

GameCube DSP User's Manual

Reverse-engineered and documented by Duddie
duddie@walla.com

March 10, 2020
v0.0.6

Contents

Disclaimer

This documentation is no way endorsed by or affiliated with Nintendo, Nintendo of America or its licenses. GameCube is a trademark of Nintendo of America. Other trademarked names used in this documentation are trademarks of their respective owners.

This documentation is provided “AS IS” and can be wrong, incomplete or in any other way useless.

This documentation cannot be used for any commercial purposes without prior agreement received from authors.

The purpose of this documentation is purely academic and it aims at understanding described hardware. It is based on academic reverse engineering of hardware.

Version History

Version	Date	Author	Change
0.0.1	2005.05.08	Duddie	Initial release
0.0.2	2005.05.09	Duddie	Added <code>\$prod</code> and <code>\$config</code> registers, table of opcodes, disclaimer.
0.0.3	2005.05.09	Duddie	Fixed BLOOP and BLOOPI and added description of the loop stack.
0.0.4	2005.05.12	Duddie	Added preliminary DSP memory map and opcode syntax.
0.0.5	2018.04.09	Lioncache	Converted document over to LaTeX.
0.0.6	2018.04.13	BhaaL	Updated register tables, fixed opcode operations

GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc. 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. 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. Secondly, 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.

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 for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

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 <https://www.gnu.org/licenses/>.

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.

Chapter 1

Overview

1.1 DSP Memory Map

The DSP accesses memory in words, so all addresses refer to words. A DSP word is 16 bits in size.

Instruction Memory (IMEM) is divided into instruction RAM (IRAM) and instruction ROM (IROM).

Exception vectors are located at the top of the RAM and occupy the first 8 words.

DSP IRAM is mapped through as first 8KB of ARAM (Accelerator RAM), therefore the CPU can DMA DSP code to DSP IRAM. This usually occurs during boot time, as the DSP ROM is not enabled at cold reset and needs to be reenabled by a small stub executed in IRAM.

0x0000	IRAM
0x0FFF	
0x8000	IROM
0x8FFF	

Chapter 2

Registers

2.1 Register names

The DSP has 32 16-bit registers, although their individual purpose and their function differ from register to register.

\$0	\$r00	\$ar0	Addressing register 0
\$1	\$r01	\$ar1	Addressing register 1
\$2	\$r02	\$ar2	Addressing register 2
\$3	\$r03	\$ar3	Addressing register 3
\$4	\$r04	\$ix0	Indexing register 0
\$5	\$r05	\$ix1	Indexing register 1
\$6	\$r06	\$ix2	Indexing register 2
\$7	\$r07	\$ix3	Indexing register 3
\$8	\$r08		
\$9	\$r09		
\$10	\$r0A		
\$11	\$r0B		
\$12	\$r0C	\$st0	Call stack register
\$13	\$r0D	\$st1	Data stack register
\$14	\$r0E	\$st2	Loop address stack register
\$15	\$r0F	\$st3	Loop counter register
\$16	\$r10	\$ac0.h	40-bit Accumulator 0 (high)
\$17	\$r11	\$ac1.h	40-bit Accumulator 1 (high)
\$18	\$r12	\$config	Config register
\$19	\$r13	\$sr	Status register
\$20	\$r14	\$prod.l	Product register (low)
\$21	\$r15	\$prod.m1	Product register (mid 1)
\$22	\$r16	\$prod.h	Product register (high)
\$23	\$r17	\$prod.m2	Product register (mid 2)
\$24	\$r18	\$ax0.l	32-bit Accumulator 0 (low)
\$25	\$r19	\$ax0.h	32-bit Accumulator 0 (high)
\$26	\$r1A	\$ax1.l	32-bit Accumulator 1 (low)
\$27	\$r1B	\$ax1.h	32-bit Accumulator 1 (high)
\$28	\$r1C	\$ac0.l	40-bit Accumulator 0 (low)
\$29	\$r1D	\$ac1.l	40-bit Accumulator 1 (low)
\$30	\$r1E	\$ac0.m	40-bit Accumulator 0 (mid)
\$31	\$r1F	\$ac1.m	40-bit Accumulator 1 (mid)

2.2 Accumulators

The DSP has two long 40-bit accumulators (**\$acX**) and their short 24-bit forms (**\$acsX**) that reflect the upper part of 40-bit accumulator. There are additional two 32-bit accumulators (**\$axX**).

Accumulators **\$acX**:

40-bit accumulator **\$acX** (**\$acX.hml**) consists of registers:

$$\text{\$acX} = \text{\$acX.h} \ll 32 \mid \text{\$acX.m} \ll 16 \mid \text{\$acX.l}$$

Short accumulators **\$acs.X**:

24-bit accumulator **\$acsX** (**\$acX.hm**) consists of the upper 24 bits of accumulator **\$acX**.

$$\text{\$acsX} = \text{\$acX.h} \ll 16 \mid \text{\$acX.m}$$

Additional accumulators **\$axX**:

$$\text{\$axX} = \text{\$axX.h} \ll 16 \mid \text{\$axX.l}$$

2.3 Stacks

The GameCube DSP contains four stack registers:

- **\$st0** – Call stack register
- **\$st1** – Data stack register
- **\$st2** – Loop address stack register
- **\$st3** – Loop counter register

Stacks are implemented in hardware and have limited depth. The data stack is limited to four values and the call stack is limited to eight values. The loop stack is limited to four values. Upon underflow or overflow of any of the stack registers exception **STOVF** is raised.

The loop stack is used to control execution of repeated blocks of instructions. Whenever there is a value in **\$st2** and the current PC is equal to the value in **\$st2**, then the value in **\$st3** is decremented. If the value is not zero, then the PC is modified by the value from call stack **\$st0**. Otherwise values from the call stack **\$st0** and both loop stacks, **\$st2** and **\$st3**, are popped and execution continues at the next opcode.

2.4 Config register

Its purpose is unknown at this time. It is written with 0x00FF and 0x0004 values.

2.5 Status register

Status register `$sr` reflects flags computed on accumulators after logical or arithmetic operations. Furthermore, it also contains control bits to configure the flow of certain operations.

Bit	Name	Comment
15	SU	Operands are signed (1 = unsigned)
14	SXM	Sign extension mode (0 = <code>set16</code> , 1 = <code>set40</code>)
13	AM	Product multiply result by 2 (when <code>AM = 0</code>)
12		
11	EIE	External interrupt enable
10		
9	IE	Interrupt enable
8	0	Hardwired to 0?
7	OS	Overflow (sticky)
6	LZ	Logic zero
5		Top two bits are equal
4	AS	Above s32
3	S	Sign
2	Z	Arithmetic zero
1	O	Overflow
0	C	Carry

2.6 Product register

The product register is a register containing the intermediate product of a multiply or multiply and accumulation operation. It's result should never be used for calculation although the register can be read or written. It reflects the state of the internal multiply unit. The product is 40 bits with 1 bit of overflow.

$$\text{\$prod} = (\text{\$prod.h} \ll 32) + ((\text{\$prod.m1} + \text{\$prod.m2}) \ll 16) + \text{\$prod.l}$$

It needs to be noted that `\$prod.m1 + \$prod.m2` overflow bit (bit 16) will be added to `\$prod.h`.

Bit `\$sr.AM` affects the result of the multiply unit. If `\$sr.AM` is equal 0 then the result of every multiply operation will be multiplied by two.

Chapter 3

Exceptions

3.1 Exception processing

Exception processing happens by setting the program counter to different exception vectors. At exception time, the exception program counter is stored at call stack `$st0` and status register `$sr` is stored at data stack `$st1`.

Operation:

```
PUSH_STACK($st0);  
$st0 = $pc;  
PUSH_STACK($st1);  
$st1 = $sr;  
$pc = exception_nr * 2;
```

3.2 Exception vectors

Exception vectors are located at address 0x0000 in Instruction RAM.

Level	Address	Name	Description
0	0x0000	RESET	
1	0x0002	STOVF	Stack under/overflow
2	0x0004		
3	0x0006		
4	0x0008		
5	0x000A	ACCOV	Accelerator address overflow
6	0x000C		
7	0x000E	INT	External interrupt (from CPU)

Chapter 4

Hardware interface

4.1 Hardware registers

Hardware registers occupy the address space at 0xFFxx in DSP memory space. Each register is 16 bits in width.

