indexé

Ce mode permet d'accéder successivement aux éléments d'un bloc ou d'une table, comme le montreront les exemples présentés plus loin. Avec l'adressage indexé, l'instruction spécifie un registre d'index et également une adresse de base. Le contenu du registre et l'adresse de base sont additionnés afin de former l'adresse finale ; de cette façon, on peut obtenir l'adresse de début d'une table en mémoire. Le registre d'index est alors employé pour accéder à tous les éléments successifs de la table, à condition de l,incrémenter ou de le décrémenter.

Dans tout adressage indexé du 65816, un des registres d'index (X ou Y) est utilisé dans le calcul de l'adresse effective des données employées par l'instruction. Il existe deux sortes d'adressage indexé : le direct indexé et l'abolu indexé.

Direct indexé

Dans ce mode, l'octet suivant le code opération est utilisé comme un déplacement et est additionné au registre direct ; le registre d'index spécifié est additionné à cette somme pour former l'adresse de l'opérande qui doit être dans la banque 0. Voici quelques exemples d'adressage direct indexé sont :

ADC $0,X
LDX $50,Y

Les instructions utilisant l'adressage direct indexé sont :

- Avec X : ADC, AND, ASL, BIT, CMP, DEC, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA, STY, STZ.
- Avec Y : LDX, STX.

voir l'annexe 24.3 "Classements par Mode d'adressage" à la page 518

Absolu indexé

Ce mode additionne les deux octets suivant le code opération au registre d'index spécifié pour former les 16 bits inférieur de l'adresse. Le registre de banque de données contient les 8 bits supérieurs de l'adresse. L'octet de poids faible de l'adresse est l'octet qui suit immédiatement le code opération. Les exemples d'adressage absolu indexé

ADC $100,Y
AND $1234,Y

Les instructions utilisant le mode absolu indexé sont :

- Avec X : ADC, AND, ASL, BIT, CMP, DEC, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA, STY, STZ.
- Avec Y : ADC, AND, CMP, EOR, LDA, LDX, ORA, SBC, STA.

Le mode absolu long indexé peut être uniquement utilisé avec le registre d'index X. Dans ce mode, l'adresse de 24 bits est stockée après le code opération et est additionnée au registre d'index X pour former l'adresse effective de l'opérande. Voici un exemple d'adressage absolu long :

CMP $FF1234,X

Les instructions utilisant le mode absolu long indexé sont : ADC, AND, CMP, EOR, LDA, ORA, SBC et STA

voir l'annexe 24.3 "Classements par Mode d'adressage" à la page 516

10.7 Préindexatíon et postindexatìon

ll existe deux modes d'indexation : la préindexation et la postindexation. La préindexation est le mode le plus souvent employé ; son adresse finale est la somme d'un déplacement ou d'une adresse, plus le contenu du registre d'index. La Figure 10.2 illustre cette solution (en supposant un champ de déplacement de B bits et un registre d,index de l6 bits).

La postindexation considère le contenu du champ de déplacement comme l'adresse du déplacement effectif et non pas comme le déplacement proprement dit. Dans la postindexation, l'adresse finale est la somme du contenu du registre d'index et du contenu du mot désigné par le champ déplacement (voir figure 10.2). ce phénomène utilise une combinaison d'adressage indirect et de préindexation. Examinons maintenant l'adressage indirect.

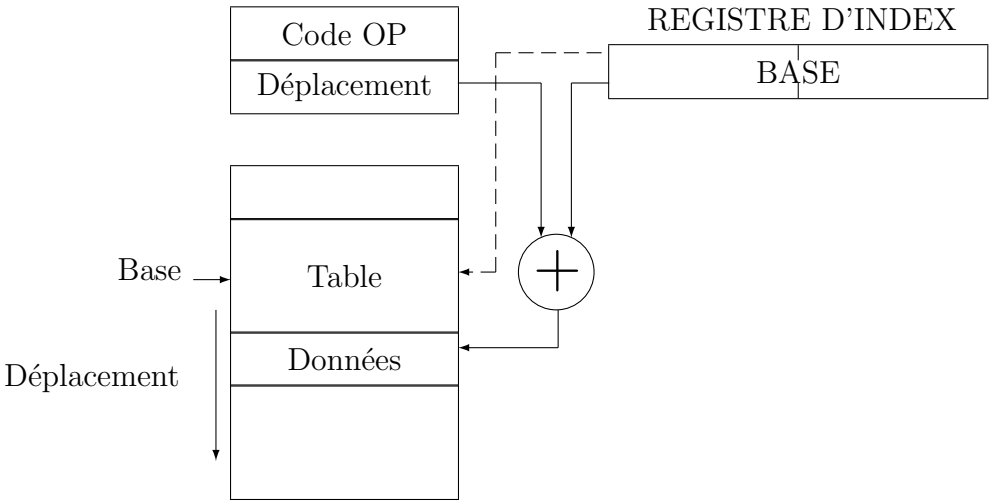


FIGURE 10.2 – Adressage - Pré-Indexation

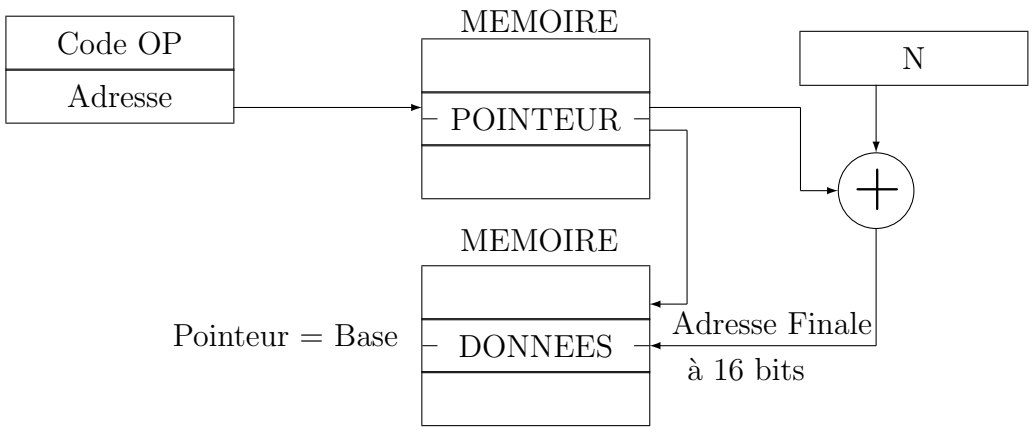


FIGURE 10.3 – Adressage indéxé indirect Post-Indexation

10.8 Adressage indirect

ll est parfois nécessaire que deux sous-programmes échangent une grande quantité de données stockées en mémoire ; d'une manière plus générale, plusieurs programmes ou sous-programmes peuvent avoir besoin d'accéder à un bloc de données commune. Afin de préservé le caractère général du programme, il est souhaitable de ne pas laisser ce bloc de donnée à un emplacement mémoire fixe.

Il se peut notamment que l'étendue de ce bloc soit augmenté ou diminuée d'une façon dynamique, et qu'il soit amené à résider dans différentes zones mémoire. De ce fait, il est difficile, voir même impossible, d'accéder à ce bloc en utilisant l'adressage absolue, sans ré-écrire chaque fois le programme.

La solution à ce problème est alors de stocker l'adresse de début de bloc dans un emplacement mémoire fixe. L'adressage indirect, par conséquent, utilise normalement un code opération (de 8 bits dans le cas du 65816) suivi d'une adresse de 16 bits ; cette adresse sert à retrouver un mot de 16 bits de la mémoire ; ce mot sert d'adresse à l'opérande.

La Figure 10.4 illustre la structure d'une instruction utilisant l'adressage indirect et dans laquelle les deux octets situés à l'adresse indiquée A1 contiennent A2 qui est alors interprétée comme l'adresse effective de la donnée à laquelle on doit accéder. L'adressage indirect est particulièrement utile lorsque les pointeurs sont employés. Différentes zones du programme peuvent alors se référer à ces indicateurs pour accéder à un mot ou à un bloc de données. L'adressage indexé indirect, autre forme d'adressage indirect, utilise, pour contenir l'adresse de la donnée désirée, un registre d'index plutôt qu'un emplacement mémoire.

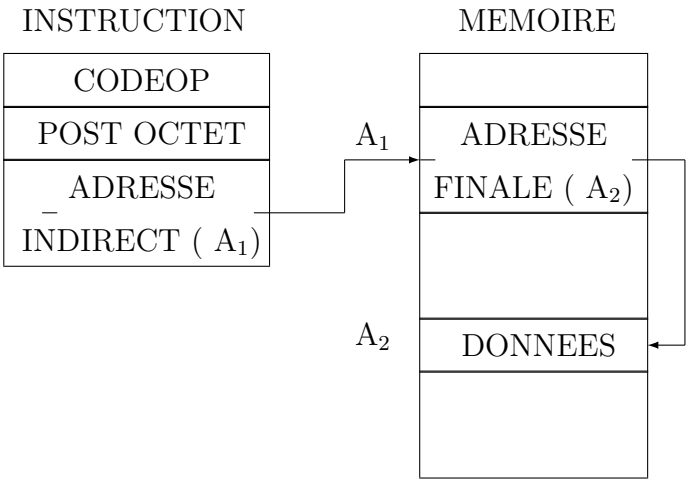


FIGURE 10.4 – Adressage indirect.

Les modes d'adressage indirect propres au 65816 utilisent les octets suivant le ode opération comme indicateur pour l'adresse du code opération. Il existe deux types d'adressage indirect : absolu et direct :

Absolu Indirect

Ce mode emploie les deux octets suivant le code opération pour former l'adresse pointant vers l'opérande utilisé par l'instruction. Seules, les instructions JMP et JML utilisent ce mode ; l'opérande prélevé est toujours l'adresse finale de saut. En voici un exemple :

JMP ($1234)

Le pointeur signal toujours la banque 0, car l'adresse a deux octets de long. Lorsque l'instruction JML est utilisée, le registre de la banque de programme est chargé à l'adresse provenant de l'indicateur, plus 2.

voir l'annexe 24.3 "Classements par Mode d'adressage" à la page 516

Direct indirect

Le mode direct indirect additionne l'octet suivant le code opération au registre direct pour former un pointeur vers les 16 bits de poids faible de l'adresse de l'opérande. Le registre de banque de données contient les huit bits de poids fort. L'adresse 16 bits de poids faible indiquée par le registre direct doit être dans la banque 0. Le mode direct indirect est signalé par des parenthèses entourant l'octet de déplacement ; en voie un exemple :

$123456 ADC ($20)
$000120 $FEEE
$AAFEEE $02

Dans cet exemple, considérons que les valeurs de D et de DBR dont respectivement : $0100 et $AA. La valeur de $20 sera additionnée à D pour extraire l'adresse $FEEE, qui est enchainée à $AA pour former l'adresse de l'opérande, $02.

Le mode d'adressage direct indirect long utilise le pointeur formé par la somme de l'octet de déplacement et le registre direct, pour extraire les trois octets de l'adresse de l'opérande. Voici un exemple de mode direct indirect long :

$123456 ADC [$20]
$000120 $AAFEEE
$AAFEEE $02

Les crochets [ ] indiquent l'adressage long.

Les instructions suivantes peuvent êtres utilisées avec les deux modes d'adressage : ADC, AND, CMP, EOR, LDA, ORA, SBC et STA.

voir l'annexe 24.3 "Classements par Mode d'adressage" à la page 519

10.9 Adressage Long

L'adressage long permet d'accéder à plus de 64K de mémoire, même si le PC et les registres d'index ont une longueur de 16 bits. Des registres d'adresse additionnels sont chaînés aux registres d'index ou au PC afin de former une adresse de 24 ou 32 bits (voir figure 10.5 ; Ils sont prévus pour l'adressage long. Un adressage absolu long requiert des octets supplémentaires dans une instruction.

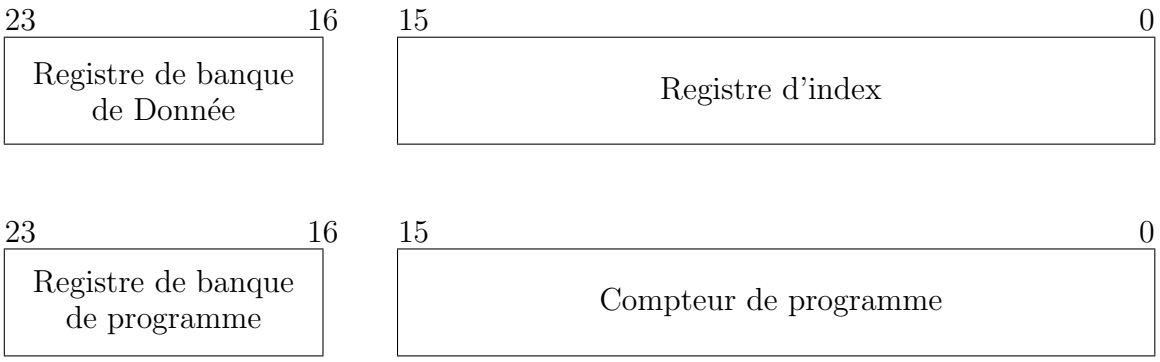


FIGURE 10.5 – Registres pour adresses longues.

10.10 Combinaison des modes

Les modes d'adressages peuvent être combinés entre eux. En particulier dans un plan d'adressage, on peut utiliser différents niveaux d'indirection. Par exemple, dans la figure 10.4, l'adresse A2 peut être interprétée comme une autre adresse indirecte, et ainsi de suite. On peut également combiner un adressage indexé avec un accès indirect ; cela permet l'accès efficace au énième mot d'un bloc de données, pourvu que l'on connaisse l'emplacement du pointeur vers l'adresse de début (voir figure 10.2).

Le 65816 possède cinq combinaisons de mode direct, indexé et indirect. Ces modes permettent une flexibilité dans le transfert des paramètres et le stockage des données d'une partie de programme à l'autre.

Direct Indirect Indexé

Le mode direct indirect indexé est également appelé indirect, Y. Le deuxième octet de l'instruction est additionné au registre direct. Le contenu de 16 bits de l'adresse indiquée par le registre direct est combiné au registre de banque de données pour former l'adresse de base. (L'adressage indiquée par le registre direct doit être dans la banque 0.) Le registre d'index Y est additionnée à la base pour former l'adresse de l'opérande. Dans l'exemple suivant :

$123456 ADC ($20),Y
$000120 $FF00
$AAFFZZ $02

on considère que D, DBR et Y valent respectivement : $0100, $AA, $00EE. La valeur 2 est additionnée à l'accumulateur.

Le mode direct indirect long indexé est similaire au mode direct indirect indexé. La seule différence étant que la somme du registre direct et de l'octet de déplacement indique une adresse de 24 bits, qui est additionné à Y pour fermer l'adresse dans la banque 0. Le registre de banque de données n'est pas utilisé, comme le montre l'exemple suivant :

$123456 ADC [$20],Y
$000120 $BBFF00
$BBFFEE $02

où D = $0100, DBR = $AA et Y = $00EE. La valeur 2 est ajoutée à l'accumulateur, les crochets []indique l'adressage long.

Ces instructions peuvent êtres utilisées avec les deux modes en adressage indirect indexé : ADC, AND, CMP, EOR, LDA, ORA, SBC et STA.

voir l'annexe 24.3 "Classements par Mode d'adressage" à la page 518

Direct Indexé Indirect

Le mode direct indexé indirect est souvent appelé indirect X. Le deuxième octet de l'instruction est additionné à la somme du registre direct et du registre d'index X. Cette somme indique une adresse de 16 bits dans la banque 0. L'adresse de 16 bits est combinée au registre de banque de données pour former l'adresse de l'opérande. L'exemple suivant additionne la valeur $02 à l'accumulateur :

$123456 ADC [$20],Y
$005120 $FF00
$AAFF00 $02

où D = $0100, DBR = $AA, et X =$5000. Les instructions suivantes peuvent êtres utilisées avec ce mode : ADC, AND, CMP, EOR, LDA, ORA, SBC et STA.

voir l'annexe 24.3 "Classements par Mode d'adressage" à la page 518

Absolue Indexé Indirect

Le mode absolu indexé indirect additionne le deuxième et troisième octet de l'instruction au registre d'index X pour former un indicateur de 16 bits dans la banque 0. L'indicateur est chargé dans le PC ; le registre de banque de programme n'est pas affecté. Ce mode d'adressage est employé avec les instructions JMP et JSL, il autorise un saut unique ou une instruction de saut à un sous-programme pour un branchement à différentes adresses, selon la valeur de X.

voir l'annexe 24.3 "Classements par Mode d'adressage" à la page 517

Pile relative Indirect Indexé

Le mode pile relative indirecte indexée utilise le deuxième octet de l'instruction pour l'additionner au pointeur de pile ; il forme ainsi in pointeur d'une adresse de base de 16 bits de poids faible dans la banque 0. Le registre de banque de données contient les huit bits de poids fort de l'adresse de base. L'adresse effective de l'opérande est la somme de l'adresse de base de 24 bits et du registre d'index Y. En voici un exemple :

$123456 ADC ($20,S),Y
$000120 $FF00
$AAFF22 $02

où S=$0100, DBR = $AA, et Y = $22. La valeur 2 est additionnée à l'accumulateur. Ce mode d'adressage est utilisé avec les instructions suivantes : ADC, AND, CMP, EOR, LDA, ORA, SBC et STA.

voir l'annexe 24.3 "Classements par Mode d'adressage" à la page 521

Notation du mode d'adresse

Lorsque l'on élabore un programme en langage assembleur, une notation s'avère nécessaire afin d'indiquer le mode d'adressage devant être employé avec une instruction. Vous trouverez, dans le tableau 10.2, la notation recommandée par le constructeur du 65816.

MODE d'ADRESSAGE	FORMAT D'OPÉRANDE
Implicite	Pas nécessaire
Immédiat	#N ou #NN
Absolu	NN
Absolu long	NNN
Direct	N
Direct Indirect Indexé	(N),Y
Direct Indirect Long Indexé	(N),Y
Direct Indexé Indirect	(N,X)
Direct Indexé avec X	N,X
Direct Indexé avec Y	N,Y
Absolu Indexé avec X	NN,X
Absolu Indexé avec Y	NN,Y
Absolu long indexé avec X	NNN,X
Absolu indirect	(NN)
Direct Indirect	(N)
Direct Indirect Long	(N)
Direct Indexé Indirect	(NN,X)
Pile Relative	N,S
Pile Relative Indexé Indirect	(N,S),Y
N valeur de 8 bits NN valeur de 16 bits NNN valeur de 24 bits	

TABLE 10.2 – Notation du mode d'adressage

Chapitre 11

Les Instructions

11.1 Classes d'Instructions

Dans ce chapitre, nous analyserons les différentes classes d'instructions disponibles dans le 65816.

Il n'existe pas de classification type des instructions. J'ai différencié ici cinq catégories d'instructions :

1 - Les Transfert de données
2 - Les Traitement de données
3 - Les Test et branchements
4 - Les Entrées / Sorties
5 - Les Contrôles

Transfert de données

Les instructions de transfert de données transmettent des données entre les périphériques d'entrées sorties. certains registres offrent même des instructions de transferts spéciales pouvant êtres utilisées pour organiser les données (par exemple, les opérations d'empilage et dépilage servent à gérer les piles de façons efficace)

Traitement de données

Ces instructions modifient les données à l'intérieur de l'ordinateur, elles peuvent être classées en 4 catégorie :

1 - Opérations arithmétique, par ex : + ou -
2 - Manipulation de bits (par ex mise a 1 ou initialisation)
3 - Opérations logiques (par ex ET OU OU exclusif)
4 - Opérations de déplacement et de décalage (permutation circulaire)

Test et branchements

Les instructions de test vérifient les valeur 0 ou 1 des bits du registre d'état ainsi que les combinaisons de ces valeurs. Il est donc souhaitable d'avoir un nombre maximal d'indicateur dans ce registre.

Il est intéressant de posséder des instructions pouvant tester :

1 - combinaison des bits.
2 - la positions d'un bit donnée dans un mot.
3 - La valeur d'un registre comparée à la valeur d'une adresse mémoire
- (supérieur à, inférieur à, égal à).

Généralement les instructions des micro-processeurs sont limités aux tests de certains bits du registre d'état.

1 - Le branchement limité à un champ de déplacement de 8 bits.
2 - Le branchement long, spécifiant une adresse complète de 16 bits.
3 - Le branchement à un sous programme, utilisé par les appels de sous-programme.

Il peut être pratique d'avoir des branchements à deux ou même trois voies dépendant, par exemple, du résultat d'une comparaison entre deux opérandes : = à, > à < à. Il est également intéressant d'avoir des opérations de saut permettant de passer plusieurs instructions. Notion qu'un saut est équivalent à un branchement.

Entrées / Sorties

Les instructions d'entrées /sorties servent à gérer des unités d'entrées/sorties. En pratique, la majorité des microprocesseur utilisent des Entrée/Sortie projetées en mémoire, ce qui signifie que les circuits d'Entrée/Sortie sont connectés au bus d'adresse comme les boitiers mémoire et sont adressés comme eux (ils apparaissent pour le programmeur comme des emplacements mémoire).

Ces opérations mémoire (adressé à un dispositif d'Entrée/Sortie) nécessitent normalement 3 octets, ce qui les rend assez lentes. Pour une gestion efficace des circuits d'Entrée/Sortie, il est généralement souhaitable d'avoir un adressage court. Il est possible d'utiliser un adressage en page directe, ne nécessitant que 2 octets, si les adresses des dispositifs d'Entrée/Sortie sont toutes dans la même page de la mémoire.

Contrôles

Ces instructions apportent des signaux de synchronisation ; elles peuvent suspendre, interrompre un programme ou simuler des interruptions .

11.2 Le jeu d'instruction

Ce chapitre est dévoué aux 94 opérations que comporte le 65816.

N'oubliez pas que le processeur utilise des nombres signés, je n'arrêterais pas de vous le rappeler car il faut faire attention.

Le Registre d'État (P) est là pour interpréter le résultat de notre instruction. C'est l'une de ses fonctions principales.

Ci-dessous vous trouverez le petit rappel du registre (P).

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Emulation	n	v	-	b	d	i	z	c
Native	n	v	m	x	d	i	z	c

n	Negative result
v	oVerflow
m	8-bit memory/accumulator
x	8-bit index registers
d	Decimal mode
i	IRQ interrupt disable
z	Zero result
c	Carry

En haut à droite vous verrez une inscription du style "6502/65816". Ceci signifie si l'instruction fonctionne en mode emulé(6502) et en mode native (65816), quand le 6502 est en bleu, ceci signifie qu'il ne supporte pas certains modes d'adressage comme le 65816[1]. Si le 6502 est absent, l'instruction ne fonctionne pas pour le mode émulé. (sauf XCE).

[1] reporter vous sur le manuel de programmation du 65816 pour avoir tous les détails.

ADC - Add with Carry

On additionne la mémoire à la valeur de l'Accumulateur avec repport (la Carry) !
Un exemple :
A=$20

ADC #$42

A => $62 si Carry = 0 (20+42+0=62)
A => $63 si Carry = 1 (20+42+1=63)

Indicateur affectés n v - - - - z c

Mieux vaut un bon exemple (en bits).[1]

Si le résultat **excède 127**

n	v	z	c		Nombres en bits	Valeur décimale
0		1		Accumulateur	0111 1111	127
			1	ADC #$42	0100 0011	127+66+1
1	1	0	0	Résultat	1100 0010	194

Si le résultat est **Positif**

n	v	z	c		Nombres en bits	Valeur décimale
0				Accumulateur	0000 0010	2
			1	ADC #$42	0100 0011	2+66+1
0	0	0	1	Résultat	0100 0101	69

Si le résultat **est à Zéro**

n	v	z	c		Nombres en bits	Valeur décimale
0				Accumulateur	1111 0000	-16
			0	ADC #$10	0001 0000	-16+16+0
0	0	1	1	Résultat	0000 0000	0

Si le résultat **Négatif**

n	v	z	c		Nombres en bits	Valeur décimale
0				Accumulateur	1100 1101	-33
			1	ADC #$10	0001 0001	-33+16+1
1	0	0	0	Résultat	1101 1110	-16

Si le résultat est **inférieur à -128**

n	v	z	c		Nombres en bits	Valeur décimale
0				Accumulateur	1000 0000	-128
			1	ADC #$05	1111 1010	-128+(-5)+1
0	1	0	1	Résultat	0111 1010	-132

[1] je rappel que ce processeur utilise des nombre "signé" 8bits (-128 à +127) ou 16bits (-32768 à 32767)

TABLE 11.1 – Valeurs possible avec ADC

n	**v**	**z**	**c**	**Résultat**	
				8bits	**16bits**
0	0	0	1	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	1	0	0	Positif Overflow, de 128 à 255 inclus	de 32768 à 65535 inclus
1	0	0	1	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	0	1	Négatif Overflow, de -1 à -128 inclus	de -1 à -32768 inclus
0	0	1	1	Nulle = 0	Nulle = 0
0	0	1	0	Nulle = 0 (addition de 2 zéros)	Nulle = (addition de 2 zéros)
0	0	1	1	=-256	=-65536

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur*		*nb d'*	*nb de*
			6502	*65816*	*Octets*	*Cycles*
Immédiat	**ADC** *#const*	69	x	x	2	2[1]
Absolue	**ADC** *addr*	6D	x	x	3	4[1]
Absolue Long	**ADC** *long*	6F		x	4	5[1]
Direct Page(DP)	**ADC** *dp*	65	x	x	2	3[1,3]
DP indirect	**ADC** *(dp)*	72		x	2	5[1,3]
DP indirect Long	**ADC** *[dp]*	67		x	2	6[1,3]
Absolue indexé par X	**ADC** *addr,X*	7D	x	x	3	4[1,4]
Absolue Long indexé par X	**ADC** *long,X*	7F		x	4	5[1]
Absolue indexé par Y	**ADC** *addr,Y*	79	x	x	3	4[1,4]
DP indexé par X	**ADC** *dp,X*	75	x	x	2	4[1,3]
DP indirectement indexé par X	**ADC** *(dp,X)*	61	x	x	2	6[1,3]
DP indirectement indexé par Y	**ADC** *(dp),Y*	71	x	x	2	5[1,3,4]
DP long indirectement indexé par Y	**ADC** *[dp],Y*	77		x	2	6[1,3]
Stack relative (SR)	**ADC** *sr,S*	63		x	2	4[1]
SR indirectement indexé indexé par Y	**ADC** *(sr,S)*,Y	73		x	2	7[1]

ADC, une instruction de groupe primaire, a disponible tous les modes d'adressage du groupe primaires et les modèles binaires

* - Ajoutez 1 octet si m=0 (mémoire et accumulateur à 16bits)

1- Ajoutez 1 cycle si m=0 (mémoire et accumulateur à 16bits)

3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)

4- Ajoutez 1 cycle si l'ajout d'index dépasse la page courante (si $00:0000 à $01:0000 par ex)

AND - And accumulator with memory

6502/65816

On fait l'opération logique "ET" entre le contenu mémoire et accumulateur

Nb1	Nb2	Résultat
0	0	0
0	1	0
1	0	0
1	1	1

Tout est dans la table !
Pour l'exemple, vous charger une valeur dans l'Accumulateur

```
LDA #$10
```

Vous faite un AND de $21 avec la valeur chargé dans l'Accumulateur

```
AND #$21
```

Le résultat se retrouvera dans l'Accumulateur, A = $31

Indicateur affectés n - - - - - z -

Si le résultat est **Positif**

n	z		Nombres en bits	Valeur décimale
			0111 1111	127
		AND	0111 1111	127
0	0	Résultat	0111 1111	127

Si le résultat **est à Zéro**

n	z		Nombres en bits	Valeur décimale
			0000 0000	0
		AND	0111 1111	127
0	0	Résultat	0000 0000	0

Si le résultat **Négatif**

n	z		Nombres en bits	Valeur décimale
			1000 0000	-128
		AND	1111 1111	-1
0	0	Résultat	1000 0000	-128

TABLE 11.2 – Valeurs possible avec AND

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Immédiat	**AND** *#const*	29	x	x	2	2[1]
Absolue	**AND** *addr*	2D	x	x	3	4[1]
Absolue Long	**AND** *long*	2F		x	4	5[1]
Direct Page(DP)	**AND** *dp*	25	x	x	2	3[1,3]
DP indirect	**AND** *(dp)*	32		x	2	5[1,3]
DP indirect Long	**AND** *[dp]*	27		x	2	6[1,3]
Absolue indexé par X	**AND** *addr,X*	3D	x	x	3	4[1,4]
Absolue Long indexé par X	**AND** *long,X*	3F		x	4	5[1]
Absolue indexé par Y	**AND** *addr, Y*	39	x	x	3	4[1,4]
DP indexé par X	**AND** *dp,X*	35	x	x	2	4[1,3]
DP indirectement indexé par X	**AND** *(dp,X)*	21	x	x	2	6[1,3]
DP indirectement indexé par Y	**AND** *(dp), Y*	31	x	x	2	5[1,3,4]
DP long indirectement indexé par Y	**AND** *[dp], Y*	37		x	2	6[1,3]
Stack relative (SR)	**AND** *sr,S*	23		x	2	4[1]
SR indirectement indexé indexé par Y	**AND** *(sr,S)*,Y	33		x	2	7[1]

AND, une instruction de groupe primaire, a disponible tous les modes d'adressage du groupe primaires et les modèles binaires

* - Ajoutez 1 octet si m=0 (mémoire et accumulateur à 16bits)

1- Ajoutez 1 cycle si m=0 (mémoire et accumulateur à 16bits)

3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)

4- Ajoutez 1 cycle si l'ajout d'index dépasse la page courante (si $00:0000 à $01:0000 par ex)

ASL - Shift memory or Accumulator Left

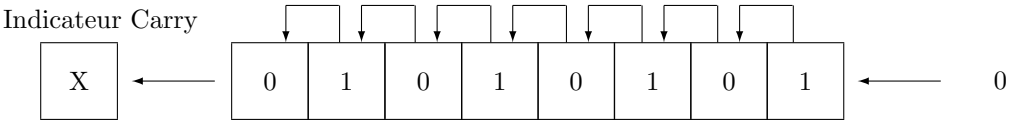
6502/65816

Décalage arithmétique vers la gauche. On peut traduire par une multiplication dans certains cas.

Quoi ça sert à faire une Multiplication ?

Oui...

Si dans l'Accumulateur on a 15, et que l'on souhaite avoir 30 :



Indicateur affectés n - - - - - z c

Bon le "Carry" ici ne sert qu'à recevoir le MSB.[1], il n'indique en rien le signe du résultat !

Tous les bits de l'opérande sont décalés d'une position sur la gauche. Le bit 7 est transféré dans la retenue (la carry), le bit 0 est remis à zéro. En mode 16 bits, le bit 15 est transféré dans la retenue (carry).

Mais nous pourrions l'interpréter comme un "overflow", on verra ça par la suite.

Pour un résultat **Positif**.[2]

n	z	c		Nombres en bits	Valeur décimale
		0		0000 1111	15
			ASL		15*2
0	0	0	Résultat	0001 1110	30

Pour un résultat **Nulle**

n	z	c		Nombres en bits	Valeur décimale
		0		1000 0000	-128
			ASL		-128*2
0	1	1	Résultat	0000 0000	0

Pour un résultat **Négatif**

n	z	c		Nombres en bits	Valeur décimale
		0		1111 1110	-2
			ASL		-2*2
0	1	1	Résultat	1111 1100	-4

TABLE 11.3 – Valeurs possible avec ASL

n	**z**	**c**	**Résultat**	
			8bits	**16bits**
0	0	0	Positif, de 1 à 126 inclus	de 1 à 32766 inclus
1	0	0	Positif Overflow, de 128 à 254 inclus	de 32768 à 65534 inclus
0	0	1	Négatif Overflow, de -129 à -255 inlcus	de -32769 à -65535 inclus
1	0	1	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	0	Nulle = 0	Nulle = 0
1	0	1	= -256	= -65536

[1] Most Signifiant Bit, le bit le plus signifiant, le bit 7 en 8bits et le bit 15 en 16bits

[2] quand le Msb est à 0, on se souvient aussi que l'on est en nombre signé

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* 6502	65816	*nb d' Octets*	*nb de Cycles*
Accumulateur	**ASL** A	0A	x	x	1	2
Absolue	**ASL** *addr*	OE	x	x	3	6^{2}
Direct Page(DP)	**ASL** *dp*	06	x	x	2	$5^{2,3}$
Absolue indexé par X	**ASL** *addr,X*	1E	x	x	3	$7^{2,4}$
DP indexé par X	**ASL** *dp,X*	16	x	x	2	$6^{2,3}$

2- Ajoutez 2 cycle si m=0 (mémoire et accumulateur à 16bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)
4- Ajoutez 1 cycle si l'ajout d'index dépasse la page courante (si $00:0000 à $01:0000 par ex)

BCC - Branch if Carry Clear

6502/65816

Branchement si la retenue (carry) est à zéro.

C'est simple comme bonjour !

```
1         CLC    ; retenue à zéro
2         LDA #$FE   ; -2(je le rappel, le bit 7 est là pour interpréter un signe)
3         ADC #$03   ; -2 + 3 = 1
4         BCC Label1
5         ADC #$10   ; Si carry = 1 on ajoute $10
6         (...)
7 Label1  ADC #$20   ; Si carry = 0 on ajoute $20
```

On regarde si l'indicateur "c" (carry) est à zéro.

Si il est à "Zero" on va au "Label1". Sinon on exécute l'instruction "ADC #$10".

L'exemple qui a été fait sert à savoir si le résultat est inférieur à zéro (<0), ou supérieur ou égal à zéro (>=0).

Si l'opération donne un résultat négatif ou égal à 0 (Carry = 0), on se branche sur "Label1". Si positif (Carry = 1), on passe à la suite (ADC #$20).

Le branchement permet d'accéder à toute instruction située entre + 129 et -126 octets par rapport au premier octet de l'instruction de branchement.

Indicateur affectés - - - - - - - -

Aucun.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Compteur de Programme relatif	**BCC** *nearlabel* ou **BTL**	90	x	x	2	$2^{5,6}$

5- Ajouter 1 cycle si la condition est vrai, si l'on branche

6- Ajouter 1 cycle si vous êtes en mode émulation (e=1)

BCS - Branch if Carry Set

6502/65816

On branche si la carry est à un. Ou ; Branchement si le résulat est positif
C'est simple comme bonjour !

```
1         CLC    ; Carry à zéro
2         LDA #$02   ; 2
3         ADC #$03   ; 2 + 3 = 5
4         BCS Label1
5         ADC #$10   ; Si carry = 1 on ajoute $10
6         (...)
7 Label1  ADC #$20   ; Si carry = 0 on ajoute $20
```

On regarde si le bit Carry est à un.
Si il est à "Un" on va au "Label1". Sinon on exécute l'instruction "ADC #$10".

L'exemple qui a été fait sert à savoir si le résultat est inférieur à zéro (<0), ou supérieur ou égal à zéro (>=0).

Si l'opération donne un résultat positif (Carry = 1), on se branche sur "Label1". Si négatif ou égal à(Carry = 0), on passe à la suite (ADC #$20).
Le branchement permet d'accéder à toute instruction située entre + 129 et -126 octets par rapport au premier octet de l'instruction de branchement.

Indicateur affectés - - - - - - - -

Aucun.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Compteur de Programme relatif	**BCS or BGE** *nearlabel*	B0	x	x	2	$2^{5,6}$

5- Ajouter 1 cycle si la condition est vrai, si l'on branche
6- Ajouter 1 cycle si vous êtes en mode émulation (e=1)

BEQ - Branch if Equal

6502/65816

On branche si l'indicateur Zero est à 1.[1]

En d'autres mots, on branche si le résultat est égal (z=1).

```
1         CLC    ; Carry à zéro
2         LDA #$FE   ; -2(je le rappel, le bit 7 est là pour interpréter un signe)
3         ADC #$02   ; -2 + 2 = 0
4         BEQ Label1
5         ADC #$10   ; Si carry = 1 on ajoute $10
6         (...)
7 Label1  ADC #$20   ; Si carry = 0 on ajoute $20
```

On regarde si le bit Zero est à un.

Si il est à un, on va au "Label1", sinon, on exécute d'instruction suivante.

L'exemple ci-dessus montre si le résultat est égal à zéro (z=1), ou pas ...

Si l'opération ADC donne un résultat nulle (z=1), on se branche sur le label "label1", si négatif ou positif (z=0), on passe à la suite (ADC #$10).

Le branchement permet d'accéder à toute instruction située entre + 129 et -126 octets par rapport au premier octet de l'instruction de branchement.

Indicateur affectés - - - - - - - -

Aucun.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Compteur de Programme relatif	**BEQ** *nearlabel*	F0	x	x	2	2[5,6]

5- Ajouter 1 cycle si la condition est vrai, si l'on branche

6- Ajouter 1 cycle si vous êtes en mode émulation (e=1)

[1] L'inverse est BNE

BIT - Test Memory Bits against Accumulator

6502/65816

Tester les bits.

La fonction ET opère sur l'accumulateur A et l'opérande en mémoire, le résultat est mis de côté. Seul le Registre d'État (P) est affecté, aucun des opérandes ne l'est. Dans le mode 8 bits, l'indicateur "n" est positionné comme le bit 7 t l'indicateur "v" est positionné comme le bit 6. En mode 16 bit, l'indicateur "n" est positionné comme le bit 15 et l'indicateur "v" est positionné comme le bit 14.

Indicateur affectés									
	-	-	-	-	-	-	z	-	adresse Immédiate seulement
	n	v	-	-	-	-	z	-	Tout autre que l'adresse immédiate

Tout autre que l'adressage immédiate

```
1 LDA #$C0 ;Mettre $C0 dans l'Accumulateur
2 STA $2210 ; Stocker l'Accumulateur dans l'adresse $2210
3 BIT $2210 ;
```

Là les indicateurs "n" et "v" seront mis à un.[1]

Si vous passer un $00 les indicateurs "n" et "v" seront à zéro, $40 l'indicateur "n" à zéro et "v" à un, $80 l'indicateur "v" à zéro et "n" à un.

n	v	z		Nombres en bits	Valeur HexaDécimale
			Valeur dans $2210	1100 0000	$C0
			BIT $2210		
1	1	0	Résultat dans le Registre	11xx xxxx	

L'indicateur "z" est à un quand le résultat est égal à zéro.

Par l'adressage immédiate

Les indicateurs "n" et "v" ne seront pas touchés. Seul l'indicateur "z" sera mis à un si le résultat de l'opération "ET"[2] est à zéro.

```
1 BIT #$C3 ; tester si l'Accumulateur "ET" $C3 = 0.
```

On peut interpréter ceci comme : "Si l'accumulateur à la valeur $C3 on peut aller à cette opération ,sinon on passe à la suite".[3]

[1] $C0 = 1100 000, bit 7 viens changer "n", et le bit 6 "v"

[2] "AND" en anglais

[3] Pour ceci, il faudra tester l'indicateur "z" après l'instruction "BIT"

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Immédiat	**BIT** *#const*	89		x	2	2[1]
Absolue	**BIT** *addr*	2C	x	x	3	4[1]
Direct Page(DP)	**BIT** *dp*	24	x	x	2	3[1,3]
Absolue indexé par X	**BIT** *addr,X*	3C		x	3	4[1,4]
DP indexé par X	**BIT** *dp,X*	34		x	2	6[1,3]

* - Ajoutez 1 octet si m=0 (mémoire et accumulateur à 16bits)
1- Ajoutez 1 cycle si m=0 (mémoire et accumulateur à 16bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)
4- Ajoutez 1 cycle si l'ajout d'index dépasse la page courante (si $00:0000 à $01:0000 par ex)

BMI - Branch if Minus

6502/65816

Facile !

On branche si l'indicateur "n" est à un ! En d'autres mots ; si le résulat de l'opération est négative.

Bon en gros, si vous êtes strictement inférieur à zéro (<0), l'indicateur "n" est à un et vous pouvez vous brancher à un label choisie, sinon vous exécutez la suite !

Exemple :

```
1          A => $15
2          On décrémente A
3          On teste si le N est activé :
4          Oui -> On se branche sur "Label1"
5          Non -> On passe à la suite du programme
6          (...)
7          BMI Label1
8          LDA #$20
9          (...)
10  Label1 ASL
```

Le branchement permet d'accéder à toute instruction située entre + 129 et -126 octets par rapport au premier octet de l'instruction de branchement.

Indicateur affectés - - - - - - - -

Aucun.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Compteur de Programme relatif	**BMI** *nearlabel*	30	x	x	2	$2^{5,6}$

5- Ajouter 1 cycle si la condition est vrai, si l'on branche
6- Ajouter 1 cycle si vous êtes en mode émulation (e=1)

BNE - Branch if Not Equal

6502/65816

Vous l'aurez devinez, c'est le contraire du BEQ !
On branche si l'indicateur Zero est à 0.
En d'autres mot, on branche si le résultat n'est pas égal (z=0).

```
1          CLC ; Carry à 0
2          LDA #$02 ; 2
3          ADC #$02 ;-2+2 = 4
4          BNE LABEL1
5          ADC #$10 ; Si zero =1 on ajoute $10
6          (...)
7  LABEL1  ADC #$20  ; Si zero=0 on ajoute $20
```

On regarde si l'indicateur Zero est à zéro.Si il est à zéro on va au label "LABEL1", si il est à un, on exécute l'instruction suivante.

L'exemple que j'ai fait sert à savoir si le résultat est différent de zéro (!=0), ou pas...

