

**68080**

**PROGRAMMER'S  
REFERENCE  
MANUAL**

**The New Instructions**

**Concept by Tommo**

Based on the original hardcopy PRM from motorola.

One page - One instruction  
Simplified

## **Index**

68080 user model

About the 080 Integer unit

About the 080 FPU

About AMMX

Integer Fusing

List of Integer, Floating point, ammx

List of ammx vector handling by type

- Integer instructions
- AMMX instructions
- Floating point instructions

Second pipe

## **080 tools:**

assembler VASM

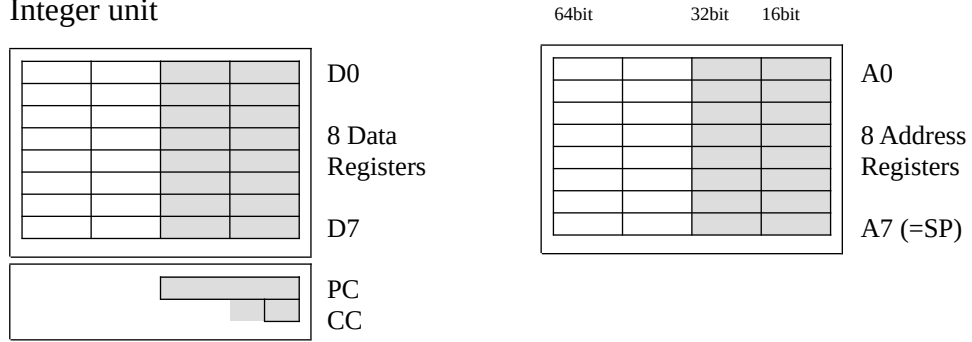
debugger Devpac's "Vamped" MonAm 3.09

feb 2024

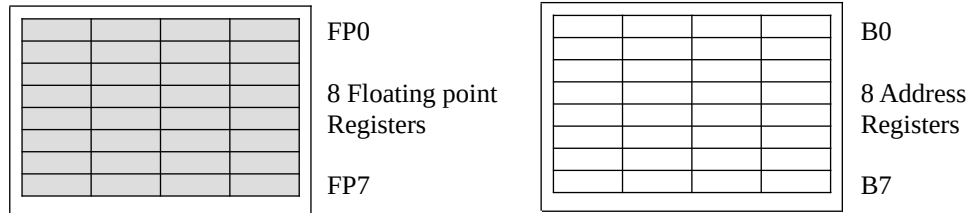
# 68080 user programming model.

Legacy model in gray.

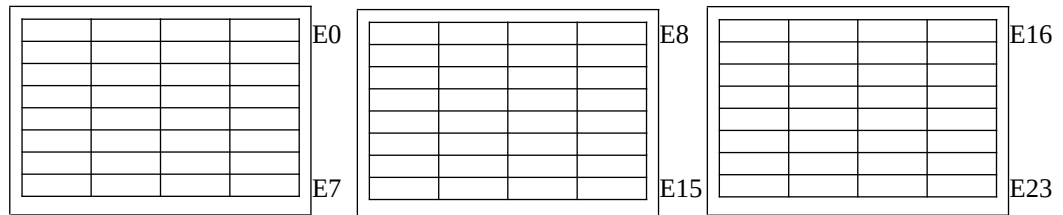
## Integer unit



## Float unit:



## General purpose for Integer & Float E0..E23



# THE 080 INTEGER UNIT.

The 68080 integer unit is a 3 operand unit. It has access to 4 times as many data registers, D0..7 & E0..23. This is done by the BANK prefix. The 3 operand is not supported by that yet. Many instructions can use e-registers, but not all.

To make existing code faster, the 3 operand can fuse instructions together as a single 3 operand instruction. Example:

```
move.l #120,d2 + add.l d1,d2 → add.l #120,d1,d2
```

Like the 060 the 080 has a second pipe, so it can also execute two instructions parallel. The Icache feeding to both pipes is not so limited as the 060, it is 16 byte.

It also has conditional instruction, a faster way to execute a single following instruction after bcc.s.

```
bcc.s skip  
<one_instruction>  
skip
```

The 080 also has 2 times as much address registers. A0..7 is the standard and B0..7 is 080 only.

The b-registers are fully interchangeable for ammx instructions, but not for the legacy instructions.

The addressing unit can handle almost all <ea> that where not possible on instructions with previous cpu's. Pc-relative is a good example of that. *However pc-relative as destination is curenly not calculated correct when there are extension words used for the source operand.*

Registers are 64bit, but integer uses 32bit so movem.l is enough to multitask older programs.