Address	Name	Description
<i>Mailboxes</i>		
0xFFFE	CMBH	CPU Mailbox H
0xFFFF	CMBL	CPU Mailbox L
0xFFFC	DMBH	DSP Mailbox H
0xFFFD	DMBL	DSP Mailbox L
<i>DMA Interface</i>		
0xFFCE	DSMAH	Memory address H
0xFFCF	DSMAL	Memory address L
0xFFCD	DSPA	DSP memory address
0xFFC9	DSCR	DMA control
0xFFCB	DSBL	Block size
<i>Accelerator</i>		
0xFFD4	ACSAH	Accelerator start address H
0xFFD5	ACSAL	Accelerator start address L
0xFFD6	ACEAH	Accelerator end address H
0xFFD7	ACEAL	Accelerator end address L
0xFFD8	ACCAH	Accelerator current address H
0xFFD9	ACCAL	Accelerator current address L
0xFFDD	ACDAT	Accelerator data
<i>Interrupts</i>		
0xFFFB	DIRQ	IRQ request

4.2 Interrupts

The DSP can raise interrupts at the CPU. Interrupts are usually used to signal that a DSP mailbox has been filled with new data.

0xFFFFB	DIRQ	IRQ Request
----	----	----I

Bit	Name	R/W	Action
0	I	W	1 - Raise interrupt at CPU

4.3 Mailboxes

4.3.1 CPU Mailbox

The CPU Mailbox (CMB) is a register that allows sending 31 bits of information from the CPU to the DSP.

0xFFFE	CMBH	CPU Mailbox H
Mddd dddd dddd dddd		

Bit	Name	R/W	Action
15	M	R	1 - Mailbox contains mail from the CPU 0 - Mailbox empty
14-0	d	R	Bits 30-16 of the mail sent from the CPU

0xFFFF	CMBL	CPU Mailbox L
dddd dddd dddd dddd		

Bit	Name	R/W	Action
15-0	d	R	Bits 15-0 of mail sent from the CPU. Reading of this register by the DSP causes the CMBH.M bit to be cleared.

Operation:

From the CPU side, software usually checks the M bit of CMBH. It takes action only in the case that this bit is 0. Said action is to write CMBH first and then CMBL. After writing to CMBL, the mail is ready to be received by the DSP.

From the DSP side, the DSP loops by probing the M bit. When this bit is 1, the DSP reads CMBH first and then CMBL. After reading CMBL, CMBH.M will be cleared.

4.3.2 DSP Mailbox

The DSP mailbox (DMB) is an interface to send 31 bits of information from the DSP to the CPU.

0xFFFFC	DMBH	DSP Mailbox H
Mddd dddd dddd dddd		

Bit	Name	R/W	Action
15	M	R	1 - Mailbox has not been received by CPU 0 - Mailbox empty
		W	Does not matter. It will be set when DMBL is written to
14-0	d	W	Bits 30-16 of mail sent from the DSP to the CPU

0xFFFFD	DMBL	DSP Mailbox L
dddd dddd dddd dddd		

Bit	Name	R/W	Action
15-0	d	W	Bits 15-0 of mail sent from the DSP to the CPU. Writing to this register by the DSP causes the DMBH.M bit to be set, indicating that the mail is ready.

Operation:

Sending mail from the DSP to the CPU can be achieved by writing mail to register DMBH and then to register DMBL in that order. After writing to DMBL, bit DMBH.M will be set, signaling that the mail is ready to be received by the CPU. If the DSP needs to receive a response from the CPU, then it usually waits for the M bit to be cleared after sending a mail. If the DSP does processing when the CPU receives a mail, then it waits for the M bit to be cleared before issuing another mail to the CPU.

4.3.3 DMA

The GameCube DSP is connected to the memory bus through a DMA channel. DMA can be used to transfer data between DSP memory (both instruction and data) and main memory.

0xFFCE	DSMAH	Memory Address H
dddd	dddd	dddd dddd

Bit	Name	R/W	Action
15–0	d	R	Bits 31–16 of the main memory address

0xFFCF	DSMAL	Memory Address L
dddd	dddd	dddd dddd

Bit	Name	R/W	Action
15–0	d	R	Bits 15–0 of the main memory address

0xFFCD	DSPA	DSP Address
dddd	dddd	dddd dddd

Bit	Name	R/W	Action
15–0	d	W	Bits 15–0 of the DSP memory address

0xFFCB	DSBL	DSP Address
dddd	dddd	dddd dddd

Bit	Name	R/W	Action
15–0	d	W	Length in bytes to transfer. Writing to this register starts a DMA transfer.

0xFFC9	DSCR	DSP Address
----	----	----

Bit	Name	R/W	Action
15–0	d	W	

4.4 Accelerator

The accelerator is used to transfer data from accelerator memory (ARAM) to DSP memory. The accelerator area can be marked with **ACSA** (start) and **ACEA** (end) addresses. Current address for the accelerator can be set or read from the **ACCA** register. Reading from accelerator memory is done by reading from the **ACDAT** register. This register contains data from ARAM pointed to by the **ACCA** register. After reading the data, **ACCA** is incremented by one. After **ACCA** grows bigger than the area pointed to by **ACEA**, it gets reset to a value from **ACSA** and the **ACCOV** interrupt is generated.

Chapter 5

Opcodes

5.1 Opcode syntax

Basic opcode syntax:

OPC <opcode parameters>

Above syntax is correct for all opcodes.

EXAMPLES:

```
JMP 0x0300
CALL loop
HALT
```

Extended syntax:

OPC'EXOPC <opcode parameters> : <extended opcode parameters>

Above syntax is correct only for arithmetic opcodes, because those can be extended with additional load/store unit behavior.

EXAMPLES:

```
DECM'L $acs0 : $acl.m, @ar0
NX'MV : $acx1.h, $ac0.l
```

5.2 Operation — Used Functions

Functions used for describing opcode operation.

`PUSH_STACK($stR)`

Description:

Pushes value onto given stack referenced by stack register `$stR`. Operation moves values down in internal stack.

Operation:

```
stack_stR[stack_ptr_stR++] = $stR;
```

`POP_STACK($stR)`

Description:

Pops value from stack referenced by stack register `$stR`. Operation moves values up in internal stack.

Operation:

```
$stR = stack_stR[--stack_ptr_stR];
```

`FLAGS(val)`

Description:

Calculates flags depending on given value or result of operation and sets corresponding bits in status register `$sr`.

`EXECUTE_OPCODE(new_pc)`

Description:

Executes opcode at the given `new_pc` address.

5.3 Bit meanings

Opcode decoding uses special naming for bits and their decimal representations to provide easier understanding of bit fields in the opcode.

Binary form	Decimal form	Meaning
d, dd, ddd, dddd	D	Destination register
s, ss, sss, ssss	S	Source register
t, tt, ttt, tttt	T	Source register
r, rr, rrr, rrrr	R	Register (either source or destination)
Aaaaa(a)	A, addrA	Address in either instruction or data memory
xxxx xxxx	X	Extended opcode
mmm(m)	M, addrM	Address in memory
iii(i)	I, Imm	Immediate value
cccc	cc	Condition (see conditional opcodes)

5.4 Conditional opcodes

Conditional opcodes are executed only when the condition described by their encoded conditional field has been met. The groups of conditional instructions are, **CALL**, **JMP**, **IF**, and **RET**.

Bits	cc	Name	Evaluated expression
0b0000	GE	Greater than or equal	
0b0001	L	Less than	
0b0010	G	Greater than	
0b0011	LE	Less than or equal	
0b0100	NE	Not equal	(\$sr & 0x4) == 0
0b0101	EQ	Equal	(\$sr & 0x4) != 0
0b0110	NC	Not carry	(\$sr & 0x1) == 0
0b0111	C	Carry	(\$sr & 0x1) != 0
0b1000		Below s32	(\$sr & 0x10) == 0
0b1001		Above s32	(\$sr & 0x10) != 0
0b1010			
0b1011			
0b1100	NZ	Not zero	(\$sr & 0x40) == 0
0b1101	ZR	Zero	(\$sr & 0x40) != 0
0b1110	O	Overflow	(\$sr & 0x2) != 0
0b1111		<always>	

Note:

There are two pairs of conditions that work similar: EQ/NE and ZR/NZ. EQ/NE pair operates on arithmetic zero flag (arithmetic 0) while ZR/NZ pair operates on logic zero flag (logic 0).