Si l'opération ADC donne un résultat Négatif ou positif (z=0) on se branche sur le Label "LABEL1", si nulle, z=1, on passe à la suite (ADC#$10).

Le branchement permet d'accéder à toute instruction située entre + 129 et -126 octets par rapport au premier octet de l'instruction de branchement. Indicateur affectés - - - - - - - -

Aucun.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Compteur de Programme relatif	**BNE** *nearlabel*	D0	x	x	2	$2^{5,6}$

5- Ajouter 1 cycle si la condition est vrai, si l'on branche
6- Ajouter 1 cycle si vous êtes en mode émulation (e=1)

BPL - Branch If Plus

6502/65816

Brancher si le résultat est positif ! contrairement à BMI.

Si Negative = 0 on branche, si =1, on exécute la suite.

Simple... non ?

Ici on test l'indicateur Negative, il sert à savoir si notre résultat est positif (>=0) ou strictement négatif (<0).

```
1          CLC ; Carry à 0
2          LDA #$02 ; 2
3          ADC #$02 ;2+2 = 4
4          BPL LABEL1
5          ADC #$10 ; Si zero =0 on ajoute $10
6          ( . . . )
7  LABEL1  ADC #$20 ; Si zero=1 on ajoute $20
```

Le branchement permet d'accéder à toute instruction située entre + 129 et -126 octets par rapport au premier octet de l'instruction de branchement.

Indicateur affectés - - - - - - - -

Aucun.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Compteur de Programme relatif	**BPL** *nearlabel*	10	x	x	2	$2^{5,6}$

5- Ajouter 1 cycle si la condition est vrai, si l'on branche
6- Ajouter 1 cycle si vous êtes en mode émulation (e=1)

BRA - BRanche Always

6502/65816

Vous l'aurez deviné tout seul, on branche toujours quelque soit le résultat !

Il y a aussi une autre instruction qui fait la même chose, mais celle-ci a une limite.

Et oui vous ne pouvez pas « brancher » n'importe où dans le programme, vous êtes limité à un déplacement de +129 à -126 adresse ! !

Votre registre PC [1] est sur 2 octets : $1234, le BRA lui indiquer "on va à cette adresse pour exécuter tel instruction". Mais le BRA ne peut lui donner l'ordre que sur l'octet le plus bas ($34).

Exemple : L'instruction est exécutée à l'adresse $1234, on ne peut brancher qu'entre $12B3 et $11B4.

Indicateur affectés - - - - - - - -

Aucun.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Compteur de Programme relatif	**BRA** *nearlabel*	80	x	x	2	3[6]

6- Ajouter 1 cycle si vous êtes en mode émulation (e=1)

[1] Program counter

BRK - Software Break

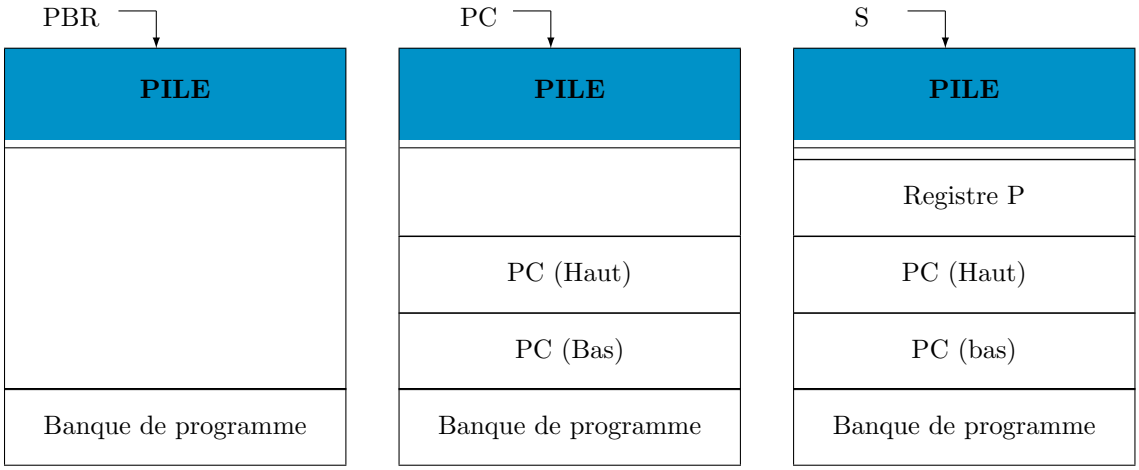
6502/65816

C'est une fonction simple, mais difficile à expliquer, je vais vous donner "Le principe", ensuite je vous donnerais des précisions.

Cette instruction **Force une interruption Logiciel.**
Dans le Registre d'État (P), l'indicateur B est à 1.

Alors la séquence est comme suit :

- BRK arrive (ta daaaam!!!)
- Le regitre de banque de programme (PBR) et le compteur de program (PC) possèdent l'adresse de la prochaine instruction à exécuter. PC = PC+2, alors le "Program Counter" est incrémenté 2 fois.
- La banque de programme est mis dans la pile, pour être sauvegarder, et pour savoir où l'on en est, au retour de la séquence d'interruption.



- Le Registre d'État (P) est ensuite sauvé dans la pile. Dans se registre n'oublier pas qu'il y a l'indicateur B (BRK cause Interrupt) qui est à 1 quand il est chargé dans la pile
- Emulation Mode E=1
- Ensuite PC pointe sur la routine d'interruption, PC=FFFE à FFFF
- Native Mode E=0
- Ensuite PC pointe sur la routine d'interruption, PC=00FFE6 à 00FFE7

Pour faire un peu plus simple :
BRK opère comme une interruption, le registre de bloc de programme est déplacé dans la pile, puis le compteur de programme et enfin le Registre P. Le contenu des adresses mémoire FFE6 et FFE7 est respectivement déposé en PCL et PCH. le registre de bloc de programme est remis à zéro. Important : A la différence d'une interruption, BREAK sauvegarde PC + 2 ; PC + 2 peut ne pas être l'instruction suivante et une correction s'avère alors nécessaire.

Indicateur affectés									
	-	-	-	b	d	i	-	-	Emulation Mode e=1
	-	-	-	-	d	i	z	-	Native mode e=0

"b"[1] est mis à un[2], ce qui signifie "Software Interrupt"
"d"[3] est à zéro, passage en mode binaire
"i"[4] est à un, c'est l'indicateur d'interruption, il dit si les interruptions sont désactivés (=1) ou pas (=0).

[1] Break ou encore brk
[2] e=0, Native mode
[3] decimale
[4] interrupt

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Stack/Interrupt	**BRK**	00	x	x	2^{+}	7^{7}

+ - BRK est sur 1 Octet, mais la valeur du compteur de programme poussé dans la stak(pile) est incrémenté de 2 suivant
l'octet signé facultatif
7- Ajouter 1 cycle si vous êtes en mode native (e=0)

BRL - Branch Always Long

/65816

On branchement long vers une instruction.

La seul limite est la distance, BRA peut brancher une instruction aux adresses se situant entre +129 et -126 par rapport au premier octet de notre instruction.

BRL permet d'accéder à toute adresse située dans un bloc (banque) mémoire de 64K [1].

Exemple : L'instruction est exécuter à l'adresse $8000, on ne peut brancher qu'entre $0000 et $FFFF.

Donc si vous suivez bien, Program Counter[2], étant sur 2 octets, est entièrement adressable par l'instruction BRL.

Voir l'instruction JMP si vous en voulez encore +.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Compteur de Programme relatif Long	**BRL** *label*	82		x	3	4

[1] 65 535 pour les non habitué

[2] PC

BVC - Branch if Overflow Clear

6502/65816

On Branche si il n'y a pas de débordement, si l'indicateur "Overflow"[1] du Registre d'État (P) est à 0.

C'est simple, si vous n'avez pas fait d'Overflow, par exemple, le résultat d'une opération soit supérieur à +127, en mode 8 bits, vous vous branchez au label, sinon vous exécutez l'instruction suivante !

```
1           CLC
2           LDA #$7F    ; 122
3           ADC #$02    ; 122+2 = 124
4           BVC LABEL1
5           ADC #$10    ; Si Overflow =1 on ajoute $10
6           (...)
7   LABEL1  ADC #$20    ; Si Overflow=0 on ajoute $20
```

Le branchement permet d'accéder à toute instruction située entre + 129 et -126 octets par rapport au premier octet de l'instruction de branchement.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur 6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Compteur de Programme relatif	**BVC** *nearlabel*	50	x	x	2	2[5,6]

5- Ajouter 1 cycle si la condition est vrai, si l'on branche
6- Ajouter 1 cycle si vous êtes en mode émulation (e=1)

[1] "v"

BVS - Branch if Overflow Set

6502/65816

On Branche si il y a un debordement, si l'indicateur "Overflow"[1] du Registre d'État (P) est à 1.

C'est simple, si vous avez fait un Overflow, par exemple, le résultat d'une opération soit supérieur à +127, vous vous brancher au label, sinon vous exécutez l'instruction suivante !

```
1         CLC
2         LDA #$7F   ; 127
3         ADC #$02   ; 127+2 = 129
4         BVS LABEL1
5         ADC #$10   ; Si Overflow =0 on ajoute $10
6 (...)
7 LABEL1 ADC #$20   ; Si Overflow=1 on ajoute $20
```

Le branchement permet d'accéder à toute instruction située entre + 129 et -126 octets par rapport au premier octet de l'instruction de branchement.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur 6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Compteur de Programme relatif	**BVS** *nearlabel*	70	x	x	2	2[5,6]

5- Ajouter 1 cycle si la condition est vrai, si l'on branche
6- Ajouter 1 cycle si vous êtes en mode émulation (e=1)

[1] "V"

CLC - Clear Carry Flag

6502/65816

Met l'indicateur Carry à 0.

Indicateur affectés - - - - - - - c

On efface la carry C.
Par exemple : On peut l'utiliser dans le cas où l'on veut passer en mode Native.

```
1 CLC
2 XCE ;(reporter vous à cette instruction, l'exemple doit y être aussi)
```

ou même avant l'instruction ADC.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**CLC**	18	x	x	1	2

CLD - Clear Decimal Mode Flag

6502/65816

On efface l'indicateur Decimal.

Indicateur affectés - - - - d - - -

L'indicateur Decimal "d" détermine si l'on est en mode binaire (d=0), ou en mode décimal (d=1).
Le bit décimal est remis à zéro, désignant le mode linaire pour les instructions ADC et SBC.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**CLD**	D8	x	x	1	2

CLI - Clear Interrupt Disable Flag

6502/65816

Remize à zéro du bit de désactivation des interruptions.

Indicateur affectés - - - - - i - -

En effaçant l'indicateur d'interruption "i", on désactive les interruptions (logiciel et matérielle).
Ce qui permet de ne pas couper une séquence de programme très importante, exemple, l'initialisation.
Le bit de désactivation des interruptions est remis à zéro ce qui remet en service les interruption. Un sous programme de gestion d'interruption doit toujours remettre le bit 1 à zéro ; sinon, d'autres interruptions peuvent être perdues.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**CLI**	58	x	x	1	2

CLV - Clear Overflow Flag

6502/65816

Effacer l'indicateur Overflow.

Indicateur affectés - v - - - - - -

Overflow = dépassement.
On efface l'overflow après une opération, par exemple, pour continuer tranquillement notre programme.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**CLV**	B8	x	x	1	2

CMP - Compare Accumulator with Memory

6502/65816

Comparer la mémoire avec l'accumulateur.

Le résultat de la comparaison n'est pas sauvegarder, c'est utilisé pour connaitre la différence entre ces deux valeurs.

Accumulator A = $20
Adresse $2100 = $22

```
1 CMP $2100
```

Accumulateur - Mémoire = Résultat
$20 - $22 = $FE, le résultat est négatif

Indicateur affectés n - - - - - z c

La valeur de Carry avant l'opération n'affectera pas celle-ci.

TABLE 11.4 – Valeurs possible avec CMP

n	z	c	Résultat	
			8bits	**16bits**
0	0	1	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	1	Nulle = 0	Nulle = 0
0	1	0	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* 6502	65816	*nb d' Octets*	*nb de Cycles*
Immédiat	**CMP** *#const*	C9	x	x	2	2[1]
Absolue	**CMP** *addr*	CD	x	x	3	4[1]
Absolue Long	**CMP** *long*	CF		x	4	5[1]
Direct Page(DP)	**CMP** *dp*	C5	x	x	2	3[1,3]
DP indirect	**CMP** *(dp)*	D2		x	2	5[1,3]
DP indirect Long	**CMP** *[dp]*	C7		x	2	6[1,3]
Absolue indexé par X	**CMP** *addr,X*	DD	x	x	3	4[1,4]
Absolue Long indexé par X	**CMP** *long,X*	DF		x	4	5[1]
Absolue indexé par Y	**CMP** *addr,Y*	D9	x	x	3	4[1,4]
DP indexé par X	**CMP** *dp,X*	D5	x	x	2	4[1,3]
DP indirectement indexé par X	**CMP** *(dp,X)*	C1	x	x	2	6[1,3]
DP indirectement indexé par Y	**CMP** *(dp),Y*	D1	x	x	2	5[1,3,4]
DP long indirectement indexé par Y	**CMP** *[dp],Y*	D7		x	2	6[1,3]
Stack relative (SR)	**CMP** *sr,S*	C3		x	2	4[1]
SR indirectement indexé indexé par Y	**CMP** *(sr,S)*,Y	D3		x	2	7[1]

CMP, une instruction de groupe primaire, a disponible tous les modes d'adressage du groupe primaires et les modèles binaires

* - Ajoutez 1 octet si m=0 (mémoire et accumulateur à 16bits)

1- Ajoutez 1 cycle si m=0 (mémoire et accumulateur à 16bits)

3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)

4- Ajoutez 1 cycle si l'ajout d'index dépasse la page courante (si $00:0000 à $01:0000 par ex)

COP - Co-Processor Enable

/65816

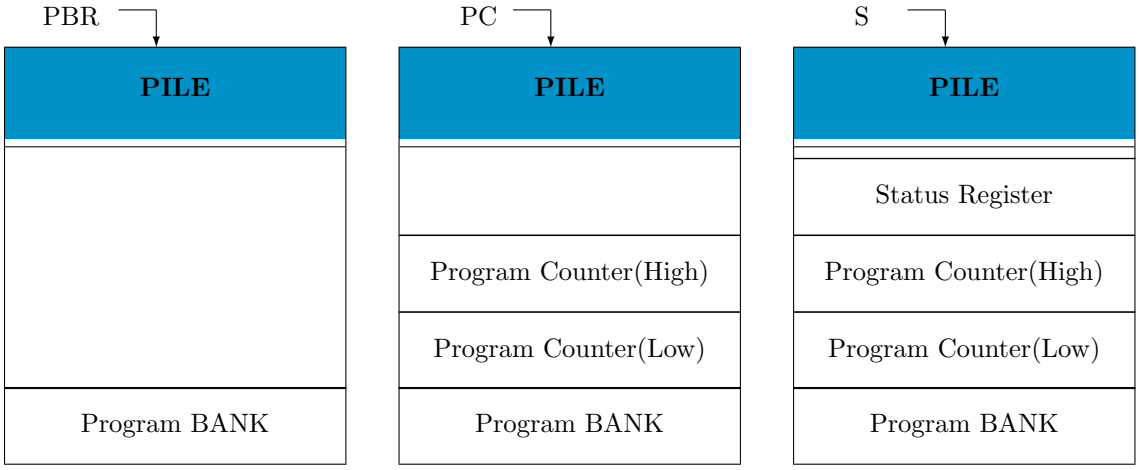
Activer le Coprocesseur

Le coprocesseur opère comme une interruption ; le registre de bloc (banque) de programme est stocké dans la pile, puis le compteur de programme et finalement le Registre d'État(P). Le contenu edes adresses mémoire FFE4 et FF45 est déposé respetivement dans PCL et PCH. Le registre du bloc des programmes est remis à zéro.

Il peut être utilisé pour appeler des instructions d'un Co-processeur graphique par exemple.

Alors la séquence est comme suit :

- COP arrive (ta daaaam ! ! !)
- Registre de Banque de Programme (PBR) et Program Counter (PC) possèdent l'adresse de la prochaine instruction à exécuter. PC = PC+2, alors le "Program Counter" est incrémenter 2 fois.
- Le PBR est mis dans la pile, pour être sauvegarder, et pour savoir où l'on en est au retour de la séquence d'interruption.
- Le registre P est ensuite sauvé dans la pile.



- L'indicateur d'Interruption "I" est mit à 1 (donc désactivation de toutes interruption), après que le Registre P soit sauvegardé.
- Emulation Mode E=1
- Ensuite PC pointe sur la routine d'interruption, PC=FFF4 à FFF5
- Native Mode E=0
- Ensuite PC pointe sur la routine d'interruption, PC=00FFE4 à 00FFE5

Indicateur affectés - - - - d i - -

L'indicateur Decimal "d" est mis à zéro après l'exécution de COP.

L'Interruption "i" n'affecte pas l'instruction COP, quelque son l'état (I=0 ou I = 1). Ce qui veut dire que si le COP veut interrompre le Processeur, il le peut même si les interruptions sont désactivées.

Mais COP met cet indicateur à 1 durant son exécution !

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Stack/Interrupt	**COP**	02		x	2^{++}	7^{7}

++ - COP est sur 1 Octet, mais la valeur du compteur de programme poussé dans la stak(pile) est incrémenté de 2 suivant
l'octet codé facultatif
7- Ajouter 1 cycle si vous êtes en mode native (e=0)

CPX - Compare Index Register X with Memory

6502/65816

On compare la mémoire et l'index X :

L'opérande en mémoire est soustrait du registre d'index X ; le résultat est ignoré. Seuls les bits du Registre d'État (P) sont affectés, aucun opérande ne l'est. Si X est plus grand ou égal à M, le bit C est mis à 1.

Index X => $50

```
1 CPX #$20
```

Résultat = X - Mémoire = $50 - $20=$30

La comparaison est seulement sur des valeurs non signées (excepté pour les comparaisons signé pour l'égalité).

Indicateur affectés n - - - - - z c

Negative "n", est mis à un si jamais le MSB (bit 7 en 8bits et bit 15 en 16bits) du résultat est à 1
Zero "z", est mis à un si le résultat est nulle
Carry "c", est mise à un si l'index "X" est plus grand ou égale, et zéro si il est inférieur.
La valeur de Carry avant l'opération n'affectera pas celle-ci.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* 6502	65816	*nb d' Octets*	*nb de Cycles*
Immédiat	**CPX** *#const*	E0	x	x	2	2^{8}
Absolue	**CPX** *addr*	EC	x	x	3	4^{8}
Direct Page(DP)	**CPX** *dp*	E4	x	x	2	$3^{8,3}$

** - Ajoutez 1 octet si x=0 (registre d'index à 16bits)
8- Ajouter 1 cycle si x =0 (registre dindex à 16 bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)

CPY - Compare Index Register Y with Memory

6502/65816

On compare la mémoire et l'index X :

L'opérande en mémoire est soustrait du registre d'index Y ; le résultat est ignoré. Seuls les bits du Registre d'État (P) sont affectés, aucun opérande ne l'est. Si X est plus grand ou égal à M, le bit C est mis à 1.

Index Y => $20

```
1 CPY #$50
```

Résultat = Y - Mémoire = $20 - $50=$D0

La comparaison est seulement sur des valeurs non signées (excepté pour les comparaisons signé pour l'égalité).

Indicateur affectés n - - - - - z c

Negative "n", est mis à un si jamais le MSB (bit 7 en 8bits et bit 15 en 16bits) du résultat est à 1
Zero "z", est mis à un si le résultat est nulle
Carry "c", est mise à un si l'index "X" est plus grand ou égale, et zéro si il est inférieur.
La valeur de Carry avant l'opération n'affectera pas celle-ci.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Immédiat	**CPY** *#const*	C0	x	x	2	2^{8}
Absolue	**CPY** *addr*	CC	x	x	3	4^{8}
Direct Page(DP)	**CPY** *dp*	C4	x	x	2	$3^{8,3}$

** - Ajoutez 1 octet si x=0 (registre d'index à 16bits)
8- Ajouter 1 cycle si x =0 (registre dindex à 16 bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)

DEC - Decrement

6502/65816

Décrémentation de l'Accumulateur ou de la Mémoire

La valeur 1 est soustraite de l'opérande spécifié ; le bit de report n'est affecté.

```
1 LDA #$20
2 DEC A
```

Le résultat est $1F.

ou

```
1 DEC $2100
```

On décrémente la valeur qui se trouve à l'adresse $2100.

Indicateur affectés n - - - - - z -

TABLE 11.5 – Valeurs possible avec DEC

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* 6502	65816	*nb d' Octets*	*nb de Cycles*
Accumulateur	**DEC** A	3A		x	1	2
Absolue	**DEC** *addr*	CE	x	x	3	6^{2}
Direct Page(DP)	**DEC** *dp*	C6	x	x	2	$5^{2,3}$
Absolue indexé par X	**DEC** *addr,X*	DE	x	x	3	$7^{2,4}$
DP indexé par X	**DEC** *dp,X*	D6	x	x	2	$6^{2,3}$

2- Ajoutez 2 cycle si m=0 (mémoire et accumulateur à 16bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)
4- Ajoutez 1 cycle si l'ajout d'index dépasse la page courante (si $00:0000 à $01:0000 par ex)

DEX - Decrement Index Register X

6502/65816

On décrémente de l'index X.

La valeur 1 est soustraite du registre d'index X ; le bit de report n'est affecté.

```
1 LDX #$00
2 DEX
```

Le résultat est $FF. [1]

Indicateur affectés n - - - - - z -

TABLE 11.6 – Valeurs possible avec DEX

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur*		*nb d' Octets*	*nb de Cycles*
			6502	*65816*		
Implicite	**DEX**	CA	x	x	1	2

[1] -1

DEY - Decrement Index Register Y

6502/65816

On décrémente de l'index Y.

La valeur 1 est soustraite du registre d'index Y ; le bit de report n'est affecté.

```
1 LDY #$00
2 DEY
```

Le résultat est $FF. [1]

Indicateur affectés n - - - - - z -

TABLE 11.7 – Valeurs possible avec DEY

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur*		*nb d' Octets*	*nb de Cycles*
			6502	*65816*		
Implicite	**DEY**	88	x	x	1	2

[1] -1

EOR - Exclusive-OR Accumulator with Memory

6502/65816

On fait un OU exclusif entre le contenue d'une adresse mémoire et l'Accumulateur.

LE OU exclusif exécute une opération logique entre ll'accumulateur A et une adresse mémoire ; le résultat est stocké en A.

A	Memory	Résultat
0	0	0
0	1	1
1	0	1
1	1	0

Accumulateur = Accumulateur EOR Mémoire
Le résultat est mis dans l'accumulateur.

Indicateur affectés n - - - - - z -

Si le résultat est **Positif**

n	z		Nombres en bits	Valeur décimale
			0011 0010	50
		EOR	0111 1111	127
0	0	Résultat	0100 1101	61

Si le résultat est **Null**

n	z		Nombres en bits	Valeur décimale
			0111 1111	127
		EOR	0111 1111	127
0	1	Résultat	0000 0000	0

Si le résultat est **Négatif**

n	z		Nombres en bits	Valeur décimale
			0100 0101	69
		EOR	1111 1101	-3
1	0	Résultat	1011 1000	-72

TABLE 11.8 – Valeurs possible avec EOR

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* 6502	65816	*nb d' Octets*	*nb de Cycles*
Immédiat	**EOR** *#const*	49	x	x	2	2[1]
Absolue	**EOR** *addr*	4D	x	x	3	4[1]
Absolue Long	**EOR** *long*	4F		x	4	5[1]
Direct Page(DP)	**EOR** *dp*	45	x	x	2	3[1,3]
DP indirect	**EOR** *(dp)*	52		x	2	5[1,3]
DP indirect Long	**EOR** *[dp]*	47		x	2	6[1,3]
Absolue indexé par X	**EOR** *addr,X*	5D	x	x	3	4[1,4]
Absolue Long indexé par X	**EOR** *long,X*	5F		x	4	5[1]
Absolue indexé par Y	**EOR** *addr,Y*	59	x	x	3	4[1,4]
DP indexé par X	**EOR** *dp,X*	55	x	x	2	4[1,3]
DP indirectement indexé par X	**EOR** *(dp,X)*	41	x	x	2	6[1,3]
DP indirectement indexé par Y	**EOR** *(dp),Y*	51	x	x	2	5[1,3,4]
DP long indirectement indexé par Y	**EOR** *[dp],Y*	57		x	2	6[1,3]
Stack relative (SR)	**EOR** *sr,S*	43		x	2	4[1]
SR indirectement indexé indexé par Y	**EOR** *(sr,S)*,Y	53		x	2	7[1]

EOR, une instruction de groupe primaire, a disponible tous les modes d'adressage du groupe primaires et les modèles binaires

* - Ajoutez 1 octet si m=0 (mémoire et accumulateur à 16bits)

1- Ajoutez 1 cycle si m=0 (mémoire et accumulateur à 16bits)

3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)

4- Ajoutez 1 cycle si l'ajout d'index dépasse la page courante (si $00:0000 à $01:0000 par ex)

INC - Increment

6502/65816

Incrémentation de l'accumulateur, ou d'une valeur dans une adresse mémoire.

On additionne 1 à l'opérande spécifié ; le bit de report n'est pas affecté.

```
1 LDA #$40
2 INC A
```

A=>$41

Avec l'adresse :

$2100=>$52

```
1 INC $2100
```

$2100=>$53

Instruction non affecté par l'indicateur Decimale "d", et par le Carry

Indicateur affectés n - - - - - z -

Si le résultat est **Positif**

n	z		Nombres en bits	Valeur décimale
			0011 0010	50
		INC		+1
0	0	Résultat	0100 1101	51

Si le résultat est **Null**

n	z		Nombres en bits	Valeur décimale
			1111 1111	-1
		INC		+1
0	1	Résultat	0000 0000	0

Si le résultat est **Negatif**

n	z		Nombres en bits	Valeur décimale
			1111 0000	-16
		INC		+1
1	0	Résultat	1111 0001	-15

TABLE 11.9 – Valeurs possible avec INC

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* 6502	65816	*nb d' Octets*	*nb de Cycles*
Accumulateur	**INC** A	1A		x	1	2
Absolue	**INC** *addr*	EE	x	x	3	6^{2}
Direct Page(DP)	**INC** *dp*	E6	x	x	2	$5^{2,3}$
Absolue indexé par X	**INC** *addr,X*	FE	x	x	3	$7^{2,4}$
DP indexé par X	**INC** *dp,X*	F6	x	x	2	$6^{2,3}$

2- Ajoutez 2 cycle si m=0 (mémoire et accumulateur à 16bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)
4- Ajoutez 1 cycle si l'ajout d'index dépasse la page courante (si $00:0000 à $01:0000 par ex)

INX - Increment Index Register X

6502/65816

On additionne 1 au registre d'index X ; le bit de report n'est pas affecté

```
1 LDX #$40
2 INX
```

X=>$41
Instruction non affecté par l'indicateur Decimale "d", et par la Carry

Indicateur affectés n - - - - - z -

Si le résultat est **Positif**

n	z		Nombres en bits	Valeur décimale
			0000 0000	00
		INX		+1
0	0	Résultat	0000 0001	51

Si le résultat est **Null**

n	z		Nombres en bits	Valeur décimale
			1111 1111	-1
		INX		+1
0	1	Résultat	0000 0000	0

Si le résultat est **Negatif**

n	z		Nombres en bits	Valeur décimale
			1000 0010	-126
		INX		+1
1	0	Résultat	1000 0011	-125

TABLE 11.10 – Valeurs possible avec INX

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**INX**	E8	x	x	1	2

INY - Increment Index Register X

6502/65816

On additionne 1 au registre d'index Y ; le bit de report n'est pas affecté

```
1 LDY #$40
2 INY
```

Y=>$41
Instruction non affecté par l'indicateur Decimale "d", et par la Carry

Indicateur affectés n - - - - - z -

Si le résultat est **Positif**

n	z		Nombres en bits	Valeur décimale
			0000 0000	00
		INY		+1
0	0	Résultat	0000 0001	51

Si le résultat est **Null**

n	z		Nombres en bits	Valeur décimale
			1111 1111	-1
		INY		+1
0	1	Résultat	0000 0000	0

Si le résultat est **Negatif**

n	z		Nombres en bits	Valeur décimale
			1000 0010	-126
		INY		+1
1	0	Résultat	1000 0011	-125

TABLE 11.11 – Valeurs possible avec INY

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**INY**	C8	x	x	1	2

JML - Jump Long

/65816

Saut Long.

Une nouvelle adresse est chargé dans le compteur de programme (PC) et le registre du bloc(banque) de programme(PBR). Les deux octets qui suivent immédiatemment le code opération pointent le premier les trois octets de la nouvelle adresse.

L'instruction JMP en mode d'adressage absolue indirect, a le même code opération que l'instruction JML.
Indicateur affectés n - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Asolue indirect long	**JML** *[addr]* ou **JMP**	DC		x	3	6

JMP - Jump

6502/65816

On ”saute” à l’adresse donnée.

En fait c’est comme un **BRA** ou un **BRL**, sauf qu’ici vous pouvez faire un adressage sur 16 ou 24 bits, et plus sur 8 bits.

Une nouvelle adresse est chargé dans le compteur de programme (PC). Cette adresse peut avoir 16 bits pour faire un branchement dans un bloc, ou 24 bits pour faire un branchement sur l’espace adressable de 16 méga octets.

En effet vous pouvez changez de banque (bloc) comme cela vous conivent. $00 :1234 à $FF :1234, c’est possible. Dans la limite des 16Mega octets ($00 :0000 à $FF :FFFF).

Indicateur affectés n - - - - - z -

Aucun

Mode d’Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d’ Octets*	*nb de Cycles*
Absolue	**JMP** *addr*	4C	x	x	3	3
Asolue indirect	**JMP** *(addr)*	6C	x	x	3	5
Absolue indirect indexé par X	**JMP** *(addr,X)*	7C		x	3	6
Absolue Long	**JMP** *long*	5C		x	4	4
Asolue indirect long	**JMP** *[addr]* ou **JML**	DC		x	3	6

JSL - Jump to Subroutine Long /65816

Branchement (saut) dans un sous programme.

Souvenez-vous l'adressage de 16 bits, $0000 à $FFFF, et quand 64K octets ne suffit pas, on peut passer à un adressage de 24bits, $00 0000 à $FF FFFF, ce qui nous fait un adressage de 16Mo ! C'est déjà plus confortable non ?

Attention le 24bits que je parle est en "adressage", il n'existe aucun mode 24bits ici ! !
N'oublier pas qu'ici on se trouve en 16 bits [1], cette instruction n'existe pas en 8 bits [2].

Le JSR permet donc d'aller chercher une routine (sous programme) dans un adressage plus grand !

Pour plus de précisions, LE PC et le registre de bloc(banque) de programme (PBR) sont stockés dans la pile et de nouvelles valeurs sont chargées dans PC et PBR à partir de la mémoire. Cette instruction autorise un branchement à un sous-programme quelle que soit l'adresse dans un espace mémoire de 16Mo.

```
1 JSR \$001245
```

On va à l'adresse $1245 de la banque $00
L'adresse de la demande (le PC) est mise dans la pile, pour revenir à cet endroit
une fois le sous programme fini.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Absolue Long	**JSL** *long* ou **JSR**	22		x	4	8

[1] mode native e=0
[2] mode emulation e=1

JSR - Jump to Subroutine

/65816

Saut à un sous-programme

Le compteur de programme (PC) est stocké dans la pile et un nouveau PC est chargé à partir de la mémoire.

Contrairement à l'instruction JSL, JSR ne peut aller dans un sous programme qui se trouve à plus de +32768 et -32765, à partir du premier octet de l'opcode.

On charge le PC dans la pile, et ont charge dans se dernier l'adresse du sous programme. Alors que JSL, on insère le PC et le PBR, ainsi on peut faire un adressage de 24bits au lieu de 16 bits.

```
1 JSR $1245
```

Et un adressage long :

```
1 JSR $001245
```

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Absolue	**JSR** *addr*	20		x	3	6
Absolue indirect indexé par X	**JSR** *(addr,X)*	FC		x	3	8
Absolue Long	**JSR** *long* ou **JSL**	22		x	4	8

LDA - Load Accumulator from Memory

6502/65816

Charger l'accumulateur depuis la mémoire.
C'est extrêmement simple :

```
1 LDA #$24
```

La valeur $24 est mise dans l'Accumulateur (A)
C'est tout...

Indicateur affectés n - - - - - z -

Si le résultat est **Positif**

n	z		Nombres en bits	Valeur décimale
			0001 0001	17
		LDA		
0	0	A	0001 0001	17

Si le résultat est **Null**

n	z		Nombres en bits	Valeur décimale
			0000 0000	0
		LDA		
0	1	A	0000 0000	0

Si le résultat est **Negatif**

n	z		Nombres en bits	Valeur décimale
			1111 1110	-2
		LDA		
1	0	A	1111 1110	-2

TABLE 11.12 – Valeurs possible avec LDA

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* 6502	65816	*nb d' Octets*	*nb de Cycles*
Immédiat	**LDA** *#const*	A9	x	x	2	2[1]
Absolue	**LDA** *addr*	AD	x	x	3	4[1]
Absolue Long	**LDA** *long*	AF		x	4	5[1]
Direct Page(DP)	**LDA** *dp*	A5	x	x	2	3[1,3]
DP indirect	**LDA** *(dp)*	B2		x	2	5[1,3]
DP indirect Long	**LDA** *[dp]*	A7		x	2	6[1,3]
Absolue indexé par X	**LDA** *addr,X*	BD	x	x	3	4[1,4]
Absolue Long indexé par X	**LDA** *long,X*	BF		x	4	5[1]
Absolue indexé par Y	**LDA** *addr,Y*	B9	x	x	3	4[1,4]
DP indexé par X	**LDA** *dp,X*	B5	x	x	2	4[1,3]
DP indirectement indexé par X	**LDA** *(dp,X)*	A1	x	x	2	6[1,3]
DP indirectement indexé par Y	**LDA** *(dp),Y*	B1	x	x	2	5[1,3,4]
DP long indirectement indexé par Y	**LDA** *[dp],Y*	B7		x	2	6[1,3]
Stack relative (SR)	**LDA** *sr,S*	A3		x	2	4[1]
SR indirectement indexé indexé par Y	**LDA** *(sr,S)*,Y	B3		x	2	7[1]

LDA, une instruction de groupe primaire, a disponible tous les modes d'adressage du groupe primaires et les modèles binaires

* - Ajoutez 1 octet si m=0 (mémoire et accumulateur à 16bits)

1- Ajoutez 1 cycle si m=0 (mémoire et accumulateur à 16bits)

3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)

4- Ajoutez 1 cycle si l'ajout d'index dépasse la page courante (si $00:0000 à $01:0000 par ex)

LDX - Load index register X from Memory

6502/65816

Charger le registre d'index X depuis la mémoire.
C'est extrêmement simple :

```
1 LDX #$24
```

La valeur $24 est mise dans l'Index X
C'est tout...

Indicateur affectés n - - - - - z -

Si le résultat est **Positif**

n	z		Nombres en bits	Valeur décimale
			0001 0001	17
		LDX		
0	0	X	0001 0001	17

Si le résultat est **Null**

n	z		Nombres en bits	Valeur décimale
			0000 0000	0
		LDX		
0	1	X	0000 0000	0

Si le résultat est **Negatif**

n	z		Nombres en bits	Valeur décimale
			1111 1110	-2
		LDX		
1	0	X	1111 1110	-2

TABLE 11.13 – Valeurs possible avec LDX

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Immédiat	**LDX** *#const*	A2		x	2	2^{8}
Absolue	**LDX** *addr*	AE	x	x	3	4^{8}
Direct Page(DP)	**LDX** *dp*	A6	x	x	2	$3^{8,3}$
Absolue indexé par X	**LDX** *addr,X*	BE		x	3	$4^{8,4}$
DP indirectement indexé par X	**LDX** *(dp,X)*	B6		x	2	$4^{8,3}$

** - Ajoutez 1 octet si x=0 (registre d'index à 16bits)
8- Ajouter 1 cycle si x =0 (registre dindex à 16 bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)
4- Ajoutez 1 cycle si l'ajout d'index dépasse la page courante (si $00:0000 à $01:0000 par ex)

LDY - Load index register Y from Memory

6502/65816

Charger le registre d'index Y depuis la mémoire.
C'est extrêmement simple :

```
1 LDY #$24
```

La valeur $24 est mise dans l'Index Y
C'est tout...

Indicateur affectés n - - - - - z -

Si le résultat est **Positif**

n	z		Nombres en bits	Valeur décimale
			0001 0001	17
		LDY		
0	0	Y	0001 0001	17

Si le résultat est **Null**

n	z		Nombres en bits	Valeur décimale
			0000 0000	0
		LDY		
0	1	Y	0000 0000	0

Si le résultat est **Negatif**

n	z		Nombres en bits	Valeur décimale
			1111 1110	-2
		LDY		
1	0	Y	1111 1110	-2

TABLE 11.14 – Valeurs possible avec LDY

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Immédiat	**LDY** *#const*	A0		x	2	2^{8}
Absolue	**LDY** *addr*	AC	x	x	3	4^{8}
Direct Page(DP)	**LDY** *dp*	A4	x	x	2	$3^{8,3}$
Absolue indexé par X	**LDY** *addr,X*	BC		x	3	$4^{8,4}$
DP indirectement indexé par X	**LDY** *(dp,X)*	B4		x	2	$4^{8,3}$

** - Ajoutez 1 octet si x=0 (registre d'index à 16bits)
8- Ajouter 1 cycle si x =0 (registre dindex à 16 bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)
4- Ajoutez 1 cycle si l'ajout d'index dépasse la page courante (si $00:0000 à $01:0000 par ex)

LSR - Logical Shift Memory or Accumulator Right

6502/65816

Décalalge logique à droite. On peut l'interprété comme un Diviseur.

Tous les bits de l'opérande sont décalés à droite d'une position. Le bit 0 est transféré dans l'indicateur de report, le bit 7 est mis à zéro ; en mode 16 bits, le bit 15 est mis à zéro.

Pour la divion :

Si dans l'Accumulateur on a 70, et que l'on veut avoir 35 !

```
1 LDA #$46 ; (70 en décimale)
2 LSR
```

Et l'on obtient 0010 0011 => $24 => 35,

```
1 LDA #$47 ; (71 en décimale)
2 LSR
```

Et l'on obtient 0010 0011 => $24 => 35,

Eh mais... Ca ne divise pas correctement !! Oui bon ca ne divise pas correctement ok, car sa ne gère pas super bien les nombres impairs je vous l'accorde.

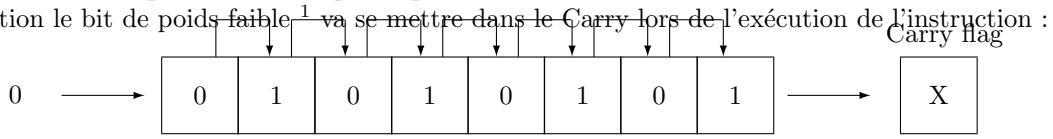
En négatif maintenant

```
1 LDA #$A5 ;(-91 en décimale)
2 LSR
```

Et l'on obtient 0101 0010 => $52 => 82

Eh ba là sa divise pas, sa fait n'importe quoi

Attention le bit de poids faible [1] va se mettre dans le Carry lors de l'exécution de l'instruction :



Mais la solution est toute bête, pour les nombre positif seulement. A la fin de la division si la carry est à 1, sa veut dire que c'est un nombre impaire (en gros il y a une Retenue !!)

```
1 LDA #$47 ; (71 en décimale)
2 LSR
```

Et l'on obtient 0010 0011 => $24 => 35, Carry=1

Donc le résultat est de 35,5. car la carry peut être interpréter comme le nombre "+0,5".

C'est très pratique pour la division non signé.

[1] LSB, le bit0 en 8 bits et 16 bits

Indicateur affectés n - - - - - z c

Bon le "Carry" ici ne sert qu'à recevoir le bit 0 [1], il n'indique en rien le signe du résultat, c'est une Retenue.

Si le résultat est **Positif** d'un nombre pair

n	z	c		Nombres en bits	Valeur décimale
		0		0100 0110	70
			LSR		
0	0	0	Résultat	0010 0011	35

Si le résultat est **Positif** d'un nombre impair

n	z	c		Nombres en bits	Valeur décimale
		0		0100 0111	71
			LSR		
0	0	1	Résultat	0010 0011	35

Si le résultat est **Nulle**

n	z	c		Nombres en bits	Valeur décimale
		0		0000 0000	0
			LSR		
0	1	0	Résultat	0000 0000	0

Si le résultat est **Negatif**[2]

Impossible (ou si vous trouvez, contacter moi !)

TABLE 11.15 – Valeurs possible avec LSR

n	**z**	**c**	**Résultat**	
			8bits	**16bits**
0	0	0	Positif entier, de 1 à 63 inclus	de 1 à 16383 inclus
0	0	1	Positif flottant , de 1.5 à 63.5 inclus	de 1.5 à 16383.5 inclus
0	1	0	Nulle = 0	Nulle = 0
0	1	1	= 0.5	= 0.5

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur*		*nb d'*	*nb de*
			6502	*65816*	*Octets*	*Cycles*
Accumulateur	**LSR** A	4A		x	1	2
Absolue	**LSR** *addr*	4E	x	x	3	6[2]
Direct Page(DP)	**LSR** *dp*	46	x	x	2	5[2,3]
Absolue indexé par X	**LSR** *addr,X*	5E	x	x	3	7[2,4]
DP indexé par X	**LSR** *dp,X*	56	x	x	2	6[2,3]

2- Ajoutez 2 cycle si m=0 (mémoire et accumulateur à 16bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)
4- Ajoutez 1 cycle si l'ajout d'index dépasse la page courante (si $00:0000 à $01:0000 par ex)

[1] le LSB

[2] Impossible car le "n" sera mis à zéro quelque soit le résultat.