## Integer instructions that use e-registers:

```
move(3/q), add(i), sub(i), (e)or(i), and(i), not, neg(x), (b)tst, (b)clr, bset,mov(s/z)  
ext(u), exg, cmp, pack, unpk, abcd, sbcd, addx, subx, as(r/l), ls(r/l), ro(x)(r/l),  
bf(clr/set/tst/chg), moviw.l
```

*! cores handle the banking not always correct yet.*

# THE 080 FLOATING POINT UNIT.

The 68881/2 units are 80 bit, 4 nibbles exponent + 16 nibbles fraction. They are not pipelined.

The 68080 float unit is 64 bit, 3 nibbles exponent + 11 nibbles fraction. It uses the 64bit double-real format. It can store 13 nibbles fraction, but 11 are used for calculation.

It is a 3 operand unit. It has access to 4 times as many float registers, FP0..7 & E0..23. This is done by the BANK prefix.

Example: `fmul.w e4,fp3,e5` ( bank 1,0,%01.110 & `fmul.w d4,fp3` )  
Convert `e4.w` to float, multiply with `fp3` and put result in float register `e5`.

E0..23 are shared with the integer unit.

The FPU is a fully pipelined unit that can accept an instruction every clock cycle. When a result is needed that is not ready yet, it waits until it is ready. The user does not have to keep track of anything.

You can make it fast by executing other instructions that are not dependent on the result in the mean while.

<ea> can be Double format from/to data-register.

The 080 has no implementation of packed bcd real size. (same on 040 & 060)

The extended format has a different layout, see `fmovem`.

B-address registers can not be used. (yet?)

## **THE 080 AMMX UNIT.**

Ammx stands for Apollo MultiMedia eXtension. It is a coprocessor with id=7. It handles 64 bit instructions.

It has access to 32 64bit wide data registers, D0..7 & E0..23. The same as the integer unit can use. The data can be a multiple bytes or words that are processed in one go. So called Single Instruction Multiple Data or SIMD.

Instructions are normally 3 operand, unlike the 2 operand 68k. It can use all the address registers, A0..7 & B0..7.

# INTEGER INSTRUCTION FUSING

1	2	comment
move.l (an)+,(am)+	move.l (an)+,(am)+	quad move
move.l (an)+,dn	move.l (an)+,dm	quad move
move.l dn,(an)+	move.l dm,(an)+	quad move
move.l dn,dm	not.b/w/l dm	& neg, addq, subq
move.l dn,dm	addi.l #,dm	& subi
move.l dn,dm	add.b/w/l dx,dm	& sub, and, or
moveq #,dn	and.(b/w/l) dx,dn	& or
move.l dn,dm	andi.w #,dm	
		all shifters except rox(r/l)
move.l dn,dm	asr.b/w/l #im,dm	& as, ls, ro(r/l)
move.l dn,dm	asr.b/w/l dx,dm	& as, ls, ro(r/l)
moveq #,dn	move.b (ea),dn	movz.b (ea),dn
moveq #,dn	move.w (ea),dn	movz.w (ea),dn
move.l (ea),dn	extb.l dn	
move.w (ea),dn	ext.l dn	movs.w (ea),dn
ext.w dn	ext.l dn	extb.l
subq.l #1,dn	bne.s	almost dbra

## INTEGER INSTRUCTIONS

addiw.l	dbcc.l	movex
addq bn	extub/w	moviw.l !new
bank	lea bn	movs !new
bcc.s+	mov3q !new	movz !new
bra.s+	move bn	perm
bsr.s+	move sr	subq bn
clr.q !new	move16	touch
cmp bn	move2 !new	
cmpiw.l	movea bn	

## FLOATING POINT INSTRUCTIONS

fdbcc.l	fmove(u)rz !new
fmove fstorei floati	fmovem

## AMMX INSTRUCTIONS

bfly	pavg	store
bsel	pcmpccb	storec
c2p	pcmpccw	storei
load	peor	storeilm
loadi	pmaxb	storem
lsdq	pmaxw	storem3
minterm	pminb	tex
pack3216	pminw	trans
packuswb	pmul	unpack1632
padd	pmula	vperm
pand	por	
pandn	psub	



Ammx vector handling by type, source or destination match:

## Vector Bit

bssel	pand	peor
minterm	pandn	por

## Vector Byte

bfly	pcmpccb	storeilm
c2p	pmaxb	storem
packuswb	pminb	storem3
padd	psub	tex
pavg	storec	vperm

## Vector Word

bfly	pmaxw	psub
pack3216	pminw	storem3
packuswb	pmul88	tex
padd	pmulh	trans
pcmpccw	pmull	unpack1632

## Vector Long

pack3216	storem3	unpack1632
pmula		

## Vector Quad

c2p	store	storem
load	storec	storem3
loadi	storei	vperm
lsdq	storeilm	





# BANK

# BANK

## Bank

**Operation:** Inform the next instruction apollo-registers are used.

**Syntax:** - none -

**Short:** Prefix for legacy instructions.

**Description:** Bank gives older instructions more bits to select more registers.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	c	c	c	1	S	S	C	C	A	A	B	B

AA extends source from 8 to 32 possible sources.