5.5 Alphabetical list of opcodes

5.5.1 ADD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0100				110d				xxxx				xxxx			

Format:

ADD \$acD, \$ac(1-D)

Description:

Adds accumulator \$ac(1-D) to accumulator register \$acD.

Operation:

\$acD += \$ac(1-D)
 FLAGS(\$acD)
 \$pc++

5.5.2 ADDARN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0000				0001				ssdd			

Format:

ADDARN \$arD, \$ixS

Description:

Adds indexing register \$ixS to an addressing register \$arD.

Operation:

\$arD += \$ixS
\$pc++

5.5.3 ADDAX

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0100				10sd				xxxx				xxxx			

Format:

ADDAX \$acD, \$axS

Description:

Adds secondary accumulator \$axS to accumulator register \$acD.

Operation:

```
$acD += $axS
FLAGS($acD)
$pc++
```


5.5.4 ADDAXL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0111				00sd				xxxx				xxxx			

Format:

```
ADDAXL $acD, $axS.l
```

Description:

Adds secondary accumulator **\$axS.l** to accumulator register **\$acD**.

Operation:

```
$acD += $axS.l
FLAGS($acD)
$pc++
```

5.5.5 ADDI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				001r				0000				0000			
iiii				iiii				iiii				iiii			

Format:

ADDI \$amR, #I

Description:

Adds a 16-bit sign-extended immediate to mid accumulator \$acD.hm.

Operation:

```
$acD.hm += #I
FLAGS($acD)
$pc += 2
```

5.5.6 ADDIS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				010d				iiii				iiii			

Format:

`ADDIS $acD, #I`

Description:

Adds an 8-bit sign-extended immediate to mid accumulator `$acD.hm`.

Operation:

```

$acD.hm += #I
FLAGS($acD)
$pc++

```

5.5.7 ADDP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0100				111d				xxxx				xxxx			

Format:

ADDP \$acD

Description:

Adds the product register to the accumulator register.

Operation:

```
$acD += $prod
FLAGS($acD)
$pc++
```

5.5.8 ADDPAXZ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1111				10sd				xxxx				xxxx			

Format:

ADDPAXZ \$acD, \$axS

Description:

Adds secondary accumulator \$axS to product register and stores result in accumulator register. Low 16-bits of \$acD (\$acD.l) are set to 0.

Operation:

```
$acD.hm = $prod.hm + $ax.h
$acD.l = 0
FLAGS($acD)
$pc++
```

5.5.9 ADDR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0100				0ssd				xxxx				xxxx			

Format:

ADDR \$acD, \$(0x18+S)

Description:

Adds register \$(0x18+S) to the accumulator \$acD register.

Operation:

\$acD += \$(0x18+S)
FLAGS(\$acD)
\$pc++

5.5.10 ANDC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0011				110d				xxxx				xxxx			

Format:

ANDC \$acD.m, \$ac(1-D).m

Description:

Logic AND middle part of accumulator \$acD.m with middle part of accumulator \$ax(1-D).m.

Operation:

\$acD.m &= \$ac(1-D).m
 FLAGS(\$acD)
 \$pc++

5.5.11 ANDCF

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				001r				1010				0000			
iiii				iiii				iiii				iiii			

Format:

ANDCF \$acD.m, #I

Description:

Sets the logic zero (LZ) flag in status register \$sr if the result of the logical AND operation involving the mid part of accumulator \$acD.m and the immediate value I is equal to immediate value I. If the logical AND operation does not result in a value equal to I, then the LZ flag is cleared.

Operation:

```
IF ($acD.m & I) == I
    $sr.LZ = 1
ELSE
    $sr.LZ = 0
ENDIF
$pc += 2
```


5.5.12 ANDF

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				001r				1100				0000			
iiii				iiii				iiii				iiii			

Format:

ANDF \$acD.m, #I

Description:

Sets the logic zero (LZ) flag in status register \$sr if the result of the logic AND operation involving the mid part of accumulator \$acD.m and the immediate value I is equal to zero. If the result is not equal to zero, then the LZ flag is cleared.

Operation:

```

IF ($acD.m & I) == 0
    $sr.LZ = 1
ELSE
    $sr.LZ = 0
ENDIF
$pc += 2

```

5.5.13 ANDI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				001r				0100				0000			
iiii				iiii				iiii				iiii			

Format:

```
ANDI $acD.m, #I
```

Description:

Performs a logical AND with the mid part of accumulator `$acD.m` and the immediate value `I`.

Operation:

```
$acD.m &= #I
FLAGS($acD)
$pc += 2
```

5.5.14 ANDR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0011				01sd				xxxx				xxxx			

Format:

ANDR \$acD.m, \$axS.h

Description:

Performs a logical AND with the middle part of accumulator \$acD.m and the high part of secondary accumulator, \$axS.h.

Operation:

\$acD.m &= \$axS.h
 FLAGS(\$acD)
 \$pc++

5.5.15 ASL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				010r				10ii				iiii			

Format:

ASL \$acR, #I

Description:

Arithmetically left shifts the accumulator \$acR by the amount specified by immediate I.

Operation:

\$acR <<= I
 FLAGS(\$acD)
 \$pc++

5.5.16 ASR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				010r				11ii				iiii			

Format:

ASR \$acR, #I

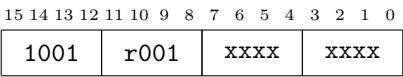
Description:

Arithmetically right shifts accumulator \$acR specified by the value calculated by negating sign-extended bits 0-6.

Operation:

\$acR >>= I
 FLAGS(\$acD)
 \$pc++

5.5.17 ASR16



Format:

```
ASR16 $acR
```

Description:

Arithmetically right shifts accumulator `$acR` by 16.

Operation:

```
$acR >>= 16
FLAGS($acD)
$pc++
```

5.5.18 BLOOP

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0				
0000	0000	011r	rrrr	
aaaa	aaaa	aaaa	aaaa	

Format:

BLOOP \$R, addrA

Description:

Repeatedly execute a block of code starting at the following opcode until the counter specified by the value from register **\$R** reaches zero. Block ends at specified address **addrA** inclusive. i.e. opcode at **addrA** is the last opcode included in loop. Counter is pushed on loop stack **\$st3**, end of block address is pushed on loop stack **\$st2** and the repeat address is pushed on call stack **\$st0**. Up to 4 nested loops are allowed.

Operation:

```

$st0 = $pc + 2
$st2 = addrA
$st3 = $R
$pc += 2

// On real hardware, the below does not happen,
// this opcode only sets stack registers
WHILE ($st3--)
    DO
        EXECUTE_OPCODE($pc)
    WHILE($pc != $st2)
        $pc = $st0
END
$pc = addrA + 1
// Remove vaues from stack

```

5.5.19 BLOOPI

15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
0001 0001	iiii iiii
aaaa aaaa	aaaa aaaa

Format:

```
BLOOPI  #I, addrA
```

Description:

Repeatedly execute a block of code starting at the following opcode until the counter specified by the immediate value **I** reaches zero. Block ends at specified address **addrA** inclusive. i.e. opcode at **addrA** is the last opcode included in loop. Counter is pushed on loop stack **\$st3**, end of block address is pushed on loop stack **\$st2** and the repeat address is pushed on call stack **\$st0**. Up to 4 nested loops are allowed.

Operation:

```

$st0 = $pc + 2
$st2 = addrA
$st3 = 1
$pc += 2

// On real hardware, the below does not happen,
// this opcode only sets stack registers
WHILE ($st3--)
    DO
        EXECUTE_OPCODE($pc)
    WHILE ($pc != $st2)
        $pc = $st0
END
$pc = addrA + 1
// Remove vaues from stack

```


5.5.20 CALL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0010				1011				1111			
aaaa				aaaa				aaaa				aaaa			

Format:

CALL addressA

Description:

Call function. Push program counter of the instruction following “call” to call stack \$st0. Set program counter to address represented by the value that follows this CALL instruction.

Operation:

```
// Must skip value that follows "call"
PUSH_STACK($st0)
$st0 = $pc + 2
$pc = addressA
```

5.5.21 CALLcc

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0010				1011				cccc			
aaaa				aaaa				aaaa				aaaa			

Format:

CALLcc addressA

Description:

Call function if condition cc has been met. Push program counter of the instruction following “call” to call stack \$st0. Set program counter to address represented by the value that follows this CALL instruction.