MVN - Block Move Next

/65816

Transfert de bloc négatif.

Ceci consiste à déplacer un bloc mémoire débutant à une adresse inférieur et terminant à une adresse supérieure.

La source, la destination et la longueur des octets à déplacer sont respectivement : index X, Y et C[1].

L'adresse **source** doit être mise dans l'index de registre X, c'est une adresse de départ que vous devez mettre.

L'adresse **destination** doit être mise dans l'index du registre Y, se doit être le début de la nouvelle adresse.

La **longueur** moins 1, doit être chargée dans le double accumulateur (C). Donc mode Native activé (e=0), et le "registre d'index 8bit" doit être en **16bits** ; indicateur x=0[2]. Ces deux indicateurs ("m","e" et "x") doivent être initialisé précédemment.

Les registres X et Y sont incrémentés aprés chaque itération. Le registre A est décrémenté aprés chaque répétition.

Exemple :

X => $2100
Y => $4200
C => $0010

```
1 MVN $00,$05
```

On déplace 11 octets de la mémoire $00-2100 à la mémoire $05-4200
Et oui on peut copier à partir de la banque $00, dans la dernière banque $FF, si elle existe...
Donc la mémoire $2100 à $2110 est déplacé dans $4200 à $4210.

Si le "registre d'index 8bits" est en **8bits** (indicateur x = 1), ou l'émulation active (e = 1), alors les blocs étant spécifiés doivent nécessairement être en page zéro puisque les Octets les plus élevés des registres d'index contiendront des zéros.

Exemple :

X => $05
Y => $86
A => $10 (C devient A quand l'index 8 bit x=1)

```
1 MVN $21,$42
```

Eh bien là on déplace 11 octets de la mémoire $00-2105 à la mémoire $00-4286.

Indicateur affectés - - - - - - - -

Aucun

[1](c'est le double Accumulateur, vous avez A qui est sur 8bits bas, B sur les 8bits haut, et là C sur 16bits, qui réuni B et A en 1)

[2]du Registre d'État(P)

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Block Move	**MVN** *srcbk,destbk*	54	x	x	3	[9]

9- Ajouter 7 cycles par octets déplacé

MVP - Block Move Previous

/65816

Transfert de bloc positif.

Ceci consiste à déplacer un bloc mémoire débutant à une adresse supérieure et terminant à une adresse inférieur.

La source, la destination et la longueur des octets à déplacer sont respectivement : index X, Y et C[1].

L'adresse **source** doit être mise dans l'index de registre X, c'est une adresse de fin que vous devez mettre.

L'adresse **destination** doit être mise dans l'index du registre Y, se doit être la fin de la nouvelle adresse.

La **longueur** moins 1, doit être chargée dans le double accumulateur (C), donc mode Native activé e=0, et "registre d'index 8bit" est en **16bits**; indicateur x=0[2]. Ces indicateur ("e" et "x") doivent être initialisés précédemment.

Les registres X et Y sont décrémentés aprés chaque itération. Le registre A est décrémentés aprés chaque répétition.

Exemple :

X => $2110
Y => $4210
C => $0010

```
1 MVP $00,$05
```

On déplace 11 octets de la mémoire $00-2110 à la mémoire $05-4210
Et oui on peut copier à partir de la banque $00, dans la dernière banque $FF, si elle existe...
Donc la mémoire $2110 à $2100 est déplacé dans $4210 à $4200.

Si le "registre d'index 8bits" est en **8bits** (indicateur x = 1), ou mode émulation activé (e = 1), alors les blocs étant spécifiés doivent nécessairement être en page zéro puisque les Octets les plus élevés des registres d'index contiendront des zéros.

Exemple :

X => $05
Y => $86
A => $10 (C devient A quand l'index 8 bit x=1)

```
1 MVP $21,$42
```

Eh bien là on déplace 11 octets de la mémoire $00-2105 à la mémoire $00-4286.

Indicateur affectés - - - - - - - -

Aucun

[1](c'est le double Accumulateur, vous avez A qui est sur 8bits bas, B sur les 8bits haut, et là C sur 16bits, qui réuni B et A en 1)

[2]du regsitre d'état du processeur (P)

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Block Move	**MVP** *srcbk,destbk*	44		x	3	[9]

9- Ajouter 7 cycles par octets déplacé

NOP - Not Operation

6502/65816

N'exécute aucune opération. Le processeur cesse toute activité pendant 2 cycles d'horloge.

Cette instruction est souvent utilisé dans les temporisations, si on doit attendre qu'un périphérique réponde, on peut faire plusieurs NOP pour attendre, et même des boucles de NOP.

Ou, elle peut utilisé de l'espae mémoire.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**NOP**	EA	x	x	1	2

ORA - OR Accumulator with Memory

6502/65816

Opération logique "OU" entre le contenue d'une adresse mémoire et l'accumulateur.
La fonction OR agit entre l'accumulateur A et l'adresse mémoire ; le résultat est stocké dans A.

Nombre 1	Nombre 2	Résultat
0	0	0
0	1	1
1	0	1
1	1	1

Tout est dans la table !

Pour l'exemple, vous charger une valeur dans l'accumulateur A

```
1 LDA #$11
```

Vous faite un OR de $21 avec la valeur chargé dans A

```
1 ORA #$21
```

Le résultat se retrouvera dans A

A = $31

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Immédiat	**ORA** *#const*	09	x	x	2	2[1]
Absolue	**ORA** *addr*	0D	x	x	3	4[1]
Absolue Long	**ORA** *long*	0F		x	4	5[1]
Direct Page(DP)	**ORA** *dp*	05	x	x	2	3[1,3]
DP indirect	**ORA** *(dp)*	12		x	2	5[1,3]
DP indirect Long	**ORA** *[dp]*	07		x	2	6[1,3]
Absolue indexé par X	**ORA** *addr,X*	1D	x	x	3	4[1,4]
Absolue Long indexé par X	**ORA** *long,X*	1F		x	4	5[1]
Absolue indexé par Y	**ORA** *addr,Y*	19	x	x	3	4[1,4]
DP indexé par X	**ORA** *dp,X*	15	x	x	2	4[1,3]
DP indirectement indexé par X	**ORA** *(dp,X)*	01	x	x	2	6[1,3]
DP indirectement indexé par Y	**ORA** *(dp),Y*	11	x	x	2	5[1,3,4]
DP long indirectement indexé par Y	**ORA** *[dp],Y*	17		x	2	6[1,3]
Stack relative (SR)	**ORA** *sr,S*	03		x	2	4[1]
SR indirectement indexé indexé par Y	**ORA** *(sr,S)*,Y	13		x	2	7[1]

ORA, une instruction de groupe primaire, a disponible tous les modes d'adressage du groupe primaires et les modèles binaires
* - Ajoutez 1 octet si m=0 (mémoire et accumulateur à 16bits)
1- Ajoutez 1 cycle si m=0 (mémoire et accumulateur à 16bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)
4- Ajoutez 1 cycle si l'ajout d'index dépasse la page courante (si $00:0000 à $01:0000 par ex)

PEA - Push Effective Absolute Address

/65816

Stocke l'adresse effective absolue dans la pile.
Stocke le mot data immédiat dans la pile.
Les deux octets de données suivant immédiatement le code opération sont stocké dans la pile.
Après l'exécution la pile possèdera l'adresse, et pas la valeur que contient l'adresse.
$5100 => $83

```
1 PEA $5100
```

La pile aura $5100 et non $83.
La pile est décrémentée 2 fois, car l'adresse est sur 16 octets.
Attention, que vous soyez en mode 16 ou 8bits, cette instruction "pousse" un mot de 16 bits dans la pile.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Stack absolute	**PEA** *addr*	F4		x	3	5

PEI - Push Effective Indirect Address /65816

Stocke l'adresse effective indirecte dans la pile. (note : A compléter).

La pile possède la valeur de l'adresse.

Attention, que vous soyez en mode 16 ou 8bits, cette instruction stocke un mot de 16 bits dans la pile.[1]

L'octet de données suivant imédiatemment le code opération est additionné au Registre de Page Zéro (D), et celui-ci est utilisé comme indicateur de deux octets pour être stocké dans la pile. Le registre D n'est pas affecté et les deux octets doivent être en banque zéro.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Stack(Direct Page Indirect)	**PEI** *(dp)*	D4		x	2	6[3]

3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)

[1] l'octet haut et l'octet bas

PER - Push Effective PC Relative Indirect Address /65816

Stocke l'adresse relative effective du compteur de programme (PC) dans la pile.

En gros vous sauvegardez un endroit de votre code, vous allez ailleurs, et vous revenez, c'est comme faire un JSR, par exemple.

Pour plus de précision ! Les deux octets de données suivant immédiatement le code opération sont additionnés au compteur d'instruction, aprés que le PC a été mis à jour pour désigner l'instruction suivante, et cette valeur est alors stocké dans la pilz. le PC et le registre de banque sont, en réalité, additionnés au PC+2.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Stack (Compteur de programme relatif long)	**PER** *label*	62			3	6

PHA - Push Accumulator

6502/65816

Stocke le contenu de l'accumulateur dans la pile.

On sauvegarde l'accumulateur dans la pile, en mode 8 bits, on enregistre un octet, et en 16bits 2 octets.

Pour être plus précis, le contenue de l'accumulateur est poussé sur la pile. Si le processeur est en mode mémoire 16 bits, les deux octets sont poussés sur la pile et l'indicateur de pile S est décrementé de 2.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* 6502	65816	*nb d' Octets*	*nb de Cycles*
Stack (push)	**PHA**	48	x	x	1	3[1]

1- Ajoutez 1 cycle si m=0 (mémoire et accumulateur à 16bits)

PHB - Push Data Bank Register

/65816

Stocke le registre de la banque de donnée dans la pile.

Souvenez vous, le registre de la banque de donnée (DBR) possède la valeur de la banque actuellement utilisée pour lire ou écrire des données. En gros si elle a la valeur $50, et que vous écrivez à l'adresse $4521, ce sera l'adresse $50-4521 dans laquelle vous écrirez.

Bon, après ce bref rappel, on sauvegarde le registre de la banque de donnée dans la pile.

Ca peut servir si on veut changer de banque faire un chargement, et revenir sur la banque précédente pour retravailler dessus !

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Stack (push)	**PHB**	8B		x	1	3

PHD - Push Direct Page Register

/65816

Stocke le Registre de Page Zéro (D) dans la pile.

Le registre D, pointe une adresse de travail, et peut adresser 64ko (0000 à FFFF) de la mémoire.

En gros si ce registre vaut $3000, et que l'on charge une valeur à l'adresse $12 (en mode 8 bits), $3000 + $12 = $3012, on va écrire dans l'adresse $3012.

Ca pourra nous servir si jamais on va dans un sous-programme qui modifie cette valeur, alors que nous en avons besoins dans notre programme principal.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Stack (push)	**PHD**	0B		x	1	4

PHK - Push Program Bank Register /65816

Stocke le Registre de Banque de Programme (PBR) dans la pile.

On se souvient que le PBR est "l'extention" de l'adresse du compteur de programme, PBR avec le PC représente 24 bit, c'est l'adresse réel de la prochaine instruction à exécuter.

On sauvegarde la banque du programme dans la pile tout simplement (ce qui prend 2 octets de place).

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Stack (push)	**PHK**	4B		x	1	3

PHP - Push Processor Status Register

6502/65816

Stocke le Registre d'État du Processeur (P).

En fait vous vous souvenez tout les indicateurs, Negative Overflow etc. . . , et bien là on les sauvegarde dans la pile.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Stack (push)	**PHP**	08	x	x	1	3

PHX - Push Index Register X

/65816

Stocke le registre d'index X sur la pile.

On sauvegarde X dans la pile [1])... tout simplement.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Stack (push)	**PHX**	DA		x	1	3[8]

8- Ajouter 1 cycle si x =0 (registre dindex à 16 bits)

[1] Pile

PHY - Push Index Register X

/65816

Stocke le registre d'index Y sur la pile.
On sauvegarde Y dans la Pile)... . tout simplement.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Stack (push)	**PHY**	5A		x	1	3[8]

8- Ajouter 1 cycle si x =0 (registre dindex à 16 bits)

PLA - Pull Accumulator

6502/65816

Dépiler l'accumulateur.

On prend la valeur qui a été précédemment empilé dans la pile, pour le mettre dans l'Accumulateur.

Si le processeur est en mode 16 bits, les octets sont dépilés et le pointeur de pile S est incrémenté de 2.

Indicateur affectés - - - - - - - -

Table 11.16 – Valeurs possible avec PLA

n	**z**	**Résultat à la sortie de la valeur de la pile**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur*		*nb d' Octets*	*nb de Cycles*
			6502	*65816*		
Stack (pull)	**PLA**	68	x	x	1	4[8]

8- Ajouter 1 cycle si x =0 (registre dindex à 16 bits)

PLB - Pull Data Bank Register /65816

Dépiler la valeur du registre de bloc de donnée.

Vous vous souvenez, que le registre de banque de donnée, est celui qui permet de savoir, où l'on pointe dans la mémoire ?

Ici on peut récupère cette donnée pour savoir où l'on pointait avant !

Bien sûr il faut l'avoir sauvegardé précédemment...

Indicateur affectés n - - - - - z -

TABLE 11.17 – Valeurs possible avec PLB

n	**z**	**Résultat à la sortie de la valeur de la pile** **8bits**	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Stack (pull)	**PLB**	AB		x	1	4

PLD - Pull Direct Page Register /65816

Dépiler le Registre de Page Zéro (D).

Le registre D, est celui qui permet de savoir dans quel page de la mémoire on se trouve. l'instruction le charge à partir de la pile.

Indicateur affectés n - - - - - z -

Table 11.18 – Valeurs possible avec PLD

n	**z**	**Résultat à la sortie de la valeur de la pile**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur*		*nb d' Octets*	*nb de Cycles*
			6502	*65816*		
Stack (pull)	**PLD**	2B		x	1	5

PLP- Pull Status Flags

6502/65816

Dépile le Registre d'État (P).

Souvenez-vous du Registre P, c'est celui qui détient les indicateurs oVerflow, Carry, Negative, Zero, etc...
Ici le Registre P est chargé à partir de la pile.

Indicateur :	**n**	v	-	b	d	i	z	c	Emulation(e=1)
	n	v	m	x	d	i	z	c	Native(e=0)

Tous les indicateurs sont remplacés par le registre qui est tiré de la pile.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Stack (pull)	**PLP**	28	x	x	1	4

PLX- Pull Index Register X from Stack /65816

Dépiler le registre d'index X.

Le registre d'index X est chargé à partir de la pile. Si le processeur est en mode 16 bits, les deux octets sont dépilé et l'indicateur de pile (S) est incrémenté de 2.

Indicateur affectés n - - - - - z -

Table 11.19 – Valeurs possible avec PLX

n	**z**	**Résultat à la sortie de la valeur de la pile**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur*		*nb d' Octets*	*nb de Cycles*
			6502	*65816*		
Stack (pull)	**PLX**	FA		x	1	4[8]

8- Ajouter 1 cycle si x =0 (registre dindex à 16 bits)

PLY - Pull Index Register Y from Stack /65816

Dépiler le registre d'index Y.

Le registre d'index Y est chargé à partir de la pile. Si le processeur est en mode 16 bits, les deux octets sont dépilé et l'indicateur de pile (S) est incrémenté de 2.

Indicateur affectés n - - - - - z -

Table 11.20 – Valeurs possible avec PLY

n	**z**	**Résultat à la sortie de la valeur de la pile**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur*		*nb d' Octets*	*nb de Cycles*
			6502	*65816*		
Stack (pull)	**PLY**	7A		x	1	4[8]

8- Ajouter 1 cycle si x =0 (registre dindex à 16 bits)

REP - Reset Status Bits

/65816

Remise à zero des bits d'état...

Cela dépend bien sur si vous êtes en mode emulation ou non.

Indicateur :	**n**	v	-	b	d	i	z	c	Emulation(e=1)
	n	v	m	x	d	i	z	c	Native(e=0)

La fontion ET logique (AND) agit sur le Registre d'État (P) et sur le complément de l'octet suivant immédiatement le code opération ; et le résultat est stocké dans P. Pour le contraire, voir l'instruction "SEP - Set Status Bits" à la page 215

Par exemple :

```
1 REP #30   ; remise à zero des bits "m" et "x" (mode native). Mémoire Accumulateur
      16bits, X/Y 16bits
2 REP #$30
3
4 REP #FF  ; remise à zero de tout les bits
5 REP #$FF
6
7 REP #00   ; remise à zero d'aucun bit (à quoi ca sert ? je ne le sais pas encore
      ...)
8 REP #$00
```

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Immédiat	**REP** *#const*	C2		x	2	3

ROL - Rotate Memory or Accumulator Left

6502/65816

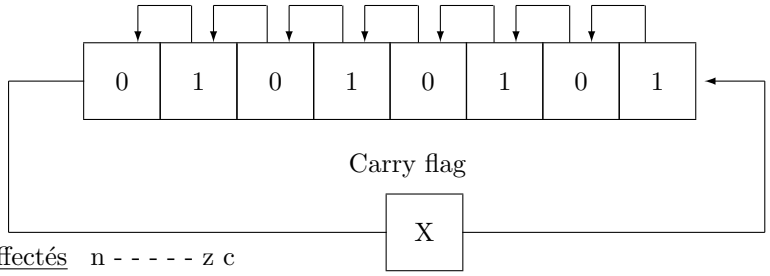
Rotation vers la gauche de la mémoire ou de l'accumulateur.

Regardez les instructions ASL et ADC, c'est ce que réunit cette opération.

On peut interpréter ceci comme une multiplication additionné de la carry.

Tous les bits de l'opérande sont décalés d'une position vers la gauche. Le bit 0 vient du bit de report(Carry) et le bit 7 dans le bit de report (Carry). En mode 16 bits, le bit 15 est transféré au bit de report(Carry).

On peut aussi envoyer l'octet dans un port en mode série (1 bit à la fois) et garder le nombre intacte ensuite [1].



Indicateur affectés n - - - - - z c

Comme je dis souvent, un bon exemple vaut mieux qu'une longue explication [2].

Remarque : Négative est à 1 si le MSB [3] est à 1.

Si le résultat est **Positif**

n	z	c		Nombres en bits	Valeur décimale
		1		0000 0000	0
			ROL		(0*2)+1
0	0	1	Résultat	0000 0001	1

Si le résultat est **Null**

n	z	c		Nombres en bits	Valeur décimale
		0		0000 0000	0
			ROL		(0*2)+0
0	1	0	Résultat	0000 0000	0

Si le résultat est **Negatif**

n	z	c		Nombres en bits	Valeur décimale
		1		1100 0001	-63
			ROL		(-63*2)+1
1	0	1	Résultat	1000 0011	-125

[1]

[2] quand on peut le faire... et surtout quand on sait comment le faire

[3] le bit7 en mode 8bits, ou le bit15 en mode 16bits

TABLE 11.21 – Valeurs possible avec ROL

n	**z**	**c**	**Résultat** **8bits**	**16bits**
0	0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	0	Positif Overflow, de 128 à 255 inclus	de 32768 à 65535 inclus
0	0	1	Négatif Overflow, de -129 à -255 inlcus	de -32769 à -65535 inclus
1	0	1	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	0	Nulle = 0	Nulle = 0
1	0	1	= -256	= -65536

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Accumulateur	**ROL** A	2A		x	1	2
Absolue	**ROL** *addr*	2E	x	x	3	6^{2}
Direct Page(DP)	**ROL** *dp*	26	x	x	2	$5^{2,3}$
Absolue indexé par X	**ROL** *addr,X*	3E	x	x	3	$7^{2,4}$
DP indexé par X	**ROL** *dp,X*	36	x	x	2	$6^{2,3}$

2- Ajoutez 2 cycle si m=0 (mémoire et accumulateur à 16bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)
4- Ajoutez 1 cycle si l'ajout d'index dépasse la page courante (si $00:0000 à $01:0000 par ex)

ROR - Rotate Memory or Accumulator Right

6502/65816

Rotation vers la droite de la mémoire ou de l'accumulateur.

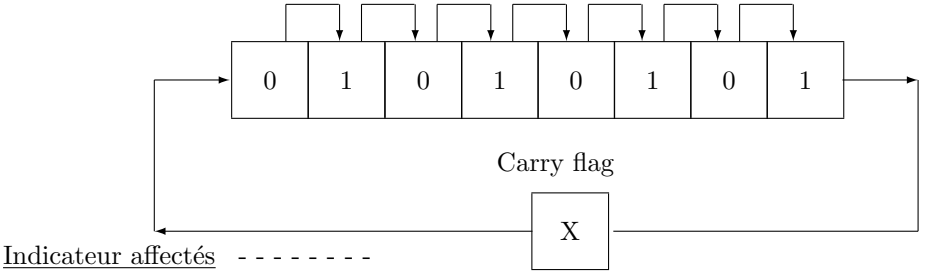
Regardez l'instructions LSR, vous ajouter $80 à la fin de l'instruction si Carry = 1 , et sa vous fait un ROR!!, c'est ce que réunit cette opération.

Sa vous évite plusieurs lignes d'instructions, et sa prend moins de temps.

Tous les bits de l'opérande sont décalés d'une position vers la droite. Le bit 0 vient du bit de report(Carry) et le bit 7 dans le bit de report (Carry). En mode 16 bits, le bit 15 est transféré au bit de report(Carry).

On peut interpréter ceci comme une division additionnée.

On peut aussi envoyer l'octet dans un port en mode série (1 bit à la fois) et garder le nombre intacte ensuite (cet exemple sera fait plus loin dans la doc).



Indicateur affectés - - - - - - - -

Aucun

Si le résultat est **Positif**

n	z	c		Nombres en bits	Valeur décimale
		0		0001 0000	32
			ROL		(32/2)+0
0	0	0	Résultat	0000 1000	16

Si le résultat est **Null**

n	z	c		Nombres en bits	Valeur décimale
		0		0000 0000	0
			ROL		(0/2)+0
0	1	0	Résultat	0000 0000	0

Si le résultat est **Negatif**

n	z	c		Nombres en bits	Valeur décimale
		1		1110 0000	-16
			ROL		(-16/2)
1	0	0	Résultat	1111 0000	-8

!!! ATTENTION !!! Avant de lancer l'instruction, vous devez initialiser la carry à 1, si le chiffre que vous voulez diviser est négatif

TABLE 11.22 – Valeurs possible avec ROR

n	z	c	Résultat 8bits	16bits
0	0	0	Positif entier, de 1 à 63 inclus	de 1 à 16383 inclus
0	0	1	Positif flottant , de 1.5 à 63.5 inclus	de 1.5 à 16383.5 inclus
1	0	0	Négatif entier, de -1 à -64 inclus	de -1 à -16384 inclus
1	0	1	Négatif flottant, de -0.5 à -63.5 inclus	de -0.5 à -16383.5 inclus
0	1	0	Nulle = 0	Nulle = 0
0	1	1	= 0.5	= 0.5

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* 6502	65816	*nb d' Octets*	*nb de Cycles*
Accumulateur	**ROR** A	6A	x	x	1	2
Absolue	**ROR** *addr*	6E	x	x	3	6^{2}
Direct Page(DP)	**ROR** *dp*	66	x	x	2	$5^{2,3}$
Absolue indexé par X	**ROR** *addr,X*	7E	x	x	3	$7^{2,4}$
DP indexé par X	**ROR** *dp,X*	76	x	x	2	$6^{2,3}$

2- Ajoutez 2 cycle si m=0 (mémoire et accumulateur à 16bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)
4- Ajoutez 1 cycle si l'ajout d'index dépasse la page courante (si $00:0000 à $01:0000 par ex)

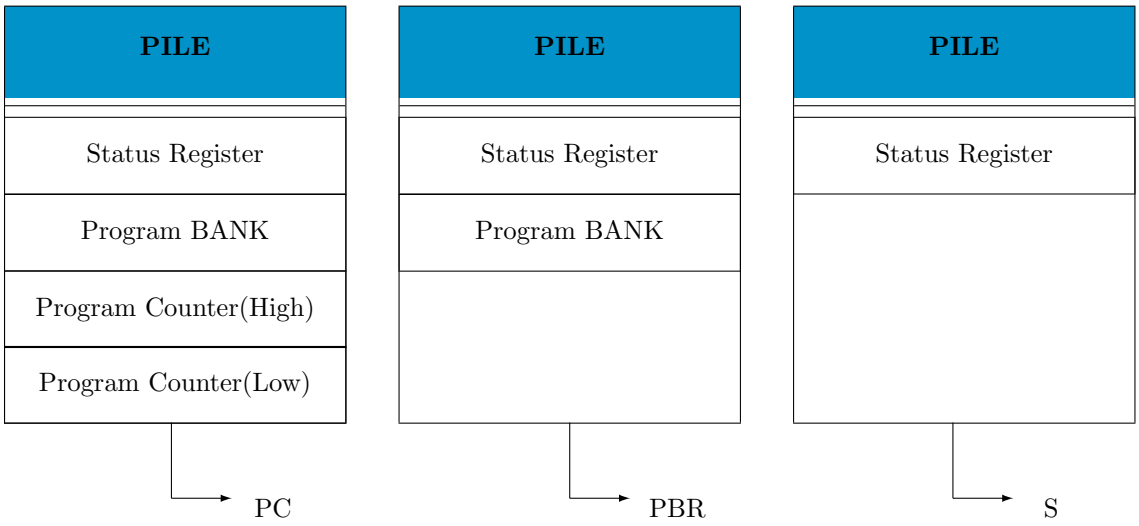
RTI - Return from Interrupt

6502/65816

Retour d'interruption.

Cette instruction sert à dire "Interruption finie, on peut reprendre de la où on était".

On récupère notre Registre de Banque de Programme (PBR), notre PC (ces deux derniers nous permettent de pointer sur la prochaine instruction à exécuter) et enfin le Registre P. Le tout ayant été sauvegardé dans la pile avant l'interruption.



Pour être plus précis, le Registre d'État (P), le PC et le registre de bloc de programme sont chargés à partir de la pile mémoire. Cette instruction a l'effet inverse d'une interruption, elle doit être placé à la fin d'un sous programme d'interruption.

Indicateur affecté	**n**	v	-	b	d	i	z	c	Emulation(e=1)
	n	v	m	x	d	i	z	c	Native(e=0)

Tous, vu que l'on récupère le Registre P, de la pile, les bits vont êtres forcément changés.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur 6502*	*Disponnible sur 65816*	*nb d' Octets*	*nb de Cycles*
Stack (RTI)	**RTI**	40	x	x	1	6[7]

7- Ajouter 1 cycle si vous êtes en mode native (e=0)

RTL - Return from Subroutine Long

/65816

Retour d'un sous programme "Long".

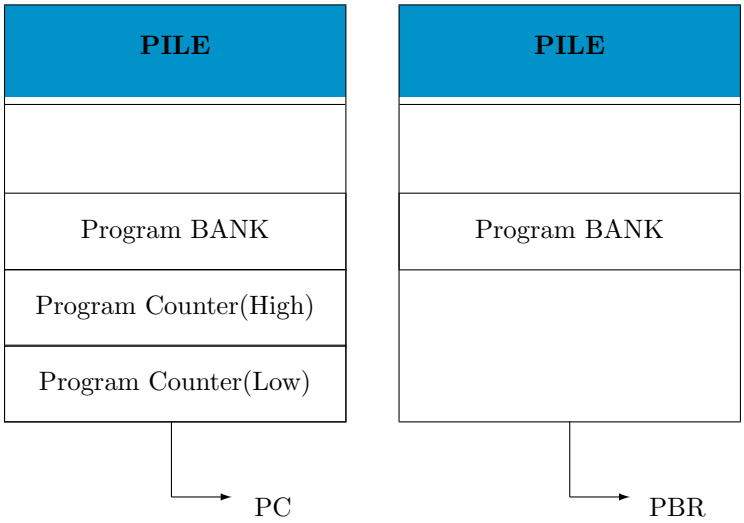
Après avoir fait un JSR, vous êtes allé dans une routine, qui se trouve dans un endroit de la mémoire, qui nécessitait de sauvegarder la banque de programme(PBR).

On va dire que vous êtes partie de l'adresse $00-0805, et vous avez appelé la routine qui se trouve à l'adresse $05-4525. Le PBR est donc passé de $00 à $05.

Et le compteur de programme (PC) qui contient l'adresse de la prochaine exécution bien évidemment.

Si vous aviez gardé la banque de programme actuelle vous auriez exécuté l'instruction qui trouvait à l'adresse $00-4525 !

Bref ici on récupère les données dans la pile, pour revenir exécuter le program principal.



Pour être plus précis, le PC et le registre de bloc de programme sont extraits de la pile mémoire. Cette instruction a l'effet inverse d'une instruction JSL.

Indicateur affectés - - - - - - - -

Aucun

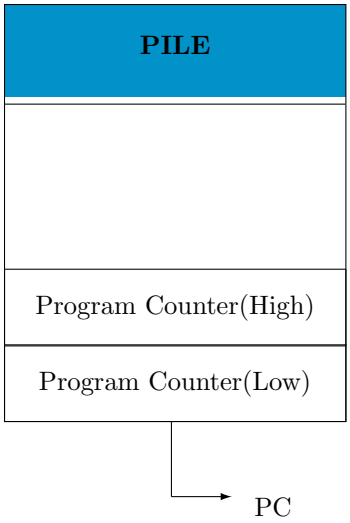
Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Stack (RTL)	**RTL**	6B		x	1	6

RTS- Return from Subroutine

6502/65816

Retour d'une routine (ou d'un sous programme, c'est la même chose).

Quand on appelle une routine, avec l'instruction JSR, on sauvegarde le "Program Counter" actuel dans la pile. Quand on quitte la routine, on prend la sauvegarde de la pile et on la charge dans le registre "Program Counter"



Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Stack (RTS)	**RTS**	60	x	x	1	6

SBC - Subtract with Borrow from Accumulator

6502/65816

On Soustrait le contenue d'une adresse mémoire à la valeur de l'"Accumulateur" avec report (Carry) !

Un exemple :

A=$50

```
1 SBC #$42
```

A=>$07, si Carry = 0 on soustrait 1(50-42-1=7)
A=>$08, si Carry = 1 (50-42-0=8)

Pour être plus précis ; L'opérande en mémoire et le complément du bit de report(Carry) sont soustraits à l'accumulateur.

Indicateur affectés n v - - - - z c

Mieux vaut un bon exemple [1]
Si le résultat **excède 127**

n	v	z	c		Nombres en bits	Valeur décimale
				Accumulateur	0111 1111	127
			0	SBC #$05	1111 0110	(127-(-10))-1
0	1	0	1	Résultat	1000 1000	136

Si le résultat est **Positif**

n	v	z	c		Nombres en bits	Valeur décimale
				Accumulateur	0000 1010	10
			0	SBC #$05	1111 1010	(10-5)-1
0	0	0	1	Résultat	0000 0100	4

Si le résultat **est à Zéro**

n	v	z	c		Nombres en bits	Valeur décimale
				Accumulateur	1111 0000	-16
			1	SBC #$10	1111 0000	(-16-16)-0
0	0	1	1	Résultat	0000 0000	0

Si le résultat **Négatif**

n	v	z	c		Nombres en bits	Valeur décimale
				Accumulateur	1110 1111	-33
			0	SBC #$10	0001 0000	(-33-16)-1
1	0	0	0	Résultat	1111 0001	-50

Si le résultat est **inférieur à -128**

n	v	z	c		Nombres en bits	Valeur décimale
				Accumulateur	1000 0001	-127
			0	SBC #$BE	0100 0010	-127-66-1
1	1	0	1	Résultat	1100 0100	-193

[1] je rappel que ce processeur utilise des nombre "signé" 8bits (-128 à +127) ou 16bits (-32768 à 32767)

TABLE 11.23 – Valeurs possible avec SBC

n	v	z	c	Résultat 8bits	Résultat 16bits
0	0	0	1	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	1	0	0	Positif Overflow, de 128 à 255 inclus	de 32768 à 65535 inclus
1	0	0	1	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	0	1	Négatif Overflow, de -1 à -255 inclus	de -32768 à -65535 inclus
0	0	1	1	Nulle = 0	Nulle = 0
0	0	1	0	Nulle = 0 (addition de 2 zéros)	Nulle = 0 (addition de 2 zéros)
0	0	1	1	=-256	=-65536

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* 6502	65816	*nb d' Octets*	*nb de Cycles*
Immédiat	**SBC** *#const*	E9	x	x	2	2[1]
Absolue	**SBC** *addr*	ED	x	x	3	4[1]
Absolue Long	**SBC** *long*	EF		x	4	5[1]
Direct Page(DP)	**SBC** *dp*	E5	x	x	2	3[1,3]
DP indirect	**SBC** *(dp)*	F2		x	2	5[1,3]
DP indirect Long	**SBC** *[dp]*	E7		x	2	6[1,3]
Absolue indexé par X	**SBC** *addr,X*	FD	x	x	3	4[1,4]
Absolue Long indexé par X	**SBC** *long,X*	FF		x	4	5[1]
Absolue indexé par Y	**SBC** *addr,Y*	F9	x	x	3	4[1,4]
DP indexé par X	**SBC** *dp,X*	F5	x	x	2	4[1,3]
DP indirectement indexé par X	**SBC** *(dp,X)*	E1	x	x	2	6[1,3]
DP indirectement indexé par Y	**SBC** *(dp),Y*	F1	x	x	2	5[1,3,4]
DP long indirectement indexé par Y	**SBC** *[dp],Y*	F7		x	2	6[1,3]
Stack relative (SR)	**SBC** *sr,S*	E3		x	2	4[1]
SR indirectement indexé indexé par Y	**SBC** *(sr,S)*,Y	F3		x	2	7[1]

SBC, une instruction de groupe primaire, a disponible tous les modes d'adressage du groupe primaires et les modèles binaires

* - Ajoutez 1 octet si m=0 (mémoire et accumulateur à 16bits)

1- Ajoutez 1 cycle si m=0 (mémoire et accumulateur à 16bits)

3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)

4- Ajoutez 1 cycle si l'ajout d'index dépasse la page courante (si $00:0000 à $01:0000 par ex)

SEC - Set the Carry Flag

6502/65816

Met le bit de report(carry) à un !

La carry est à 1, cela se fait généralement avant une instruction SBC. Indicateur affectés - - - - - - - c

L'indicateur "c" est égal à un après l'opération.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**SEC**	38	x	x	1	2

SED - Set Decimal Mode Flag

6502/65816

Positionne le bit décimal à 1.

L'indicateur décimale est mis à un, en spécifiant le mode décimal pour ADC et SBC.
Indicateur affectés - - - - d - - -

On travaillera en décimale après l'instruction.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**SED**	F8	x	x	1	2

SEI - Set Interrupt Disable Flag

6502/65816

Mise à un du bit d'interruption.

L'indicateur d'interruption "i" du Registre d'État (P) est mis à un. Les interruptions sont innutilisables. Cela est utilisé pendant les sous-programmes de gestion d'interruption.

Indicateur affectés - - - - - i - -

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**SEI**	78	x	x	1	2

SEP - Set Status Bits

/65816

Positionne les registres d'état (P).

L'opération logique OU(OR) agit sur le registre P et sur l'octet suivant immédiatement le code opération, et le résultat est stocké dans le registre P.

Par exemple si on veut mettre les bits 1 et 4 à un, on applique la valeur $12 (00010010b) à SEP.

```
1 SEP #12
```

Les autres bits du registre P ne seront pas touchés par le changement. Si vous voulez mettre à "0" certains bit du registre, voir l'instruction "REP - Reset Status Bits" à la page 202

Indicateur Affectés	**n**	v	-	-	d	i	z	c	Emulation(e=1)
	n	v	m	x	d	i	z	c	Native(e=0)

Les bits 5 et 6, en mode émulation, ne peuvent être touchés.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Immédiat	**SEP** *#const*	E2		x	2	3

STA - Sore Accumulator to Memory

6502/65816

Stocke l'accumulateur en mémoire.

```
1 LDA #$20
2 STA $2100
```

On charge la valeur $20 dans l'Accumulateur, et ensuite on le stock à l'adresse $2100.
Cette adresse vaudra $20 par la suite.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Absolue	**STA** *addr*	8D	x	x	3	4[1]
Absolue Long	**STA** *long*	8F		x	4	5[1]
Direct Page(DP)	**STA** *dp*	85	x	x	2	3[1,3]
DP indirect	**STA** *(dp)*	92		x	2	5[1,3]
DP indirect Long	**STA** *[dp]*	87		x	2	6[1,3]
Absolue indexé par X	**STA** *addr,X*	9D	x	x	3	5[1]
Absolue Long indexé par X	**STA** *long,X*	9F		x	4	5[1]
Absolue indexé par Y	**STA** *addr, Y*	99	x	x	3	5[1]
DP indexé par X	**STA** *dp,X*	95	x	x	2	4[1,3]
DP indirectement indexé par X	**STA** *(dp,X)*	81	x	x	2	6[1,3]
DP indirectement indexé par Y	**STA** *(dp), Y*	91	x	x	2	6[1,3,4]
DP long indirectement indexé par Y	**STA** *[dp], Y*	97		x	2	6[1,3]
Stack relative (SR)	**STA** *sr,S*	83		x	2	4[1]
SR indirectement indexé indexé par Y	**STA** *(sr,S)*,Y	93		x	2	7[1]

STA, une instruction de groupe primaire, a disponible tous les modes d'adressage du groupe primaires et les modèles binaires
1- Ajoutez 1 cycle si m=0 (mémoire et accumulateur à 16bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)

STP - Stop the Processor /65816

Arrêt de l'horloge

C'est le mode "sleep" du processeur. On le met en veille, si vous préférez, jusqu'à ce qu'une interruption vienne le réveiller.

Par contre, il vaut tenir compte des changements lors de son réveil. Vous ne reviendrez à l'endroit où vous vous serez arrêté.

Pour le moment je ne pense pas que l'on en est à mettre en veille le processeur. (par souci d'écologie éteignez tout c'est toujours mieux).

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**STP**	DB		x	1	3[10]

10- Utilise 3 cycles pour éteindre le processeur, des cylces additionnels sont requis pour le redémarrer par reset

STX - Store Index Register X to Memory

6502/65816

Stock le registre d'index X en mémoire.

```
1 LDX #$20
2 STX $2100
```

On charge la valeur $20 dans l'index X, et ensuite on le stock à l'adresse $2100.
Cette adresse vaudra $20 par la suite.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Absolue	**STX** *addr*	8E	x	x	3	4^{8}
Direct Page(DP)	**STX** *dp*	86	x	x	2	$3^{8,3}$
DP indexé par Y	**STX** *dp,Y*	96	x	x	2	$4^{8,3}$

8- Ajouter 1 cycle si x =0 (registre dindex à 16 bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)

STY - Store Index Register Y to Memory

6502/65816

Stock le registre d'index Y en mémoire.

```
1 LDY #$20
2 STY $2100
```

On charge la valeur $20 dans l'index Y, et ensuite on le stock à l'adresse $2100.
Cette adresse vaudra $20 par la suite.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Absolue	**STY** *addr*	8C	x	x	3	4^{8}
Direct Page(DP)	**STY** *dp*	84	x	x	2	$3^{8,3}$
DP indexé par Y	**STY** *dp,Y*	94	x	x	2	$4^{8,3}$

8- Ajouter 1 cycle si x =0 (registre dindex à 16 bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)

STZ - Store Zero to Memory

6502/65816

Stock zéro en mémoire.

Si l'on a une adresse à mettre à zéro, cette instruction sera idéale.
Au lieu de faire

```
1 LDA #$00
2 STA $4200
```

Et de prendre 6 cycles d'horloge (dans le cas le plus rapide), ici nous n'en prendrons que 4 cycles.
On a juste :

```
1 STZ $4200
```

On gagne du temps, et aussi l'utilisation inutile de l'accumulateur (ou des index X et Y).

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Absolue	**STZ** *addr*	9C	x	x	3	4[1]
Direct Page(DP)	**STZ** *dp*	64	x	x	2	3[1,3]
Absolue indexé par X	**STZ** *addr,X*	9E	x	x	3	5[1]
DP indexé par X	**STZ** *dp,X*	74	x	x	2	4[2,3]

1- Ajoutez 1 cycle si m=0 (mémoire et accumulateur à 16bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)

TAX - Transfer Accumulator to Index Register X

6502/65816

Transférer le contenue de l'Accumulateur dans le registre d'index X.

Le contenue de l'accumulateur n'est pas affecté.

Attention à une chose, votre initialisation des tailles, celle de la mémoire/accumulateur (bit m du registre P) et celle du registre des index (bit x du registre P).

Petit rappel: L'accumulateur est appelé A, en mode 8bits, et C en 16 bits. En 16 bits, l'accumulateur C est représenté par A, qui prend l'octet le plus bas, et B, qui prend l'octet le plus haut.