BB extends destination from 8 to 32 possible sources.

CCccc is xored to BBbbb to create a third operand.

If CCccc <> 0 then BBbbb xor CCccc → DDddd

instr a ? b → d

else

instr a ? b → b

endif

SS Size is the length of the whole bundle = opcode length + bank\_length

0 = 4 bytes

1 = 6 bytes

2 = 8 bytes

3 = 10 bytes

Data and Float registers share the same extra registers when banked.

AA & BB:

0 = original

1 = E0 - E7

2 = E8 - E15

3 = E16 - E23

Note:

For a single operand instruction, both AA and BB should be the same.

# Bcc

# Bcc

## Branch Conditional

**Operation:** If cc then PC + dn → PC

**Syntax:** Bcc.S+ <label>

Bcc.B <label>

**Short:** Conditional jump to label.

**Description:** If the condition is true then the program execution continues at location (PC) + displacement. The displacement is always even. For short it can appear as odd, then it extends the range by 2. This extended size is named “b” “s2” or “s+”

**Condition Codes:** not affected

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	Condition				Short Extended Displacement							1

Range bcc.s : -128 .. +126

Range bcc.s+: -256 .. -130 & 128 .. 254

Note:

Variant on bcc.s

# BRA

# BRA

## Branch

**Operation:** PC + dn → PC

**Syntax:** BRA.S+ <label>

BRA.B <label>

**Short:** Program continues at label.

**Description:** Program execution continues at location (PC) + displacement. The displacement is always even. For short it can appear as odd, then it extends the range by 2. This extended size is named “b” “s2” or “s+”

**Condition Codes:** not affected

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

Range bra.s : -128 .. +126

Range bra.s+: -256 .. -130 & 128 .. 254

Note:

Variant on bra.s

# BSR

# BSR

## Branch to SubRoutine

**Operation:**  $SP - 4 \rightarrow SP ; PC \rightarrow (SP) ; PC + dn \rightarrow PC$

**Syntax:** BSR.S+ <label>

BSR.B <label>

**Short:** Push PC to stack & program continues at label.

**Description:** Pushes the long-word address of the instruction immediately following the BSR instruction onto the system stack. The program execution continues at location (PC) + displacement. The displacement is always even. For short it can appear as odd, then it extends the range by 2. This extended size is named "b" "s2" or "s+"

**Condition Codes:** not affected

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

Range bra.s : -128 .. +126

Range bra.s+: -256 .. -130 & 128 .. 254

Note:

Variant on bsr.s

# CLR

# CLR

## Clear

**Operation:** 0 → <ea>

**Syntax:** CLR <ea>

**Short:** Clears destination.

**Description:** Clears the destination to zero. Size is byte, word, long or quad.

### Condition Codes:

X	N	Z	V	C
-	0	1	0	0

### Quad:

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

### Others:

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

Size 0=byte, 1=word, 2= long

### Example:

clr.q d0

ea d0

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Result:

ea d0

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

### Note:

Handy for fast clearing memory.



# CMP

# CMP

## Compare

**Operation:**  $D_n - B_n \rightarrow cc$

**Syntax:** `CMP Bn,Dn`

**Short:** Subtract & use only the condition.

**Description:** Subtracts the source from the destination and sets the condition codes according to the result. Size is long.

### Condition Codes:

X	N	Z	V	C
-	*	*	*	*

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

# CMPIW

# CMPIW

## Compare Immediate Word extended Long

**Operation:** <ea> – data → cc

**Syntax:** CMPIW.L #<data>, <ea>

**Short:** Subtract & use only the condition.

**Description:** Sign extend immediate word data to long, subtract it from the destination and sets the condition codes according to the result. Size is long.

### Condition Codes:

X	N	Z	V	C
–	*	*	*	*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	0	Mode			Register		
16-bit word data															

Note:

There is no subiw.l

# DBcc

# DBcc

## Test, Decrement & Branch Conditional

**Operation:** If not cc then ( Dn - 1 → Dn ; if Dn <> - 1 then PC + dn → PC )

**Syntax:** DBcc.L Dn,<label>

**Short:** Test failed? then Decrement Dn & conditional jump to label.

**Description:** Controls a loop of instructions.If condition is true the loop ends and the program continues with the next instruction.

Else Dn is decremented by 1.

If Dn = - 1 the loop also ends and the program continues with the next instruction.

If Dn <> - 1 the loop continues and the program execution continues at location (PC) + displacement.