Operation:

```
// Must skip value that follows "call"
IF (cc)
    PUSH_STACK($st0)
    $st0 = $pc + 2
    $pc = addressA
ELSE
    $pc += 2
ENDIF
```

5.5.22 CALLR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				0111				rrr1				1111			

Format:

CALLR \$R

Description:

Call function. Push program counter of the instruction following “call” to call stack \$st0. Set program counter to register \$R.

Operation:

```
PUSH_STACK($st0)
$st0 = $pc + 1
$pc = $R
```

5.5.23 CLR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1000				r001				xxxx				xxxx			

Format:

CLR \$acR

Description:

Clears accumulator \$acR.

Operation:

\$acR = 0
FLAGS(\$acR)
\$pc++

5.5.24 CLRL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1111				110r				xxxx				xxxx			

Format:

CLRL \$acR.l

Description:

Clears \$acR.l - low 16 bits of accumulator \$acR.

Operation:

\$acR.l = 0
 FLAGS(\$acR)
 \$pc++

5.5.25 CLRP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1000				0100				xxxx				xxxx			

Format:

CLRP

Description:

Clears product register \$prod.

Operation:

```
$prod = 0 // See note below
$pc++
```

Note:

Actually product register gets cleared by setting registers with following values:

```
$14 = 0x0000
$15 = 0xffff0
$16 = 0x00ff
$17 = 0x0010
```

5.5.26 CMP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1000				0010				xxxx				xxxx			

Format:

CMP

Description:Compares accumulator `$ac0` with accumulator `$ac1`.**Operation:**

```

$sr = FLAGS($ac0 - $ac1)
$pc++

```

5.5.27 CMPI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				001r				1000				0000			
iiii				iiii				iiii				iiii			

Format:

```
CMPI $amD, #I
```

Description:

Compares mid accumulator `$acD.hm` (`$amD`) with sign-extended immediate value `I`. However, flags are set with regards to the whole accumulator register.

Operation:

```
res = ($acD.hm - I) | $acD.l
FLAGS(res)
$pc += 2
```


5.5.28 CMPIS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				011d				iiii				iiii			

Format:

`CMPIS $acD, #I`

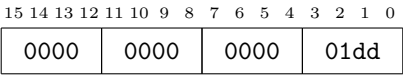
Description:

Compares accumulator with short immediate. Comparison is performed by subtracting the short immediate (8-bit sign-extended) from mid accumulator `$acD.hm` and computing flags based on whole accumulator `$acD`.

Operation:

`FLAGS($acD - #I)`
`$pc++`

5.5.29 DAR



Format:

DAR `$arD`

Description:

Decrement address register `$arD`.

Operation:

`$arD--`
`$pc++`

5.5.30 DEC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0111				101d				xxxx				xxxx			

Format:

DEC \$acD

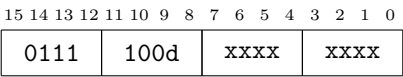
Description:

Decrements accumulator \$acD.

Operation:

\$acD--
FLAGS(\$acD)
\$pc++

5.5.31 **DECM**



Format:

```
DECM $acsD
```

Description:

Decrements 24-bit mid-accumulator **\$acsD**.

Operation:

```
$acsD--  
FLAGS($acD)  
$pc++
```

5.5.32 HALT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0000				0020				0001			

Format:

HALT

Description:

Stops execution of DSP code. Sets bit DSP_CR_HALT in register DREG_CR.

Operation:

DREG_CR |= DSP_CR_HALT;

5.5.33 IAR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0000				0000				10dd			

Format:

IAR \$arD

Description:

Increment address register \$arD.

Operation:

\$arD++
\$pc++

5.5.34 IFcc

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0010				0111				cccc			

Format:

IFcc

Description:

Executes the following opcode if the condition described by cccc has been met.

Operation:

```

IF (cc)
    EXECUTE_OPCODE($pc + 1)
ELSE
    $pc += 2
ENDIF

```

5.5.35 ILRR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				001d				0001				00ss			

Format:

ILRR \$acD.m, @\$arS

Description:

Move value from instruction memory pointed by addressing register **\$arS** to mid accumulator register **\$acD.m**.

Operation:

\$acD.m = MEM[\$arS]
\$pc++

5.5.36 ILRRD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				001d				0001				01ss			

Format:

ILRRD \$acD.m, @\$arS

Description:

Move value from instruction memory pointed by addressing register **\$arS** to mid accumulator register **\$acD.m**. Decrement addressing register **\$arS**.

Operation:

```
$acD.m = MEM[$arS]
$arS--
$pc++
```

5.5.37 ILRRI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				001d				0001				10ss			

Format:

ILRRI \$acD.m, @\$S

Description:

Move value from instruction memory pointed by addressing register **\$arS** to mid accumulator register **\$acD.m**. Increment addressing register **\$arS**.

Operation:

\$acD.m = MEM[\$arS]
 \$arS++
 \$pc++

5.5.38 ILRRN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				001d				0001				11ss			

Format:

ILRRN \$acD.m, @\$arS

Description:

Move value from instruction memory pointed by addressing register **\$arS** to mid accumulator register **\$acD.m**. Add corresponding indexing register **\$ixS** to addressing register **\$arS**.

Operation:

```
$acD.m = MEM[$arS]
$arS += $ixS
$pc++
```

5.5.39 INC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0111				011d				xxxx				xxxx			

Format:

INC \$acD

Description:

Increments accumulator \$acD.

Operation:

\$acD++
FLAGS(\$acD)
\$pc++

5.5.40 INCM

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0111				010d				xxxx				xxxx			

Format:

INCM *\$acsD*

Description:

Increments 24-bit mid-accumulator *\$acsD*.

Operation:

\$acsD++
FLAGS(*\$acD*)
\$pc++

5.5.41 JMP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0010				1001				1111			
aaaa				aaaa				aaaa				aaaa			

Format:

JMP addressA

Description:

Jumps to **addressA**. Set program counter to the address represented by the value that follows this **JMP** instruction.

Operation:

\$pc = addressA

5.5.42 Jcc

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0010				1001				cccc			
aaaa				aaaa				aaaa				aaaa			

Format:

Jcc addressA

Description:

Jumps to `addressA` if condition `cc` has been met. Set program counter to the address represented by the value that follows this Jcc instruction.

Operation:

```

IF (cc)
    $pc = addressA
ELSE
    $pc += 2
ENDIF

```

5.5.43 JMPR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				0111				rrr0				1111			

Format:

JMPR \$R

Description:

Jump to address; set program counter to a value from register \$R.

Operation:

\$pc = \$R

5.5.44 LOOP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0000				010r				rrrr			

Format:

LOOP \$R

Description:

Repeatedly execute the following opcode until the counter specified by the value from register **\$R** reaches zero. Each execution decrements the counter. Register **\$R** remains unchanged. If register **\$R** is set to zero at the beginning of loop then the looped instruction will not get executed.

Operation:

```

counter = $R
WHILE (counter--)
    EXECUTE_OPCODE($pc + 1)
END
$pc += 2

```

5.5.45 LOOPI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				0000				iiii				iiii			

Format:

LOOPI #I

Description:

Repeatedly execute the following opcode until the counter specified by immediate value I reaches zero. Each execution decrements the counter. If immediate I is set to zero at the beginning of loop then the looped instruction will not get executed.

Operation:

```

counter = I
WHILE (counter--)
    EXECUTE_OPCODE($pc + 1)
END
$pc += 2

```

5.5.46 LR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0000				110d				dddd			
mmm				mmm				mmm				mmm			

Format:

LR \$D, @M

Description:

Move value from data memory pointed by address M to register \$D. Perform an additional operation depending on destination register.

Operation:

\$D = MEM[M]
\$pc += 2

5.5.47 LRI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0000				100d				dddd			
iiii				iiii				iiii				iiii			

Format:

```
LRI $D, #I
```

Description:

Load immediate value I to register \$D. Perform and additional operation depending on destination register.

Operation:

```
$D = I
$pc += 2
```

5.5.48 LRIS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				1ddd				iiii				iiii			

Format:

LRIS \$(0x18+D), #I

Description:

Load immediate value I (8-bit sign-extended) to accumulator register \$(0x18+D). Perform an additional operation depending on destination register.

Operation:

\$(0x18+D) = I
\$pc++

5.5.49 LRR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				1000				0ssd				dddd			

Format:

LRR \$D, @\$S

Description:

Move value from data memory pointed by addressing register **\$S** to register **\$D**. Perform an additional operation depending on destination register.