Exemple :

A = $1234 quand on est en 16 bits (m=0).[1]
Maintenant on passe en 8bits (m=1), A = $34, et B=$12.

m	x	Accumulateur	Index X	Résultat
0	0	16bits	16bits	C($1234) => X($1234)
0	1	16bits	8bits	C($1234) => X($34)
1	0	8bits	16bits	B($12) A($34) => X($1234)
1	1	8bits	8bits	A($34) => X($34)

Indicateur affectés n - - - - - z -

n - Si le bit 7[2] est à un. En gros si on a un nombre négative dans l'index X après transfert.
z - Si le nombre, transféré dans l'index X, est égale à zéro.

Table 11.24 – Valeurs possible avec TAX

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur*		*nb d'*	*nb de*
			6502	*65816*	*Octets*	*Cycles*
Implicite	**TAX**	AA	x	x	1	2

[1] A en mode 16 bits est appelé "C", ici , c'est pour dire "Accumulateur"

[2] en 8 bits, sinon bit 15 en 16 bits

TAY - Transfer Accumulator to Index Register Y

6502/65816

Transférer le contenue de l'Accumulateur dans le registre d'index Y.

Le contenue de l'accumulateur n'est pas affecté.

Attention à une chose, votre initialisation des tailles, celle de la mémoire/accumulateur (bit m du registre P) et celle du registre des index (bit x du registre P).

Petit rappel: L'accumulateur est appelé A, en mode 8bits, et C en 16 bits. En 16 bits, l'accumulateur C est représenté par A, qui prend l'octet le plus bas, et B, qui prend l'octet le plus haut.

Exemple :

A = $1234 quand on est en 16 bits (m=0).[1]
Maintenant on passe en 8bits (m=1), A = $34, et B=$12.

m	x	Accumulateur	Index X	Résultat
0	0	16bits	16bits	C($1234) => Y($1234)
0	1	16bits	8bits	C($1234) => Y($34)
1	0	8bits	16bits	B($12) A($34) => Y($1234)
1	1	8bits	8bits	A($34) => Y($34)

Indicateur affectés n - - - - - z -

n - Si le bit 7[2] est à un. En gros si on a un nombre négative dans l'index Y après transfert.
z - Si le nombre, transféré dans l'index Y, est égale à zéro.

Table 11.25 – Valeurs possible avec TAY

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur*		*nb d'*	*nb de*
			6502	*65816*	*Octets*	*Cycles*
Implicite	**TAY**	A8	x	x	1	2

[1] A en mode 16 bits est appelé "C", ici , c'est pour dire "Accumulateur"

[2] en 8 bits, sinon bit 15 en 16 bits

TCD - Transfer 16-Bit Accumulator to Direct Page Register

/65816

On transfert l'Accumulateur C de 16bibts dans le Registre de Page Zéro (D). Le contenue C n'est pas changé.

Petit rappel: L'accumulateur est appelé A, en mode 8bits, et C en 16 bits. En 16 bits, l'accumulateur C est représenté par A, qui prend l'octet le plus bas, et B, qui prend l'octet le plus haut.

Vous vous dites, "on aurait pu appeler l'instruction TAD, pour "transfer A to Direct page"", voilà la raison du C. [1].

Bref le Registre de Page Zéro (D) est celui qui dit quelle instruction est à exécuter. Là, vous donner directement l'ordre au processeur d'aller à cet emplacement mémoire pour exécuter une instruction, sans rien sauvegarder.

Indicateur affectés n - - - - - z -

n - Si le bit 15[2] est à un. En gros si on a un nombre négative dans l'index Y après transfert.
z - Si le nombre, transféré dans l'index Y, est égale à zéro.

Table 11.26 – Valeurs possible avec TCD

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de $01 à $7F inclus	de $0001 à $7FFF inclus
1	0	Négatif, de $80[3]à $FF[4]inclus	de $8000[5]à $FFFF[6]inclus
0	1	Nulle =$00	Nulle = $00

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**TCD** ou **TAD**	5B		x	1	2

[1]Je crois que vous pouvez faire un TAD si vous voulez faire cette opération en 8 bits
[2]en 16bits, sinon bit 7 en 8bits
[0]-127
[0]-1
[0]-32768
[0]-1

TCS - Transfer Accumulator to Stack Pointer /65816

On transfert la valeur de l'accumulateur C (16bits) dans le pointeur de pile.

Sa vous permet de dire à la pile "je veux pointer sur cette adresse dans la pile", pour la changer ou la charger...

Alternativement vous pouvez appelez TAS, qui est un transfert de l'accumulateur A sur 8 bits.

Le contenue de l'accumulateur C n'est pas changé. Cett instruction modifie l'adresse de la pile dans la mémoire.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**TCS** ou **TAS**	1B		x	1	2

TDC - Transfer Direct Page Register to 16-Bit Accumulator

/65816

On tranfert le Registre de Page Zéro (D) dans l'Accumulateur C (donc 16bits).

Si l'on veut que 8 bits, on fait un TDA (attention de bien initialiser l'indicateur m).

Le contenue du registre D n'est pa changé.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite & ou **TDA**	**TDC**	7B		x	1	2

TRB - Test and Reset Memory Bits Against Accumulator

6502/65816

Tester et remet à zéro les bits

On fait un ET logique entre, la valeur qui se trouve dans l'Accumulateur, et l'opérande en mémoire.

Le résultat est stocké dans l'opérande en mémoire. Tout bit positionné à 1 dans A sera mis à 0 dans l'opérande en mémoire.

On compare si deux nombres sont égaux ou non. L'indicateur "z" nous servira de témoin.
Si l'on touche à l'indicateur "m", on pourra tester des nombres de 8 ou 16bits.

A = $15
$4200 = $56

```
1 TRB $4200
```

```
1     $15 (0001 0101)
2 AND $56 (0101 0110)
3     $14 (0001 0100)
```

Le résultat n'est pas égal à zéro, donc les deux valeurs ne sont pas égales.

Indicateur affectés - - - - - - z -

z - C'est avec cet indicateur que l'on peut savoir si les deux valeurs sont égales ou non. Si le résultat est égal à zéro, alors cet indicateur est à un.

Table 11.27 – Valeurs possible avec TRB

z	Résultat
0	Mémoire AND l'Accumulateur = KO
1	Mémoire AND l'Accumulateur = OK

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* 6502	65816	*nb d' Octets*	*nb de Cycles*
Absolue	**TRB** *addr*	1C		x	3	6^{2}
Direct Page(DP)	**TRB** *dp*	14		x	2	$5^{2,3}$

2- Ajoutez 2 cycle si m=0 (mémoire et accumulateur à 16bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)

TSB - Test and Set Memory Bits Against Accumulator

6502/65816

Teste et positionne les bits à 1.

On fait un OU logique entre, la valeur qui se trouve dans l'Accumulateur, et l'opérande en mémoire.

Le résultat est stocké dans l'opérande en mémoire. Tout bit positionné à 1 dans A sera mis à 1 dans l'opérande en mémoire.

On compare si deux nombres sont égaux ou non. L'indicateur "z" nous servira de témoin.
Si l'on touche à l'indicateur "m", on pourra tester des nombres de 8 ou 16bits.

A = $15
$4200 = $56

```
1 TSB $4200
```

```
1     $15 (0001 0101)
2 OR underline{$56 (0101 0110)}
3     $57 (0101 0111)
```

Le résultat n'est pas égal à zéro, donc les deux valeurs ne sont pas égales.

Indicateur affectés - - - - - - z -

z - C'est avec cet indicateur que l'on peut savoir si les deux valeurs sont égales ou non. Si le résultat est égal à zéro, alors cet indicateur est à un.

TABLE 11.28 – Valeurs possible avec TSB

z	Résultat
0	Mémoire OR l'Accumulateur = KO
1	Mémoire OR l'Accumulateur = OK

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Absolue	**TSB** *addr*	0C		x	3	6^{2}
Direct Page(DP)	**TSB** *dp*	04		x	2	$5^{2,3}$

2- Ajoutez 2 cycle si m=0 (mémoire et accumulateur à 16bits)
3- Ajoutez 1 cycle si l'octet bas du Registre de Page Zéro (D) est autre que 0 (DL <>0)

TSC - Transfer Stack Pointer to 16-Bit Accumulator

6502/65816

On transfert le pointeur de pile dans l'Accumulateur C (donc 16bits).
Le contenu de la pile et du pointeur de pile ne sont pas affecté.
Si l'on veut que 8 bits, on fait un TSA (attention de bien initialiser l'indicateur m).

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**TSC**	3B		x	1	2
Implicite	**TSA**			x	1	

TSX - Transfert Stack Pointer to Index Register X 6502/65816

On transfert le pointeur de pile dans le registre d'index X.

Le contenu de la pile et du pointeur de pile ne sont pas affecté.

Si vous voulez transférer les 16 bit, mettez l'indicateur x à 0, sinon vous transférez 8 bits (le bits le plus bas) avec x = 1.

Indicateur affectés n - - - - - z -

n - Si le bit 15 [1] est à un. En gros si on a un nombre négative dans le Registre de Page Zéro (D) après transfert.

z - Si le nombre, transféré dans le registre D, est égale à zéro.

Table 11.29 – Valeurs possible avec TSX

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de $01 à $7F inclus	de $0001 à $7FFF inclus
1	0	Négatif, de $80[2]à $FF[3]inclus	de $8000[4]à $FFFF[5]inclus
0	1	Nulle =$00	Nulle = $00

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur*		*nb d'*	*nb de*
			6502	*65816*	*Octets*	*Cycles*
Implicite	**TSX**	BA	x	x	1	2

[1] en 16 bits, sinon bit 7 en 8 bits

[0] -127

[0] -1

[0] -32768

[0] -1

TXA - Transfer Index Register X to Accumulator

6502/65816

Transférer le registre d'index X dans l'Accumulateur A.

Le contenue de X n'est pas affecté.
Attention à une chose, votre initialisation des tailles, celle de la mémoire/accumulateur (bit m du registre P) et celle du registre des index (bit x du registre P).

Petit rappel: L'accumulateur est appelé A, en mode 8bits, et C en 16 bits. En 16 bits, l'accumulateur C est représenté par A, qui prend l'octet le plus bas, et B, qui prend l'octet le plus haut.

Exemple :

```
16bits          8bits
LDX $1250       LDX $1250
TXA             TXA
C=$1250         B =$12 A=$50
```

m	x	Index X	Accumulateur	Résultat
0	0	16bits	16bits	X($1234) => C($1234)
0	1	16bits	8bits	X($34) => A($34)
1	0	8bits	16bits	X($1234 => B($12) A($34)
1	1	8bits	8bits	X($34) => A($34)

Indicateur affectés n - - - - - z -

n - Si le bit 7[1] est à un. En gros si on a un nombre négative dans l'index Y après transfert.
z - Si le nombre, transféré dans l'index Y, est égale à zéro.

Table 11.30 – Valeurs possible avec TXA

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur*		*nb d'*	*nb de*
			6502	*65816*	*Octets*	*Cycles*
Implicite	**TXA**	8A	x	x	1	2

[1] en 8 bits, sinon bit 15 en 16 bits

TXS - Transfer Index Register to Stack Pointer

6502/65816

Transférer le registre d'index X dans le pointer de pile.

Le contenue de X n'est pas changé. Cette instruction modifie l'adresse de la pile en mémoire. Après cette opération, notre pointer pile pointera à l'adresse que l'index X avait.

L'indicateur e=1 : en mode émulation, votre transfert sera sur 8 bits.
L'indicateur x=1 : l'index X est à 8 bits, vous n'aurez que l'octet le plus bas du pointer de pile
L'indicateur x=0 : l'index X est à 16bits, la valeur complète sera transférer.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur*		*nb d' Octets*	*nb de Cycles*
			6502	*65816*		
Implicite	**TXS**	9A	x	x	1	2

TXY - Transfer Index Register X to Y

/65816

Transférer le registre d'index X dans le registre d'index Y.

Le contenue de X n'est pas changé.
Attention à une chose, votre initialisation des tailles, celle du registre des index (bit x du registre P).

Exemple :

```
16bits          8bits
LDX $1250       LDX $50
TXY             TXY
Y=$1250         Y=$50
```

x	Index	Résultat
0	16bits	X($1234) => Y($1234)
1	8bits	X($34) => Y($34)

Indicateur affectés n - - - - - z -

n - Si le bit 7[1] est à un. En gros si on a un nombre négative dans l'index Y après transfert.
z - Si le nombre, transféré dans l'index Y, est égale à zéro.

TABLE 11.31 – Valeurs possible avec TXY

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur*		*nb d' Octets*	*nb de Cycles*
			6502	*65816*		
Implicite	**TXY**	9B		x	1	2

[1] en 8 bits, sinon bit 15 en 16 bits

TYA - Transfer Index Register Y to Accumulator

6502/65816

Transférer le registre d'index Y dans l'Accumulateur A.

Le contenue de Y n'est pas affecté.
Attention à une chose, votre initialisation des tailles, celle de la mémoire/accumulateur (bit m du registre P) et celle du registre des index (bit x du registre P).

Petit rappel: L'accumulateur est appelé A, en mode 8bits, et C en 16 bits. En 16 bits, l'accumulateur C est représenté par A, qui prend l'octet le plus bas, et B, qui prend l'octet le plus haut.

Exemple :

```
16bits          8bits
LDY $1250       LDY $1250
TXA             TXA
C=$1250         B =$12 A=$50
```

m	x	Index Y	Accumulateur	Résultat
0	0	16bits	16bits	Y($1234) => C($1234)
0	1	16bits	8bits	Y($34) => A($34)
1	0	8bits	16bits	Y($1234 => B($12) A($34)
1	1	8bits	8bits	Y($34) => A($34)

Indicateur affectés n - - - - - z -

n - Si le bit 7[1] est à un. En gros si on a un nombre négative dans l'index Y après transfert.
z - Si le nombre, transféré dans l'index Y, est égale à zéro.

Table 11.32 – Valeurs possible avec TYA

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**TYA**	98	x	x	1	2

[1] en 8 bits, sinon bit 15 en 16 bits

TYX - Transfer Index Register Y to X

/65816

Transférer le registre d'index Y dans le registre d'index X.

Le contenue de Y n'est pas changé.
Attention à une chose, votre initialisation des tailles, celle du registre des index (bit x du registre P).

Exemple :

```
16bits          8bits
LDY $1250       LDY $50
TYX             TYX
X=$1250         X=$50
```

x	Index	Résultat
0	16bits	Y($1234) => X($1234)
1	8bits	Y($34) => X($34)

Indicateur affectés n - - - - - z -

n - Si le bit 7[1] est à un. En gros si on a un nombre négative dans l'index Y après transfert.
z - Si le nombre, transféré dans l'index Y, est égale à zéro.

TABLE 11.33 – Valeurs possible avec TYX

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur*		*nb d' Octets*	*nb de Cycles*
			6502	*65816*		
Implicite	**TYX**	BB		x	1	2

[1] en 8 bits, sinon bit 15 en 16 bits

WAI - Wait for Interrupt /65816

Attente d'une interruption.

Ici on met le processeur en mode "Sleep" attendant qu'une interruption vienne le réveiller. Cette caractéristique peut être utilisé pour réduire le temps de latence en plaçant l'instruction WAI au début du programme d'interruption et en mettant à 1 le bit d'interruption (désactivation). Lorsqu'une interruption apparaît, l'instruction suivant WAI sera exécuté.

Quand celle-ci arrive, on reprend de là où l'on c'est arrêté.

Indicateur affectés - - - - - - - -

Aucun

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**WAI**	CB		x	1	3[11]

11- Utilise 3 cycles pour éteindre le processeur, des cylces additionnels sont requis pour le redémarrer par interruption

WDM - Reserved for Future Expansion

/65816

Réservé pour de future utilisation, NE JAMAIS l'UTILISER.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
	WDM	42		x	2^x	x

x - Bytes and cycle counts subject to change in future processors which expand WDM into 2-byte opcode portions of instructions of varying lengths.

XBA - Exchange the B and A Accumulateurs /65816

Permute les accumulateurs B et A et vice-versa.

Les accumulateurs A et B sont permutés, le bit M du Registre d'État (P) doit être à 1, l'instruction XBA est employé lorsque les données de 8 bits sont utilisées et qu'un registre supplémentaire, B, est nécessaire.

ACCUMULATEUR = $4568	
16bits	8bits
C=$4568	B=$45 A=$68

Voilà à quoi correspond notre Accumulateur, C représente 16bits, B représente les 8bits les plus haut,et A les 8 bits les plus bas.

Le A sera à la place du B et B à la place du A.

Il faudra être en mode 8 bits pour appeler cette instruction, sinon il faudra appeler l'instruction SWA,pour le mode 16bits.

Indicateur affectés n - - - - - z -

n=1 Si le nouveau A est négatif, sinon n=0.
z=1 Si le nouveau A est égal zéro, sinon z=0.

TABLE 11.34 – Valeurs possible avec XBA

n	**z**	**Résultat**	
		8bits	**16bits**
0	0	Positif, de 1 à 127 inclus	de 1 à 32767 inclus
1	0	Négatif, de -1 à -128 inclus	de -1 à -32768 inclus
0	1	Nulle = 0	Nulle = 0

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**XBA**	EB		x	1	3
Implicite	**SWA**			x	1	

XCE - Exchange Carry and Emulation Bits /65816

Permutte les bits Carry et Emulation.

Les bits ”c” et ”e” dans le Registre d'État (P) sont permutés. Cette instruction est utilisé pour changer le fonctionnement du 65816, mode natif en mode émulation du 6502.

On met notre carry à un ou à zéro, et on appel cette instruction pour passer du mode Native à l'émulation ou inversement.

Indicateur affectés n - - - - - z -

e - prend la valeur de la carry
c - prend la valeur du bit ”e”
m - existe qu'en mode native, il disparait si on passe en mode Emulation.
x - est remplacé par l'indicateur ”b” en mode Emulation.
b - est remplacé par l'indicateur ”x” en mode Native.

Mode d'Adressage	*Syntax*	*Opcode* (Hex)	*Disponnible sur* *6502*	*65816*	*nb d' Octets*	*nb de Cycles*
Implicite	**XCE**	FB		x	1	2

Chapitre 12

Trucs et Astuces

12.1 Les divisions avec LSR

voir l'instruction "LSR - Logical Shift Memory or Accumulator Right" à la page 178 pour vous en souvenir.

Pour faire une division signé.
Exemple : -1 divisé par 2 donne 127 au lieu de -0,5.
On applique l'opération 127 - 128 = -1, et si le carry = 1 alors -1 + 0,5 = -0,5.

-64 divisé par 2 donne 96 au lieu de -32
96 - 128 = -32 ici le carry vaut zéro, donc -32

Sa prend plus de temps mais sa fonctionne !

```
1 LDA #$9C  ;(-100)
2 LSR       ;(-100/2 = 78???)
3 EOR #$80  ;(-128 + 78 = 50 ahhh)
```

Accumulateur = $CE (50)

Vous avez le résultat en négatif (après si vous voulez gérer la virgule, je vous laisse le plaisir de coder). Faite le en binaire pour vous en rendre compte.

Il y a plus simple... vous utilisez l'instruction ROR, qui le fait "pratiquement" automatiquement.

12.2 Les divisions avec ROR

voir l'instruction "ROR - Rotate Memory or Accumulator Right" à la page 205 pour vous en souvenir. C'est pratiquement la même chose que la division avec LSR, mais à une différence, vous savez si votre nombre est négatif à la fin ! Votre indicateur "n" sera à un à la fin de cette instruction.

Exemple :

Initialiser le Carry à 1.
-1 divisé par 2 donne -1 au lieu de -0,5.

Initialiser le Carry à 0.
-1 divisé par 2 donne 127 au lieu de -0,5.

Voilà la différence, si la carry est correctement initialisée, vous aurez votre résultat juste.
Ensuite il suffit de faire : Si carry = 1 [1] votre devez appliquer +0.5 à votre résulat.

Voici un exemple de ce qui pourrais se passer :

n	z	c		Nombres en bits	Valeur décimale
		0		1110 0000	-16
			ROL		(-16/2)
0	0	0	Résultat	0111 0000	8

Les différences avec la multiplication LSR :

- On peut ajouter "+0.5" dans le résultat sans faire d'autres opérations.
- Et sa prend autant de temps qu'une instruction LSR.
- ROR peut diviser les nombres négatifs.
- Mais il faut initialiser la carry avec ROR pour savoir si on divise un nombre négatif ou positif.

[1] Après l'instruction, bien-sûr

12.3 Les Multiplications avec ASL

voir l'instruction "ASL - Shift memory or Accumulator Left" à la page 136 pour vous en souvenir.

Nous voilà à un exemple "bête et méchant". On ne peut pas multiplier quand le résultat va de -256 à 255.[1]

interprétation du résultat : Votre nombre est maintenant "INTERPRÈTE" sur 9bits :

1 0000 0000

Le carry (le bit à 1) fait office de 9ème bit. Ca évite de passer en mode 16 bits quand notre résultat doit être compris entre -256 et +254.

Pour un résultat **Positif Overflow**

n	z	c		Nombres en bits	Valeur décimale
1		0		0100 0001	65
			ASL		65*2
0	0	1	Résultat	1000 0010	130

Pour un résultat **Négatif Overflow**

n	z	c		Nombres en bits	Valeur décimale
1		0		1011 1111	-65
			ASL		-65*2
0	0	1	Résultat	0111 1110	-130

Pour un résultat **Nulle Overflow**

n	z	c		Nombres en bits	Valeur décimale
1		0		1000 0000	-128
			ASL		-128*2
0	1	1	Résultat	0000 0000	-256

65*2 = 130 normalement, et ici -126.
-65*2 = -130 normalement, et ici 126.
Mais la carry nous sert de "8ème bit"[2], et nous permet de l'interpréter comme "signe"!
Si la Carry=0 et Négative=0 on est positif.
Ce qui nous ferais des nombres allant de 254 à -256 ($0FE à $100).
Faites des exemples à la main pour vous en convaincre.

Cette astuce est utilisée si l'on ne veut pas s'occuper des flottants.

[1] -256 à 255 en 8bits et -32768 à 32767 en 16bits

[2] dans le mode 8bits, sinon c'est "16ème bit" en mode 16bits

12.4 Les Multiplications avec ROL

voir l'instruction "ROL - Rotate Memory or Accumulator Left" à la page 203 pour vous en souvenir.

On peut utiliser la carry comme un indicateur de signe, en complément de l'indicateur Négative. ATTENTION le carry doit être initialisée avant l'instruction, elle ajoutera au nombre "+0.5" si il est à un (c=1) ou "+0" si il est à 0 (c=0).

Ceci est utilisé pour "interpréter" le résultat d'une opération qui dépasserait les limites d'un nombre signé : -128 à 127 pour 8 bits, et -32768 à 32767 pour 16 bits.

Et avec le Carry on pourra étendre de : -256 à 255 en 8bits, et -65536 à 65535 en 16 bits.

C'est exactement la même chose que la section "Les Multiplications avec ASL"[1], sauf que l'on fait "+1" ici quand le Carry est à un avant l'instruction.

Pour un résultat **Positif Overflow**

n	z	c		Nombres en bits	Valeur décimale
		1		0111 1111	127.5
			ROL		(127*2)+1
1	0	0	Résultat	1111 1111	255

Pour un résultat **Nulle Overflow**

n	z	c		Nombres en bits	Valeur décimale
		1		1000 0000	-127.5
			ROL		(-128*2)+1
0	1	1	Résultat	0000 0000	-256

Pour un résultat **Negatif Overflow**

n	z	c		Nombres en bits	Valeur décimale
		0		1001 1100	-100
			ROL		(100*2)+0
1	0	1	Résultat	0011 1000	-200)

Les différences avec la multiplication ASL :

- On peut ajouter "+1" dans le résultat sans faire d'autres opérations.
- Et sa prend autant de temps qu'une instruction ASL.
- On peut aller jusqu'au nombre 255 et 127[2], contre 254 et 126[3] avec ASL).
- ASL n'a que des résultats pair, ROL peut faire les pair et impair.(
- Il faut initialiser à la carry avec ROL.

[1] voir la section 12.2 "Les divisions avec ROR" à la page 242

[2] en 8bits et 65534 et 32767 en 16bits

[3] en 8bits et 65536 et 32766 en 16bits

Cinquième partie

LA SUPER NINTENDO

Chapitre 13

Les Notions

13.1 Objet

Un objet est un élement constitué d'un ou de plusieurs tiles, dont la taille varie de 8x8 jusqu'à 64x64. Il peut être déplacé à différents endroit sur l'écran. Par exemple, des éléments comme Mario, ou un missile se déplacent sur l'écran. Cela se produit en effaçant leurs anciennes positions, et en les affichant à leur nouvelles, en donnant ainsi un effet d'animation.

Le nombre maximum d'objet, que peut supporter la console, est de 128. Je préciseque que l'objet est une frame.

2 tailles peuvent êtres sélectionnées dans une frame. 1 taille peut être sélectionné pour chaque Objets.

Il y a 8 palettes de couleur pour les Objets, dont une peut être choisis pour chacun d'eux. Une palette peut avoir 16 codes de couleurs sur 32768 différentes. Cependant, chaque Objet dans l'image est dessiné par 16 couleurs. Les Objets ont des priorités, Si deux d'entre eux se retrouvent au même endroit, le plus prioritaire est affiché.

Au final ils possèdent les fonctions de retournement entre haut et bas, droite et gauche Ordre de priorité du BG [1] des Ordre de priorité permutable.

Nombre d'Objet Affichable	128			
Taille de l'Objet	8x8	16x16	32x32	64x64
Nombre de ligne affichable	32 pièces (convertie en taille de 8x8			
Nombre de couleur par Objet	16			
Nombre de palettes	8			
Nombre couleur à l'écran	128			
Attribution	fonction H-FLIP, V-FLIP Priorité d'affichage (Sélectionner la priorité pour le Fond d'écran (BG)			

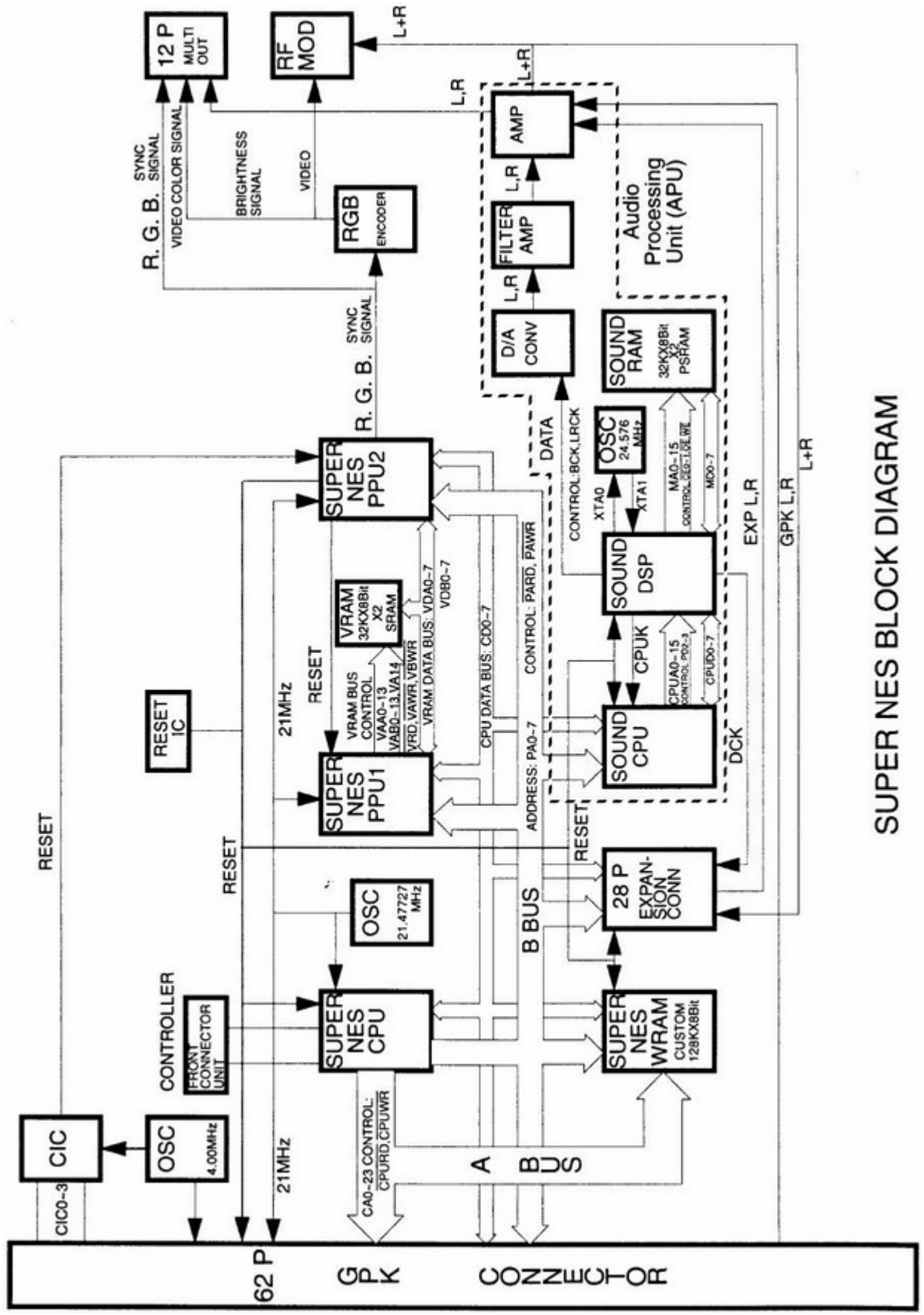
[1] BackGround(fond d'écran)

Chapitre 14

Les éléments principaux de la Super Nintendo

Ce chapitre vous fournie un fonctionnement générale de la console Super Nintendo. Référez vous au Diagramme, pour comprendre les paragraphes de ce chapitre.

FIGURE 14.1 – Diagramme des blocs des fonctions de la Super Nintendo



14.1 CPU

C'est l'unité de traitement Centrale(Central Processing Unit) pour la Super Nintendo. Il coordonne toutes les fonctions de la Super Nintendo et des périphériques, qui sont attaché à la console.

14.2 PPU1 et PPU2

Ces deux unités travaillent ensembles comme Unité de traitement d'image (Picture Processing Unit) sur la Super Nintendo. Les images sont générées, a partir de ces deux PPU, pour envoyer le travaille maché au CPU de la Super Nintendo.

En générale„ PPU1 est utilisé pour afficher les caractères de données du fond d'écran, rotation, et mise à l'échelle[1]. Tandis que le PPU2 traite les effets spéciaux, comme les fenêtres, mosaïque, et les fondus d'images [2].

14.3 WRAM

La ram de travail[3] est une RAM de 128k-octets personnalisée, utilisé par le CPU pour le stockage des données. Le DMA (Direct Memory Adressing, Adressage Memoire Direct) peut être utilisé par le CPU, pour un transfert rapide de données volumineuses.

14.4 VRAM

La Vram est composée de deux S-RAM de 32k-octets. Cette unité est utilisée par le PPU1, pour stocker les caractères de données du fond d'écran, tant qu'il en a besoin pour l'affichage.

14.5 APU(Audio processing Unit)

L'unitée de traitement Audio exécute toutes les fonctions du son pour la Super Nintendo, et est composé des unités suivante :

CPU audio

Le CPU du son, est l'unité de traitement central pour l'APU. Il controle les fonctions du son, beaucoup plus que de la même façon, que le CPU contrôle les fonctions de la Super Nintendo.

DSP audio

Le DSP a 8 cannaux de PCM (Pulse Code Modulated, Modulation par impulsions codées), un générateur de bruit, echo, balayage(sweep), enveloppe, et d'autres cicuits pour reproduire la qualité de la tonalité, depuis les donnée de la RAM.

RAM audio

La RAM audio est composée de 2 32k octets de SRAM. le programme et les données de tonalité, sont chargé depuis le pak du jeu, à la RAM audio, par le CPU Audio. Le CPU audio et le DSP se partagent la RAM.

Convertion Numérique/Analogique

Convertie le son digital en son analogique, qui est filtré et amplifié, pour produire une sortie ,MONO à travers le modulateur RF, et STEREO à travers le connecteur à sortie multiple.

[1] scaling
[2] fades
[3] work RAM

Chapitre 15

Mapping mémoire

Le mapping mémoire de la Super Nintendo est disponible à l'annexe Mapping Mémoire page 2 à 5. Ce dernier représentes 3 mode, le mode 20, 21 et 25.

Sur la l'annexe MM[1] page 1, vous avez l'emplacement des zones où se situent les registres, RAM, Mémoire, vitesse de la mémoire etc...

15.1 Horloge CPU

L'horloge CPU peut commuté automatiquement, cela dépend de l'adresse à laquelle on accède par le CPU. 3 vitesses d'horloges sont possibles : 3.58Mhz, 2.68Mhz et 1.79Mhz. La vitesse des périphériques (ROM, RAM, LSI, etc.) détermineront la vitesse qui sera utilisée.

Si une vitesse moyenne, de la ROM et RAM, (temps d'accès < à 200ns) est utilisé par la cartouche, elle correspondra avec les zones d'adresses pour 2.68Mhz.

Si une grande vitesse (temps d'accès : < 120ns) est utilisé, Il correspondra avec les zones d'adresse pour 3.58Mhz.

Référez vous à "Fréquence & Mapping d'Adresse " pour les correspondances entre les adresses et l'horloge.

Deux horloges (2.68Mz & 3.58Mhz) peuvent êtres sélectionnées par le paramètre $D0 du registre $420D pour la rangée mémoire "2" illustrée à la page . Le paramètre par défaut est "2.68Mhz". Le CPU a une horloge interne de 3.58Mhz.[2]

15.2 Mapping Mémoire du CPU

Référez vous à l'annexe MM[3] page 1. Les 8Ko de WRAM correspondent à l'adresse $0000 à $1FFF des banques $00 à $3F,$80 à $BF et $7E.

La WRAM est utilisée comme banque commune. Les 8Ko peuvent être accessibles depuis n'importe quel banque décrites ci-dessous.

Les 120ko de la WRAM correspondent à l'adresse $7E-2000 à $7E-FFFF et $7F-0000 à $7F-FFFF[4].

Cependant la WRAM (128Ko au total) est inclue dans l'unité de la Super Nintendo. Ses 128Ko (RAM 1 et RAM 2) sont de la mémoire consécutive et peuvent êtres accessibles à partir du Bus d'adresse[5].

Les adresses $2000 - $5000 des banque $00 - $3F et $80 - $BF sont réservées comme un registre S-PPU, DMA, etc... Parce-que ses registres essentielles sont réservés comme une banque commune. Les registres S-PPU et DMA peuvent êtres accessibles depuis n'importe quel Banque au-dessus.

15.3 les registres de bases

de $2000 à $5FFF des banques $00 à $3F et $80 à $BF, sont réservé au processeur de la Super Nintendo.

[1] Mapping Mémoire
[2] Indifféremment de l'adresse DMA, qui sera exécuté avec une horloge de 2.68Mhz
[3] Mapping Mémoire
[4] $BANK-ADRESSE
[5] B-Bus Adress

Ces travaillerons à 3.58Mhz, excepté la zone $4000 à $41FF, qui est pour le contrôleur, et sera à 1,79Mhz

Ensuite les extensions, les DSP, travaillerons dans la zone 6000 à 7FFF des banques $00 à $3F et $80 à $BF. Travaillant à 2.68Mhz.

l'espace des zones mémoires sont comprise entre $2000 à $7FFF, soit 24ko accessibles à partir des Banques définis. A confirmer cependant.

La Vram 8ko

Celle-ci est un peu éparpiller, mais comporte un avantage. On peut la lire à partir de nombreuses banques.
La zone est de 8ko $0000 et $1FFF.
Que l'on soit en banque $00 à $3F, $7E, $80 à $BF. On sera toujours sur les même 8ko de la Vram.
Celle-ci travaille en 2.68Mhz.

La Vram 120ko

Si on veux une Vram plus grosse, on aura celle qui se trouve à l'adresse $7E :2000 à $7E :FFFF, et $7F :0000 à $7F :FFFF, soit 120Ko cadencé à 2,68Mhz.

zones mémoire

C'est ici que l'on va intégrer nos ROM de jeux. Vous avez la première zone qui est cadencé à 2.68Mhz, $8000 à $FFFF aux banques $00 à $3F, et de $40 :0000 à $7D :FFFF.

Ensuite la seconde zone mémoire qui vas pouvoir swicher entre 2 vitesses, 2.68Mhz ou 3.58Mhz, qui se situe de $8000 à $FFFF aux banques $80 à BF, et de $C0 :0000 à $FF :FFFF.

Les connections physiques

Vous remarquerez en haut à gauche du dessins "A16" "A23" "A15" "A0", se sont les pins qui vont êtres mises à 1 ou à 0 pour accéder à une adresse.

De A15 à A0, vous aurez les adresses de $0000 à $FFFF. Quand vous ferez un accès mémoire, celles si vont prendre un état haut ou bas, de tel sorte à accéder à une certaines zone mémoire.

Pareil pour A23 à A16, ces pins vous commutés pour informer la Banque dans laquelle on travaille. de $00 à $FF.

Par exemple, on veux aller écrire quelques chose dans à l'adresse $7EADFE, en mode hexadécimale, sa ne vous dit rien, normale, il faut le traduire en bit pour que cela signifie quelque chose.

D'après le tableaux ci-dessous, la première ligne représente les Pin de notre cartouche, la seconde représente l'état (1 ou 0) de la pin. L'adresse est représentée à la 3^{e}ligne

Pin	A23	A22	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12
adr bin	0	1	1	1	1	1	1	0	1	0	1	0
adr hexa	7				E				A			

Pin	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00
adr bin	1	1	0	1	1	1	1	1	1	1	1	0
adr hexa	D				F				E			

Voilà la base du mapping mémoire de la Super Nintendo, différentes zones de travailles, la RAM à un endroit et la ROM dans un autre. Se sera sur ce tableaux qu'il faudra se baser pour savoir où l'on écrit.

15.4 les différents modes

Les modes les plus utilisés, sont à l'annexe Mapping Mémoire page 3 à 5.

Vous avez sûrement entendu parlé de "Low ROM" et "High ROM", pour information c'est une histoire de connections. En effet la "Low ROM" n'a pas la pin "A23" de connecté, et de se fait, ne peut accéder aux adresses strictement supérieur à $7F :FFFF.

Pour ceux qui ne comprennent pas, quand on enlève la pin "A23", nous avons donc une adresse codé sur 23 bits (A22 à A0). Quand on prend l'octet qui sert à changer de Banque, (pin A23 à A16), vous avez 8 bits, soit 256 possibilités ($2^8 = 256$) donc de $00 à $FF. Maintenant vous vous retrouvez avec 7 bits, les possibilités seront de 128 ($2^7 = 128$), allant de $00 à $7F.[1].

Mode 20 (Lorom) - 3,9Mo (31,5Mb) MAX

Annexe MM page 3.

Le programme se trouve entre $8000 et $FFFF aux banques $00 à $7D, soit :

$[(FFFF - 8000) + 1] * (7D + 1) = 4\text{Moctets} = 32\text{Mbits MAX}$ [2]

Bien maintenant que vous savez que le Mode 20 sert pour se connecté avec une "Low Rom" alias LoROM, vous allez êtres surpris quand je vous dits que vous pouvez comme même utilisez les zones supérieurs à $7F :FFFF.

Par exemple si vous écrivez à l'adresse $C0 :ABDF, vous irez écrire à $40 :ABDF. De plus prés, $C0 = %1100 0000, sachant que l'A23 n'est pas connecté, et sera donc considérée comme un 0, vous aurez le chiffre %0100 0000 = $40.

D'après le document, on a une extension de RAM possible à l'adresse $0000 à $8000 des Banques $70 à $7D, soit 32Ko (256 Kbits) MAX. Ne vous méprenez pas en vous disant "tient mais de $70 :0000 à $7D :FFFF on a 448ko non ?", eh bien non. A ce que j'ai pus comprendre, que l'on soit en banque $70 ou $7B, on écrit dans la même mémoire. A confirmer.

Mode 21 (Hirom)- 4Mo (32Mb) MAX

Annexe MM page 4.

Le programme se trouve en mémoire haute, de l'adresse $C0 :0000 à $FF :FFFF, soit 4Moctets (32Mbits) MAX.

Vous pouvez donc travaillez soit en 2.68Mhz ou en 3.58Mhz, ce qui ne l'est pas pour la LoROM.

Le programme qui se trouve entre $8000 à $FFFF aux Banques $C0 à $FF sont aussi accessibles aux Banques $00 à $3F et $80 à $BF.

Faites attention de ne rien écrire dans les Banques $40 à $7D, il n'y a strictement rien.

Apparemment, vous disposez d'une extension mémoire de 128ko (1Mbits), localisé en banque $30 à $3F aux adresses $6000 à $7FFF, ici, le compte est bon, il n'y a qu'un emplacement pour écrire dans la RAM étendu.

Mode 25 (Hirom)- 7,9Mo (63Mb) MAX

Annexe MM page 5.

Ce genre de mapping mémoire n'est présent que sur une minorité de jeux.

Le programme est séparé en 2 zone mémoire. $40 :0000 à $7D :FFFF, et de $C0 :0000 à $FF :FFFF.

Respectivement, la première zone n'est accessible qu'en 2,68Mhz, l'autre peut commuter avec la vitesse 3,68Mhz.

Les zones mémoires $8000 à $FFFF des Banques $00 à $3D est une images des Banques $40 à $7D.

Pareil pour les Banques $C0 à $FF sur les Banques $80 à $BF. Les zones comprises entre $0000 et $7FFF n'ont pas d'images.