The displacement is always even. When it appears as odd, then the counter is a long, not a word.

**Condition Codes:** not affected

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	1	Condition				1	1	0	0	1	Register			
16 bit Displacement														1		

Note:

Variant on dbcc. Here Dn is a long counter, not a word counter.

Dbra is accepted by most assemblers for dbf. With dbf no condition is tested, only a count terminates the loop. This crippled unofficial version is ironically about the only one used of the group.







# MOVE

# MOVE

## Move

**Operation:** Bn → <ea>

**Syntax:** MOVE Bn,<ea>

**Description:** Move B-address register into destination. Size is long.

### Condition Codes:

X	N	Z	V	C
-	*	*	0	0

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

Can be banked so <ea> includes En.

Example:

```
move.l b0,d1
```

bn b0

								0	0	1	2	3	4	5	6
--	--	--	--	--	--	--	--	---	---	---	---	---	---	---	---

Result:

ea d1

								0	0	1	2	3	4	5	6
--	--	--	--	--	--	--	--	---	---	---	---	---	---	---	---

# MOVE sr

# MOVE sr

## Move from status register

**Operation:** sr → d

**Syntax:** MOVE sr,<ea>

**Description:** Moves status register to the destination.

**Condition Codes:** not affected

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

Size is word

Note:

PRIVILEGED INSTRUCTION.

Except on the 68000 & 68080 where reading the status register may be done in user mode.



# MOVE16

# MOVE16

## Move 16-byte block

**Operation:** memory: source → destination

**Syntax:** MOVE16 (Ax)+,(Ay)+

MOVE16 (An)+,abs.l

MOVE16 (An),abs.l

MOVE16 abs.l,(An)+

MOVE16 abs.l,(An)

**Description:** Moves 16 bytes memory to the destination.

The absolute is always a long extension word.

**Condition Codes:** not affected

first:

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

Others:

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

mode m=0: (An)+ else (An)

direction d=0: register → absolute else absolute → register

Note:

Move16 is seen as line-F coprocessor with id=3, like touch.

Introduced on the 68040 with a 16 byte alignment restriction,

move16 does **NOT** has to be aligned on the 68080.

# MOVE2

# MOVE2

## Move two

**Operation:** Source pair → destination pair

**Syntax:** MOVE2 <ea>,b:c

MOVE2 b:c,<ea>

**Short:** Move a pair.

**Description:** Moves from or to memory two registers. Size byte, word or long.

**Condition Codes:** are taken from b, the first.

X	N	Z	V	C
-	*	*	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	1	1	1	0	Size			Mode			Register		
B				d	0	C			0	1	0	0	0	1		

S=Size 0=byte, 1=word, 2= long.

B & C: data or address-register.

direction d=0: <ea>,b else b,<ea>

Example:

move2.w (a0),d2:d3

ea (a0) memory content where a0 points to

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

b d2 & d3

										0	0	0	0	1	2	3	4
										0	0	0	0	4	5	6	7

Note:

move2 extends unsigned, movem extends signed.

<ea> must refer to memory, else unexpected result.

# MOVE2

# MOVE2

## old Move two

**Operation:** Source pair → destination pair

**Syntax:** MOVE2 <ea>,b:b+1

MOVE2 b:b+1,<ea>

**Short:** Move a pair.

**Description:** Moves a source pair to destination pair. A destination register is extended unsigned to a long. Conditions are taken from the first. Size can be byte, word or long.

**Condition Codes:** are taken from first, not the second.

X	N	Z	V	C
-	*	*	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	Size		Mode			Register		
b				0	d	0	0	0	0	0	1	0	0	0	1

S=Size 0=byte, 1=word, 2= long

b: 0..6=data register 8..14=address register 0..6 , must be even.

direction d=0: <ea>,b else b,<ea>

Example:

move2.w (a0),d2:d3

ea (a0) memory content where a0 points to

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

b d2 & d3

								0	0	0	0	1	2	3	4
								0	0	0	0	4	5	6	7

Note:

move2 extends unsigned, movem extends signed.

<ea> must refer to memory, else unexpected result.

# MOVEA

# MOVEA

## Move Address

**Operation:** <ea> → Bn

**Syntax:** MOVEA <ea>,Bn

**Description:** Move the source into a B-address register.

**Condition Codes:** not affected

**Constraints:** No LEA (Bn),Bm or MOVEA.L Bn,Bm

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

Size is long. Can be banked so <ea> includes En.