Operation:

\$D = MEM[\$S]
\$pc++

5.5.50 LRRD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				1000				1ssd				dddd			

Format:

LRRD \$D, @\$S

Description:

Move value from data memory pointed by addressing register \$S to register \$D. Decrements register \$S.
Perform additional operation depending on destination register.

Operation:

\$D = MEM[\$S]
\$S--
\$pc++

5.5.51 LRRl

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				1001				0ssd				dddd			

Format:

LRRl \$D, @\$S

Description:

Move value from data memory pointed by addressing register **\$S** to register **\$D**. Increments register **\$S**.
 Perform additional operation depending on destination register.

Operation:

\$D = MEM[\$S]
 \$S++
 \$pc++

5.5.52 LRRN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				1001				1ssd				dddd			

Format:

LRRN \$D, @\$\$S

Description:

Move value from data memory pointed by addressing register \$\$S to register \$D. Add indexing register \$(0x4+S) to register \$\$S. Perform additional operation depending on destination register.

Operation:

\$D = MEM[\$\$S]
 \$\$S += \$(4+S)
 \$pc++

5.5.53 LRS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0010				0ddd				mmmm				mmmm			

Format:

LRS \$(0x18+D), @M

Description:

Move value from data memory pointed by address **M** (8-bit sign-extended) to register \$(0x18+D). Perform additional operation depending on destination register.

Operation:

\$(0x18+D) = MEM[M]
\$pc++

5.5.54 LSL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				010r				00ii				iiii			

Format:

LSL \$acR, #I

Description:

Logically left shifts accumulator \$acR by the amount specified by value I.

Operation:

```
$acR <<= I
FLAGS($acD)
$pc++
```

5.5.55 LSL16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1111				000r				xxxx				xxxx			

Format:

LSL16 \$acR

Description:

Logically left shifts accumulator \$acR by 16.

Operation:

```
$acR <<= 16
FLAGS($acD)
$pc++
```

5.5.56 LSR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				010r				01ii				iiii			

Format:

LSR \$acR, #I

Description:

Logically right shifts accumulator \$acR by the amount calculated by negating sign-extended bits 0–6.

Operation:

```
$acR >>= I
FLAGS($acD)
$pc++
```

5.5.57 LSR16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1111				010r				xxxx				xxxx			

Format:

LSR16 \$acR

Description:

Logically right shifts accumulator \$acR by 16.

Operation:

```
$acR >>= 16
FLAGS($acD)
$pc++
```

5.5.58 MADD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1111				001s				xxxx				xxxx			

Format:

MADD \$axS.l, \$axS.h

Description:

Multiply low part \$axS.l of secondary accumulator \$axS by high part \$axS.h of secondary accumulator \$axS (treat them both as signed) and add result to product register.

Operation:

```
$prod += $axS.l * $axS.h
$pc++
```

See also:

\$sr.AM bit affects multiply result.

5.5.59 MADDC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110				10st				xxxx				xxxx			

Format:

MADDc \$acS.l, \$axT.h

Description:

Multiply middle part of accumulator \$acS.m by high part of secondary accumulator \$axT.h (treat them both as signed) and add result to product register.

Operation:

```
$prod += $acS.l * $axT.h
$pc++
```

See also:

\$sr.AM bit affects multiply result.

5.5.60 MADDX

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110				00st				xxxx				xxxx			

Format:

MADDX \$(0x18+S*2), \$(0x19+T*2)

Description:

Multiply one part of secondary accumulator **\$ax0** (selected by **S**) by one part of secondary accumulator **\$ax1** (selected by **T**) (treat them both as signed) and add result to product register.

Operation:

```
$prod += $(0x18+S*2) * $(0x19+T*2)
$pc++
```

See also:

\$sr.AM bit affects multiply result.

5.5.61 MOV

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0110				110d				xxxx				xxxx			

Format:

MOV \$acD, \$ac(1-D)

Description:

Moves accumulator \$ax(1-D) to accumulator \$axD.

Operation:

\$acD = \$ax(1-D)
 FLAGS(\$acD)
 \$pc++

5.5.62 MOVAX

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0110				10sd				xxxx				xxxx			

Format:

MOVAX \$acD, \$axS

Description:

Moves secondary accumulator \$axS to accumulator \$axD.

Operation:

\$acD = \$axS
 FLAGS(\$acD)
 \$pc++

5.5.63 MOVNP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0111				111d				xxxx				xxxx			

Format:

MOVNP \$acD

Description:

Moves negated multiply product from the \$prod register to the accumulator register \$acD.

Operation:

```
$acD = -$prod
FLAGS($acD)
$pc++
```

5.5.64 MOVP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0110				111d				xxxx				xxxx			

Format:

MOVP \$acD

Description:

Moves multiply product from the \$prod register to the accumulator register \$acD.

Operation:

```
$acD = $prod
FLAGS($acD)
$pc++
```

5.5.65 MOVZ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1111				111d				xxxx				xxxx			

Format:

MOVZ \$acD

Description:

Moves multiply product from the \$prod register to the accumulator \$acD and sets \$acD.l to 0.

Operation:

```
$acD.hm = $prod.hm
$acD.l = 0
FLAGS($acD)
$pc++
```

5.5.66 MOVR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0110				0ssd				xxxx				xxxx			

Format:

MOVR \$acD, \$(0x18+S)

Description:

Moves register \$(0x18+S) (sign-extended) to middle accumulator \$acD.hm. Sets \$acD.l to 0.

Operation:

```
$acD.hm = $(0x18+S)
$acD.l = 0
FLAGS($acD)
$pc++
```

5.5.67 MRR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				11dd				ddds				ssss			

Format:

MRR \$D, \$S

Description:

Move value from register \$S to register \$D. Perform additional operation depending on destination register.

Operation:

\$D = \$S
\$pc++

5.5.68 MSUB

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1111				011s				xxxx				xxxx			

Format:

```
MSUB $axS.l, $axS.h
```

Description:

Multiply low part `$axS.l` of secondary accumulator `$axS` by high part `$axS.h` of secondary accumulator `$axS` (treat them both as signed) and subtract result from product register.

Operation:

```
$prod -= $axS.l * $axS.h
$pc++
```

See also:

`$sr.AM` bit affects multiply result.

5.5.69 MSUBC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110				11st				xxxx				xxxx			

Format:

MSUBC \$acS.m, \$axT.h

Description:

Multiply middle part of accumulator \$acS.m by high part of secondary accumulator \$axT.h (treat them both as signed) and subtract result from product register.

Operation:

```
$prod -= $acS.m * $axT.h
$pc++
```

See also:

\$sr.AM bit affects multiply result.

5.5.70 MSUBX

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110				01st				xxxx				xxxx			

Format:

MSUBX \$(0x18+S*2), \$(0x19+T*2)

Description:

Multiply one part of secondary accumulator **\$ax0** (selected by **S**) by one part of secondary accumulator **\$ax1** (selected by **T**) (treat them both as signed) and subtract result from product register.

Operation:

```
$prod -= $(0x18+S*2) * $(0x19+T*2)
$pc++
```

See also:

\$sr.AM bit affects multiply result.

5.5.71 MUL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1001				s000				xxxx				xxxx			

Format:

MUL `$axS.l`, `$axS.h`

Description:

Multiply low part `$axS.l` of secondary accumulator `$axS` by high part `$axS.h` of secondary accumulator `$axS` (treat them both as signed).

Operation:

```
$prod = $axS.l * $axS.h
$pc++
```

See also:

`$sr.AM` bit affects multiply result.

5.5.72 MULAC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1001				s10r				xxxx				xxxx			

Format:

MULAC `$axS.l`, `$axS.h`, `$acR`

Description:

Add product register to accumulator register `$acR`. Multiply low part `$axS.l` of secondary accumulator `$axS` by high part `$axS.h` of secondary accumulator `$axS` (treat them both as signed).

Operation:

```

$acR += $prod
$prod = $axS.l * $axS.h
$pc++

```

See also:

`$sr.AM` bit affects multiply result.

5.5.73 MULC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
110s					t000			xxxx				xxxx			

Format:

```
MULC $acS.m, $axT.h
```

Description:

Multiply mid part of accumulator register `$acS.m` by high part `$axS.h` of secondary accumulator `$axS` (treat them both as signed).

Operation:

```
$prod = $acS.m * $axS.h
$pc++
```

See also:

`$sr.AM` bit affects multiply result.