Le programme commence dans la zone haute ($C0 :0000 à $FF :FFFF) pour finir en zone basse ($40 :0000 à $7D :FFFF).

[1] Je rappel que le $00 est une possibilitée

[2] les +1 sont mis pour prendre en compte la possibilité "0" Je rappel, 1 et 0 donnent 2 possibilités

Chapitre 16

Les informations de la Cartouche

Ces informations sont stockées sur 48 octets aux adresses :
En mode 20 : 0x007FB0 à 0x007FDF
En mode 21 : 0x00FFB0 à 0x00FFDF
En mode 23 (SA-1) : 0x007FB0 à 0x007FDF
En mode 25 : 0x40FFB0 à 0x40FFDF

Elles servent à référencer une cartouche finale dans la base de donnée Nintendo, et accessoirement, à faire fonctionner les jeux sur la Super Nintendo :p
Exemple pour le mode 20 :

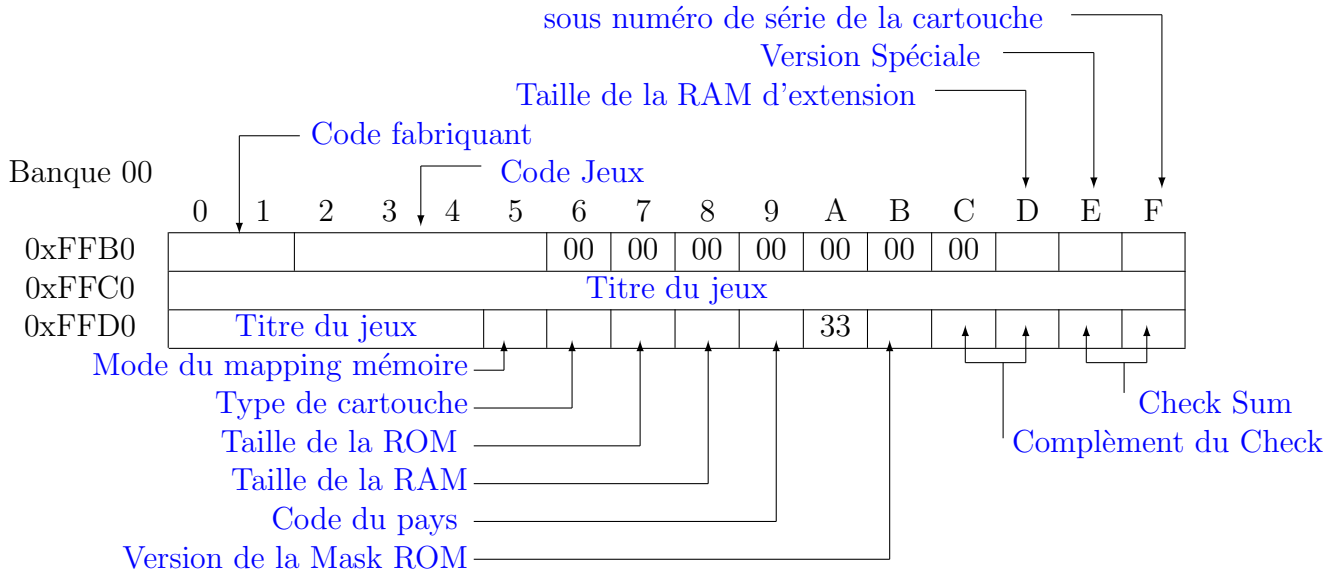


TABLE 16.1 – Information Cartouche

16.1 Code fabriquant

(0xFFB0,0xFFB1)

Entrez 2 octets ASCII assigné par Nintendo en majuscule. Vue que l'on a pas de licence, et encore moins de contrat, avec Nintendo, on peut mettre n'importe quoi !

Par exemple : si on met "01", on a 0x30 et 0x31. 0x30 se met dans 0xFFB0 et 0x31 dans 0xFFB1.

16.2 Code Jeux

(0xFFB2 0xFFB5)

Entrez 4 octets du code du jeu assigné par Nintendo en ASCII et en majuscule.

Par exemple le code du jeux est "SMWJ", le code ASCII que l'on rentrera est :

0x53 ('S') $\Rightarrow$ 0xFFB2
0x4D('M') $\Rightarrow$ 0xFFB3
0x57('W') $\Rightarrow$ 0xFFB4
0x4A ('J') $\Rightarrow$ 0xFFB5

Si code de jeu est sur 2 octets, alors les deux derniers octets des code "Space" (espace).

Par exemple, le code du jeux est "MW"

0x4D('M') $\Rightarrow$ 0xFFB2
0x57('W') $\Rightarrow$ 0xFFB3
0x20 (' ') $\Rightarrow$ 0xFFB4
0x20 (' ') $\Rightarrow$ 0xFFB5

16.3 Valeur fixée

(0xFFB6-0xFFBC)

Dans les adresses 0xFFB6 à 0xFFBC, on insére 0x00. C'est pas moi c'est Nintendo !!!

16.4 Taille de la RAM d'extension

(0xFFBD)

Entre la taille de la RAM d'extension instée dans le jeux, en utilisant la table ci-dessous. Si la taille n'est pas listé ci-dessous, choisissez la taille listé suivant (si RAM > 1Mbits alors 0xFFBD > 0x07).

Attention, ceci est la RAM d'extension, et non la RAM du CPU du coprocesseur !

Par exemple la SUPER-FX a une RAM commune à la Super Nintendo, alors que le SA-1 à une RAM intégré à son processeur ! Dans ce dernier cas il faudra informer le registre 0xFFD8 et mettre celui-ci à 0x00.

0xFFBD	Taille de l'expansion de la RAM
0x00	Aucun
0x01	16 Kbit
0x03	64 Kbit
0x05	256 Kbit
0x06	512 Kbit
0x07	1 Mbit

Par exemple, si vous voulez mettre 4Mo de RAM (soit 64Mbit), la valeur sera de 0x0D.

Si on met 0xFF cela signifie que nous aurons $4.5x10^74$ Mbit soit $2^248MBit$ ou encore $2^244Moctets$, il y a de la marge avant d'en arriver là ...

16.5 Version Spéciale

(0xFFBE)

Ceci est utilisé que dans des circonstances spéciales, comme un évennement promotionnel.

Le code 0x00 doit être entré dans des circonstance normale. On pourra eventuellement l'utilisé pour sortir des version "BETA".

16.6 sous numéro de série de la cartouche

(0xFFBF)

Ceci est utilisé dans le cas où deux jeux cohabitent dans la même cartouche de jeux. Sinon habituellement on utilise 0x00.

16.7 Titre du jeux

(0xFFC0-0xFFD4)

Ici on entre le nom du jeu en ASCII, référez vous au tableau des codes ASCII ci dessous pour indiquer les caratères dont vous pouvez utiliser. Si il n'y a rien à mettre, assignez ces adresses à 0x00.

	00	10	20	30	40	50	60	70	80	~	F0
0			SP	0		P	’	p			
1			!	1	A	Q	a	q			
2			”	2	B	R	b	r			
3			#	3	C	S	c	s			
4			$	4	D	T	d	t			
5			%	5	E	U	e	u			
6			&	6	F	V	f	v			
7			’	7	G	W	g	w			
8			(	8	H	X	h	x			
9			)	9	I	Y	i	y			
A			*	:	J	Z	j	z			
B			+	;	K	[	k	{			
C			,	<	L	¥	l	|			
D			-	=	M	]	m	}			
E			.	>	N	^	n	~			
F			//	?	O	_	o				

Table 16.2 – Code ASCII pour le titre du jeux

16.8 Mode du mapping mémoire

(0xFFD5)

Ceci est utilisé pour informer le mode du mapping mémoire que l'on utilise, et la vitesse d'horloge du CPU de la Super Nintendo. MODE 20,21,23,25. Sélectionnez le code approprié dans la table ci-dessous.

0xFFD5	Mode	Clock du CPU de la Super Nintendo
0x20	Mode 20	2.68Mhz (vitesse normale)
0x21	Mode 21	2.68Mhz (vitesse normale)
0x22	RFU[1]	- - - - - - - - - - -
0x23	Mode 23(SA-1)	2.68Mhz (vitesse normale)
0x25	Mode 25	2.68Mhz (vitesse normale)
0x30	Mode 30	3.68Mhz (vitesse normale)
0x31	Mode 31	3.68Mhz (vitesse normale)
0x35	Mode 35	3.68Mhz (vitesse normale)

Table 16.3 – Mapping mémoire

16.9 Type de cartouche

(0xFFD6)

Indique la configuration de la cartouche. utilisez l'une des deux table selon a présence d'un coprocesseur ou non.

0xFFD6	Configuration de la cartouche
0x00	ROM seulement
0x01	ROM + RAM
0x02	ROM + RAM + Batterie

Table 16.4 – Type de cartouche sans co-processeur

0xFFD6		Configuration de la cartouche
Bits haut	Bits bas	
0x0*	-	Co-processeur = DSP
0x1*	-	Co-processeur = SuperFx
0x2*	-	Co-processeur = OCB1
0x3*	-	Co-processeur = SA-1
0xE*	-	Co-processeur = Autre
0xF*	-	Co-processeur = Processeur personnalisé
-	0x3*	ROM + Co-processeur
-	0x4*	ROM + Co-processeur + RAM
-	0x5*	ROM + Co-processeur + RAM + Batterie
-	0x6*	ROM + Co-processeur + Batterie

Table 16.5 – Type de cartouche Avec co-processeur

Exemple : Si on a une cartouche avec un co-processeur qui n'est pas dans la norme Nintendo, et que l'on a une ROM (obligatoire) une RAM et une batterie (bon une pile qui sauvegarde des données quoi) on doit mettre la valeur 0xF5 selon le second tableau.

[0] **R**éservé pour une **F**uture **U**tilisation

16.10 Taille de la ROM

(0xFFD7)

On se réfère au tableau suivant pour informer la taille de la ROM.

0xFFD7	Taille de la ROM
0x09	3 - 4 Mbit
0x0A	5 - 8 Mbit
0x0B	9 - 16 Mbit
0x0C	17 - 32 Mbit
0x0D	33 - 64 Mbit

Table 16.6 – Taille de la ROM

Et on ne pas avoir une ROM de plus de 64 Mbit (4Moctets), car le mapping mémoire ne nous le permet pas.

16.11 Taille de la RAM

(0xFFD8)

C'est ici que l'on insère la taille de la RAM du CPU. Si on en a pas, 0x00 est alors imposé. La valeur 0x00 est aussi imposé si la RAM installé est utilisé uniquement par le co-processeur, ou interne à celui-ci.

Par exemple, la RAM utilisé par le SuperFx est renseigné dans le registre 0xFFBD, et 0x00 dans 0xFFD8.

Encore un exemple, le SA-1 possède une RAM interne, cette taille doit être renseigné ici.

0xFFD7	Taille de la RAM
0x00	Aucune RAM
0x01	16K bit
0x03	64K bit
0x05	256K bit
0x06	512K bit
0x07	1M bit

Table 16.7 – Taille de la RAM CPU

16.12 Code du pays

(0xFFD9)

Ou encore, code de destination, pour informer dans quel pays ou continent ira le jeux, pour avoir un peu sa trace, et pour recouper le code du jeux en 0xFFB2 à 0xFFB5.

0xFFD9	Destination - (language)	Code de reconnaissance de la ROM (Code Jeux)
0x00	Japon	J
0x01	Amérique du nord (USA & Canada)	E
0x02	Toute l'Europe	P
0x03	Sandinavie	W
0x06	France	F
0x07	Hollande	H
0x08	Espagne	S
0x09	Allemagne	D
0x0A	Italie	I
0x0B	Chine	C
0x0D	Korée	K
0x0E	Commmon	A
0x0F	Canada	N
0x10	Brésil	B
	Nintendo Gateway System[2]	G
0x11	Australie	U
0x12	Autre Version	X
0x13	Autre Version	Y
0x14	Autre Version	Z

Table 16.8 – Code Pays

[0]filiale qui développe des partenariats divers à travers le monde, avec des hôtels, des compagnies aériennes ou des constructeurs automobiles. Elle met aussi en place des systèmes vidéoludiques embarqués.(source : Wikipedia.org)

16.13 Valeur fixe

(0xFFDA)

On fixe la valeur à 0x33.

16.14 Version de la Mask ROM

(0xFFDB)

Stocker le numéro de la version de la Mask ROM sortie sur le marché comme produit. Le numéro de la version de votre ROM en gros...

Le numéro commence à 0x00 à la production, et s'incrémente à chaque nouvelle révision de version.

16.15 Complèment du Check

(0xFFDC, 0xFFDD)

Stockez le premier compplement des deux octets bas du check sum du programme dans l'ordre : 0xFFDC et 0xFFDD. Reférez vous à "Check Sum", pour son calcul.

$$\frac{(0xFFDE, 0xFFDF)}{CheckSum} + \frac{(0xFFDC, 0xFFDD)}{ComplementduCheck} = 0xFFFF \tag{16.1}$$

16.16 Check Sum

(0xFFDE, 0xFFDF)

Stockez dans la zone du complément du check (0xFFDC et 0xFFDD) la valeur 0xFF, et dans la zone du Check Sum (0xFFDE et 0xFFDF) la valeur 0x00.

Ajoutez alors chaque octet dans les données de la ROM. Si la taille de la ROM ne peut pas être exprimée en 2^nMbit, tel que 10Mbit ou 20Mbit, ajouté le reste jusqu'à ce que le total de 2^nMbit soit atteind.

Par exemple, si le programme fait 12Mbit, exécutez le calcul comme si c'étaient 16Mbit.

$2^3 Mbit < 12Mbit > 2^4 Mbit$
$8Mbit = 2^3 Mbit$, reste $4Mbit$
On va donc prend 2^4Mbit comme checksum.

Pour 10Mbit executez le calcul comme si c'était 16Mbits. (Total des premier 8Mbit) + [(total des dernier 2Mbit)x4] = Check Sum

Pour 20Mbit executez le calcul comme si c'était 32Mbits. (Total des premier 8Mbit) + [(total des dernier 4Mbit)x4] = Check Sum

Pour 24Mbit executez le calcul comme si c'était 32Mbits. (Total des premier 8Mbit) + [(total des dernier 8Mbit)x2] = Check Sum

Par la suite, stocker les 2 octets bas de la valeur du check sum dans la zone du check sum (0xFFDE,0xFFDF), 0xFFDE contiendra l'octet bas et 0xFFDF l'octet haut.

Ensuite, stocker les deux octets bas du complément du check dans les registres 0xFFDC et 0xFFDD.

Chapitre 17

Registres Du PPU

Dans ce chapitre vous allez découvrir les registres de la Super Nintendo, ceux qui vont nous servir à contrôler tous les éléments de la Super Nintendo.

Ci-dessous, je vous donne quelques définitions qui vous serrons utile lors de la lecture :

Le VBLANK est le moment ou le ”rayon” qui balaye l'écran de haut en bas arrive en bas, il doit donc se repositionner en haut, le ”rayon” est donc désactiver le temp de l'opération, on parle donc de VBLANK.

Le FORCED BLANK est une intervention qui interrompt l'affichage de l'écran. Le ”rayon” qui balaye l'écran est alors interrompu, et remit à zéro.

CG = Color Generator = Graphique. C'est mieux que de traduire par ”Générateur de couleur”.

= Couleur ! ! Attention, ma traduction alterne parfois entre ”couleur” et ”graphique”, c'est la même chose ! Eh oui c'est embêtant la langue française quand on a plusieurs mots pour définir un seul objet ! ! !

La Super Nintendo peut afficher, grâce à ces registres, 4 plans, BG1 BG2 BG3 BG4, et un Objet, OBJ. Vous allez voir leur initialisation pour les afficher, l'odre, la priorité etc...

17.1 $2100 INIDISP

Initialisation de l'écran (INIt DISplay)

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
BLANK-ING				**Brillance (0 à 15)**			
				F3	F2	F1	F0

- **BLANKING** : Gestion de l'écran 1 : Eteint 0 : Allumer
- **Brillance de l'écran**

F3	F2	F1	F0	Brillance
1	1	1	1	Brillant
1	1	1	0	
		(...)		
0	0	0	1	
0	0	0	0	Noir

17.2 $2101 OBJSEL

Taille de l'objet et paramétrage de la zone de donnée de l'objet (OBJect SELection). voir la section 13.1 "Objet" à la page 247 pour sa description.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
Taille d'Objet			**Nom d'objet**		**Adress de l'objet**		
S2	S1	S0	N1	N0	BA-2	BA-1	BA-0

- Détermine la taille de des objets (Voir l'Annexe PPU p3 à l'Annexe PPU p4)

  DOT = point.

S2	S1	S0	Taille d'objet 0(SM)	Taille d'objet 1(LG)
0	0	0	8 DOT	16 DOT
0	0	1	8 DOT	32 DOT
0	1	0	8 DOT	64 DOT
0	1	1	16 DOT	32 DOT
1	0	0	16 DOT	64 DOT
1	0	1	32 DOT	64 DOT

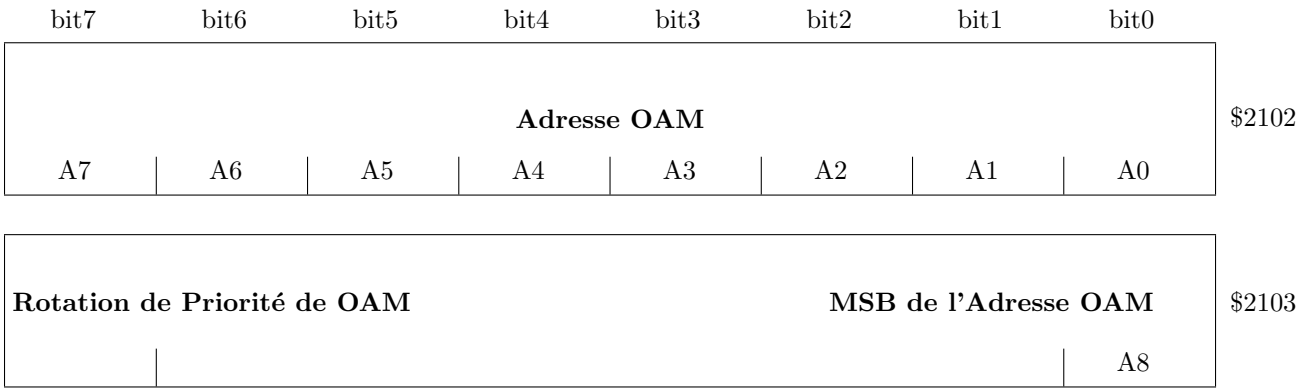
- **Nom d'objet, N1,N0** (Voir l'Annexe PPU p1 à l'Annexe PPU p2)
- **Adresse de base de l'objet BA-2 BA-1 BA-0**

  Désigne l'adresse du segment des données Objets. Pour ensuite les stocker dans la VRAM. (Voir l'Annexe PPU p1 à l'Annexe PPU p2)

  BA-2 est réservé pour une expansion future donc = 0.

17.3 $2102 OAMADDL - $2103 OAMADDH

OAM EST UNE MÉMOIRE ATTRIBUÉ À UN OBJET. OAMADD est un registre 16 bits, c'est pourquoi nous avons L et H, L=Low (bit0 à bit7) et H=High(bit8 à bit 15). Il contient l'adresse de l'objet qui est en mémoire. c'est un pointeur en fait !



- C'est cette adresse qui doit être initialisé en avance, quand on veut écrire ou lire l'objet.
- Pour paramétrer l'ordre de priorité de l'objet, on écrit "1" dans le bit 7 du registre $2103 et mettre la priorité sur l'objet (0 à 127) dans D1 à D7 du registre $2102. Ensuite on remet "0" dans le bit 7 de $2103.

- L'adresse qui à été paramétré juste avant chaque secteur,(commençant pas V-BLANK[1]) sera paramétré un fois de plus aux registres $2102 et $2103 automatiquement. Cependant, l'adresse de contrôle sera automatiquement paramétré durant la période "FORCED BLANK"[2].

[1] "Registres Du PPU" à la page 265
[2] "Registres Du PPU" à la page 265

17.4 $2104 OAM DATA

Données pour l'écriture de OAM.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
Donnée OAM								$2104
A7	A6	A5	A4	A3	A2	A1	A0	

- Se sont les données OAM qui vont écrire dans n'importe quel adresse de l'OAM (lire l'Annexe PPU p3).
- Après l'accès aux registres $2102 et $2103, les données doivent êtres écrites dans l'ordre ; 8bit les plus bas, et les 8bits les plus hauts, dans le registre $2104. L'adresse OAM sera incrémenté automatiquement, quand les données OAM seront en train d'écrire dans l'ordre BAS à HAUT.
- Les données peuvent êtres écrites durant les périodes V-BLANK et FORCED BLANK.

17.5 $2105 BG MODE

PARAMÉTRAGE ARRIÈRE PLAN BG = Back Ground = arrière plan. Ici on change le mode de l'arrière plan, et on paramètre la taille des caractères.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
Taille du BG				**BG3**	**Mode du BG**		
BG 4	BG 3	BG 2	BG 1	PRIO	M2	M1	M0

- **Taille du BG** : Désigne la taille de chaque caractères (voir l'Annexe PPU p21 à l'Annexe PPU p22).
  0 : 8x8 DOT/caractère
  1 : 16x16 DOT/caractère

- **BG3** : Mettre la priorité la plus haute dans BG3, durant le mode BG 0 ou 1 (voir l'Annexe PPU p5).
  0 : OFF
  1 : ON

- **Mode BG** : Voir le résumé du Mode BG à l'Annexe PPU p5.

17.6 $2106 MOSAIC

Taille des Mosaïque, et choix de l'écran pour l'afficher.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
Taille de la Mosaïque				**Activation de la Mosaïque**			
BG4	BG3	BG2	BG1	M3	M2	M1	M0

- **Activation de la Mosaïque** :

  ON/OFF du mode mosaïques pour chaque arrière plan
  0 : OFF
  1 : ON

- **Taille de la Mosaïque** : Désigne la taille de la mosaïque (voir l'Annexe PPU p7)

  Mode 256

M3	M2	M1	M0	Size
0	0	0	0	1x1 DOT
0	0	0	1	2x2 DOT
0	0	1	0	3x3 DOT
	...			...
1	1	1	0	15x15 DOT
1	1	1	1	16x16 DOT

  Mode 512

M3	M2	M1	M0	Size Non entrelacé
0	0	0	0	2x1 DOT
0	0	0	1	4x4 DOT
0	0	1	0	6x6 DOT
	...			...
1	1	1	0	3$30 DOT
1	1	1	1	32x32 DOT

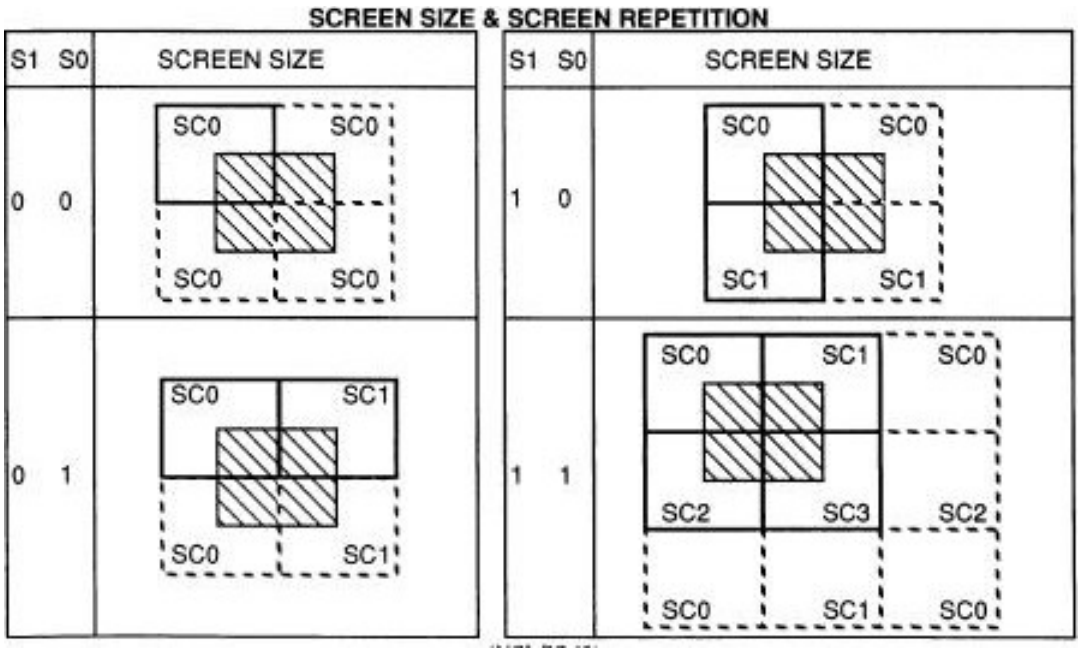
17.7 $2107 BG1SC à $210A BG4SC

$2107 BG1SC - $2108 BG2SC - $2109 BG3SC - $210A BG4SC

PARAMÉTRE DES ÉCRANS SC = SCreen = Ecran. Adresses pour stocker les données écran de chaque arrière plan et paramètre la taille de l'écran.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
BG1 SC Base Address						**BG1 SC Size**		$2107
A5	A4	A3	A2	A1	A0	S1	S0	
BG2 SC Base Address						**BG2 SC Size**		$2108
A5	A4	A3	A2	A1	A0	S1	S0	
BG3 SC Base Address						**BG3 SC Size**		$2109
A5	A4	A3	A2	A1	A0	S1	S0	
BG4 SC Base Address						**BG4 SC Size**		$210A
A5	A4	A3	A2	A1	A0	S1	S0	

- **Taille d'écran :** (voir l'Annexe PPU p21 à l'Annexe PPU p22) Désigne La taille de l'arrière plan.
- **Base d'adresse d'arrière plan (6bits haut)** Désigne le segment dans lequel les données dans la Vram sont stockés. (1K-Word/segment)



17.8 $210B BG12NBA - $210C BG34NBA

PARAMÈTRE L'ARRIÈRE PLAN Désignation de la zone de donnée de Caractères en arrière d'écran.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
BG2 Nom de l'adresse de base				**BG1 Nom de l'adresse de base**				$210B
A3	A2	A1	A0	A3	A2	A1	A0	
BG4 Nom de l'adresse de base				**BG3 Nom de l'adresse de base**				$210C
A3	A2	A1	A0	A3	A2	A1	A0	

- **Adresse de base d'arrière plan (4 bit Haut) :** désigne le segment d'adresse dans la Vram dans laquelle donnée de l'arrière plan sont stockés. (4k-Word/segment)

17.9 $210D BG1H0FS - $210E BG1V0FS

Défilement Horizontal et Verticale pour BG-1 (arrière plan 1)

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
			H013	H012	H011	H010	H09	
BG 1 Défilement-H (Inférieur-Supérieur)								$210D
H07	H06	H05	H04	H03	H02	H01	H00	

			V013	V012	V011	V010	V09	
BG 1 Défilement-V (Inférieur-Supérieur)								$210E
V07	V06	V05	V04	V03	V02	V01	V00	

- 10 bits MAXIMUM (0 à 1023) peuvent être désignés pour la valeur de défilement H/V
- En MODE-7 la taille de 13 bits MAXIMUM (-4096 à 4095) peut être mise.(voir l'Annexe PPU p10 à l'Annexe PPU p11)
- En écrivant dans le registre deux fois de suite, ce dernier est paramétrè dans l'ordre : bits bas (0 à 7) et bits haut (8 à 12).

17.10 $210F BG2H0FS à $2114 BG4V0FS

$210F BG2H0FS $2110 BG2V0FS - $2111 BG3H0FS $2112 BG3V0FS - $2113 BG4H0FS $2114 BG4V0FS

Défilement Horizontal et Verticale pour BG-2 3 4

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
						H010	H09	$210F
BG 1 Défilement-H (Inférieur-Supérieur)								$2111
H07	H06	H05	H04	H03	H02	H01	H00	$2113

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
						V010	V09	$2110
BG 1 Défilement-V (Inférieur-Supérieur)								$2112
V07	V06	V05	V04	V03	V02	V01	V00	$2114

- La valeur de défilement H/V est sur 10 bits Maximum (0 à 1023) (voir l'Annexe PPU p10).
- En écrivant dans le registre deux fois de suite, ce dernier est paramétrè dans l'ordre : bits bas (0 à 7) et bits haut (8 à 12).

17.11 $2115 VMAINC

Désignation de la valeur incrémentée par l'adresse VRAM.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
H/L				**Adresse Vram Graphique Plein**		**Mode Séquence Incrément Ecran**	
INC				G1	G0	I1	I0

- **bit7** : Désigne le timing d'incrémentassions pour l'adresse

  0 : L'adresse sera incrémenté après que les données soient écrites dans le registre $2118, ou lu dans le registre $2139

  1 : L'adresse sera incrémenté après que les données soient écrites dans le registre $2119, ou lu dans le registre $213A

- **Désigne la valeur d'incrémentation pour la Vram** (voir l'Annexe PPU p8)

G1	G0	I1	I0	Valeur d'incrémentassions
0	1	0	0	Incrémenté par 8 (pour 32 fois) (formations de 2 bits)
1	0	0	0	Incrémenté par 8 (pour 64 fois) (formations de 4 bits)
1	1	0	0	Incrémenté par 8 (pour 128 fois) (formations de 8 bits)
0	0	0	0	Incrémenter Adresse 1 par 1
0	0	0	1	Incrémenter Adresse 32 par 32
0	0	1	0	Incrémenter Adresse 128 par 128
0	0	1	1	Incrémenter Adresse 128 par 128

17.12 $2116 VMADDL - $2117 VMADDH

Adresse pour la lecture et écriture en Vram.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
Adresse VRAM (Octet Inférieur)								$2116
A7	A6	A5	A4	A3	A2	A1	A0	

Adresse VRAM (Octet Supérieur)								$2117
A15	A14	A13	A12	A11	A10	A9	A8	

- C'est l'adresse initiale pour lire ou écrire depuis la VRAM.
- La donnée est lue ou écrite par l'adresse passé initialement. A chaque fois que la donnée est écrit ou lut dans la VRAM, l'adresse sera incrémentée automatiquement.
- La valeur à être incrémentée est déterminé par "SC INCREMENT" dans le registre $2115 et la valeur de "FULL GRAPHIQUE".

17.13 $2118 VMDATAL - $2119 VMDATAH

Données pour l'écriture dans la Vram.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
Adresse VRAM (Octet Supérieur)								$2118
A7	A6	A5	A4	A3	A2	A1	A0	

Adresse VRAM (Octet Supérieur)								$2119
A15	A14	A13	A12	A11	A10	A9	A8	

- Se sont les données de l'arrière plan et d'objet. Elle peuvent êtres écrites dans n'importe quel zone mémoire de la Vram.
- Selon le paramètre "H/L INC" du registre $2115 , la donnée peut être écrite dans la VRAM suivant le tableau ci-dessous :

H/L INC	Écrit dans le Registre	Opération
0	Écrire seulement dans $2118	La donnée est écrite dans les 8bit bas de la Vram et l'adresse sera incrémenté automatiquement
1	Écrire seulement dans $2119	La donnée est écrite dans les 8bit haute de la Vram et l'adresse sera incrémenté automatiquement
0	Écrire dans $2119 puis $2118	Quand les données sont mise dans l'octet bas et l'octet haut, l'adresse sera incrémenté automatiquement
1	Écrire dans $2118 puis $2119	Quand les données sont mise dans l'octet haut et l'octet bas, et l'adresse sera incrémenté automatiquement

17.14 $211A M7SEL

Paramètres Initial du mode 7

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
Hors Ecran						**Basculement** d'Ecran	
01	00					V	H

- **Hors Ecran**

01	00	Procédure or zone
0	0	Répétition de l'écran si on est hors de la zone d'affichage
1	0	Le fond d'écran de couleur unique est hors de la zone d'affichage
1	1	Répétition du caractère #0 si hors de la zone d'affichage

- **Basculement d'écran Horizontal et vertical en mode 7**

V	H	Image
0	O	Image normal
0	1	Retournement horizontal
1	0	Retournement vertical
1	1	Retournement horizontal & vertical

17.15 $211B M7A à $2120 M7Y

$211B M7A - $211C M7B - $211D M7C - $211E M7D - $211F M7X - $2120 M7Y

Rotation-Élargissement-Réduction en mode 7. Paramètres de coordination central et paramétrage pour des opération de multiplication complémentaire

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
MP16	MP15	MP14	MP13	MP12	MP11	MP10	MP9	
Paramètre A de la matrice (Inférieur-Supérieur)								$211B
MP7	MP6	MP5	MP4	MP3	MP2	MP1	MP0	

MP16	MP15	MP14	MP13	MP12	MP11	MP10	MP9	
Paramètre B de la matrice (Inférieur-Supérieur)								$211C
MP7	MP6	MP5	MP4	MP3	MP2	MP1	MP0	

MP16	MP15	MP14	MP13	MP12	MP11	MP10	MP9	
Paramètre C de la matrice (Inférieur-Supérieur)								$211D
MP7	MP6	MP5	MP4	MP3	MP2	MP1	MP0	

MP16	MP15	MP14	MP13	MP12	MP11	MP10	MP9	
Paramètre D de la matrice (Inférieur-Supérieur)								$211E
MP7	MP6	MP5	MP4	MP3	MP2	MP1	MP0	

- Les 8 bits de données doivent êtres écrites deux fois, le bas et ensuite haut.
- Ensuite, le paramètre de rotation, d'élargissement et de réduction doivent être en 16 bits.
- La valeur basse à une virgule devra être déposée dans les 8 bits inférieur. Le MSB [1] du demi-octet haut sera signé, c'est à dire qu'il signifiera si le chiffre est négatif ou positif.
- **Formule pour la rotation, l'élargissement et la réduction** (Voir l'Annexe PPU p16)

$$\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} X_1 & X_0 \\ Y_1 & Y_0 \end{bmatrix} + \begin{bmatrix} X_0 \\ Y_0 \end{bmatrix}$$

$A = \cos\gamma \times (1/\alpha)$
$B = \sin\gamma \times (1/\alpha)$
$C = -\sin\gamma \times (1/\beta)$
$D = \cos\gamma \times (1/\beta)$

γ : Angle de rotation
α : Rapport de réduction pour X(H)
β : Rapport de réduction pour Y(v)

$X_0 \bullet Y_0$: Coordonnée centrales
$X_1 \bullet Y_1$: coordonnée d'affichage
$X_2 \bullet Y_2$: Coordonnée avant Calcul

[1] le bit le plus significatif, le bit MP15

- Mettre la valeur ”A” dans le registre $211B, ”B” ”C” et ”D” dans $211C, $211D, $211E
- Dans une multiplication complémentaire, (16-bit x 8-bit) les registres $211B et $211C sont utilisés. le $211B possède le nombre 16-bit (écrit 2 fois), et le $211C le 8-bit (écrit une fois).

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
			MP13	MP12	MP11	MP10	MP9	
Position centrale X_0 (Inférieur-Supérieur)								$211F
MP7	MP6	MP5	MP4	MP3	MP2	MP1	MP0	

			MP13	MP12	MP11	MP10	MP9	
Position centrale Y_0 (Inférieur-Supérieur)								$2120
MP7	MP6	MP5	MP4	MP3	MP2	MP1	MP0	

- La coordonnée centrale (X_0Y_0) pour la réduction, élargissement et la réduction, peuvent êtres désignés par ce registre
- La valeur de coordonnée X_0 & Y_0 peuvent êtres désigné par 13-bit (complément à 2).
- Le registre doit être initialisé dans l’ordre suivant, les 8 bits inférieurs, ensuite les 5 bits supérieurs, ce qui nous fait 13 bits au total à initialiser.

17.16 $2121 CGADD

Adresse pour lire et écrire dans la Ram Graphique (CG-RAM).

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
Paramètre CG-RAM							
A7	A6	A5	A4	A3	A2	A1	A0

- C'est l'adresse initiale pour lire ou écrire dans la CG-RAM.[1]
- La donnée est lue par l'adresse qui est dans ce registre, et à chaque fois qu'il y aura une lecture ou une écriture, l'adresse de la CG-RAM s'incrémentera automatiquement.

[1] voir 17

17.17 $2122 CGDATA

Donnée pour l'écrire de la CG-RAM.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
	D15	D14	D13	D12	D11	D10	D9
Donnée CG-RAM (Inférieur-Supérieur)							
D7	D6	D5	D4	D3	D2	D1	D0

- C'est le registre qui contient la donnée graphique [1] à être écrite dans n'importe quelle adresse de la ram graphique(CG-RAM).
- Le mapping mémoire de BG1 à BG4 et donnée Objet dans la Ram Graphique (CG-RAM) seront déterminant, ils seront exécutés par chaque mode sélectionné par le "MODE BG" du registre $2105. (Voir l'Annexe PPU p17)
- Il y a 8 palettes de couleur pour chaque fond d'écran BG1 à BG4. La sélection de la palette est sur 3 bit sur les données graphique (CG) "COLOR" (Voir l'Annexe PPU p10).
- les données RAM Graphique (CG-RAM) sont sur 15 bits, il faudra donc faire deux accès consécutif pour écrire dans les 8 bits inférieur, et ensuite dans les 7 bits supérieurs. Ensuite l'adresse sera incrémenté de 1 automatiquement.
- NOTE : Même chose que pour le registre de donnée OAM, après que avoir rentré une adresse, les données doivent être écrites dans le demi-octet inférieur, et ensuite dans le supérieur.La donnée ne peut être écrite que durant les périodes H/V BLANK et FORCED BLANK.("Registres Du PPU" à la page 265).
- Voici comment on peut jouer avec ce registre : Le format est en RVB, 5 bits par couleur (32 possibilités).

  bit 10 à 14 => Rouge

  bit 5 à 9 => Vert

  bit 0 à 4 => Bleu

  Si l'on veut mettre Bleu =31, Vert = 7, Rouge = 1

bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Couleur	Bleu					Vert					Rouge				
Valeur	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1

```
1   LDA #%11100000   ; Charger l'octet bas
2   STA $2122
3   LDA #%00000000   ; Charger l'octet Haut
4   STA $2122
```

[1] voir 17

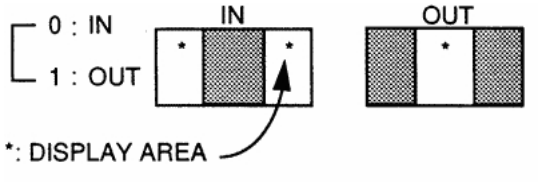
17.18 $2123 W12SEL - $2124 W34SEL - $2125 WOBJSEL

Paramètre mask de la fenêtre (BG1 à BG4, OBJ, COULEUR).

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
Fenêtre BG2				**Fenêtre BG1**				$2123
W2 EN	IN/OUT	W1 EN	IN/OUT	W2 EN	IN/OUT	W1 EN	IN/OUT	
Fenêtre BG4				**Fenêtre BG3**				$2124
W2 EN	IN/OUT	W1 EN	IN/OUT	W2 EN	IN/OUT	W1 EN	IN/OUT	
Fenêtre COULEUR				**Fenêtre OBJET**				$2125
W2 EN	IN/OUT	W1 EN	IN/OUT	W2 EN	IN/OUT	W1 EN	IN/OUT	

- **IN/OUT**

  La zone de masquage de la fenêtre, peut être indiqué par l'extérieur ou l'intérieur de la Frame. Cette dernière étant indiquée par la position de la fenêtre.



- **W1 EN**

  Indique si la fenêtre-1(W1) est active ou pas.
  0 : OFF
  1 : ON

- **W2 EN**

  Indique si la fenêtre-2(W2) est active ou pas.
  0 : OFF
  1 : ON

La "FENETRE de COULEUR" (COLOR WINDOW) est une fenêtre principale et secondaire. (Voir le registre $2130)

17.19 $2126 WH0 à $2129 WH3

$2126 WH0 - $2127 WH1 - $2128 WH2 - $2129 WH3 INDIQUE LA POSITION DE LA FENÊTRE (Voir l'Annexe PPU p18).