# MOVEC

# MOVEC

## Move Control register

**Operation:** Control → d  
**Syntax:** MOVEC Rc,Rn  
 MOVEC Rn,Rc ! Super mode only  
**Short:** Control register request/set  
**Description:** Some usefull event counters to look at.  
**Condition Codes:** not affected

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	1	0	1	d
a	Register			Control Register											

a=1: address register else data register  
 direction d=0: Rc to Rn else Rn to Rc

- \* \$808 PCR Procesor Configuration Register
- \* \$809 CCC Clock Cycle Counter
- \* \$80A IEP1 Instructions Executed Pipe 1
- \* \$80B IEP2 Instructions Executed Pipe 2
- \* \$80C BPC Branches Predicted Correct
- \* \$80D BPW Branches Predicted Wrong
- \* \$80E DCH Data Cache Hits
- \* \$80F DCM Data Cache Miss
  
- \* \$00A STR STalls Register
- \* \$00B STC STalls Cache
- \* \$00C STH STalls Hazard
- \* \$00D STB STalls Buffer
- \* \$00E MWR Memory Writes

Event counters increase by one when that event happens.

Note:  
 PRIVILEGED INSTRUCTION.  
 On 68080 reading a control register may be done in user mode.

# MOVEC

# MOVEC

## Move Control register

808 PCR Processor Configuration Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	1	ede	a	0	0	0	0	dfp	ess

id & revision are read only, writing has no effect.

bit(s)		Default	Comment
31-16	id	\$0440 = 080	\$0430 = 060
15-8	revision	\$01	(\$01 on 060)
7	edebg	0	1=slow mode, bit 6 selects what.
6	amiga	0	1=A1200, 0=A500
1	dfp	0	1=disable float point unit
0	ess	1	1=enable super scalar (second pipe)

Note:

PRIVILEGED INSTRUCTION.

On 68080 reading a control register may be done in user mode.







# MOVS

# MOVS

## Move with Sign extend

**Operation:** <ea> → Dn

**Syntax:** MOVS.B <ea>,Dn

MOVS.W <ea>,Dn

**Description:** Move the source operand to data register and sign extend to long.

### Condition Codes:

X	N	Z	V	C
-	*	*	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	Dn			1	0	S	Mode			Register		

S=Size 0=byte else word.

Example:

movs.b d0,d1

ea d0

																				C	4
--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	---	---

Result:

dn d1

																					F	F	F	F	F	F	C	4
--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	---	---	---	---	---	---	---	---



# PERM

# PERM

## Permute

**Operation:** Pick bytes from a → d

**Syntax:** PERM #n,Ra,Rb

**Short:** Change order and place in destination.

Where #n contains the picking order from a.

**Condition Codes:** not affected

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	1	1	0	0	a			
b				pos0				pos1			pos2		pos3		

a & b=0..7 data register 8..15 address register

Example:

perm #@3320,d0,d1

a d0

								0	0	1	1	2	2	3	3
--	--	--	--	--	--	--	--	---	---	---	---	---	---	---	---

Result:

b d1

								3	3	3	3	2	2	0	0
--	--	--	--	--	--	--	--	---	---	---	---	---	---	---	---

# SUBQ

# SUBQ

## Sub Quick

**Operation:**  $B_n - \text{data} \rightarrow B_n$

**Syntax:** SUBQ #<data>,B<sub>n</sub>

**Short:** Subtracts data from destination.

**Description:** Subtracts an immediate value of one to eight from the destination. Destination is a B-addr register. Size is long.

### Condition Codes:

X	N	Z	V	C
*	*	*	*	*

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

# TOUCH

# TOUCH

## Touch data

**Operation:** <ea> → void

**Syntax:** TOUCH <ea>

**Condition Codes:** not affected

**Short:** Preload data cache.

**Description:** Preload the data cache. Use it 12 to 15 cycles before needed. For the occasional speedy need for data that is not detectable as a sequential flow.

**Constraints:** Only two <ea> mode supported: address index & indirect. This includes the full extension format.

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

Mode = 2 & 6

supported <ea> modes examples:

touch (A1) ; indirect  
touch 6000(A1,D1\*4) ; with index  
touch (6000,A1,D1\*4) ; (same)  
touch ([6000,A1],D1\*4,7000) ; post indexed  
touch ([6000,A1,D1\*4],7000) ; pre indexed

Note:

Touch is seen as line-F coprocessor with id=3, like move16.

The second pipe accepts touch.