5.5.74 MULCAC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
110s					t10r					xxxx					xxxx				

Format:

MULCAC \$acS.m, \$axT.h, \$acR

Description:

Multiply mid part of accumulator register \$acS.m by high part \$axS.h of secondary accumulator \$axS (treat them both as signed). Add product register before multiplication to accumulator \$acR.

Operation:

```
temp = $prod
$prod = $acS.m * $axS.h
$acR += temp
$pc++
```

See also:

\$sr.AM bit affects multiply result.

5.5.75 MULCMV

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
110s					t11r					xxxx					xxxx				

Format:

MULCMV \$acS.m, \$axT.h, \$acR

Description:

Multiply mid part of accumulator register \$acS.m by high part \$axS.h of secondary accumulator \$axS (treat them both as signed). Move product register before multiplication to accumulator \$acR.

Operation:

```
temp = $prod
$prod = $acS.m * $axS.h
$acR = temp
$pc++
```

See also:

\$sr.AM bit affects multiply result.

5.5.76 MULCMVZ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
110s				t01r				xxxx				xxxx			

Format:

MULCMVZ \$acS.m, \$axT.h, \$acR

Description:

Multiply mid part of accumulator register \$acS.m by high part \$axS.h of secondary accumulator \$axS (treat them both as signed). Move product register before multiplication to accumulator \$acR. Set low part of accumulator \$acR.l to zero.

Operation:

```
temp = $prod
$prod = $acS.m * $axS.h
$acR.hm = temp.hm
$acR.l = 0
$pc++
```

See also:

\$sr.AM bit affects multiply result.

5.5.77 MULMV

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1001				s11r				xxxx				xxxx			

Format:

MULMV `$axS.l`, `$axS.h`, `$acR`

Description:

Move product register to accumulator register `$acR`. Multiply low part `$axS.l` of secondary accumulator Register `$axS` by high part `$axS.h` of secondary accumulator `$axS` (treat them both as signed).

Operation:

```

$acR = $prod
$prod = $axS.l * $axS.h
$pc++

```

See also:

`$sr.AM` bit affects multiply result.

5.5.78 MULMVZ

15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
1001	s01r xxxx xxxx

Format:

MULMVZ \$axS.l, \$axS.h, \$acR

Description:

Move product register to accumulator register \$acR and clear low part of accumulator register \$acR.l.
 Multiply low part \$axS.l of secondary accumulator \$axS by high part \$axS.h of secondary accumulator \$axS (treat them both as signed).

Operation:

```
$acR.hm = $prod.hm
$acR.l = 0
$prod = $axS.l * $axS.h
$pc++
```

See also:

\$sr.AM bit affects multiply result.

5.5.79 MULX

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
101s				t000				xxxx				xxxx			

Format:

MULX \$ax0.S, \$ax1.T

Description:

Multiply one part \$ax0 by one part \$ax1 (treat them both as signed). Part is selected by S and T bits.
Zero selects low part, one selects high part.

Operation:

```
$prod = (S == 0) ? $ax0.l : ax0.h * (T == 0) ? $ax1.l : $ax1.h
$pc++
```

See also:

\$sr.AM bit affects multiply result.

5.5.80 MULXAC

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0															
101s	t01r	xxxx	xxxx												

Format:

MULXAC \$ax0.S, \$ax1.T, \$acR

Description:

Add product register to accumulator register \$acR. Multiply one part \$ax0 by one part \$ax1 (treat them both as signed). Part is selected by S and T bits. Zero selects low part, one selects high part.

Operation:

```
$acR += $prod
$prod = (S == 0) ? $ax0.l : ax0.h * (T == 0) ? $ax1.l : $ax1.h
$pc++
```

See also:

\$sr.AM bit affects multiply result.

5.5.81 MULXMV

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
101s				t11r				xxxx				xxxx			

Format:

MULXMV \$ax0.S, \$ax1.T, \$acR

Description:

Move product register to accumulator register \$acR. Multiply one part \$ax0 by one part \$ax1 (treat them both as signed). Part is selected by S and T bits. Zero selects low part, one selects high part.

Operation:

```
$acR = $prod
$prod = (S == 0) ? $ax0.l : ax0.h * (T == 0) ? $ax1.l : $ax1.h
$pc++
```

See also:

\$sr.AM bit affects multiply result.

5.5.82 MULXMVZ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
101s				t01r				xxxx				xxxx			

Format:

MULXMVZ \$ax0.S, \$ax1.T, \$acR

Description:

Move product register to accumulator register \$acR and clear low part of accumulator register \$acR.l. Multiply one part \$ax0 by one part \$ax1 (treat them both as signed). Part is selected by S and T bits. Zero selects low part, one selects high part.

Operation:

```
$acR.hm = $prod.hm
$acR.l = 0
$prod = (S == 0) ? $ax0.l : ax0.h * (T == 0) ? $ax1.l : $ax1.h
$pc++
```

See also:

\$sr.AM bit affects multiply result.

5.5.83 **NEG**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0111				110d				xxxx				xxxx			

Format:

NEG \$acD

Description:

Negates accumulator \$acD.

Operation:

\$acD =- \$acD
FLAGS(\$acD)
\$pc++

5.5.84 NOP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0000				0000				0000			

Format:

NOP

Description:

No operation.

Operation:

\$pc++

5.5.85 NX

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1000				-000				xxxx				xxxx			

Format:

NX

Description:

No operation, but can be extended with extended opcode.

Operation:

\$pc++

5.5.86 ORC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0011				111d				xxxx				xxxx			

Format:

ORC $\$acD.m$, $\$ac(1-D).m$

Description:

Logic OR middle part of accumulator $\$acD.m$ with middle part of accumulator $\$ax(1-D).m$.

Operation:

```

$acD.m |= $ac(1-D).m
FLAGS($acD)
$pc++

```

5.5.87 ORI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				001r				0110				0000			
iiii				iiii				iiii				iiii			

Format:

```
ORI $acD.m, #I
```

Description:

Logical OR of accumulator mid part `$acD.m` with immediate value `I`.

Operation:

```
$acD.m |= #I
FLAGS($acD)
$pc += 2
```

5.5.88 ORR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0011				10sd				xxxx				xxxx			

Format:

```
ORR $acD.m, $axS.h
```

Description:

Logical OR middle part of accumulator **\$acD.m** with high part of secondary accumulator **\$axS.h**.

Operation:

```
$acD.m |= $axS.h
FLAGS($acD)
$pc++
```

5.5.89 RET

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0010				1101				1111			

Format:

RET

Description:

Return from subroutine. Pops stored PC from call stack `$st0` and sets `$pc` to this location.

Operation:

```
$pc = $st0
POP_STACK($st0)
```

5.5.90 RETcc

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0010				1101				cccc			

Format:

RETcc

Description:

Return from subroutine if condition `cc` has been met. Pops stored PC from call stack `$st0` and sets `$pc` to this location.

Operation:

```

IF (cc)
    POP_STACK($st0)
ELSE
    $pc += 2
ENDIF

```

5.5.91 RTI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0010				1111				1111			

Format:

RTI

Description:

Return from exception. Pops stored status register **\$sr** from data stack **\$st1** and program counter PC from call stack **\$st0** and sets **\$pc** to this location.

Operation:

```

$sr = $st1
POP_STACK($st1)
$pc = $st0
POP_STACK($st0)

```


5.5.92 SBSET

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				0010				0000				0iii			

Format:

SBSET #I

Description:

Set bit of status register `$sr`. Bit number is calculated by adding 6 to immediate value I.

Operation:

```
$sr |= (I + 6)
$pc++
```

5.5.93 SBCLR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				0011				0000				0iii			

Format:

SBCLR #I

Description:

Clear bit of status register **\$sr**. Bit number is calculated by adding 6 to immediate value I.

Operation:

```
$sr &= ~(I + 6)
$pc++
```

5.5.94 SI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				0110				mmmm				mmmm			
iiii				iiii				iiii				iiii			

Format:

SI @M, #I

Description:

Store 16-bit immediate value I to a memory location pointed by address M (M is an 8-bit sign-extended value).

Operation:

MEM[M] = I
\$pc += 2

5.5.95 SR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0000				111s				ssss			
mmmm				mmmm				mmmm				mmmm			

Format:

SR @M, \$S

Description:

Store value from register \$S to a memory pointed by address M. Perform additional operation depending on destination register.