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
position H0/H1/H2/H3 de la Fenêtre								$2126 $2127 $2128
P7	P6	P5	P4	P3	P2	P1	P0	$2129

Position H0 de la fenêtre $2126 : Fenêtre-1 position gauche - 0 à 255
Position H1 de la fenêtre $2127 : Fenêtre-1 position droite - 0 à 255
Position H2 de la fenêtre $2128 : Fenêtre-2 position gauche - 0 à 255
Position H3 de la fenêtre $2129 : Fenêtre-2 position droite - 0 à 255

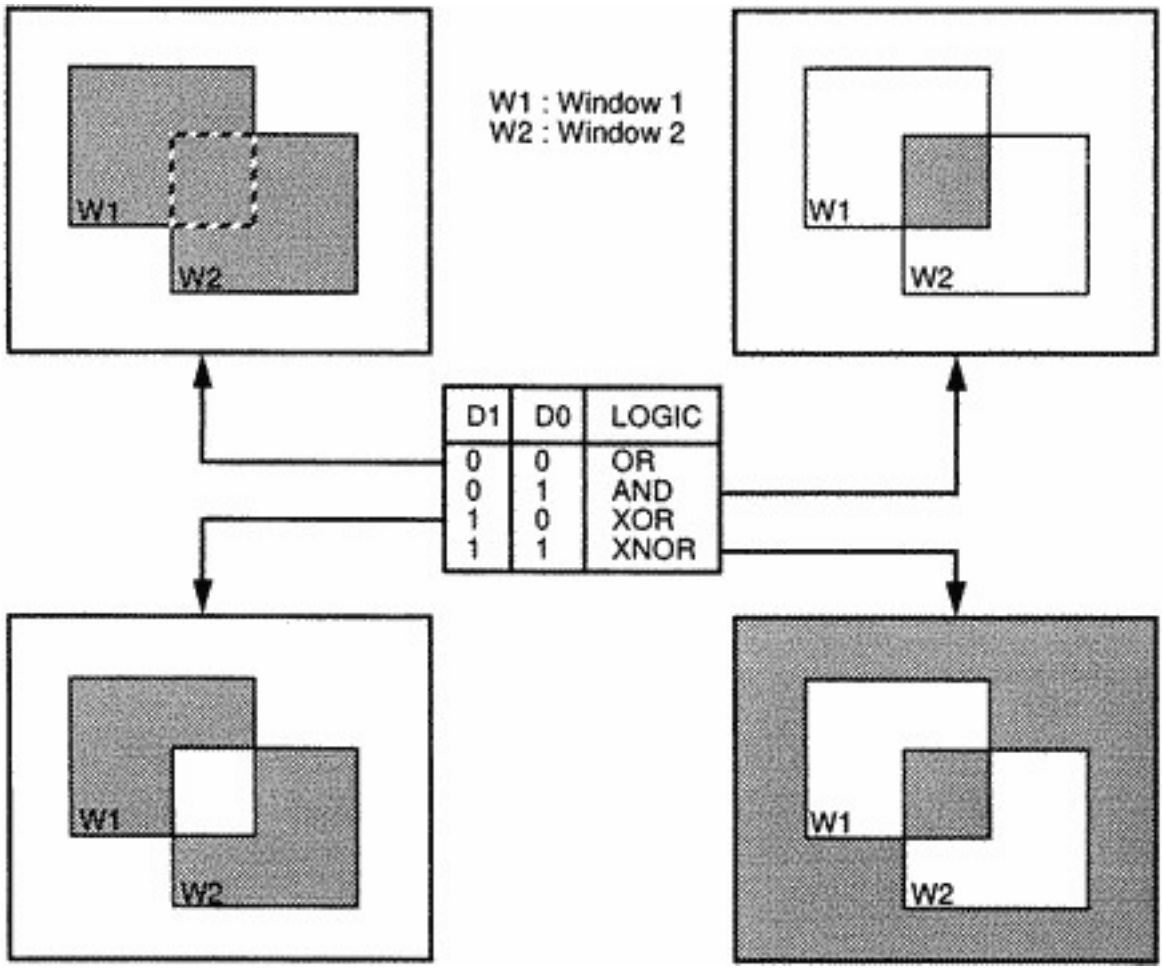
- Si la valeur de la position gauche > la valeur de la position droite, alors les position de la fenêtre sera à 0.

17.20 $212A WBGLOG $212B WOBJLOG

Paramètre logique du masque pour la fenêtre-1 et 2 sur chaque écran.

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
$212A	BG4		BG3		BG2		BG1	
	D1	D0	D1	D0	D1	D0	D1	D0
$212B					FENETRE LOGIQUE			
					COULEUR		OBJ	
					D1	D0	D1	D0

- On met les masques logique pour les fenêtre-1 et 2. Quand les deux fenêtres sont ”IN”, la portion partagé sera masqué automatiquement comme si-dessous :



- NOTE : ”IN/OUT” des registres $2123 à $2125 deviennent des ”non logique” pour chaque fenêtre-1 et fenêtre-2.

17.21 $212C TM

Ecran de premier plan

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
			PREMIER PLAN				
			OBJ	BG4	BG3	BG2	BG1

- Le registre "PREMIER PLAN" désigne l'écran ,BG1 à BG4, ou OBJET, qui va être en premier plan.
- Il désigne aussi quel est l'écran qui va être ajouter pour l'addition ou la soustrait sur l'écran.
  0 : Désactivé
  1 : Activé

17.22 $212D TS

Ecran de second plan

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
			SECOND PLAN				
			OBJ	BG4	BG3	BG2	BG1

- Le registre "SECOND PLAN" désigne l'écran ,BG1 à BG4, ou OBJET, qui va être en second plan(fond d'écran,).
- Il désigne aussi quel est l'écran qui va être ajouter pour l'addition ou la soustrait sur l'écran.
  0 : Désactivé
  1 : Activé
- Quand l'addition ou la soustraction de l'écran est fonctionnel, le SECOND plan est un écran à ajouter ou à soustraire sur l'écran PRINCIPAL.

17.23 $212E TMW

Fenêtre de Masque pour le premier plan

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
			(Fenêtre) PREMIER PLAN				
			OBJ	BG4	BG3	BG2	BG1

- Avant d'aller dans ce registre, vérifiez la zone de fenêtre indiqué par les registres $2123 à $2129, et l'écran à afficher défini par le registre $212C.
  0 : Désactivé
  1 : Activé

17.24 $212F TSW

Fenêtre de Masque pour le second plan

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
			(Fenêtre) SECOND PLAN				
			OBJ	BG4	BG3	BG2	BG1

- Avant d'aller dans ce registre, vérifiez la zone de fenêtre indiqué par les registres $2123 à $2129, et l'écran à afficher défini par le registre $212C.
  0 : Désactivé
  1 : Activé

- Quand l'addition ou la soustraction de l'écran est fonctionnel, le SECOND plan est un écran à ajouter ou à soustraire sur l'écran PRINCIPAL.

17.25 $2130 CGSWSEL

Paramètres Initiales pour décider de l'ajout de couleur ou d'écran.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
Fenêtre Graphique ON/OFF						**CC ADD ENABLE**	**DIRECT SELECT**
MAIN SW(A)		SUB SW(B)					
M1	M0	S1	S0				

- **Fenêtre Graphique**

  Quand la Fenêtre Graphique est fonctionnelle, on peut alors assigné une zone de fenêtre pour l'écran principal et secondaire.

M1 (S1)	M0 (S0)	Fonction
0	0	ON (Tout le temp)
0	1	ON (Dans la fêntre uniquement)
1	0	ON (Hors de la fenêtre uniquement)
1	1	OFF (Tout le temp)

- **CC ADD ENABLE**

  Activer l'addition/soustraction de la couleur fixe.

  Indique laquelle des 2 sortes de données, devra être ajoutés/soustraite entre elle ou pas, celles-ci sont les ensembles de couleur fixé par le registre $2132 et les données graphique qui sont paramétrés dans la CGRAM.

  0 : Addition/Soustraction pour les couleur fixées
  1 : Addition/soustraction pour l'écran de fond

- **Direct Select**

  (Voir l'Annexe PPU p17.)

  La donnée VRAM (données Graphique & Personnage) deviennent les données graphique directement. (Seul les modes 3,4 & 7) sont sélectionnés).

  0 : Déactivé
  1 : Activé

17.26 $2131 CGADSUB

Paramétrage de l'Addition/Soustraction et soustraction pour chaque objet en fond d'écran et couleurs de fond.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
ADD SUB	**1/2 Enable**	**Activation de l'Addition et de la soustraction**					
		BACK	OBJ	BG4	BG3	BG2	BG1

- **Sélection de l'Addition/Soustraction des données Graphique**

  Dans le cas de l'exécution de l'addition/soustraction d'écran, indiquez l'addition ou le mode de soustraction.

  0 : Mode Addition sélectionné
  1 : Mode Soustraction sélectionné

- **Indique "1/2 de la donnée graphique"**

  Quand l'addition (ou soustraction) de la constante de couleur ou l'addition (ou soustraction) d'écran est exécuté, cela signifie que le résultat de RGB dans le secteur d'addition/soustraction devrait être « 1/2 » ou pas. Le secteur arrière (couleur constante) sur l'écran secondaire, ne deviendra pas "1/2".

  0 : Désactivé
  1 : Activé (fonction 1/2 :ON)

- **Donnée graphique Addition/Soustraction activé**

  Désigne la donnée graphique de BG1 à BG4, OBJ, ou Back dans l'écran principal. Pour l'addition/soustraction des données graphique de l'écran secondaire (ou données graphique fixe).

  0 : Désactivé
  1 : Activé (Addition/Soustraction :ON)

- NOTE : Quand OBJ est désigné, la fonction de l'addition/Soustraction est disponible seulement quand la palette de couleur de l'OBJ est de 4 à 7.

17.27 $2132 COLDATA

Donnée Graphique fixé, pour l'Addition/Soustraction de la couleur fixé.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
FENETRE LOGIQUE							
P7	P6	P5	P4	P3	P2	P1	P0

Donnée Couleur constante			**Données de brillant de couleur**				
BLUE	GREEN	RED	D4	D3	D2	D1	D0

- **Désignation de couleur** : Sélection de la couleur désirée

  L'éclat R/G/B doit être paramétré utilisant 5 bit de données. Exemple :

Rouge :	$C0, $3F	(B=$00, V=$00, R=$1F)
Vert :	$A0, $5F	(B=$00, V=$1F, R=$00)
Bleue :	$60, $9F	(B=$1F, V=$00, R=$00)
Blanc :	$FF	
Noir :	$E0	

- **Données Constante de couleur** Paramètre la donnée constante de couleur pour l'addition/soustraction de la constante de couleur.

17.28 $2133 SETINI

Paramètre initial de l'écran.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
EXT. SYNC.	**EXT.** INPUT			**PSEUDO 512**	**224 / 239**	**OBJ-V SELECT**	**INTER-LACE**

- **Synchronisation Externe**

  Utilisé pour super-positionner les images, etc... Normallement, "0" doit être écrit.

- **Mode EXTBG(Ecran étendu)**

  Active la donnée fourni depuis le LSI externe. Pour la Super Nintendo, activer quand l'écran, avec priorité est activé, en mode 7.

- **Mode 512 pseudo horizontal**

  Une résolution imaginaire de 512(horizontal) peut être créé par décalage de demi de point(dot) de l'écran secondaire vers la gauche , alternant tous les champs.

  0 : Disable
  1 : Enable

- **Affichage de la direction verticale du fond d'écran(224/239)**

  Décale la ligne d'affichage du champs de 224 lignes ou 239 lignes. (Dans le cas du mode d'entrelacement, se sera un double point(dot)).

  0 : 224 Lines
  1 : 239 Lines

- **Affichage de la direction verticale de l'Objet (OBJ-V)**[1].

  Dans le mode d'entrelacement, sélectionnez soit 1 point par la ligne ou 1 point a répété toutes les 2 lignes. Si "1" est écrit, l'Objet semble être réduit de la moitié de sa taille verticale.

- **Choix de l'entrelacement(1) ou non(0) du balayage(Interlace)**

  Voir $2105.

[1]Si "D1" est paramétré dans le monde "non-entrelacé", les lignes paire et impaire de l'Objet sera afficher alternativement à chaque champ

17.29 $2134 MPYL - $2135 MPYM - $2136 MPYH

Résultat de la multiplication

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
MPY(LOW)								$2134
M7	M6	M5	M4	M3	M2	M1	M0	
MPY(MID)								$2135
M15	M14	M13	M12	M11	M10	M9	M8	
MPY(HIGH)								$2136
M22	M21	M20	M19	M18	M17	M16	M15	

- Ceci est le résultat de la multiplication (complément à 2) et peut être lu en plaçant 16-bit dans le registre $211B and 8 bits dans $211C.

17.30 $2137 SLHV

Verrou(Latch) Logiciel pour compteur H/V

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
Verrou Logiciel pour compteur H/V							
SL7	SL6	SL5	SL4	SL3	SL2	SL1	SL0

- C'est le registre, qui génère les impulsions pour le verrouillage de la valeur du compteur H/V.
- La valeur de compteur H/V doit être verrouillé, quand le registre $2137 est lu. La donnée qui a été lu est une donnée insignifiante.
- La valeur du compteur verrouillé doit être référencé par le registre $213C et $213D.

17.31 $2138 OAMDATA

Lire la donnée depuis OAM.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
Donnée OAM (Bas,Haut)							
D7	D6	D5	D4	D3	D2	D1	D0

- Le registre peut lire la donnée a n'importe quel adresse de l'OAM.

  Quand l'adresse est paramétré dans les registres $2102, $2103 et le registre $2138 est alor accessible, la donnée peut être lu, dans l'odre, Bas 8-bits et Haut 8-bits.

  Après, l'adresse sera incrémenté automatiquement. et la donnée de la prochaine adresse pourra être lu.

- NOTE : La donnée peut être lu durant les périodes H/V BLANK et FORCED BLANK.[1].

[1] "Registres Du PPU" à la page 265

17.32 $2139 VMDATAL - $213A VMDATAH

Lire les données de la VRAM

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
Donnée VRAM(Bas)								$2139
D7	D6	D5	D4	D3	D2	D1	D0	

Donnée VRAM(Haut)								$213A
D15	D14	D13	D12	D11	D10	D9	D8	

- Le registre peut lire la donnée a n'importe quel adresse de la VRAM.
- L'adresse initiale doit être paramétré par les registres $2116 et $2117. La donnée peut être lu par l'adresse qui a été paramétrée initialement.
- En lisant les données sans interruption, la première donnée, pour l'incrémentation de l'adresse, doit être lu comme donnée "factice", après que l'adresse ait été paramétré.
- La quantité à être incrémenté sera déterminé par "Incrémentation Ecran" du registre $2115 et le paramétrage de la valeur du "Graphique Plein".
- NOTE : La donnée peut être lu durant les périodes H/V BLANK et FORCED BLANK.[1].

[1] "Registres Du PPU" à la page 265

17.33 $213B CGDATA

Lire les données de la CGRAM (Ram Graphique)

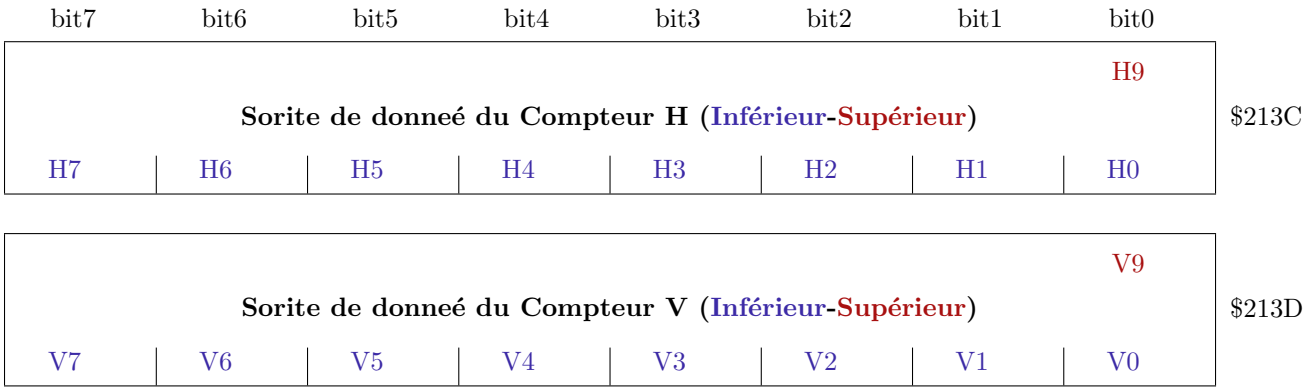
bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
	D15	D14	D13	D12	D11	D10	D9
CG DATA (Inférieur-Supérieur)							
D7	D6	D5	D4	D3	D2	D1	D0

- Le registre peut lire la donnée a n'importe quel adresse de la CGRAM.
- L'adresse initiale doit être paramétré par le registre $2121. Les 8 bits bas sont lus en premier, les 7 bits hauts en second. L'adresse courante sera incrémenter automatiquement pendant la lecture des 7 bits haut.
- NOTE : La donnée peut être lu durant les périodes H/V BLANK et FORCED BLANK.[1].

[1] "Registres Du PPU" à la page 265

17.34 $213C OPHCT - $213D OPVCT

Donnée du compteur H/V par verrou(LATCH) Externe ou Logiciel



- Le Compteur H/V est verrouillé par la lecture du registre $2137, et sa valeur du compteur H/V doit être lue par ce registre.
- Le Compteur H/V est alors verrouillé par le verrou externe, et sa valeur peut être lu par ce registre.
- Si le registre $213C ou $213D est lu après la lecture du registre $213F, les 8 bit bas seront lu en premier, le bit haut en second.

17.35 $213E STAT77

Bit d'état du PPU & Numéro de Version

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
TIME OVER	**RANGE** OVER	**MASTER** /SLAVE		**5C77 numéro de Version**			

- **TIME OVER et RANGE OVER**
  (L'état de l'affichage de l'Objet sur ligne horizontale)
  RANGE : Quand la quantité de l'Objet (sans se soucier de la taille) devien 33pccs ou plus, 1 y sera mis
  TIME : Quand la quantité de l'Objet qui est convertie à "8x8 size" est 35pcs ou plus, 1 y sera mis

- **Selection du mode MASTER / SLAVE** MODE LSI, normalement il est configuré à "0".

- NOTE : La donnée peut être lu durant les périodes H/V BLANK et FORCED BLANK.[1].

[1] "Registres Du PPU" à la page 265

17.36 $213F STAT78

bit d'état du PPU & Numéro de Version

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
FIELD	**EXT.** LATCH	**NTSC** /PAL		**5C78 numéro de Version**			

- **FIELD**

  C'est un bit d'état, qui indique si le 1erou 2^{e}champ (FIELD) est scanné au mode d'entrelacement. (La définition est différente depuis le champ de NTSC)

  0 : 1erChamp
  1 : 2^{e}Champ

- **Indicateur de verrou Externe (EXT. LATCH)**

  Quand le signal externe (light Pen, etc..) est appliqué, il active le verrou de valeur de compteur H/V. C'est la connection au port I/O d7 dans la Super Nintendo. (voir registre CPU $4001).

- **Method d'affichage NTSC/PAL**

  0 : NTSC
  1 : PAL

- NOTE : Quand ce registre est lu, les registres $213C et $213D seront initialisés individuellement dans l'ordre Bas et Haut.

17.37 $2140 APUIO0 à $2143 APUIO3

$2140 APUIO0 - $2141 APUIO1 - $2142 APUIO2 - $2143 APUIO3

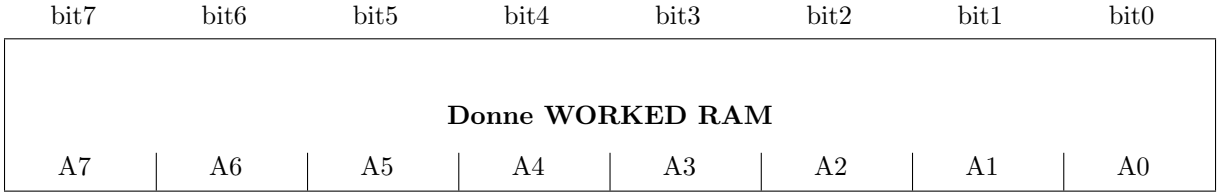
Communication avec le port APU

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
PORTS I/O de l'APDU								$2140 $2141 $2142
P7	P6	P5	P4	P3	P2	P1	P0	$2143

- Les ports fournissent plus de registres pour IN/OUT, qui sont 8 registres au total dans l'APDU. Donc, les différents registres serons accessibles, en lisant ou en écrivant pour la même adresse.
- Référez vous à la page (Contacter moi pour l'info si je n'ai pas mis la page) pour plus de détail

17.38 $2180 WMDATA

Donnée à lire et écrire dans la WRAM



- Donnée à lire et écrire consécutivement à n'importe quel adresse de la WRAM.
- La donnée est lu et écrite à l'adresse paramétré par le registre $2181 à $2183, et l'adresse est automatiquement incrémenté chaque fois que la donnée est lu ou écrite.

17.39 $2181 WMADDL à $2183 WMADDH

$2181 WMADDL $2182 WMADDM $2183 WMADDH

Adresse pour lire et écrire dans la WRAM

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
Adresse WORK RAM(Bas)								$2181
A7	A6	A5	A4	A3	A2	A1	A0	
Adresse WORK RAM(Milieu)								$2182
A15	A14	A13	A12	A11	A10	A9	A8	
Adresse WORK RAM(Haut)								$2183
							A16	

- Adresse à être paramétré avant la WRAM soit lue et écrite.
- De A0 à A16, au registre $2181 à $2183 c'est l'adresse basse de 17 bits, représentant les adresses en mémoire de $7E :0000 à $7F :FFFF

Chapitre 18

Registres Du CPU

18.1 $4200 NMITIMEN

Active l'indicateur pour V-BLANK, interruption du timer et lecture du contrôleur (la manette) standard

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
Activer NMI		**Activer Timer** V - EN	H - EN				**Activer** Contrôle Standard

- **Activation du NMI**

  Active NMI quand V-BLANK commence. (Quand on allume,ou reboot, la console, il sera à "0").

  0 : NMI désactivé
  1 : NMI Activé

- **Activation du Timer**

  V-EN : Activation du compteur du timer Verticale
  H-EN : Activation du compteur du timer Horizontal

V EN	H EN	Fonction
0	0	Désactive H et V
0	1	Active H seulement. IRQ est appliqué par la valeur du compteur du timer H
1	0	Active V seulement. IRQ est appliqué par la valeur du compteur du timer V
1	1	Active V et H. l'IRQ est appliqué par la valeur du compteur du timer H et V

- **Activation du contrôleur Standard**
  0 : Désactive de la lecture automatique du contrôleur
  1 : Activation de la lecture automatique du contrôleur

- NOTE : La lecture de la donnée peut démarrer au commencement de la période V-BLANK, mais il prend 3 ou 4 lignes de balayage, pour compléter la lecture.

18.2 $4201 WRIO

Port I/O programmable (Port de Sortie)

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
PORT IO (Entrée Sortie)							
D7	D6	D5	D4	D3	D2	D1	D0

- C'est le port d'I/O (Entrée/Sortie) programmable. L'écriture de donnée sera directement en sortie.
- Quand le port est utilisé comme Entrée, on doit mettre "1" dans les bit que l'on souhaite lire. La lecture se fait en $4213.
- Seul D6 et D7 peuvent êtres utilisés par la Super Nintendo. pour lire les contrôleurs (manettes).
- Les contrôleurs 1 et 4 (manettes, connecteur 1) ont le signal D6 et les contrôleurs 2 et 3 (manettes, connecteur 2) ont le signal D7.
- Le signal à D7 est également une entrée latch externe (voir registre $213F)

18.3 $4202 WRMPYA - $4203 WRMPYB

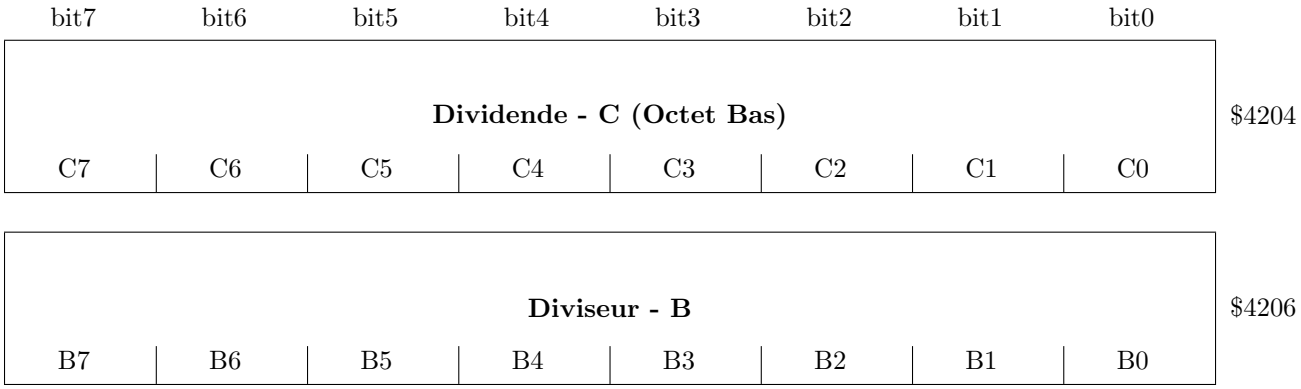
Multiplication

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
Multiplicande - A								$4202
A7	A6	A5	A4	A3	A2	A1	A0	
Multiplicateur - B								$4203
B7	B6	B5	B4	B3	B2	B1	B0	

- C'est un registre qui peut paramétrer un multiplicande (A) et un multiplicateur (B) pour une multiplication en Absolue de "$A(8bits) \times B(16bits) = C(16bits)$".
- Le produit (C) peut être lu par les registres $4216 et $4217
- Paramétré, dans l'ordre, "A" et "B". Cette opération démarrera dés que B sera paramétré, et finira après 8 cycles d'horloge.
- Une fois que le registre A est paramétré, il ne sera pas effacé, jusqu'à ce qu'une nouvelle donnée soit paramétré.

18.4 $4204 WRDIVL - $4205 WRDIVH - $4206 WRDIVB

Division



- C'est un registre qui peut paramétrer un Dividende (C) et un Diviseur (B) pour une division en Absolue de "$C(16bits) \times B(8bits) = A(16bits)$".
- Le Quotient A peut être lu aux registres $4214 et $4215. Et le rest sera aux registre $4216 et $4217.
- Paramétré dans l'ordre, C et B. L'opération commencera dés que B sera entré, et finira après une période de 16 cycles d'horloge.
- Une fois que le registre C est paramétré, il ne sera pas effacé, jusqu'à ce qu'une nouvelle donnée soit paramétré.

18.5 $4207 HTIMEL - $4208 HTIMEH

Paramètre du timer du compteur Horizontal

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
Timer du compteur H								$4207
H7	H6	H5	H4	H3	H2	H1	H0	

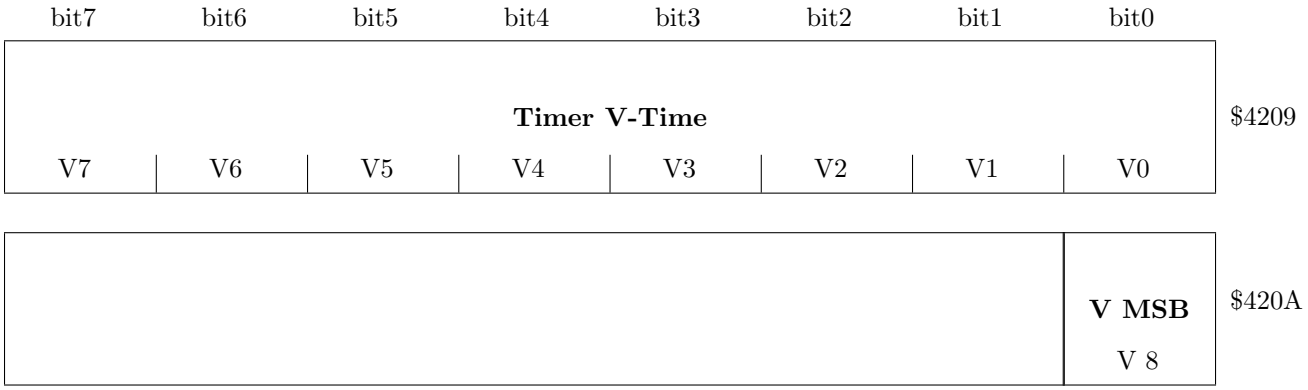
	H MSB	$4208
	H 8	

- C'est le registre qui pramètre la valeur du timer du compteur Horizontal.
- La valeur du compteur est entre 0 et 339, qui est compté depuis la gauche de l'écran.
- Quand le compteur de coordonnée devient le paramétré de la valeur compté, l'IRQ sera appliqué.
- Et en même temps, le "timer IRQ" du registre $4211 (lire Reset) sera à 1. l' Activation/désactivation de l'interruption, sera déterminé par le registre $4200.

- NOTE : Ce compteur continue est remis à zéro à chaque balayage de ligne. Une fois que la valeur est stocké, il est possible de l'appliqué à l'IRQ chaque fois que le balayages de lignes viens à la même position verticale de l'écran.

18.6 $4209 VTIMEL - $4208 VTIMEH

Paramètre du timer du compteur Verticale



- C'est le registre qui pramètre la valeur du timer du compteur Verticale.
- La valeur du compteur est entre 0 et 261(626), qui est compté depuis en le haut de l'écran. (Ce nombre de lignes décris, est différent du nombre de lignes actuelles sur l'écran).
- Quand le compteur de coordonnée devient le paramétré de la valeur compté, l'IRQ sera appliqué.
- Et en même temps, le "timer IRQ" du registre $4211 (lire Reset) sera à 1. L' Activation/désactivation de l'interruption, sera déterminé par le registre $4200.

- NOTE : Ce compteur continue comme le comteur H et sera mis à zéro chaque fois que les 262 lignes seront balayé. Une fois que la valeur est stocké, il est possible de l'appliqué à l'IRQ chaque fois que le balayages de lignes viens à la même position verticale de l'écran.

18.7 $420B MDMAEN

Paramétrage des Canaux pour le DMA générale et de son démarrage

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
Activation de l'indicateur DMA général								$420B
CH7EN	CH6EN	CH5EN	CH4EN	CH3EN	CH2EN	CH1EN	CH0EN	

- Le DMA générale a 8 cannaux au total (CH0 à CH7). On peut sélectionner une des 8 canaux maximum.
- Le canal utilisée doit être à "1" (ex : on veut utilisé le canal 4, le registre aura la valeur %0001 0000).
- Dés que le registre est paramétré, (après quelques cycles d'horloge), le transfert du DMA générale démarre.
- Quand le canal du DMA générale a fini, l'indicateur est effacé.

- NOTE 1 : Parce que la zone de donnée (registre $4300) de chaque canaux est en commum avec la donnée de chaque canal H-DMA, la châine désigné par le registre $420C de le canal H-DMA, ne peut être utilisé. (il est interdit de mettre à 1 le bit de la châine). Au final, 8 canaux(CH0 à CH7) devrait être assigné par le H-DMA et le DMA générale.

- NOTE 2 : Si le H-BLANK arrive durant l'opération du DMA générale et du H-DMA est démarré, le DMA générale sera discontinue dans le milieu , et redémarrera après que le H-DMA finira.

- NOTE 3 : Si 2, ou plus, de canaux sont désignés, le transfert du DMA sera exécuté continuellement accordant l'ordre de priorité décrit l'Annexe CPU p1). Le CPU sera alors stopé tant que le DMA générale aura fini son transfert.

18.8 $420C MDMAEN

Paramétrage des Canaux pour le H-DMA

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
Activer l'indicateur H-DMA								$420B
CH7EN	CH6EN	CH5EN	CH4EN	CH3EN	CH2EN	CH1EN	CH0EN	

- Le H-DMA a 8 canaux au total (CH0 à CH7).
- Le registre est utilisé pour désigner le cannal de sortie des 8 canaux (8 MAX).
- Le canal utilisé doit être à 1. dés que le H-BLANK démarre (après quelques cycles machine), le transfert H-DMA démarre.

- NOTE : Une fois que l'indicateur est paramétré, il ne sera effacé qu'à la prochaine écriture dans le registre. Les paramètres initiaux sont fais automatiquement pour chaque champs et la même parité de transfert sera répétée. L'indicateur est alors visé par la période V-BLANK, alors le transfert DMA sera exécuté proprement pendant la prochaine frame.

18.9 $420D MEMSEL

SÉLECTION DE LA VITESSE D'ACCÈS À LA ZONE MÉMOIRE 2 Annexe MM[1] page 1.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
							2.68 / 3.58

- Vitesse d'accès à la zone mémoire 2
  0 : 2.68Mhz
  1 : 3.58Mhz (Seulement si le vitesse mémoire "high" est utilisé)
- La Mémoire 2 est à l'adresse $8000 à $FFFF de la banque $80 à $BF et toute les adresses de la banque $C0 à $FF.
- Quand la console démarre, ou reboot, ce paramètre est à 0.

[1] Mapping Mémoire

18.10 $4210 RDNMI

Indicateur NMI[1] par V-BLANK et Numéro de version

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
BLANK-NMI				**Numéro version SNES-CPU**			

- **NMI-BLANK** Quand le paramètre "Activer NMI", dans le registre $4200, est à 1, se bit montrera son état.
  0 : Le NMI ne s'est pas produit
  1 : Le NMI s'est produit

- NOTE : Il est nécessaire de mettre à zéro par lecture de l'indicateur pendant le déroulement de NMI l'Annexe CPU p3)

[1]L'indicateur est à 1 au commencement du V-BLANK, et à 0 à la fin du V-BLANK, il peut alors être remis à zéro par lecture de ce registre.

18.11 $4211 TIMEUP

L'INDICATEUR IRQ PAR TIMER DE COMPTEUR H/V [1]

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
TIMER IRQ							

- **TIMER IRQ** L'indicateur IRQ par le timer du compteur H/V :
  Cet indicateur est "READ REST". Il est lu comme une mise à zéro. Si "Activer timer" est activé par le registre $4200, l'IRQ sera appliqué et l'indicateur sera paramétré dés que le timer du compteur H/V porte la valeur stocké.
  0 : Le timer du compteur H/V est activé ou désactivé
  1 : L'état du timer du compteur H/V est arrivé à échéance

[1] Même si V-EN = "0" et H-EN = 0 sont paramétrés par le registre "Activer Timer" su registre $4200, cet indicateur sera remis à zéro.

18.12 $4212 HVBJOY

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
V BLANK	**H BLANK**						**Activer Contrôleur Centrale**

- **V-BLANK** Iindicateur de la periode V-blank : Il montre si le scan est dans la période du V-BLANK ou non.
  0 : Hors de la période V-BLANK
  1 : Dans la période V-BLANK
- **H-BLANK** Indicateur de la période H-Blank : Il montre si le scan est dasn le période H-Blank ou non.
  0 : Hors de la période H-Blank
  1 : Dans la période V-Blank
- **Activer Contrôleur Central** Cet indicateur montre le timing, du contrôleur standard, à lire. (Cependant, il est limité dans le cas où le "Activer contrôleur Standard" du registre $4200 est à 1).
  0 : C'est la période où le contrôleur standard ne lit pas de donnée ou est désactivé.[2]
  1 : C'est la période où le contrôleur standard lit la donnée

[0](Dans le cas ou "Activer contrôleur Standard" du registre $4200 est à 0)

18.13 $4213 RDIO

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
Port I/O							
D7	D6	D5	D4	D3	D2	D1	D0

- C'est la le port I/O (Entrée/Sortie) programmable en entrée. La donnée qui est paramétré dans le port d'entrée doit être lu directement.
- Le bit mis à 1 dans le registre $4201 est utilisé définit comme une entrée du port.
- Seul D6 et D7 peuvent êtres utilisés par la Super Nintendo. pour lire les contrôleurs (manettes).
- Les contrôleurs 1 et 4 (manettes, connecteur 1) ont le signal D6 et les contrôleurs 2 et 3 (manettes, connecteur 2) ont le signal D7.
- Le signal à D7 est également une entrée latch externe (voir registre $213F)

18.14 $4214 RDDIVL - $4215 RDDIVH

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
Quotient A(octet Bas)								$4214
A7	A6	A5	A4	A3	A2	A1	A0	
Quotient A(octet Haut)								$4215
A15	A14	A13	A12	A11	A10	A9	A8	

- C'est le Quotient (A), qui est le résultat absolue de la division du $C(16bits)/B(8bits) = A(16bits)$.
- Le dividende C et le diviseur B sont paramétrés dans les registres $4204, $4205 et $4206.

18.15 $4216 RDMPYL - $4217 RDMPYH

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
RÉSULTAT Multiplication OU RESTE division(octet Bas)								$4214
A7	A6	A5	A4	A3	A2	A1	A0	

RÉSULTAT Multiplication OU RESTE division(octet Haut)								$4215
A15	A14	A13	A12	A11	A10	A9	A8	

Utilisé pour la Multiplication

- C'est le produit C, qui sera le résultat de la multiplication de $A(8bits) \times B(8bits) = C(16bits)$.
- Le multiplicande A et le multiplicateur B sont paramétrés dans les registres $4202 et $4203.

Utilisé pour la Division

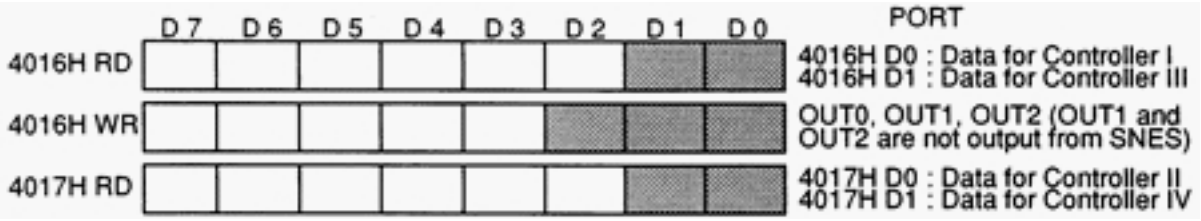
- C'est le Reste de la division de $C(16bits)\ B(8bits) = A(16bits) \ldots Reste(8ou16bits)$
- Le dividende C et le diviseur B sont paramétrés dans les registres $4204, $4205 et $4206.

18.16 $4218 STD CNTRL1L à $421F STD CNTRL4H

$4218 STD CNTRL1L - $4219 STD CNTRL1H - $421A STD CNTRL2L - $421B STD CNTRL2H - $421C STD CNTRL3L - $421D STD CNTRL3H - $421E STD CNTRL4L - $421F STD CNTRL4H

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
Contrôleur Standard 1 (octet Bas)								$4218
A	X	L	R					
Bouton	**Bouton**	**Bouton**	**Bouton**					
Contrôleur Standard 1 (octet Haut)								$4219
B	Y	Select	Start	PAD Directionnel				
Bouton	**Bouton**	**Bouton**	**Bouton**	UP	DOWN	LEFT	RIGHT	
Contrôleur Standard 2 (octet Bas)								$421A
A	X	L	R					
Bouton	**Bouton**	**Bouton**	**Bouton**					
Contrôleur Standard 2 (octet Haut)								$421B
B	Y	Select	Start	PAD Directionnel				
Bouton	**Bouton**	**Bouton**	**Bouton**	UP	DOWN	LEFT	RIGHT	
Contrôleur Standard 3 (octet Bas)								$421C
A	X	L	R					
Bouton	**Bouton**	**Bouton**	**Bouton**					
Contrôleur Standard 3 (octet Haut)								$421D
B	Y	Select	Start	PAD Directionnel				
Bouton	**Bouton**	**Bouton**	**Bouton**	UP	DOWN	LEFT	RIGHT	
Contrôleur Standard 4 (octet Bas)								$421E
A	X	L	R					
Bouton	**Bouton**	**Bouton**	**Bouton**					
Contrôleur Standard 4 (octet Haut)								$421F
B	Y	Select	Start	PAD Directionnel				
Bouton	**Bouton**	**Bouton**	**Bouton**	UP	DOWN	LEFT	RIGHT	

- Les contrôleurs 3 et 4 sont des extensions.
- Registre $4016 et $4017 peut être utilisé comme la NES.



Valeurs : 1 : connecté
0 : non connecté

NOTE : Soit les contrôleurs standard sont connectés à la Super Nintendo ou ne peuvent êtres déterminé par la lecture du registre de 17 bits $4016 et $4017 (Référez vous au "Contrôleur Standard").

18.17 $43X0 Transfert DMA

(X => numéro de canal : 0 à 7)

Paramètre pour le transfert de donnée avec le DMA

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
canal d'origine	**Type de canal**		**Adresse du BUS A**		**canal du transfert du mot(WORD) sélectionné**		
			INC/DEC	Fixé	D2	D1	D0

- **Canal d'origine** Désigne l'origine du transfert. voir l'Annexe CPU p1)

  0 : BUS A -> BUS B (Mémoire CPU -> PPU)
  1 : BUS B -> BUS A (PPU -> Mémoire CPU)

- **Type de Canal (H-DMA seulement)** Désigne le mode d'adresse pour l'accès à la donnée (voir l'Annexe CPU p2)

  0 : Adressage Direct
  1 : Adressage Indirect

- **Adresse du Bus A (DMA générale seulement)** Adresse fixé pour le BUS A (Mémoire CPU) et la Incrémente/Décrémente automatiquement.

  D3 0 : Incrémentation/décrémentation d'adresse automatique
  1 : Adresse fixé (utilisé pour effacer la VRAM etc...

  D4 0 : Incrémentation automatique
  1 : Décrémentation automatique (Dans le cas où D3 est à 0)

- **Canal de tranfert** Sélection du mot à transférer par le DMA.

  DMA générale : Adresse B change, par chaine, la méthode désigné.

bit2	bit1	bit0	Adresse à écrire
0	0	0	Adresse 1
0	0	1	Adresse 2 (Vram etc) Bas,haut
0	1	0	Adresse 1 (écrire 2 fois)
0	1	1	Adresse 2 (écrire 2 fois) bas,bas,haut,haut
1	0	0	Adresse 4 bas,haut,bas,haut

  H-DMA : Le nombre d'octets à être transférer par ligne et écrire la méthode désigné.

bit2	bit1	bit0	nb d'octets à être transféré	Adresse à écrire
0	0	0	1 octets	Adresse 1
0	0	1	2 octets	Adresse 2 (Vram etc) Bas,haut
0	1	0	2 octets	Adresse 1 (écrire 2 fois)
0	1	1	4 octets	Adresse 2 (écrire 2 fois) bas, bas, haut, haut
1	0	0	4 octets	Adresse 4 bas,haut,bas,haut

18.18 $43X1 Transfert DMA

(X => numéro de canal : 0 à 7)
Adresse du BUS B pour le DMA

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
Adresse B							
BA7	BA6	BA5	BA4	BA3	BA2	BA1	BA0

- Ce registre peut paramétrer l'adresse du BUS B
- C'est l'adresse de destination ("Destination du Transfert") ou l'adresse d'origine ("Origine du Transfert") qui est déterminé par le bit 7 (adresse d'origine) dur registre $43X0.