# AMMX INSTRUCTIONS

bfly	pavg	store
bsel	pcmpccb	storec
c2p	pcmpccw	storei
load	peor	storeilm
loadi	pmaxb	storem
lsdq	pmaxw	storem3
minterm	pminb	tex
pack3216	pminw	trans
packusbw	pmul	unpack1632
padd	pmula	vperm
pand	por	
pandn	psub	

# AMMX

# AMMX

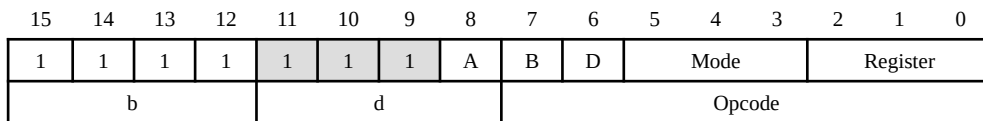
## Apollo Multi Media eXtension

**Operation:** handles 64 bit instructions

**Syntax:** instruction <vea>,b,d  
where d is the destination

**Condition Codes:** not affected

**Description:** AMMX is a line-F coprocessor with id=7. It handles 64 bit. So some say **Apply More Magic eXtension**.



B is msb of b. D is msb of d.

b & d are data-registers. d0..7 / e0..23

vector effective address or <vea> = A, Mode, Register

A=0	Mode			Register			A=1
Data register Dn	0	0	0	Register			Data register E8..15
Data register E0..7	0	0	1	Register			Data register E16..23
Address indirect (An)	0	1	0	Register			(Bn)
Address post inc (An)+	0	1	1	Register			(Bn)+
Address pre decr -(An)	1	0	0	Register			-(Bn)
d16(An)	1	0	1	Register			d16(Bn)
d8(An,Xn.w x Size)	1	1	0	Register			d8(Bn,Xn.w x Size)
d16(pc)	1	1	1	0	1	0	
d8(pc,Xn.w x Size)	1	1	1	0	1	1	
Abs.w	1	1	1	0	0	0	
Abs.l	1	1	1	0	0	1	
#imm.Q	1	1	1	1	0	0	#imm.W
- vperm -	1	1	1	1	1	1	

**Note:**

#imm.W is repeated. \$1234.w expands to Quad \$1234123412341234.

# BFLY

# BFLY

## Butterfly

**Operation:**  $b + a \rightarrow d$  ,  $b - a \rightarrow d2$

**Syntax:** BFLYB <vea>,b,d:d2

BFLYW <vea>,b,d:d2

**Condition Codes:** not affected

**Short:** Butterfly operation, vector short addition and subtraction.

**Description:** Bflyb is 8byte vector, It calculates 8additions and 8 substractions.

Bflyw is 4 word vector. This is for 4additions and 4substractions.

**Constraints:** The destination register pair needs to be consecutive, starting with an even register (e.g. bflyw (a0),E8,E0:E1 ).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	A	B	D	Mode			Register			
b				d				0	0	0	0	1	1	1	0	S

S=Size 0=byte, 1=word.

Example:

bflyb (a0),e1,e6:e7

vea (a0) memory content where a0 points to

0	4	0	4	0	4	0	3	1	4	0	4	0	5	8	8
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

b e1

0	0	F	F	7	F	3	3	7	4	5	5	6	6	7	7
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d e6 & e7

0	4	0	3	3	8	3	6	8	8	5	9	6	B	F	F
F	C	F	B	7	B	3	0	6	0	5	1	6	1	E	F

Note:

There is no saturation.(limiting)



# BSEL

# BSEL

## Bit Select

**Operation:** 64x b=1 ? then a → d

**Syntax:** BSEL <vea>,mask,d

**Condition Codes:** not affected

**Short:** Bitwise selection from <vea> , taken if b=1

**Description:** Masked bits are taken from <vea> , unmasked stays d.

This instruction allows a bit-by-bit selection of data from two sources into the destination. Typically, this is applied in conjunction with a prior pcmp instruction.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b(mask)				d				0	0	1	0	1	0	0	1

Example:

bsel d0,d1,d2

vea d0

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

b d1

0	0	0	F	F	F	C	0	0	0	C	F	F	F	F	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

d d2

5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d d2

5	5	5	3	4	5	5	5	5	5	9	B	C	D	E	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# C2P

# C2P

## Chunky to Planair

**Operation:** bit re-order source → d

**Syntax:** C2P <vea>,d

**Condition Codes:** not affected

**Short:** Chunky to planar conversion.

**Description:** Chunky-to-Planar conversion, bit-wise transpose.

Planar-to-Chunky conversion is the same as Chunky-to-Planar.

From a 8 byte source all bits from place n are put in destination byte n, in the order of source. So all msb are placed in the top byte of the destination and the lsb are all placed in the lowest byte.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	0	D	Mode			Register		
0	0	0	0	d				0	0	1	0	1	0	0	0

Example:

c2p d0,d1

vea d0

F	E	0	0	0	0	0	0	0	0	0	0	0	0	0	7
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d d1

8	0	8	0	8	0	8	0	8	0	8	1	8	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(\$FE=%1111 1110 , \$07=%0000 0111)

# LOAD

# LOAD

## Load source into register

**Operation:** <vea> → d

**Syntax:** LOAD <vea>,d

**Condition Codes:** not affected

**Short:** Load 64 bit into destination register.

**Description:** Load is the AMMX equivalent of move <ea>,dn

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	A	0	D	Mode			Register			
0	0	0	0	d				0	0	0	0	0	0	0	0	1

Load is always quad word.