Operation:

MEM[M] = \$S
\$pc += 2

5.5.96 SRR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				1010				0dds				ssss			

Format:

SRR @D, \$S

Description:

Store value from source register \$S to a memory location pointed by addressing register \$D. Perform additional operation depending on source register.

Operation:

MEM[\$D] = \$S
\$pc++

5.5.97 SRRD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				1010				1dds				ssss			

Format:

SRRD @D, S

Description:

Store value from source register **S** to a memory location pointed by addressing register **D**. Decrement register **D**. Perform additional operation depending on source register.

Operation:

MEM[D] = S
 D--
 pc++

5.5.98 SRRI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				1011				0dds				ssss			

Format:

SRRI @D, \$S

Description:

Store value from source register **\$S** to a memory location pointed by addressing register **\$D**. Increment register **\$D**. Perform additional operation depending on source register.

Operation:

MEM[\$D] = \$S
 \$D++
 \$pc++

5.5.99 SRRN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001				1011				1dds				ssss			

Format:

SRRN @D, \$S

Description:

Store value from source register \$S to a memory location pointed by addressing register \$D. Add indexing register \$(0x4+D) to register \$D. Perform additional operation depending on source register.

Operation:

```
MEM[$D] = $S
$D += $(4+D)
$pc++
```


5.5.100 SRS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0010				1sss				mmmm				mmmm			

Format:

SRS @M, \$(0x18+S)

Description:

Store value from register \$(0x18+S) to a memory pointed by address M (8-bit sign-extended). Perform additional operation depending on destination register.

Operation:

MEM[M] = \$(0x18+S)
\$pc++

5.5.101 SUB

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0101				110d				xxxx				xxxx			

Format:

```
SUB $acD, $ac(1-D)
```

Description:

Subtracts accumulator `$ac(1-D)` from accumulator register `$acD`.

Operation:

```
$acD -= $ac(1-D)
FLAGS($acD)
$pc++
```

5.5.102 SUBAX

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0101				10sd				xxxx				xxxx			

Format:

SUBAX \$acD, \$axS

Description:

Subtracts secondary accumulator \$axS from accumulator register \$acD.

Operation:

\$acD -= \$axS
FLAGS(\$acD)
\$pc++

5.5.103 SUBP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0101				111d				xxxx				xxxx			

Format:

SUBP \$acD

Description:

Subtracts product register from accumulator register.

Operation:

\$acD -= \$prod
FLAGS(\$acD)
\$pc++

5.5.104 SUBR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0101				0ssd				xxxx				xxxx			

Format:

SUBR \$acD, \$(0x18+S)

Description:

Subtracts register \$(0x18+S) from accumulator \$acD register.

Operation:

\$acD -= \$(0x18+S)
FLAGS(\$acD)
\$pc++

5.5.105 TST

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1011				r001				xxxx				xxxx			

Format:

TST \$acR

Description:

Test accumulator \$acR.

Operation:

FLAGS(\$acR)
\$pc++

5.5.106 TSTAXH

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1000				011r				xxxx				xxxx			

Format:

TSTAXH \$axR.h

Description:

Test hight part of secondary accumulator \$axR.h.

Operation:

FLAGS(\$axR.h)
\$pc++

5.5.107 XORI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				001r				0010				0000			
iiii				iiii				iiii				iiii			

Format:

```
XORI $acD.m, #I
```

Description:

Logical XOR (exclusive OR) of accumulator mid part \$acD.m with immediate value I.

Operation:

```
$acD.m ^= #I
FLAGS($acD)
$pc += 2
```


5.5.108 XORR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0011				00sd				xxxx				xxxx			

Format:

`XORR $acD.m, $axS.h`

Description:

Logical XOR (exclusive OR) middle part of accumulator `$acD.m` with high part of secondary accumulator `$axS.h`.

Operation:

`$acD.m ^= $axS.h`
`FLAGS($acD)`
`$pc++`

5.6 Extended opcodes

Extended opcodes do not exist on their own. These opcodes can only be attached to opcodes that allow extending (8 lower bits of opcode not used by opcode). Extended opcodes do not modify the program counter (`$pc` register).

5.7 Alphabetical list of extended opcodes

5.7.1 'DR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xxxx				xxxx				0000				01rr			

Format:

'DR \$arR

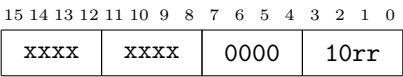
Description:

Decrement addressing register \$arR.

Operation:

\$arR--

5.7.2 'IR



Format:

'IR \$arR

Description:

Increment addressing register \$arR.

Operation:

\$arR++

5.7.3 'L

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xxxx				xxxx				01dd				d0ss			

Format:

```
'L $(0x18+D), @$S
```

Description:

Load register `$(0x18+D)` with value from memory pointed by register `$S`. Post increment register `$S`.

Operation:

```
$(0x18+D) = MEM[$S]
$S++
```

5.7.4 'LN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xxxx				xxxx				01dd				d1ss			

Format:

'LN \$(0x18+D), @\$\$S

Description:

Load register \$(0x18+D) with value from memory pointed by register \$\$S. Add indexing register register \$(0x04+S) to register \$\$S.

Operation:

\$(0x18+D) = MEM[\$\$S]
 \$\$S += \$(0x04+S)

5.7.5 'LS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xxxx				xxxx				01dd				d1ss			

Format:

```
'LS $(0x18+D), $acS.m
```

Description:

Load register `$(0x18+D)` with value from memory pointed by register `$ar0`. Store value from register `$acS.m` to memory location pointed by register `$ar3`. Increment both `$ar0` and `$ar3`.

Operation:

```
$(0x18+D) = MEM[$ar0]
MEM[$ar3] = $acS.m
$ar0++
$ar3++
```


5.7.6 'LSM

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xxxx				xxxx				10dd				100s			

Format:

```
'LSM $(0x18+D), $acS.m
```

Description:

Load register `$(0x18+D)` with value from memory pointed by register `$ar0`. Store value from register `$acS.m` to memory location pointed by register `$ar3`. Add corresponding indexing register `$ix3` to addressing register `$ar3` and increment `$ar0`.

Operation:

```
$(0x18+D) = MEM[$ar0]
MEM[$ar3] = $acS.m
$ar0++
$ar3 += $ix3
```

5.7.7 'LSNM

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xxxx				xxxx				10dd				110s			

Format:

```
'LSNM $(0x18+D), $acS.m
```

Description:

Load register `$(0x18+D)` with value from memory pointed by register `$ar0`. Store value from register `$acS.m` to memory location pointed by register `$ar3`. Add corresponding indexing register `$ix0` to addressing register `$ar0` and add corresponding indexing register `$ix3` to addressing register `$ar3`.

Operation:

```
$(0x18+D) = MEM[$ar0]
MEM[$ar3] = $acS.m
$ar0 += $ix0
$ar3 += $ix3
```

5.7.8 'LSN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xxxx				xxxx				10dd				010s			

Format:

```
'LSN $(0x18+D), $acS.m
```

Description:

Load register `$(0x18+D)` with value from memory pointed by register `$ar0`. Store value from register `$acS.m` to memory location pointed by register `$ar3`. Add corresponding indexing register `$ix0` to addressing register `$ar0` and increment `$ar3`.

Operation:

```
$(0x18+D) = MEM[$ar0]
MEM[$ar3] = $acS.m
$ar0 += $ix0
$ar3++
```

5.7.9 'MV

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xxxx				xxxx				0001				ddss			

Format:

'MV \$(0x18+D), \$(0x1c+S)

Description:

Move value of register \$(0x1c+S) to the register \$(0x18+D).

Operation:

\$(0x18+D) = \$(0x1c+S)

5.7.10 'NR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xxxx				xxxx				0000				11rr			

Format:

'NR \$arR

Description:

Add corresponding indexing register \$ixR to addressing register \$arR.

Operation:

\$arR += \$ixR

5.7.11 'S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xxxx				xxxx				001s				s0dd			

Format:

'S @ \$D, \$(0x1c+D)

Description:

Store value of register \$(0x1c+S) in the memory pointed by register \$D. Post increment register \$D.

Operation:

MEM[\$D] = \$(0x1c+D)
\$S++

5.7.12 'SL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xxxx				xxxx				10dd				001s			

Format:

'SL \$acS.m, \$(0x18+D)

Description:

Store value from register \$acS.m to memory location pointed by register \$ar0. Load register \$(0x18+D) with value from memory pointed by register \$ar3. Increment both \$ar0 and \$ar3.