  BUS A <- -> BUS B

  La direction est paramétrée par "Origine du Transfert"
  (L'adresse actuelle du BUS B est $00 :21XX, où XX est la valeur par le registre).

- Quand le H-DMA est exécuté, il sera l'adresse de destination.

18.19 $43X2 $43X3 $43X4

(X => numéro de canal : 0 à 7)
Table d'adresse pour le BUS A du DMA

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
table d'Adresse A1 (octet Bas)								$42X2
A7	A6	A5	A4	A3	A2	A1	A0	
table d'Adresse A1 (octet Haut)								$42X3
A15	A14	A13	A12	A11	A10	A9	A8	
Banque de la table A								$42X4
A23	A22	A21	A20	A19	A18	A17	A16	

- C'est le registre pour paramétrer l'adresse du BUS A.
- C'est l'adresse de destination ("Destination du Transfert") ou l'adresse d'origine ("Origine du Transfert") qui est déterminé par le bit 7 (adresse d'origine) dur registre $43X0.
- Dans le mode H-DMA, l'adresse d'origine est désigné, excepté dans un cas spéciale. Donc, pour la zone CPU désigné par l'adresse, la donnée (voir l'Annexe CPU p2) peut être paramétrée par le mode l'adressage absolue ou par le mode d'adressage direct.
- Cette adresse deviens l'adresse basique sur le BUS A durant la période du transfert du DMA, et l'adresse sera incrémentée ou décrémenté par rapport à celle-ci. (Quand le DMA générale a été exécuté, il sera décrémenté).

18.20 $43X5 $43X6 $43X7

(X => numéro de canal : 0 à 7)

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
Adresse basse de la Donnée								
DA7	DA6	DA5	DA4	DA3	DA2	DA1	DA0	$43X5
Nombre de bits à transférer(Octet Bas)								
B7	B6	B5	B4	B3	B2	B1	B0	

Adresse Haute de la Donnée								
DA15	DA14	DA13	DA12	DA11	DA10	DA9	DA8	$43X6
Nombre de bits à transférer(Octet Haut)								
B15	B14	B13	B12	B11	B10	B9	B8	$42X7
Banque de la donnée								
DA23	DA22	DA21	DA20	DA19	DA18	DA17	DA16	

- **Dans le cas du H-DMA** Ce registre dans lequel l'adresse indirect sera stocké automatiquement dans le mode d'adressage indirect. Cette adresse indirect pointe sur la donnée, pour plus de description (voir l'Annexe CPU p2). Ce n'est pas nécessaire de lire ou d'écrire directement par le CPU, excepté dans certains cas.
- **Dans le cas du DMA générale** C'est ce registre qui peut paramétré le nombre d'octets à êtres transférer. Cependant, le nombre d'octet est limité à 64Ko ($0001 à $FFFF), le nombre $0000 signifie $10000 dans ce cas. Donc si vous mettez 0, vous aurez un déplacement de 64ko.

18.21 $43X8 $43X9

(X => numéro de canal : 0 à 7)

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
Table d'adresse A2 (octet Bas)								$42X8
A7	A6	A5	A4	A3	A2	A1	A0	

Table d'adresse A2 (octet Haut)								$42X9
A15	A14	A13	A12	A11	A10	A9	A8	

- C'est l'adresse qui est utilisé à accéder au CPU et la RAM. Il sera incrémenté automatiquement (voir l'Annexe CPU p2)
- La donnée de ce registre est utilisé comme une adresse basique, celle-ci est paramétré par "Table d'adresse A1" (registre $43X2). Ensuite, parce qu'il sera incrémenté (ou décrémenté) automatiquement, il est nécessaire de paramétré l'adresse dans ce registre par le CPU directement.
- **Pour le H-DMA Seulement** Cependant, la donnée qui est transféré, à besoin d'être changée de force. Cela peut être fait par le paramétrage de l'adresse mémoire du CPU, depuis ce registre . Dans un tel cas, l'adresse du CPU, qui est accédé actuellement, sera changé par la lecture de ce registre.

18.22 $43XA

(X => numéro de canal : 0 à 7)

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
Nombre de lignes							
Continue	L6	L5	L4	L3	L2	L1	L0

- Ce registre montre le nombre de lignes pour le transfert du H-DMA. (voir l'Annexe CPU p2)
- Ce nombre de lignes écrit dans la mémoire du CPU, sera le nombre basique de lignes. Il n'est pas nécessaire d'écrire la donnée dans ce registre directement par le CPU .

Sixième partie

LE COMPILATEUR WLA

Ici ont va pouvoir décortiquer les directives de compilation du WLA que Ville Helin a soigneusement développé depuis 1998 ! (balèze)

Je ne vais pas tout traduire car c'est assez complet, et assez ... long à vrai dire. Au fur et à mesure que j'avancerais dans la programmation, des subsections en français vont apparaitre.

Par contre je fait en sorte de classer toutes les directives par ordre alphabétique !

Chapitre 19

Directives du compilateur WLA

Here's the order in which the data is placed into the output :

1. Data and group 3 directives outside subsections.
2. Group 2 directives.
3. Data and group 3 directives inside subsections.
4. Group 1 directives.

Here are the supported directives (with examples) in WLA :

19.1 Group 1 :

- .COMPUTESNESCHECKSUM

19.2 Group 2 :

- .EMPTYFILL $C9
- .ENDEMUVECTOR
- .ENDNATIVEVECTOR
- .ENDSNES
- .EXPORT work_x
- .FASTROM
- .HIROM
- .LOROM
- .OUTNAME "other.o"
- .SLOWROM
- .SMC
- .SNESEMUVECTOR
- .SNESHEADER
- .SNESNATIVEVECTOR

19.3 Group 3 :

- .8BIT
- .16BIT
- .24BIT
- .ACCU 8
- .ASC "HELLO WORLD!"
- .ASCTABLE

- .ASCIITABLE
- .ASM
- .BACKGROUND ”parallax.gb”
- .BANK 0 SLOT 1
- .BASE $80
- .BLOCK ”Block1”
- .BR
- .BREAKPOINT
- .BYT 100, $30, %1000, ”HELLO WORLD!”
- .DB 100, $30, %1000, ”HELLO WORLD!”
- .DBCOS 0.2, 10, 3.2, 120, 1.3
- .DBM filtermacro 1, 2, ”encrypt me”
- .DBRND 20, 0, 10
- .DBSIN 0.2, 10, 3.2, 120, 1.3
- .DEFINE IF $FF0F
- .DEF IF $FF0F
- .DS 256, $10
- .DSB 256, $10
- .DSTRUCT
- .DSW 128, 20
- .DW 16000, 10, 255
- .DWCOS 0.2, 10, 3.2, 1024, 1.3
- .DWM filtermacro 1, 2, 3
- .DWRND 20, 0, 10
- .DWSIN 0.2, 10, 3.2, 1024, 1.3
- .ELSE
- .ENDASM
- .ENDB
- .ENDE
- .ENDIF
- .ENDM
- .ENDME
- .ENDR
- .ENDRO
- .ENDS
- .ENDST
- .ENUM $C000
- .EQU IF $FF0F
- .FAIL
- .FCLOSE FP_DATABIN
- .FOPEN ”data.bin” FP_DATABIN
- .FREAD FP_DATABIN DATA
- .FSIZE FP_DATABIN SIZE
- .IF DEBUG == 2
- .IFDEF IF
- .IFDEFM \2
- .IFEQ DEBUG 2
- .IFEXISTS ”main.s”
- .IFGR DEBUG 2
- .IFGREQ DEBUG 2
- .IFLE DEBUG 2

- .IFLEEQ DEBUG 2
- .IFNDEF IF
- .IFNDEFM \2
- .IFNEQ DEBUG 2
- .INCBIN ”sorority.bin”
- .INCDIR ”/usr/programming/gb/include/”
- .INCLUDE ”cgb_hardware.i”
- .INDEX 8
- .INPUT NAME
- .MACRO TEST
- .MEMORYMAP
- .ORG $150
- .ORGA $150
- .PRINTT ”Here we are...\n”
- .PRINTV DEC DEBUG+1
- .RAMSUBSECTION ”Vars” BANK 0 SLOT 1
- .REDEF IF $0F
- .REDEFINE IF $0F
- .REPEAT 6
- .REPT 6
- .ROMBANKMAP
- .ROMBANKS 2
- .ROMBANKSIZE $4000
- .SEED 123
- .SECTION ”Init” FORCE
- .SHIFT
- .SLOT 1
- .STRUCT enemy_object
- .SYM SAUSAGE
- .SYMBOL SAUSAGE
- .UNBACKGROUND $1000 $1FFF
- .UNDEFINE DEBUG
- .UNDEF DEBUG
- .WORD 16000, 10, 255

19.4 Descrition

.8BIT

There are a few mnemonics that look identical, but take different sized arguments. Here's a list of such 6502 mnemonics :

ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, ORA, ROL, SBC, STA, STX and STY.

For example :

```
1 LSR 11        ; $46 $0B
2 LSR $A000     ; $4E $00 $A0
3
4 ;The first one could also be
5
6 LSR 11        ; $4E $0B $00
```

.8BIT is here to help WLA to decide to choose which one of the opcodes it selects. When you give **.8BIT** (default) no 8bit address/value is expanded to 16bits.

By default WLA uses the smallest possible size. This is true also when WLA finds a computation it can't solve right away. WLA assumes the result will be inside the smallest possible bounds, which depends on the type of the mnemonic.

You can also use the fixed argument size versions of such mnemonics by giving the size with the operand (i.e., operand hinting). Here are few examples :

```
1 LSR 11.B    ; $46 $0B
2 LSR 11.W    ; $46 $0B $00
```

In WLA-65816 .ACCU/.INDEX/SEP/REP override .8BIT/.16BIT/.24BIT when considering the immediate values, so be careful. Still, operand hints override all of these, so use them to be sure.

This is not a compulsory directive.

.16BIT

Analogous to .8BIT. .16BIT forces all addresses and immediate values to be expanded into 16bit range, when possible, that is.

```
1 LSR 11          ; $46 $0B
```

that would be the case, normally, but after .16BIT it becomes

```
1 LSR 11          ; $4E $0B $00
```

This is not a compulsory directive.

.24BIT

Analogous to .8BIT and .16BIT. .24BIT forces all addresses to be expanded into 24bit range, when possible, that is.

```
1 AND $11          ; $25 $11
```

that would be the case, normally, but after .24BIT it becomes

```
1 AND $11          ; $2F $11 $00 $00
```

If it is not possible to expand the address into .24BIT range, then WLA tries to expand it into 16bit range.

This is not a compulsory directive.

.ACCU 8

Forces WLA to override the accumulator size given with SEP/REP. .ACCU doesn't produce any code, it only affects the way WLA interprets the immediate values (8 for 8 bit operands, 16 for 16 bit operands) for opcodes dealing with the accumulator.

So after giving .ACCU 8

```
1 AND #6
```

will produce $29 $06, and after giving .ACCU 16

```
1 AND #6
```

will yield $29 $00 $06.

Note that SEP/REP again will in turn reset the accumulator/index register size.

This is not a compulsory directive.

.ASCIITABLE

.ASCIITABLE's only purpose is to provide character mapping for .ASC.

Take a look at the example :

```
1 .ASCIITABLE
2 MAP "A" TO "Z" = 0
3 MAP "!" = 90
4 .ENDA
```

Here we set such a mapping that character 'A' is equal to 0, 'B' is equal to 1, 'C' is equal to 2, and so on, and '!' is equal to 90.

After you've given the .ASCIITABLE, use .ASC to define bytes using this mapping (.ASC is an alias for .DB, but with .ASCIITABLE mapping).

For example, .ASC "ABZ" would define bytes 0, 1 and 90.

Note that the following works as well :

```
1 .ASCIITABLE
2 MAP 'A' TO 'Z' = 0
3 MAP 65 = 90
4 .ENDA
```

Also note that the characters that are not given any mapping in .ASCIITABLE map to themselves (i.e., 'A' maps to 'A', etc.).

This is not a compulsory directive.

.ASCTABLE

.ASCTABLE is an alias for .ASCIITABLE.

This is not a compulsory directive.

.ASC ”HELLO WORLD !”

.ASC is an alias for .DB, but if you use .ASC it will remap the characters using the mapping given via .ASCII-TABLE.

This is not a compulsory directive.

.ASM

Tells WLA to start assembling. Use .ASM to continue the work which has been disabled with .ENDASM. .ASM and .ENDASM can be used to mask away big blocks of code. This is analogous to the ANSI C comments (/*...*/), but .ASM and .ENDASM can be nested, unlike the ANSI C counterpart.

This is not a compulsory directive.

.BACKGROUND ”parallax.gb”

This chooses an existing ROM image (parallax.gb in this case) as a background data for the project. You can overwrite the data with OVERWRITE subsections only, unless you first clear memory blocks with .UNBACKGROUND after which there's room for other subsections as well.

Note that .BACKGROUND can be used only when compiling an object file.

.BACKGROUND is useful if you wish to patch an existing ROM image with new code or data.

This is not a compulsory directive.

.BANK 0 SLOT 1

Defines the ROM bank and the slot it is inserted into in the memory. You can also type the following :

```
1 .BANK 0
```

This tells WLA to move into BANK 0 which will be put into the DEFAULTSLOT of .MEMORYMAP. This is a compulsory directive.

.BASE $80

Defines the base value for the 65816 CPU bank number (used only in 24bit addresses). Here are few examples of how to use .BASE (both examples assume the label resides in the first ROM bank) :

```
1 .BASE  $00
2   JSL  label   ; if label address is $1234, this will assemble into
3                ; JSL $001234
4 .BASE $80
5   JSL label    ; again, label is $1234, but this time the result will be
6                ; JSL $801234
7
8 .BASE defaults to $00.
```

Use .LOROM or .HIROM to define the ROM mode. Note that the address of the label will also contribute to the 65816 CPU bank number (CPU bank number == .BASE + CPU ROM bank of the label).

This is not a compulsory directive.

.BLOCK ”Block1”

Begins a block (called ”Block1” in the example). These blocks have only one function : to display the number of bytes they contain. When you embed such a block into your code, WLA displays its size when it assembles the source file.

Use .ENDB to terminate a .BLOCK. Note that you can nest .BLOCKs.

This is not a compulsory directive.

.BR

Inserts a breakpoint that behaves like a .SYM without a name. Breakpoints can only be seen in WLALINK's symbol file.

This is not a compulsory directive.

.BREAKPOINT

.BREAKPOINT is an alias for .BR.

This is not a compulsory directive.

.BYT 100, $30, %1000, "HELLO WORLD !"

.BYT is an alias for .DB.

This is not a compulsory directive.

.COMPUTESNESCHECKSUM

When this directive is used WLA computes the SNES ROM checksum and inverse checksum found at $7FDC-$7FDF (LoROM) or $FFDC$FFDF (HiROM).

Note that this directive can only be used with wla-65816. Also note that the ROM size must be at least 32KB for LoROM images and 64KB for HiROM images.

.LOROM or .HIROM must be issued before .COMPUTESNESCHECKSUM.

This is not a compulsory directive.

.DB 100, $30, %1000, "HELLO WORLD !"

Defines bytes.

This is not a compulsory directive.

.DBCOS 0.2, 10, 3.2, 120, 1.3

Defines bytes just like .DSB does, only this time they are filled with cosine data. .DBCOS takes five arguments.

The first argument is the starting angle. Angle value ranges from 0 to 359.999..., but you can supply WLA with values that are out of the range - WLA fixes them ok. The value can be integer or float.

The second one descibes the amount of additional angles. The example will define 11 angles.

The third one is the adder value which is added to the angle value when next angle is calculated. The value can be integer or float.

The fourth and fifth ones can be seen from the pseudo code below, which also describes how .DBCOS works. The values can be integer or float.

Remember that cos (and sin) here returns values ranging from -1 to 1.

.DBCOS A, B, C, D, E

```
1 for (B++; B > 0; B--) {
2     output_data((D * cos(A)) + E)
3     A = keep_in_range(A + C)
4 }
```

This is not a compulsory directive.

.DBM filtermacro 1, 2, ”encrypt me”

Defines bytes using a filter macro. All the data is passed to ”filtermacro” in the first argument, one byte at a time, and the byte that actually gets defined is the value of definition ”_OUT” (”_out” works as well). The second macro argument holds the offset from the beginning (the first byte) in bytes (the series being 0, 1, 2, 3, ...).

Here’s an example of a filter macro that increments all the bytes by one :

```
1 .MACRO increment
2 .REDEFINE _out \1+1
3 .ENDM
```

This is not a compulsory directive.

.DBRND 20, 0, 10

Defines bytes, just like .DSB does, only this time they are filled with (pseudo) random numbers. We use the integrated Mersenne Twister to generate the random numbers. If you want to seed the random number generator, use .SEED.

The first parameter (20 in the example) defines the number of random numbers we want to generate. The next two tell the range of the random numbers, i.e. min and max.

Here's how it works :

```
1 .DBRND A, B, C
2
3 for (i = 0; i $<$ A; i++)
4     output_data((rand() % (C-B+1)) + B);
```

This is not a compulsory directive.

.DBSIN 0.2, 10, 3.2, 120, 1.3

Analogous to .DBCOS, but does sin() instead of cos().

This is not a compulsory directive.

.DEFINE IF $FF0F

Assigns a number or a string to a definition label.

By default all defines are local to the file where they are presented. If you want to make the definition visible to all the files in the project, use .EXPORT.

Here are some examples :

```
1 .DEFINE X 1000
2 .DEFINE FILE "level01.bin"
3 .DEFINE TXT1 "hello and welcome", 1, "to a new world...", 0
4 .DEFINE BYTES 1, 2, 3, 4, 5
5 .DEFINE COMPUTATION X+1
6 .DEFINE DEFAULTV
```

All definitions with multiple values are marked as data strings, and .DB is about the only place where you can later on use them.

```
1 .DEFINE BYTES 1, 2, 3, 4, 5
2 .DB 0, BYTES, 6
```

is the same as

```
1 .DB 0, 1, 2, 3, 4, 5, 6
```

If you omit the definition value (in our example "DEFAULTV"), WLA will default to 0.

Note that you must do your definition before you use it, otherwise WLA will use the final value of the definition. Here's an example of this :

```
1 .DEFINE AAA 10
2 .DB AAA               ; will be 10.
3 .REDEFINE AAA 11
```

but

```
1 .DB AAA               ; will be 11.
2 .DEFINE AAA 10
3 .REDEFINE AAA 11
```

You can also create definitions on the command line. Here's an example of this :

wla-gb -vl -DMOON -DNAME=john -DPRICE=100 -DADDRESS=$100 math.s

MOON's value will be 0, NAME is a string definition with value "john", PRICE's value will be 100, and ADDRESS's value will be $100.

Note that

```
1 .DEFINE AAA = 10    ; the same as ".DEFINE AAA 10".
```

works as well.

This is not a compulsory directive.

.DEF IF $FF0F

.DEF is an alias for .DEFINE.

This is not a compulsory directive.

.DS 256, $10

.DS is an alias for .DSB.

This is not a compulsory directive.

.DSB 256, $10

Defines 256 bytes of $10.

This is not a compulsory directive.

.DSTRUCT

.DSTRUCT WATERDROP INSTANCEOF WATER DATA "TINGLE", 40, 120

Defines an instance of struct water, called waterdrop, and fills it with the given data. Before calling .DSTRUCT we must have defined the structure, and in this example it could be like :

```
1 .STRUCT water
2 name   ds 8
3 age    db
4 weight dw
5 .ENDST
```

Note that the keywords INSTANCEOF and DATA are optional, so

```
1 .DSTRUCT waterdrop, water, "tingle", 40, 120
```

also works. And one can define instances without supplying values to all struct members :

```
1 .DSTRUCT waterdrop, water, "somedrop"
```

Note that WLA fills the missing bytes with the data defined with .EMPTYFILL, or $00 if no .EMPTYFILL has been issued.

In this example you would also get the following labels :

```
1 waterdrop
2 waterdrop.name
3 waterdrop.age
4 waterdrop.weight
```

This is not a compulsory directive.

.DSW 128, 20

Defines 128 words (two bytes) of 20.

This is not a compulsory directive.

.DW 16000, 10, 255

Defines words (two bytes each). .DW takes only numbers and characters as input, not strings.

This is not a compulsory directive.

.DWCOS 0.2, 10, 3.2, 1024, 1.3

Analogous to .DBCOS (but defines words).

This is not a compulsory directive.

.DWM filtermacro 1, 2, 3

Defines 16bit words using a filter macro. All the data is passed to "filtermacro" in the first argument, one word at a time, and the word that actually gets defined is the value of definition "_OUT" ("_out" works as well). The second macro argument holds the offset from the beginning (the first word) in bytes (the series being 0, 2, 4, 6, ...).

Here's an example of a filter macro that increments all the words by one :

```
1 .MACRO increment
2 .REDEFINE _out \1+1
3 .ENDM
```

This is not a compulsory directive.

.DWRND 20, 0, 10

Analogous to .DBRND (but defines words).

This is not a compulsory directive.

.DWSIN 0.2, 10, 3.2, 1024, 1.3

Analogous to .DBCOS (but defines words and does sin() instead of cos()).

This is not a compulsory directive.

.ELSE

If the previous .IFxxx failed then the following text until .ENDIF is acknowledged.

This is not a compulsory directive.

.EMPTYFILL $C9

This byte is used in filling the unused areas of the ROM file. EMPTYFILL defaults to $00.

This is not a compulsory directive.

.ENDASM

Tells WLA to stop assembling. Use .ASM to continue the work.

This is not a compulsory directive.

.ENDB

Terminates .BLOCK.

This is not a compulsory directive, but when .BLOCK is used this one is required to terminate it.

.ENDE

Ends the enumeration.

This is not a compulsory directive, but when .ENUM is used this one is required to terminate it.

.ENDEMUVECTOR

Ends definition of the emulation mode interrupt vector table.

This is not a compulsory directive, but when .SNESEMUVECTOR is used this one is required to terminate it.

.ENDIF

This terminates any .IFxxx directive.

This is not a compulsory directive, but if you use any .IFxxx then you need also to apply this.

.ENDM

Ends a .MACRO.

This is not a compulsory directive, but when .MACRO is used this one is required to terminate it.

.ENDME

Terminates .MEMORYMAP.

This is not a compulsory directive, but when .MEMORYMAP is used this one is required to terminate it.

.ENDNATIVEVECTOR

Ends definition of the native mode interrupt vector table.

This is not a compulsory directive, but when .SNESNATIVEVECTOR is used this one is required to terminate it.

.ENDR

Ends the repetition.

This is not a compulsory directive, but when .REPEAT is used this one is required to terminate it.

.ENDRO

Ends the rom bank map.

This is not a compulsory directive, but when .ROMBANKMAP is used this one is required to terminate it.

.ENDS

Ends the subsection.

This is not a compulsory directive, but when .SECTION is used this one is required to terminate it.

.ENDSNES

This ends the SNES header definition.

This is not a compulsory directive, but when .SNESHEADER is used this one is required to terminate it.

.ENDST

Ends the structure definition.

This is not a compulsory directive, but when .STRUCT is used this one is required to terminate it.

.ENUM $C000

Starts enumeration from $C000. Very useful for defining variables.

To start a descending enumeration, put ”DESC” after the starting address. WLA defaults to ”ASC” (ascending enumeration).

You can also add ”EXPORT” after these if you want to export all the generated definitions automatically.

Here’s an example of .ENUM :

.EXPORT work_x

Exports the definition "work_x" to outside world. Exported definitions are visible to all object files and libraries in the linking procedure. Note that you can only export value definitions, not string definitions.

You can export as many definitions as you wish with one .EXPORT :

```
1  .EXPORT NUMBER, NAME, ADDRESS, COUNTRY
2  .EXPORT NAME, AGE
```

This is not a compulsory directive.

```
1  ...
2  .STRUCT mon                  ; check out the documentation on
3  name ds 2                    ; .STRUCT
4  age  db
5  .ENDST
6  .ENUM $A000
7  _scroll_x DB                 ; db  - define byte (byt and byte work also)
8  _scroll_y DB
9  player_x: DW                 ; dw  - define word (word works also)
10 player_y: DW
11 map_01:   DS  16             ; ds  - define size (bytes)
12 map_02    DSB 16             ; dsb - define size (bytes)
13 map_03    DSW  8             ; dsw - define size (words)
14 monster   INSTANCEOF mon 3   ; three instances of structure mon
15 dragon    INSTANCEOF mon     ; one mon
16 .ENDE
17 ...
```

Previous example transforms into following definitions :

```
1  .DEFINE _scroll_x       $A000
2  .DEFINE _scroll_y       $A001
3  .DEFINE player_x        $A002
4  .DEFINE player_y        $A004
5  .DEFINE map_01          $A006
6  .DEFINE map_02          $A016
7  .DEFINE map_03          $A026
8  .DEFINE monster         $A036
9  .DEFINE monster.name    $A036
10 .DEFINE monster.age     $A038
11 .DEFINE monster.1       $A036
12 .DEFINE monster.1.name  $A036
13 .DEFINE monster.1.age   $A038
14 .DEFINE monster.2       $A039
15 .DEFINE monster.2.name  $A039
16 .DEFINE monster.2.age   $A03B
17 .DEFINE monster.3       $A03C
18 .DEFINE monster.3.name  $A03C
19 .DEFINE monster.3.age   $A03E
20 .DEFINE dragon          $A03F
21 .DEFINE dragon.name     $A03F
22 .DEFINE dragon.age      $A041
```

DB, DW, DS, DSB, DSW and INSTANCEOF can also be in lowercase. You can also use a dotted version of the symbols, but it doesn't advance the memory address. Here's an exmple :

```
1  .ENUM $C000 DESC EXPORT
2  bigapple_h db
3  bigapple_l db
4  bigapple:  .dw
```

```
5 .ENDE
```

And this is what is generated :

```
1 .DEFINE bigapple_h $BFFF
2 .DEFINE bigapple_l $BFFE
3 .DEFINE bigapple   $BFFE
4 .EXPORT bigapple, bigapple_l, bigapple_h
```

This way you can generate a 16bit variable address along with pointers to its parts.

If you want more flexible variable positioning, take a look at .RAMsubsections.

This is not a compulsory directive.

.EQU IF $FF0F

.EQU is an alias for .DEFINE.

This is not a compulsory directive.

.FAIL

Terminates the compiling process.

This is not a compulsory directive.

.FASTROM

Sets the ROM memory speed bit in $FFD5 (.HIROM) or $7FD5 (.LOROM) to indicate that the SNES ROM chips are 120ns chips.

This is not a compulsory directive.

.FCLOSE FP_DATABIN

Closes the filehandle FP_DATABIN.

This is not a compulsory directive.

.FOPEN ”data.bin” FP_DATABIN

Opens the file ”data.bin” for reading and associates the filehandle with name ”FP_DATABIN”.

This is not a compulsory directive.

.FREAD FP_DATABIN DATA

Reads one byte from ”FP_DATABIN” and creates a definition called ”DATA” to hold it. ”DATA” is an ordinary definition label, so you can .UNDEFINE it.

Here’s an example on how to use .FREAD :

```
1 .fopen "data.bin" fp
2 .fsize fp t
3 .repeat t
4 .fread fp d
5 .db d+26
6 .endr
7 .undefine t, d
```

This is not a compulsory directive.

.FSIZE FP_DATABIN SIZE

Creates a definition called ”SIZE”, which holds the size of the file associated with the filehandle ”FP_DATABIN”. ”SIZE” is an ordinary definition label, so you can .UNDEFINE it.

This is not a compulsory directive.

.HIROM

With this directive you can define the SNES ROM mode to be HiROM. Issuing .HIROM will override the user's ROM bank map when WLALINK computes 24bit addresses and bank references. If no .HIROM or .LOROM are given then WLALINK obeys the banking defined in .ROMBANKMAP.

.HIROM also sets the ROM mode bit in $FFD5.

This is not a compulsory directive.

.IF DEBUG == 2

If the condition is fulfilled the following piece of code is acknowledged until .ENDIF/.ELSE occurs in the text, otherwise it is skipped. Operands must be immediate values or strings.

The following operators are supported :

```
1 <  - less than
2 <= - less or equal to
3 >  - greater than
4 >= - greater or equal to
5 == - equals to
6 != - doesn't equal to
```

All IF (yes, including .IFDEF, .IFNDEF, etc) directives can be nested.

This is not a compulsory directive.

.IFDEF IF

If "IF" is defined, then the following piece of code is acknowledged until .ENDIF/.ELSE occurs in the text, otherwise it is skipped.

This is not a compulsory directive.

.IFDEFM \2

If the specified argument is defined (argument number two, in the example), then the following piece of code is acknowledged until .ENDIF/.ELSE occurs in the macro, otherwise it is skipped.

This is not a compulsory directive. .IFDEFM works only inside a macro.

.IFEQ DEBUG 2

If the value of DEBUG equals to 2, then the following piece of code is acknowledged until .ENDIF/.ELSE occurs in the text, otherwise it is skipped. Both arguments can be computations, defines or immediate values.

This is not a compulsory directive.

.IFEXISTS ”main.s”

If ”main.s” file can be found, then the following piece of code is acknowledged until .ENDIF/.LESE occurs in the text, otherwise it is skipped.

By writing the following few lines you can include a file if it exists without breaking the compiling loop if it doesn't exist.

```
1 .IFEXISTS  FILE
2 .INCLUDE  FILE
3 .ENDIF
```

This is not a compulsory directive.

.IFGR DEBUG 2

If the value of DEBUG is greater than 2, then the following piece of code is acknowledged until .ENDIF/.ELSE occurs in the text, otherwise it is skipped. Both arguments can be computations, defines or immediate values.

This is not a compulsory directive.

.IFGREQ DEBUG 2

If the value of DEBUG is greater or equal to 2, then the following piece of code is acknowledged until .ENDIF/.ELSE occurs in the text, otherwise it is skipped. Both arguments can be computations, defines or immediate values.

This is not a compulsory directive.

.IFLE DEBUG 2

If the value of DEBUG is less than 2, then the following piece of code is acknowledged until .ENDIF/.ELSE occurs in the text, otherwise it is skipped. Both arguments can be computations, defines or immediate values.

This is not a compulsory directive.

.IFLEEQ DEBUG 2

If the value of DEBUG is less or equal to 2, then the following piece of code is acknowledged until .ENDIF/.ELSE occurs in the text, otherwise it is skipped. Both arguments can be computations, defines or immediate values.

This is not a compulsory directive.

.IFNDEF IF

If "IF" is not defined, then the following piece of code is acknowledged until .ENDIF/.ELSE occurs in the text, otherwise it is skipped.

This is not a compulsory directive.

.IFNDEFM \2

If the specified argument is not defined, then the following piece of code is acknowledged until .ENDIF/.ELSE occurs in the macro, otherwise it is skipped.

This is not a compulsory directive. .IFNDEFM works only inside a macro.

.IFNEQ DEBUG 2

If the value of DEBUG doesn't equal to 2, then the following piece of code is acknowledged until .ENDIF/.ELSE occurs in the text, otherwise it is skipped. Both arguments can be computations, defines or immediate values.

This is not a compulsory directive.

.INCBIN ”sorority.bin”

Includes the specified data file into the source file. .INCBIN caches all files into memory, so you can .INCBIN any data file millions of times, but it is loaded from hard drive only once.

You can optionally use SWAP after the file name, e.g.,

```
1 .INCBIN "kitten.bin" SWAP
```

.INCBIN data is divided into blocks of two bytes, and inside every block the bytes are exchanged (like ”SWAP r” does to nibbles). This requires that the size of the file is even.

You can also force WLA to skip n bytes from the beginning of the file by writing for example :

```
1 .INCBIN "kitten.bin" SKIP 4
```

Four bytes are skipped from the beginning of kitten.bin and the rest is incbinned.

It is also possible to incbin only n bytes from a file :

```
1 .INCBIN "kitten.bin" READ 10
```

Will read ten bytes from the beginning of kitten.bin.

You can also force WLA to create a definition holding the size of the file :

```
1 .INCBIN "kitten.bin" FSIZE size_of_kitten
```

Want to circulate all the included bytes through a filter macro? Do this :

```
1 .INCBIN "kitten.bin" FILTER filtermacro
```

The filter macro is executed for each byte of the included data, data byte being the first argument, and offset from the beginning being the second parameter, just like in the case of .DBM and .DWM.

And you can combine all these four commands :

```
1 .INCBIN "kitten.bin" SKIP 10 READ 8 SWAP FSIZE size_of_kitten FILTER filtermacro
```

This example shows how to incbin eight bytes (swapped) after skipping 10 bytes from the beginning of file ”kitten.bin”, and how to get the size of the file into a definition label ”size_of_kitten”. All the data bytes are circulated through a filter macro.

Note that the order of the extra commands is important.

If the file’s not found in the .INCDIR directory, WLA tries to find it in the current working directory.

This is not a compulsory directive.

.INCDIR "/usr/programming/gb/include/"

Changes the current include root directory. Use this to specify main directory for the following .INCLUDE and .INCBIN directives. If you want to change to the current working directory (WLA also defaults to this), use

```
1 .INCDIR ""
```

This is not a compulsory directive.

.INCLUDE ”cgb_hardware.i”

Includes the specified file to the source file. If the file's not found in the .INCDIR directory, WLA tries to find it in the current working directory.

This is not a compulsory directive.

.INDEX 8

Forces WLA to override the index (X/Y) register size given with SEP/REP. .INDEX doesn't produce any code, it only affects the way WLA interprets the immediate values (8 for 8 bit operands, 16 for 16 bit operands) for opcodes dealing with the index registers.

So after giving .INDEX 8

```
1   CPX #10
```

will produce $E0 $A0, and after giving .INDEX 16

```
1   CPX #10
```

will yield $E0 $00 $A0.

Note that SEP/REP again will in turn reset the accumulator/index register size.

This is not a compulsory directive.

.INPUT NAME

.INPUT is much like any Basic-language input : .INPUT asks the user for a value or string. After .INPUT is the variable name used to store the data.

.INPUT works like .REDEFINE, but the user gets to type in the data.

Here are few examples how to use input :

```
1  .PRINTT "The name of the ROM? "
2  .INPUT NAME
3  .NAME NAME
4  ...
5  .PRINTT "Give the .DB amount.\n"
6  .INPUT S
7  .PRINTT "Give .DB data one at a time.\n"
8  .REPEAT S
9      .INPUT B
10     .DB B
11 .ENDR
12 ...
```

This is not a compulsory directive.

.LOROM

With this directive you can define the SNES ROM mode to be LoROM. Issuing .LOROM will override the user's ROM bank map when WLALINK computes 24bit addresses and bank references. If no .HIROM or .LOROM are given then WLALINK obeys the banking defined in .ROMBANKMAP.

WLA defaults to .LOROM.

This is not a compulsory directive.

.MACRO TEST

Begins a macro called 'TEST'.

You can use '' inside a macro to e.g., separate a label from the other macro 'TEST' occurrences. '' is replaced with an integer number indicating the amount of times the macro has been called previously so it is unique to every macro call. '' can also be used inside strings inside a macro or just as a plain value. Look at the following examples for more information.

You can also type '\!' to get the name of the source file currently being parsed.

Also, if you want to use macro arguments in e.g., calculation, you can type '\X' where X is the number of the argument. Another way to refer to the arguments is to use their names given in the definition of the macro (see the examples for this).

Remember to use .ENDM to finish the macro definition. Note that you cannot use .INCLUDE inside a macro. Note that WLA's macros are in fact more like procedures than real macros, because WLA doesn't substitute macro calls with macro data. Instead WLA jumps to the macro when it encounters a macro call at compile time.

You can call macros from inside a macro. Note that the preprocessor does not expand the macros. WLA traverses through the code according to the macro calls, so macros really define a very simple programming language.

Here are some examples :

```
1  .MACRO  NOPMONSTER
2      .REPT 32      ; evil...
3      NOP
4      .ENDR
5  .ENDM
6
7  .MACRO LOAD_ABCD
8      LD A, \1
9      LD B, \1
10     LD C, \1
11     LD D, \1
12     NOPMONSTER
13     LD HL, 1$<<$\1
14 .INCBIN \1
15
16 .ENDM
17
18 .MACRO QUEEN
19
20 QUEEN@:
21     LD    A, \1
22     LD    B, \1
23 CALL QUEEN@
24     .DB   "@", 0       ; will translate into a zero terminated string
25     ; holding the amount of macro QUEEN calls.
26     .DB   "@", 0; will translate into a string containing
27     ; @.
28     .DB   @; will translate into a number indicating
29     ; the amount of macro QUEEN calls.
30     .ENDM
31
32 .MACRO LOAD_ABCD_2 ARGS ONE, TWO, THREE, FOUR, FIVE
33               LD A, ONE
34               LD B, TWO
35               LD C, THREE
36               LD D, FOUR
37               NOPMONSTER
38               LD HL, 1$<<$ONE
39 .INCBIN FIVE
```

```
40 .ENDM
41
42 .MACRO TEST NARGS 3
43                 .DB \1, \1, \1
44 .ENDM
```

And here's how they can be used :

```
1 NOPMONSTER
2 LOAD_ABCD $10, $20, $30, XYZ, "merman.bin"
3 QUEEN 123
4 LOAD_ABCD_2 $10, $20, $30, XYZ, "merman.bin"
5 TEST 1, 2, 3vspace{1ex}
```

Note that you must separate the arguments with commas.

If you want to give names to the macro's arguments you can do that by listing them in order after supplying ARGS after the macro's name.

Every time a macro is called a definition NARGS is created. It shows only inside the macro and holds the number of arguments the macro was called with. So don't have your own definition called NARGS. Here's an example :

```
1 .MACRO LUPIN
2       .IF NARGS != 1
3           .FAIL
4       .ENDIF
5
6       .PRINTT "Totsan! Ogenki ka?\n"
7 .ENDM
```

This is not a compulsory directive.

.MEMORYMAP

Begins the memory map definition. Using .MEMORYMAP you must first describe the target system's memory architecture to WLA before it can start to compile the code. .MEMORYMAP gives you the freedom to use WLA Z80/6502/65C02/6510/65816/HUC6280/SPC-700 to compile data for numerous different real Z80/6502/65C02/6510/65816/HUC6280/SPC-700 based systems.

Examples :

```
.MEMORYMAP
      DEFAULTSLOT 0
      SLOTSIZE $4000
      SLOT 0 $0000
      SLOT 1 $4000
.ENDME

.MEMORYMAP
      DEFAULTSLOT 0
      SLOT 0 $0000 $4000
      SLOT 1 $4000 $4000
.ENDME
.MEMORYMAP
DEFAULTSLOT 0
SLOT 0 START $0000 SIZE $4000
SLOT 1 START $4000 SIZE $4000
.ENDME

.MEMORYMAP
DEFAULTSLOT 1
SLOTSIZE $6000
SLOT 0 $0000
SLOTSIZE $2000
SLOT 1 $6000
SLOT 2 $8000
.ENDME
```

Here's a real life example from Adam Klotblixt. It should be interesting for all the ZX81 coders :

```
...
.MEMORYMAP
DEFAULTSLOT 1
SLOTSIZE $2000
SLOT 0 $0000
SLOTSIZE $6000
SLOT 1 $2000
.ENDME

.ROMBANKMAP
BANKSTOTAL 2
BANKSIZE $2000
BANKS 1
BANKSIZE $6000
BANKS 1
.ENDROvspace{1ex}

.BANK 1 SLOT 1
.ORGA $2000
...
```

SLOTSIZE defines the size of the following slots, unless you explicitly specify the size of the slot, like in the second and third examples. You can redefine SLOTSIZE as many times as you wish.

DEFAULTSLOT describes the default slot for banks which aren't explicitly inserted anywhere. Check

.BANK definition for more information.

SLOT defines a slot and its starting address. SLOT numbering starts at 0 and ends to 255 so you have 256 slots at your disposal.

This is a compulsory directive, and make sure all the object files share the same .MEMORYMAP or you can't link them together.

Note that both START and SIZE are optional!

.ORG $150

Defines the starting address. The value supplied here is relative to the ROM bank given with .BANK.

This is a compulsory directive.

.ORGA $150

Defines the starting address. The value supplied here is absolute and used directly in address computations. WLA computes the right position in ROM file. By using .ORGA you can instantly see from the source file where the following code is located in the 16bit memory.

Here's an example :

```
1  .MEMORYMAP
2        SLOTSIZE $4000
3        DEFAULTSLOT 0
4        SLOT 0 $0000
5        SLOT 1 $4000
6  .ENDMEvspace{1ex}
7
8  .ROMBANKMAP
9  BANKSTOTAL 2
10 BANKSIZE $4000
11 BANKS 2
12 .ENDRO
13
14 .BANK 0 SLOT 1
15 .ORGA $4000
16
17 MAIN:     JP   MAIN
```

Here "MAIN" is at $0000 in the ROM file, but the address for "MAIN" is $4000.

This is a compulsory directive.

.OUTNAME ”other.o”

Changes the name of the output file. Here’s and example :

```
1 wla-gb -o test.s
```

would normally output ”test.o”, but if you had written

```
1 .OUTNAME "new.o"
```

somewhere in the code WLA would write the output to new.o instead.

This is not a compulsory directive.

.PRINTT ”Here we are...\n”

Prints the given text into stdout. Good for debugging stuff. PRINTT takes only a string as argument, and the only supported formatting symbol is ’\n’ (line feed).

This is not a compulsory directive.

.PRINTV DEC DEBUG+1

Prints the value of the supplied definition or computation into stdout.