Immediate data can be word size, this will expand repeated to quad .

load.w #1234,d1

Result:

d d1

1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# LOADI

# LOADI

**Load Indirect source into register**

**Operation:** <vea> → (d)

**Syntax:** LOADI <vea>,d

**Condition Codes:** not affected

**Short:** Load 64 bit indirect into destination register

**Description:** Loadi is the indexed variant of load. For many cases, the normal store instruction is more appropriate and convenient. While this indexed variant requires to preload the index register, it helps for example at places where the source register is to be changed conditionally. Also, you may think of storing a list of AMMX registers in a loop instead of in a row to keep code size small (where appropriate).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	A	0	D	Mode			Register			
0	0	0	1	d				0	0	0	0	0	0	0	0	1

(d) value → register

00 - 07 = D0 - D7

08 - 15 = A0 - A7

16 - 23 = B0 - B7

40 - 47 = E0 - E7

48 - 55 = E8 - E15

56 - 63 = E16 - E23

Example:

if d1=47 then

loadi (a0),d1

would do the same as

load (a0),e7

# LSdQ

# LSdQ

## Logical Shift Quad

**Operation:**  $b \ll a \rightarrow d$   
 $b \gg a \rightarrow d$

**Syntax:** LSLQ <vea>,b,d  
 LSRQ <vea>,b,d  
 where <vea> modulo 64 = shift count

**Condition Codes:** not affected

**Short:** 64 Bit shift left or right.

**Description:** LSdQ is a 64 Bit shift operation. The shift is modulo 64, the same as the 32bit variant. Zeroes are shifted into the LSB/MSB.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	1	1	1	0	0	dir

Direction: 0=left 1=right

Example:

lslq d0,d1,d2

vea d0

															0	C
--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	---	---

(= decimal 12)

b d1

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d d2

3	4	5	6	7	8	9	A	B	C	D	E	F	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# MINTERM

# MINTERM

## Min term

**Operation:**  $a ? b ? c \rightarrow d$

**Syntax:** MINTERM a0-a3,d

**Condition Codes:** not affected

**Short:** Diverse bitwise logical operation on 3 operands.

**Description:** Acts similar to blitter.

**Constraints:** This instruction does not support memory operands. The four inputs must be consecutive registers. The first source a is constrained to a multiple of 4 (i.e. D0-D3,D4-D7,E0-E3,...,E20-E23).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	0	D	0	0	a	a	0	0
0	0	0	0	d				0	0	1	0	1	0	1	0

Example:

minterm d0-d3,d6

a d0-d3

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	a
0	0	0	F	F	F	C	0	0	0	C	F	F	F	F	0	b
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	c
														E	2	Minterm logical operation

Result:

d d6

5	5	5	3	4	5	5	5	5	5	9	B	C	D	E	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Minterm bitlookup table: upper=1 lower=0 (A=1 a=0)

\* = \$E2

- 0 000 abc
- 1 001 abC \*
- 2 010 aBc
- 3 011 aBC
- 4 100 Abc
- 5 101 AbC \*
- 6 110 ABc \*
- 7 111 ABC \*

# PACK3216

# PACK3216

## Pack 32 bit color to 16 bit color

**Operation:** b & d convert → <vea>

**Syntax:** PACK3216 b,d,<vea>

**Condition Codes:** not affected

**Short:** Pack 32 Bit ARGB data into 16 Bit RGB565

**Description:** Convert gfx mode. Compress 2 x 2 32 bit color into 4 16 bit color.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	0	0	0	1	1	1

Not allowed <vea> : #imm

Example of red, green, purple & blue:

pack3216 d0,d1,e2

b d0

		F	F	0	0	0	0			0	0	F	F	0	0
--	--	---	---	---	---	---	---	--	--	---	---	---	---	---	---

d d1

		F	F	0	0	F	F			0	0	0	0	F	F
--	--	---	---	---	---	---	---	--	--	---	---	---	---	---	---

Result:

vea e2

F	8	0	0	0	7	E	0	F	8	1	F	0	0	1	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# PACKUSWB

# PACKUSWB

Pack Unsigned Saturated signed Word to Byte

**Operation:** b & d convert → <vea>

**Syntax:** PACKUSWB b,d,<vea>

**Condition Codes:** not affected

**Short:** Pack 2x4 signed words into 8 unsigned byte, saturate to 0..255

**Description:** Convert signed words to unsigned bytes. Result is saturated/limited when over the limit.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	0	0	0	1	1	0