Operation:

\$(0x18+D) = MEM[\$ar3]
 MEM[\$ar0] = \$acS.m
 \$ar0++
 \$ar3++

5.7.13 'SLM

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xxxx				xxxx				10dd				101s			

Format:

```
'SLM $acS.m, $(0x18+D)
```

Description:

Store value from register `$acS.m` to memory location pointed by register `$ar0`. Load register `$(0x18+D)` with value from memory pointed by register `$ar3`. Add corresponding indexing register `$ix3` to addressing register `$ar3` and increment `$ar0`.

Operation:

```
$(0x18+D) = MEM[$ar3]
MEM[$ar0] = $acS.m
$ar0++
$ar3 += $ix3
```


5.7.14 'SLMN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xxxx				xxxx				10dd				111s			

Format:

'SLMN \$acS.m, \$(0x18+D)

Description:

Store value from register \$acS.m to memory location pointed by register \$ar0. Load register \$(0x18+D) with value from memory pointed by register \$ar3. Add corresponding indexing register \$ix0 to addressing register \$ar0 and add corresponding indexing register \$ix3 to addressing register \$ar3.

Operation:

```
$(0x18+D) = MEM[$ar3]
MEM[$ar0] = $acS.m
$ar0 += $ix0
$ar3 += $ix3
```

5.7.15 'SLN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xxxx				xxxx				10dd				011s			

Format:

```
'SLN $acS.m, $(0x18+D)
```

Description:

Store value from register `$acS.m` to memory location pointed by register `$ar0`. Load register `$(0x18+D)` with value from memory pointed by register `$ar3`. Add corresponding indexing register `$ix0` to addressing register `$ar0` and increment `$ar3`.

Operation:

```
$(0x18+D) = MEM[$ar3]
MEM[$ar0] = $acS.m
$ar0 += $ix0
$ar3++
```

5.7.16 'SN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xxxx				xxxx				001s				s1dd			

Format:

'SN @\$D, \$(0x1c+D)

Description:

Store value of register \$(0x1c+S) in the memory pointed by register \$D. Add indexing register register \$(0x04+D) to register \$D.

Operation:

MEM[\$D] = \$(0x1c+D)
\$D += \$(0x04+D)

5.8 Instructions sorted by opcode

NOP		*	0000 0000 0000 0000
DAR		*	0000 0000 0000 01aa
IAR		*	0000 0000 0000 10aa
XXX	NOT USED	*	0000 0000 0000 11xx
ADDARN		*	0000 0000 0001 bbaa
HALT		*	0000 0000 0010 0001
LOOP		*	0000 0000 010r rrrr
BLOOP		*	00000 0000 011r rrrr
LRI		*	0000 0000 100r rrrr iiii iiii iiii iiii
XXX	NOT USED	*	0000 0000 101x xxxx
LR		*	0000 0000 110r rrrr mmmm mmmm mmmm mmmm
SR		*	0000 0000 111r rrrr mmmm mmmm mmmm mmmm
IF cc		*	0000 0010 0111 cccc
JMP cc		*	0000 0010 1001 cccc
CALL cc		*	0000 0010 1011 cccc
RET cc		*	0000 0010 1101 cccc
ADDI		*	0000 001r 0000 0000 iiii iiii iiii iiii
XORI		*	0000 001r 0010 0000 iiii iiii iiii iiii
ANDI		*	0000 001r 0100 0000 iiii iiii iiii iiii
ORI		*	0000 001r 0110 0000 iiii iiii iiii iiii
CMPI		*	0000 001r 1000 0000 iiii iiii iiii iiii
ANDCF		*	0000 001r 1010 0000 iiii iiii iiii iiii
ANDF		*	0000 001r 1100 0000 iiii iiii iiii iiii
ILRR		*	0000 001r 0001 mmaa
ADDIS		*	0000 010d iiii iiii
CMPIs		*	0000 011d iiii iiii
LRIS		*	0000 1rrr iiii iiii
LOOPI		*	0001 0000 iiii iiii aaaa aaaa aaaa aaaa
BLOOPI		*	0001 0001 iiii iiii aaaa aaaa aaaa aaaa
SBSET		*	0001 0010 ???? ?iii
SBCLR		*	0001 0011 ???? ?iii
LSL/LSR		*	0001 010r 0sss ssss
ASL/ASR		*	0001 010r 1sss ssss
SI		*	0001 0110 iiii iiii mmmm mmmm mmmm mmmm
CALLR		*	0001 0111 rrr1 1111
JMPR		*	0001 0111 rrr0 1111
LRR(I D X)		*	0001 100x xaar rrrr
SRR(I D X)		*	0001 101x xaar rrrr
MRR		*	0001 11dd ddds ssss
LRS		*	0010 0rrr mmmm mmmm
SRS		*	0010 1rrr mmmm mmmm
XORR		*	0011 00sr xxxx xxxx
ANDR		*	0011 01sr xxxx xxxx
ORR		*	0011 10sr xxxx xxxx
ANDC		*	0011 110r xxxx xxxx
ORC		*	0011 111r xxxx xxxx

ADDR		*	0100 0ssd xxxx xxxx
ADDAX		*	0100 10sd xxxx xxxx
ADD		*	0100 110d xxxx xxxx
ADDP		*	0100 111d xxxx xxxx
SUBR		*	0101 0ssd xxxx xxxx
SUBAX		*	0101 10sd xxxx xxxx
SUB		*	0101 110d xxxx xxxx
SUBP		*	0101 111d xxxx xxxx
MOVR		*	0110 0ssd xxxx xxxx
MOVAX		*	0110 10sd xxxx xxxx
MOV		*	0110 110d xxxx xxxx
MOVP		*	0110 111d xxxx xxxx
ADDAXL		*	0111 00sr xxxx xxxx
INCM		*	0111 010r xxxx xxxx
INC		*	0111 011r xxxx xxxx
DECM		*	0111 100r xxxx xxxx
DEC		*	0111 101r xxxx xxxx
NEG		*	0111 110r xxxx xxxx
MOVNP		*	0111 111r xxxx xxxx
NX		*	1000 x000 xxxx xxxx
CLR		*	1000 x001 xxxx xxxx
CMP		*	1000 0010 xxxx xxxx
???	UNUSED	*	1000 0011 xxxx xxxx
CLRP		*	1000 0100 xxxx xxxx
TSTAXH		*	1000 011x xxxx xxxx
M0/M2			1000 101x xxxx xxxx
CLR15/SET15			1000 110x xxxx xxxx
SET40/16			1000 111x xxxx xxxx
MUL		*	1001 a000 xxxx xxxx
ASR16		*	1001 r001 xxxx xxxx
MULMVZ		*	1001 a01r xxxx xxxx
MULAC		*	1001 a10r xxxx xxxx
MULMV		*	1001 a11r xxxx xxxx
MULX		*	101b a000 xxxx xxxx
???			1010 r001 xxxx xxxx
MULXMVZ		*	101b a01r xxxx xxxx
MULXAC		*	101b a10r xxxx xxxx
MULXMV		*	101b a11r xxxx xxxx
MULC		*	110s a000 xxxx xxxx
CMP		*	110x r001 xxxx xxxx
MULCMVZ		*	110s a01r xxxx xxxx
MULCAC		*	110s a10r xxxx xxxx
MULCMV		*	110s a11r xxxx xxxx
MADDX		**	1110 00st xxxx xxxx
MSUBX		**	1110 01st xxxx xxxx
MADDC		**	1110 10st xxxx xxxx
MSUBC		**	1110 11st xxxx xxxx
LSL16		*	1111 000r xxxx xxxx
MADD		*	1111 001s xxxx xxxx

LSR16	*	1111 010r xxxx xxxx
MSUB	*	1111 011s xxxx xxxx
ADDPAXZ	*	1111 10ar xxxx xxxx
CLRL	*	1111 110r xxxx xxxx
MOVPZ	*	1111 111r xxxx xxxx

Extension Opcodes

[D I N]R	*	xxxx xxxx 0000 nnaa
MV	*	xxxx xxxx 0001 ddss
S[N]	*	xxxx xxxx 001r rnaa
L[N]	*	xxxx xxxx 01dd diss
LS[NM M N]	*	xxxx xxxx 10dd ba0r
SL	*	xxxx xxxx 10dd ba1r
LD[NM M N]		xxxx xxxx 11mn barr
LD2[NM M N]		xxxx xxxx 11rm ba11