GameCube DSP User's Manual

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

March 10, 2020
v0.0.6

Contents

Disclaimer

Version History

Version	**Date**	**Author**	**Change**
0.0.1	2005.05.08	Duddie	Initial release
0.0.2	2005.05.09	Duddie	Added `$prod` and `$config` 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

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`
 	
`0x0FFF`
	`IRAM`

`0x8000`
 	
`0x8FFF`
	`IROM`

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:

```
$acX = $acX.h << 32 | $acX.m << 16 | $acX.l
```

Short accumulators `$acs.X`:

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

```
$acsX = $acX.h << 16 | $acX.m
```

Additional accumulators `$axX`:

```
$axX = $axX.h << 16 | $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 = `set16`, 1 = `set40`)
13	`AM`	Product multiply result by 2 (when `AM = 0`)
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.

```
$prod = ($prod.h << 32) + (($prod.m1 + $prod.m2) << 16) + $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**
Mailboxes		
`0xFFFE`	`CMBH`	CPU Mailbox H
`0xFFFF`	`CMBL`	CPU Mailbox L
`0xFFFC`	`DMBH`	DSP Mailbox H
`0xFFFD`	`DMBL`	DSP Mailbox L
DMA Interface		
`0xFFCE`	`DSMAH`	Memory address H
`0xFFCF`	`DSMAL`	Memory address L
`0xFFCD`	`DSPA`	DSP memory address
`0xFFC9`	`DSCR`	DMA control
`0xFFCB`	`DSBL`	Block size
Accelerator		
`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
Interrupts		
`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.

`0xFFFB`	`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.

`0xFFFC`	`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

`0xFFFD`	`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 : $ac1.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

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1001	r001	xxxx	xxxx

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 = I
$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 = 0xfff0
$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

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

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

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
0111	100d	xxxx	xxxx

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
mmmm	mmmm	mmmm	mmmm

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 LRRI

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

Format:

```
LRRI $D, @$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 MOVPZ

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

Format:

```
MOVPZ $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

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

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