Computation must be solvable at the time of printing (just like definitions values). PRINTV takes two parameters. The first describes the type of the print output. ”DEC” means decimal, ”HEX” means hexadecimal.

Use PRINTV with PRINTT as PRINTV doesn't print linefeeds, only the result. Here's an example :

```
1 .PRINTT "Value of "DEBUG" = $"
2 .PRINTV HEX DEBUG
3 .PRINTT "\n"
```

This is not a compulsory directive.

.RAMSUBSECTION "Vars" BANK 0 SLOT 1

RAMsubsections accept only variable labels and variable sizes, and the syntax to define these is identical to .ENUM (all the syntax rules that apply to .ENUM apply also to .RAMsubsection). Additionally you can embed structures (.STRUCT) into a RAMsubsection. Here's an example :

```
.RAMsubsection "Some of my variables" BANK 0 SLOT 1
vbi_counter:    db
player_lives:   db
.ENDS
```

RAMsubsections behave like FREE subsections, but instead of filling any banks RAM subsections will occupy area inside slots. You can fill different slots with different variable labels. It's recommend that you create separate slots for holding variables (as ROM and RAM don't usually overlap).

Here's another example :

```
.MEMORYMAP
SLOTSIZE $4000
DEFAULTSLOT 0
SLOT 0 $0000    ; ROM slot 0.
SLOT 1 $4000    ; ROM slot 1.
SLOT 2 $A000    ; variable RAM is here!
.ENDME

.STRUCT game_object
x DB
y DB
.ENDST

.RAMsubsection "vars 1" BANK 1 SLOT 2
moomin      DW
phantom     DB
nyanko      DB
enemy       INSTANCEOF game_object
.ENDS
```

If no other RAM subsection is used, then this is what you will get :

```
.DEFINE moomin  $A000
.DEFINE phantom $A002
.DEFINE nyanko  $A003
.DEFINE enemy   $A004
.DEFINE enemy.x $A004
.DEFINE enemy.y $A005
```

Note that the BANK value is only used when referring labels using notation " :label". BANK in .RAMsubsection is optional so you can leave it away if you think you don't need to know the bank number for a label inside a RAM subsection.

This is not a compulsory directive.

.REDEF IF $0F

.REDEF is an alias for .REDEFINE.

This is not a compulsory directive.

.REDEFINE IF $0F

Assigns a new value or a string to an old definition. If the definition doesn't exist, .REDEFINE performs .DEFINE's work.

When used with .REPT REDEFINE helps creating tables :

```
1 .DEFINE CNT 0
2
3 .REPT 256
4 .DB CNT
5 .REDEFINE CNT CNT+1
6 .ENDR
```

This is not a compulsory directive.

.REPEAT 6

Repeats the text enclosed between ".REPEAT x" and ".ENDR" x times (6 in this example). You can use .REPEATs inside .REPEATs. 'x' must be >= 0.

This is not a compulsory directive.

.REPT 6

.REPT is an alias for .REPEAT.

This is not a compulsory directive.

.ROMBANKMAP

Begins the ROM bank map definition. You can use this directive to describe the project's ROM banks. Use .ROMBANKMAP when not all the ROM banks are of equal size.

Note that you can use .ROMBANKSIZE and .ROMBANKS instead of .ROMBANKMAP, but that's only when the ROM banks are equal in size.

Some systems based on a real Z80 chip, 6502/65C02/6510/65816/HUC6280/SPC-700 CPUs and Pocket Voice cartridges for Game Boy require the usage of this directive.

Examples :

```
1  .ROMBANKMAP
2        BANKSTOTAL 16
3        BANKSIZE $4000
4        BANKS 16
5  .ENDRO
6
7  .ROMBANKMAP
8        BANKSTOTAL 510
9        BANKSIZE $6000
10       BANKS 1
11       BANKSIZE $2000
12       BANKS 509
13 .ENDRO
```

The first one describes an ordinary ROM image of 16 equal sized banks. The second one defines a 4MB Pocket Voice ROM image. In the PV ROM image the first bank is $6000 bytes and the remaining 509 banks are smaller ones, $2000 bytes each.

BANKSTOTAL tells the total amount of ROM banks. It must be defined prior to anything else.

BANKSIZE tells the size of the following ROM banks. You can supply WLA with BANKSIZE as many times as you wish.

BANKS tells the amount of banks that follow and that are of the size BANKSIZE which has been previously defined.

This is not a compulsory directive when .ROMBANKSIZE and .ROMBANKS are defined.

You can redefine .ROMBANKMAP as many times as you wish as long as the old and the new ROM bank maps match as much as possible. This way you can enlarge the size of the project on the fly.

.ROMBANKS 2

Indicates the size of the ROM in rombanks. This value is converted to a standard GB ROM size indicator value found at $148 in a GB ROM, and there this one is put into.

This is a compulsory directive unless .ROMBANKMAP is defined.

You can redefine .ROMBANKS as many times as you wish as long as the old and the new ROM bank maps match as much as possible. This way you can enlarge the size of the project on the fly.

.ROMBANKSIZE $4000

Defines the ROM bank size. Old syntax is ".BANKSIZE x".

This is a compulsory directive unless .ROMBANKMAP is defined.

.SECTION ”Init” FORCE

Section is a continuous area of data which is placed into the output file according to the section type and .BANK and .ORG directive values.

The example begins a section called ”Init”. Before a section can be declared, .BANK and .ORG must be used unless WLA is in library file output mode. Library file's sections must all be FREE ones. .BANK tells the bank number where this section will be later relocated into. .ORG tells the offset for the relocation from the beginning of .BANK.

You can supply the preferred section size (bytes) inside the section name string. Here's an example :

```
1 .SECTION "Init_100" FREE
```

will create a section (”Init”) with size of 100 bytes, unless the actual data overflows from the section, in which case the section size is enlarged to contain all the data. Note that the syntax for explicit section size defining is : ”NAME_X”, where ”NAME” is the name of the section and ”X” is the size (decimal or hexadecimal value).

You can also give the size of the section the following way :

```
1 .SECTION "Init" SIZE 100 FREE
```

It's possible to force WLALINK to align the FREE, SEMIFREE and SUPERFREE sections by giving the alignment as follows :

```
1 .SECTION "Init" SIZE 100 ALIGN 4 FREE
```

And if you want that WLA returns the ORG to what it was before issuing the section, put RETURNORG at the end of the parameter list :

```
1 .SECTION "Init" SIZE 100 ALIGN 4 FREE RETURNORG
```

By default WLA advances the ORG, so, for example, if your ORG was $0 before a section of 16 bytes, then the ORG will be 16 after the section.

Note also that if your section name begins with double underlines (e.g., ” __UNIQUE_SECTION!!! ”) the section will be unique in the sense that when WLALINK recieves files containing sections which share the same name, WLALINK will save only the first of them for further processing, all others are deleted from memory with corresponding labels, references and calculations.

If a section name begins with an exclamation mark (’!’) it tells WLALINK to not to drop it, even if you use WLALINK's ability to discard all unreferenced sections and there are no references to the section.

FORCE after the name of the section tells WLA that the section _must_ be inserted so it starts at .ORG. FORCE can be re-placed with FREE which means that the sec-tion can be inserted somewhere in the defined bank, where there is room. You can also use OVERWRITE to insert the section into the memory regardless of data collisions. Using OVERWRITE you can easily patch an existing ROM image just by .BACKGROUND'ing the ROM image and inserting OVERWRITE sections into it. SEMIFREE sections are also possible and they behave much like FREE sections. The only difference is that they are positioned somewhere in the bank starting from .ORG. SUPERFREE sections are also available, and they will be positioned into the first suitable place inside the first suitable bank (candidates for these suitable banks have the same size with the slot of the section, no other banks are considered). You can also leave away the type specifier as the default type for the section is FREE.

You can name the sections as you wish, but there is one special name.
A section called ”BANKHEADER” is placed in the front of the bank where it is defined. These sections contain data that is not in the memory map of the machine, so you can't refer to the data of a BANKHEADER section, but you can write references to outside. So no labels inside BANKHEADER sections. These special sections are useful when writing e.g., MSX programs. Note that library files don't take BANKHEADER sections.

Here's an example of a ”BANKHEADER” section :

```
1 .BANK 0
2 .ORG 0
3 .SECTION "BANKHEADER"
```

```
4       .DW MAIN
5       .DW VBI
6   .ENDS
7
8   .SECTION "Program"
9   MAIN:       CALL    MONTY_ON_THE_RUN
10  VBI:        PUSH    HL
11          ...
12      POP HL
13      RETI
14  .ENDS
```

Here's an example of an ordinary section :

```
1   .BANK 0
2   .ORG $150
3   .SECTION "Init" FREE
4       DI
5       LD  SP, $FFFE
6       SUB A
7       LD  ($FF00+R_IE), A
8   .ENDS
```

This tells WLA that a FREE section called "Init" must be located somewhere in bank 0. If you replace FREE with SEMIFREE the section will be inserted somewhere in the bank 0, but not in the $0-$14F area. If you replace FREE with SUPERFREE the section will be inserted somewhere in the

Here's the order in which WLA writes the sections :

1. FORCE
2. SEMIFREE & FREE
3. SUPERFREE
4. OVERWRITE

Before the sections are inserted into the output file, they are sorted by size, so that the biggest section gets processed first and the smallest last.

You can also create a RAM section. For more information about them, please read the .RAMsection directive explanation.

This is not a compulsory directive.

.SEED 123

Seeds the random number generator.

This is not a compulsory directive. The random number generator is initially seeded with the output of time(), which is, according to the manual, "the time since the Epoch (00 :00 :00 UTC, January 1, 1970), measured in seconds". So if you don't .SEED the random number generator yourself with a constant value, .DBRND and .DWRND give you different values every time you run WLA.

In WLA DX 9.4a and before we used the stdlib's srand() and rand() functions making the output differ on different platforms. Since v9.4 WLA DX contains its own Mersenne Twister pseudo random number generator.

.SHIFT

Shifts the macro arguments one down (\1 becomes \1, \1 -> \1, etc.).
.SHIFT can thus only be used inside a .MACRO.

This is not a compulsory directive.

.SLOT 1

Changes the currently active memory slot. This directive is meant to be used with SUPERFREE subsections, where only the slot number is constant when placing the subsections.

This is not a compulsory directive.

.SLOWROM

Clears the ROM memory speed bit in $FFD5 (.HIROM) or $7FD5 (.LOROM) to indicate that the SNES ROM chips are 200ns chips.

This is not a compulsory directive.

.SMC

Forces WLALINK to compute a proper SMC header for the ROM file.

SMC header is a chunk of 512 bytes. WLALINK touches only its first three bytes, and sets the rest to zeroes. Here's what will be inside the first three bytes :

0 - low byte of 8KB page count.
1 - high byte of 8KB page count.
2 - bit 7 - 0
bit 6 - 0
bit 5 - 0 lorom, 1 hirom
bit 4 - 0 lorom, 1 hirom
bit 3+2 - sram size - 00 256Kb, 01 64Kb, 10 16Kb, 11 0Kb
bit 1 - 0
bit 0 - 0

This is not a compulsory directive.

.SNESEMUVECTOR

Begins definition of the emulation mode interrupt vector table.

```
1 .SNESEMUVECTOR
2       COP       COPHandler
3       UNUSED    $0000
4       ABORT     BRKHandler
5       NMI       VBlank
6       RESET     Main
7       IRQBRK    IRQBRKHandler
8 .ENDEMUVECTOR
```

These can be defined in any order, but they will be placed into memory starting at $7FF4 ($FFF4 in HiROM) in the order listed above. All the vectors default to $0000.

This is not a compulsory directive.

.SNESHEADER

This begins the SNES header definition, and automatically defines
.COMPUTESNESCHECKSUM. From here you may define any of the following :

ID "ABCD" - inserts a one to four letter string starting at $7FB2 (lorom) or $FFB2 (hirom).

NAME "Hello World!" - identical to a freestanding .NAME.

LOROM - identical to a freestanding .LOROM.

HIROM - identical to a freestanding .HIROM.

SLOWROM - identical to a freestanding .SLOWROM.

FASTROM - identical to a freestanding .FASTROM.

CARTRIDGETYPE $00 - Places the given 8-bit value in $7FD6 ($FFD6 in HiROM). Some possible values I've come across but cannot guarantee the accuracy of :

$00 ROM only
$01 ROM and RAM
$02 ROM and Save RAM
$03 ROM and DSP1 chip
$04 ROM, RAM and DSP1 chip
$05 ROM, Save RAM and DSP1 chip
$13 ROM and Super FX chip

ROMSIZE $09 - Places the given 8-bit value in $7FD7 ($FFD7 in HiROM).
Possible values include (but may not be limited to :)

$08 - 2 Megabits
$09 - 4 Megabits
$0A - 8 Megabits
$0B - 16 Megabits
$0C - 32 Megabits

SRAMSIZE $01 - Places the given 8-bit value into $7FD8 ($FFD8 in HiROM).
I believe these are the only possible values :

$00 - 0 kilobits
$01 - 16 kilobits
$02 - 32 kilobits
$03 - 64 kilobits

COUNTRY $00 - Places the given 8-bit value into $7FD9 ($FFD9 in HiROM).
$00 is Japan and $01 is the United States, and there several more for other regions that I cannot recall off the top of my head.

LICENSEECODE $00 - Places the given 8-bit value into $7FDA ($FFDA in HiROM.) You must find the legal values yourself as there are plenty of them.;)

VERSION $01 - Places the given 8-bit value into $7FDB ($FFDB in HiROM) This is supposedly interpreted as version 1.byte, so a $01 here would be version 1.01.

This is not a compulsory directive.

.SNESNATIVEVECTOR

Begins definition of the native mode interrupt vector table.

```
1 .SNESNATIVEVECTOR
2       COP       COPHandler
3       BRK       BRKHandler
4       ABORT     ABORTHandler
5       NMI       VBlank
6       UNUSED    $0000
7       IRQ       IRQHandler
8 .ENDNATIVEVECTOR
```

These can be defined in any order, but they will be placed into memory starting at $7FE4 ($FFE4 in HiROM) in the order listed above.

All the vectors default to $0000.

This is not a compulsory directive.

.STRUCT enemy_object

Begins the definition of a structure. These structures can be placed inside RAMsubsections and ENUMs. Here's an example :

```
.STRUCT enemy\_object
id      dw                  ; the insides of a .STRUCT are 1:1 like in .ENUM
x       db                  ; except that no structs inside structs are
y       db                  ; allowed.
data    ds   10
info    dsb  16
stats   dsw   4
.ENDST
```

This also creates a de-fi-ni-tion "_sizeof_[struct name]", in our e-xam-ple this would be "_size-of_en-emy_ob-ject", and the value of this de-fi-ni-tion is the si-ze of the ob-ject, in bytes (2+1+1+10+16+4*2 = 38 in the ex-am-ple).

You'll get the following definitions as well :

enemy_object.id (== 0)
enemy_object.x (== 2)
enemy_object.y (== 3)
enemy_object.data (== 4)
enemy_object.info (== 14)
enemy_object.stats (== 30)

After defining a .STRUCT you can create an instance of it in a .RAMsubsection / .ENUM by typing

<instance name> INSTANCEOF <struct name> [optional, the number of structures]

Here's an example :

```
.RAMsubsection "enemies" BANK 4 SLOT 4
enemies    INSTANCEOF enemy_object 4
enemyman   INSTANCEOF enemy_object
enemyboss  INSTANCEOF enemy_object
.ENDS
```

This will create labels like "enemies", "enemies.id", "enemies.x", "enemies.y" and so on. Label "enemies" is followed by four "enemy_object" structures, and only the first one is labeled. After there four come "enemyman" and "enemyboss" instances.

Take a look at the documentation on .RAMsubsection & .ENUM, they have more examples of how you can use .STRUCTs.

A WORD OF WARNING : Don't use labels b, B, w and W inside a struct as e.g., WLA sees enemy.b as a byte sized reference to enemy. All other labels should be safe.

lda enemy1.b ; load a byte from zeropage address enemy1 or from the address
; of enemy1.b ? ? ? i can't tell you, and WLA can't tell you...

This is not a compulsory directive.

.SYM SAUSAGE

WLA treats symbols ("SAUSAGE" in this example) like labels, but they only appear in the symbol files WLALINK outputs. Useful for finding out the location where WLALINK puts data.

This is not a compulsory directive.

.SYMBOL SAUSAGE

.SYMBOL is an alias for .SYM.

This is not a compulsory directive.

.UNBACKGROUND $1000 $1FFF

After issuing .BACKGROUND you might want to free some parts of the backgrounded ROM image for e.g., FREE subsections. With .UNBACKGROUND you can define such regions. In the example a block starting at $1000 and ending at $1FFF was released (both ends included). You can issue .UNBACKGROUND as many times as you wish.

This is not a compulsory directive.

.UNDEFINE DEBUG

Removes the supplied definition label from system. If there is no such label as given no error is displayed as the result would be the same.

You can undefine as many definitions as you wish with one .UNDEFINE :

```
1 .UNDEFINE NUMBER, NAME, ADDRESS, COUNTRY
2 .UNDEFINE NAME, AGE
```

This is not a compulsory directive.

.UNDEF DEBUG

.UNDEF is an alias for .UNDEFINE.

This is not a compulsory directive.

.WORD 16000, 10, 255

.WORD is an alias for .DW.

This is not a compulsory directive.

Septième partie

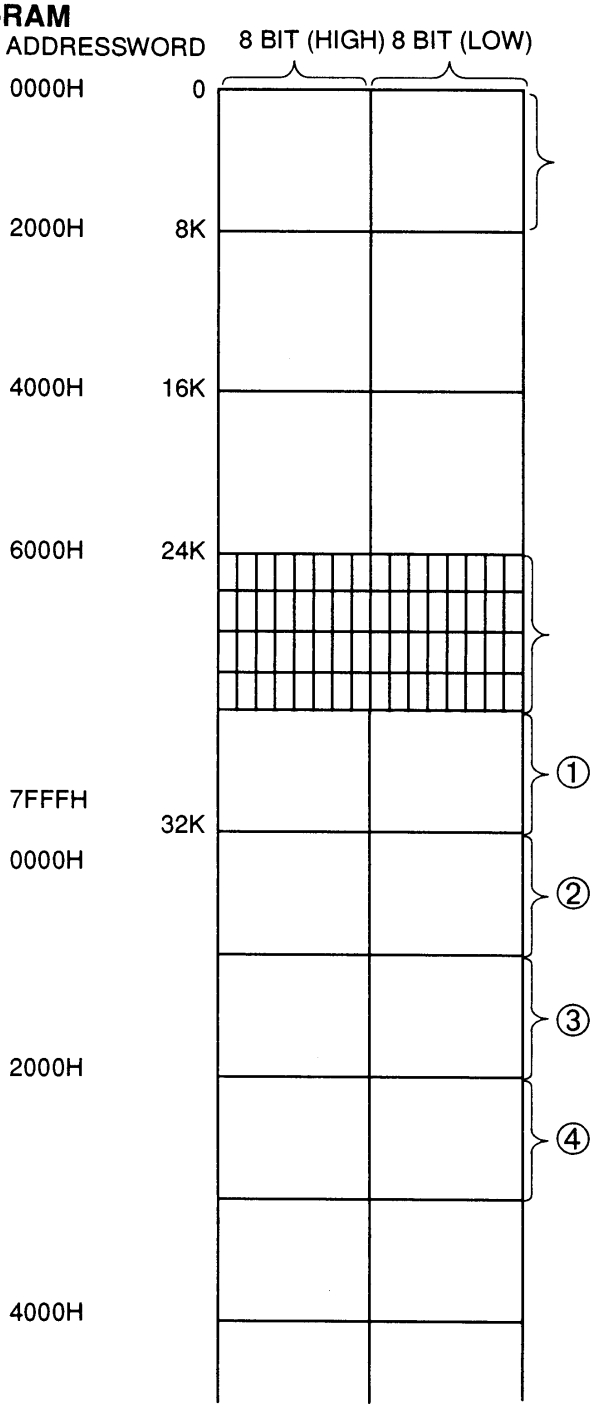
ANNEXES

Chapitre 20

Registre PPU

Appendix A. PPU Registers

V-RAM



8K-WORD: This is an area which is designated by "OBJ NAME BASE ADDRESS" of the register <2101H>. (32K-WORD / 4-Partition.) [The BA2 of the register <2101H> "OBJ NAME BASE ADDRESS" is used for expansion purposes, and it will normally be ignored.]

[In case BA1=1 and BA0=1 are set by "OBJ NAME BASE ADDRESS"]

4K-WORD: This is a lower 4K-WORD of the area (8K-WORD) designated by "OBJ NAME BASE ADDRESS" of the register <2101H>. The combination of this 4K-WORD and the 4K-WORD remaining will be determined by "OBJ NAME SELECT" of the register <2101H>.

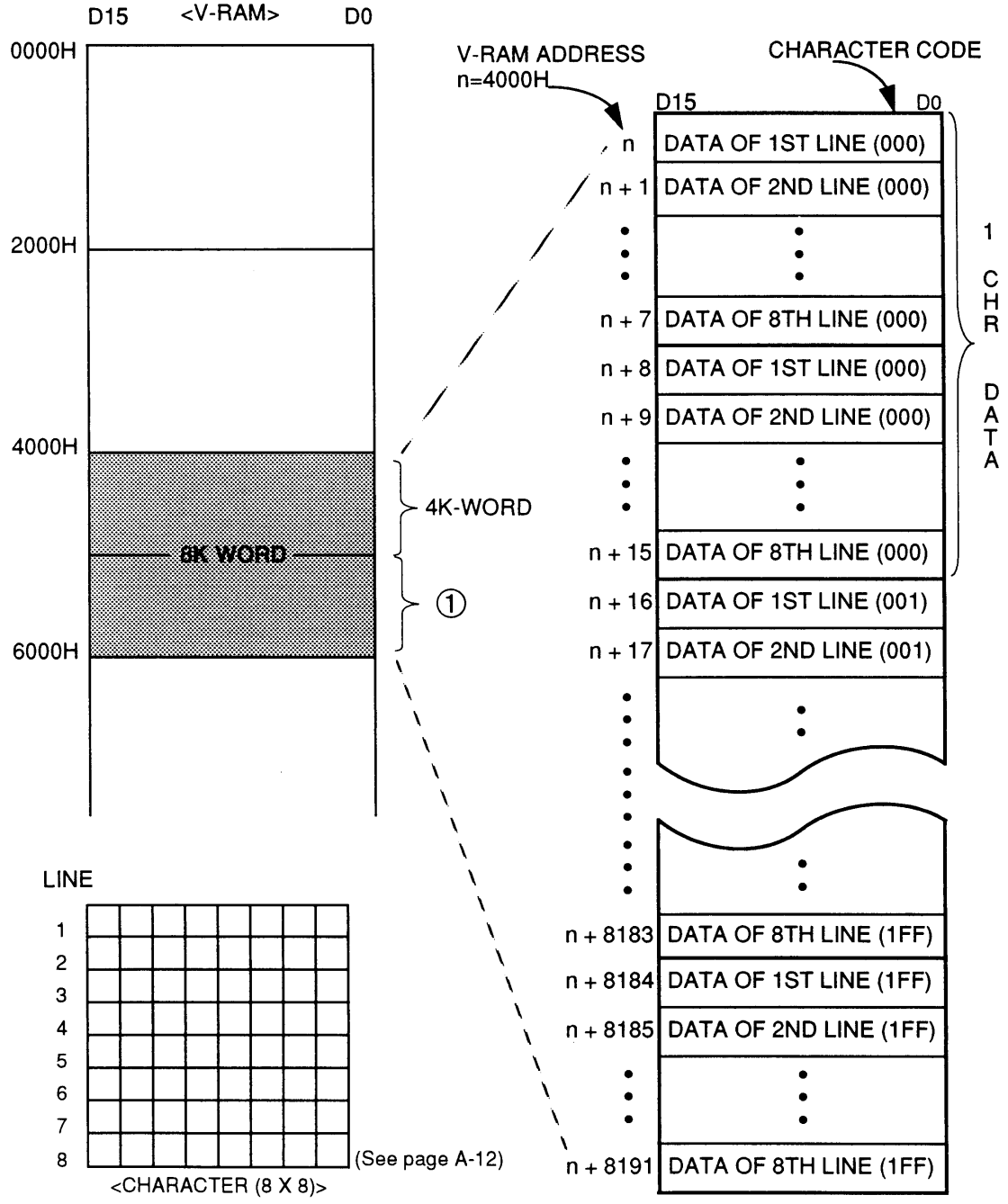
OBJ Name Select

N1	N0	COMBINATION
0	0	4K - WORD + ①
0	1	4K - WORD + ②
1	0	4K - WORD + ③
1	1	4K - WORD + ④

(NCL PG 68)

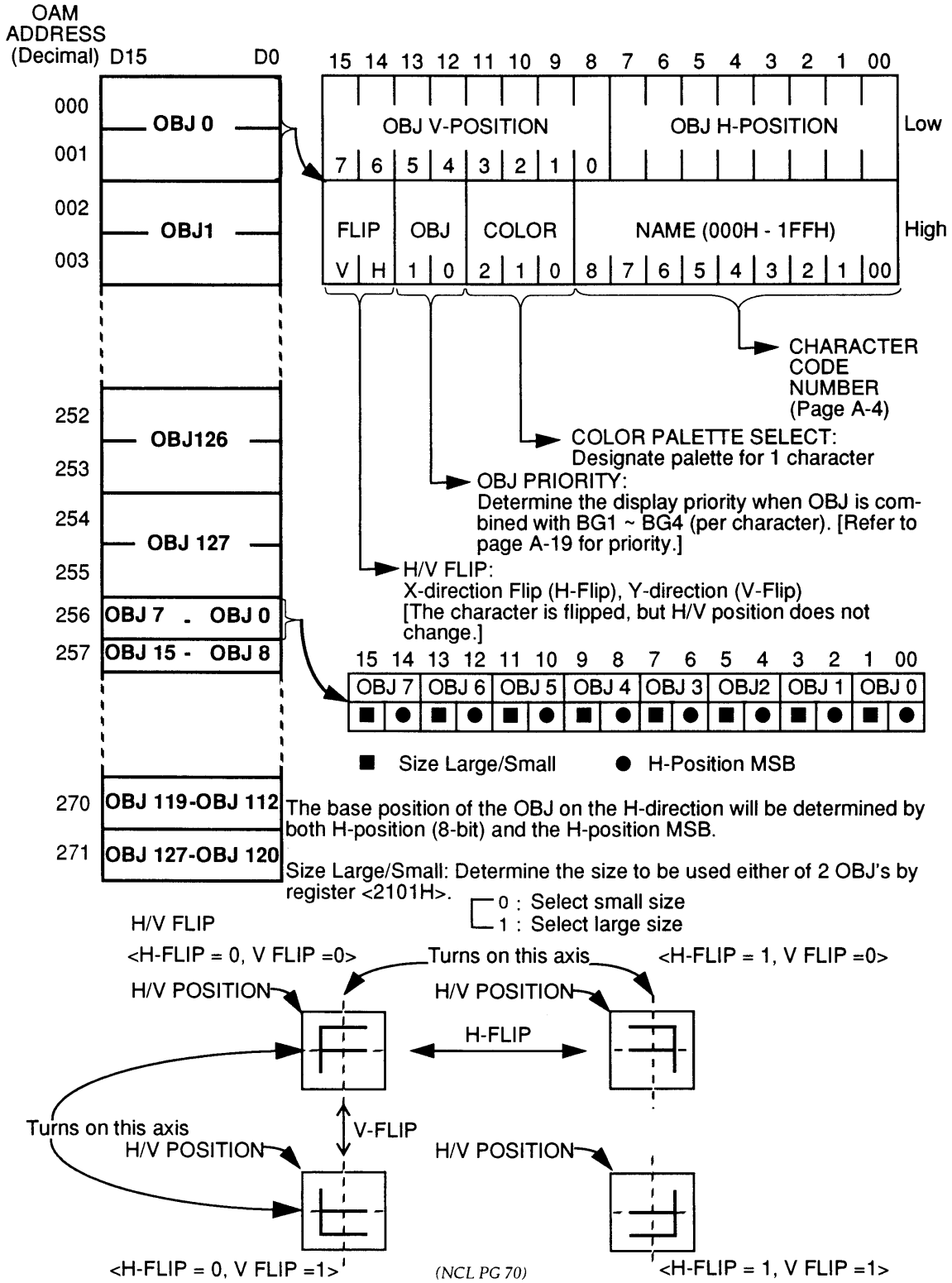
OBJECT DATA TO BE STORED

4 BIT CONSTRUCTION [8 x 8 x 4 Bit (16 WORD) / CHARACTER] (Refer to page A-12)
8 x 8 (Character Size) x 4 (Bit Construction) x 512 (Number of character) ⟶ 16K-BYTE
[In case BA1=1 and BA0=0 are set by "OBJ NAME BASE ADDRESS" and also N1=0 and N0=0 are set by "OBJ NAME SELECT"]



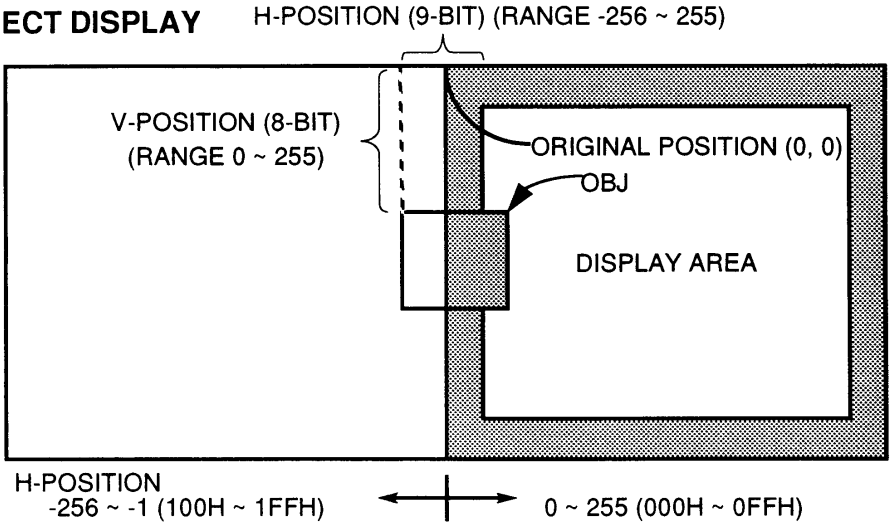
(NCL PG 69)

OBJECT DATA



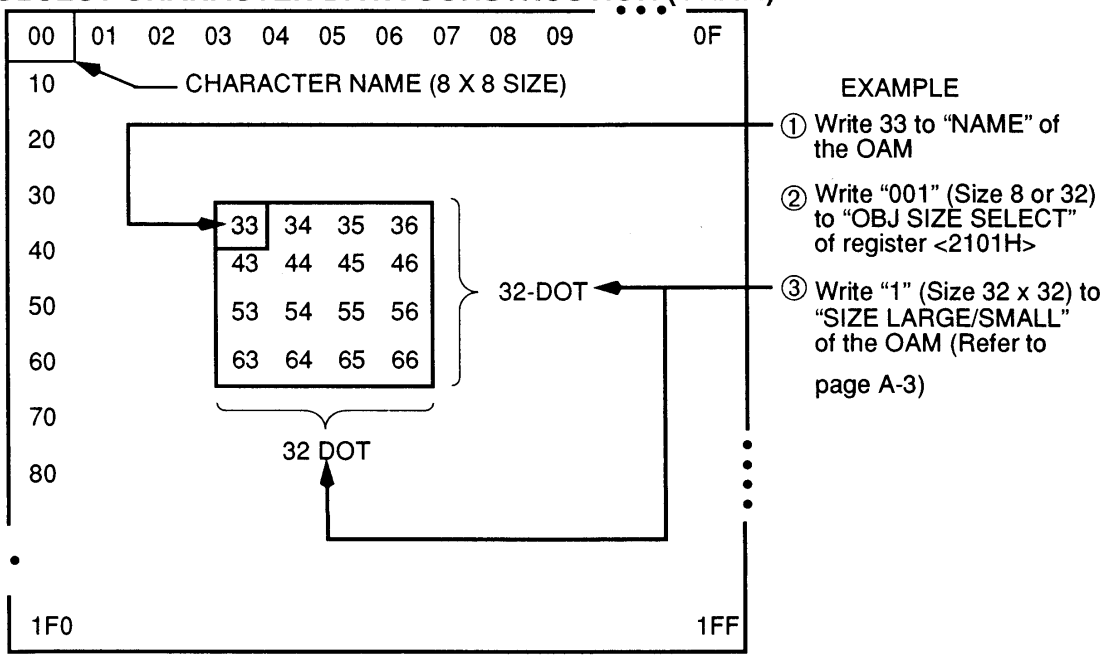
(NCL PG 70)

OBJECT DISPLAY



(NOTE-1) The H-position is a complementary expression of 2 (9-bit).
(NOTE-2) The coordinate of the OBJ displayed is shifted down compared to the coordinate of the BG displayed. [Interlace: 2-dot / Non-Interlace: 1-dot] (See page A-10.)
(NOTE-3) "100H" is basically prohibited to use for 9-bit of the H-Position. (If it is used, it must be counted as OBJ quantity displayed even if it is not displayed on the screen.)

OBJECT CHARACTER DATA CONSTRUCTION (VRAM)



In case the character code is 000 through 0FF, the V-RAM address per character data (16-word) will be "n (Name Base Address) + N (Name) x 16 ~ n + N x 16 + 15." If the character code is 100 through 1FF, it will be "n + Ns (Name Select) x 4K + N x 16 ~ n + Ns x 4K + N x 16 + 15."

(NCL PG 71)

OBJECT

# OF CELLS DISPLAYED	1 2 8			
CELL SIZE	8X8	16X16	32X32	64X64
# OF LINES DISPLAYED	32-pcs (converted to 8x8 size)			
# OF CELL-COLOR	1 6			
# OF PALETTE	8			
# OF COLOR ON SCREEN	1 2 8			
ATTRIBUTE	H-FLIP, V-FLIP FUNCTION DISPLAY PRIORITY (Select priority against BG)			

BG

MODE	# OF SCREENS DIS-PLAYED	SCREEN	# OF CELL DOT	# OF CELL COLOR	# OF PALETTES	# OF COLORS PER SCREEN	FUNCTION
0	MAX 4	BG1	8 X 8	4	8	32	①②③ ⑤⑥⑦⑧ ⑩
		BG2	OR	4	8	32	①②③ ⑤⑥⑦⑧ ⑩
		BG3		4	8	32	①②③ ⑤⑥⑦⑧ ⑩
		BG4	16 X 16	4	8	32	①②③ ⑤⑥⑦⑧ ⑩
1	MAX 3	BG1		16	8	128	①②③ ⑤⑥⑦⑧ ⑩
		BG2		16	8	128	①②③ ⑤⑥⑦⑧ ⑩
		BG3		4	8	32	①②③ ⑤⑥⑦⑧ ⑩
2	MAX 2	BG1		16	8	128	①②③ ⑤⑥⑦⑧ ⑩⑪
		BG2		16	8	128	①②③ ⑤⑥⑦⑧ ⑩⑪
3	MAX 2	BG1		256	1	256	①②③ ⑤⑥⑦⑧⑨⑩
		BG2		16	8	128	①②③ ⑤⑥⑦⑧ ⑩
4	MAX 2	BG1		256	1	256	①②③ ⑤⑥⑦⑧⑨⑩⑪
		BG2		4	8	32	①②③ ⑤⑥⑦⑧ ⑩⑪
5	MAX 2	BG1	▼	16	8	128	①②③ ⑤ ⑦⑧ ⑫
		BG2		4	8	32	①②③ ⑤ ⑦⑧ ⑫
6	1	BG1	16X8	16	8	128	①②③ ⑤ ⑦⑧ ⑪⑫
7	1	BG1	8 X 8	256	1	256	① ③④⑤⑥⑦⑧⑨⑩
EXT BG	1	BG2	8 X 8	128	1	128	① ③④⑤⑥⑦⑧⑨⑩

(NCL PG 72)

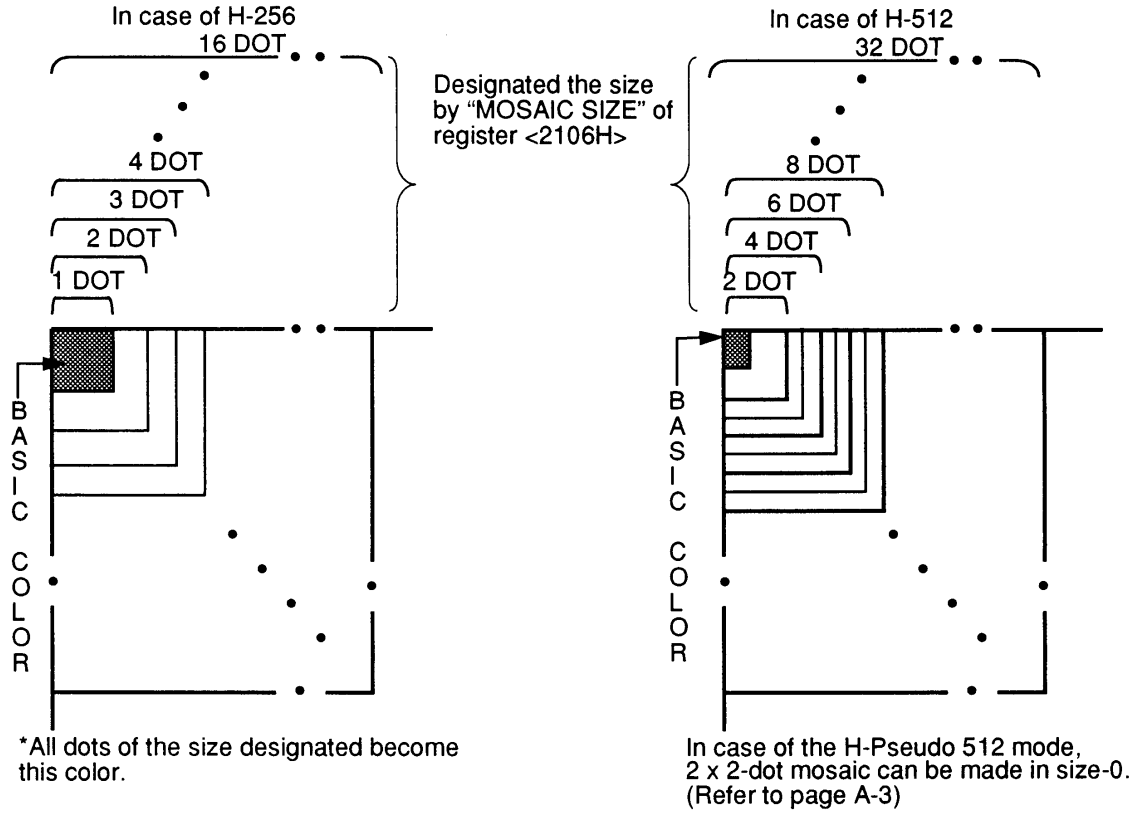
[Main Function of BG]

1. HV Scroll (each screen)
2. HV Flip (each character)
3. Mosaic (Refer to Chapter 4)
4. Rotate, Enlarge, Reduce (Refer to Chapter 5)
5. Window Mask (Refer to Chapter 6)
6. Screen Addition and Subtraction (Refer to ¶7.1)
7. Fixed Color Addition and Subtraction (Refer to ¶7.2)
8. Color Window (Refer to ¶7.2)
9. CG Direct Select (Refer to Chapter 8)
10. Horizontal Pseudo 512 (Refer to Chapter 9)
11. Offset Change (Refer to Chapter 12)
12. Horizontal 512 Mode (Refer toChapter 19)

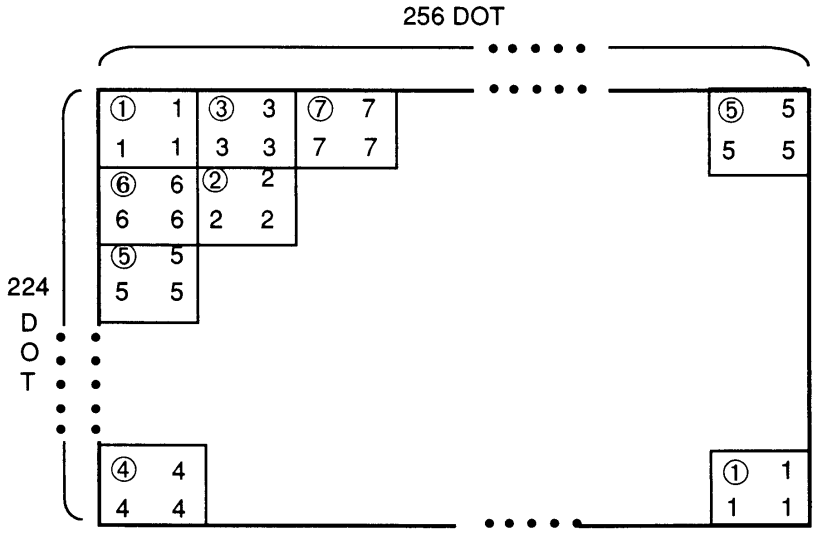
[Other Function]

- Priority (each character/mode 0 ~ 6)
- Screen HV Rotate (mode 7)

MOSAIC SCREEN



MOSAIC SCREEN DISPLAY EXAMPLE (BG SCREEN)
(When the mosaic size is 2 x 2-dot in the 256-mode)



⑪ is the basic color data

(NCL PG 73)