Not allowed <vea> : #imm

Example:

packuswb d0,d1,(a2)

b d0

F	8	0	0	0	7	E	0	0	0	F	E	0	0	1	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

d d1

0	0	0	1	0	0	0	2	0	0	0	3	4	5	6	7
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

vea (a2) memory content where a2 points to

0	0	F	F	F	E	1	2	0	1	0	2	0	3	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



# PADD

# PADD

## Vector add

**Operation:**  $a + b \rightarrow d$   
**Syntax:** PADDDB <vea>,b,d  
 PADDW <vea>,b,d  
 PADDUSB <vea>,b,d  
 PADDUSW <vea>,b,d

**Condition Codes:** not affected

**Short:** Vector add.

**Description:** Paddb is 8byte vector, It calculates 8 additions. Paddw is 4 word vector. This is for 4 additions. Unsigned Saturated has a lower limit and an upper limit. Result above maximum are clipped to maximum.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	0	1	0	U	1	S

S=Size 0=byte, 1=word.

U=1 Unsigned Saturated else signed & no limiting.

Example:

paddb d0,d1,d2

vea d0

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

b d1

F	C	1	2	F	F	0	2	F	F	0	5	0	0	1	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d d2

F	D	3	5	4	4	6	9	8	8	B	0	C	D	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result paddusb d0,d1,d2

d d2

F	D	3	5	F	F	6	9	F	F	B	0	C	D	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result paddusw d0,d1,d2

d d2

F	D	3	5	F	F	F	F	F	F	F	F	C	E	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# PAND

# PAND

## Vector and

**Operation:** 64x a & b → d  
**Syntax:** PAND <vea>,b,d  
**Condition Codes:** not affected

**Short:** Bitwise logical operation.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	0	0	1	0	0	0

Example:  
pand d0,d1,d2

vea d0

1	2	F	F	1	2	F	F	0	0	F	F	0	0	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

b d1

1	2	1	2	F	F	F	F	0	0	0	0	F	F	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d d2

1	2	1	2	1	2	F	F	0	0	0	0	0	0	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# PANDN

# PANDN

## Vector and not

**Operation:** 64x (not a) & b → d

**Syntax:** PANDN <vea>,b,d

**Condition Codes:** not affected

**Short:** Bitwise logical operation, vea bits get flipped before logical “and” operation. Result is stored in d.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	0	0	1	0	1	1

Example:

pandn d0,d1,d2

vea d0

1	2	F	F	1	2	F	F	0	0	F	F	0	0	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

b d1

1	2	1	2	F	F	F	F	0	0	0	0	F	F	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d d2

0	0	0	0	E	D	0	0	0	0	0	0	F	F	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# PAVGB

# PAVGB

## Vector average

**Operation:**  $8 \times (a + b + 1) \gg 1 \rightarrow d$

**Syntax:** PAVGB <vea>,b,d

**Condition Codes:** not affected

**Short:** Average 8 unsigned bytes with 8 unsigned bytes.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	0	0	1	1	0	0

Example:

pavgb d0,d1,d2

vea d0

0	1	2	3	4	5	6	7	4	0	5	0	6	0	7	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

b d1

0	0	5	3	6	5	E	8	4	1	6	2	8	2	A	3
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d d2

0	1	3	B	5	5	A	8	4	1	5	9	7	1	8	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# PCMPccB

# PCMPccB

## Vector compare

**Operation:** 8 x b - a → condition → d

**Syntax:** PCMPEQB <vea>,b,d

PCMPHIB <vea>,b,d

PCMPGEB <vea>,b,d

PCMPGTB <vea>,b,d

**Condition Codes:** not affected

**Short:** Byte-by-byte vector compare.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	1	0	CC		0	

CC:

eq = 000 (ne)

hi = 001 (ls) unsigned

ge = 110 (lt) signed

gt = 111 (le) signed

hs (lo) unsigned is not implemented , see next page.

Example:

pcmpgtb d0,d1,d2

vea d0

0	1	0	5	0	3	0	4	F	F	0	0	7	0	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

b d1

0	5	0	1	0	3	F	F	0	4	7	0	8	0	0	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d d2

F	F	0	0	0	0	0	0	F	F	F	F	0	0	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result pcmphib d0,d1,d2

d d2

F	F	0	0	0	0	F	F	0	0	0	0	F	F	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# PCMPccW

# PCMPccW

## Vector compare

**Operation:**  $4 \times b - a \rightarrow \text{condition} \rightarrow d$

**Syntax:** PCMPEQW <vea>,b,d  
PCMPHIW <vea>,b,d  
PCMPGEW <vea>,b,d  
PCMPGTW <vea>,b,d

**Condition Codes:** not affected

**Short:** Word-by-word vector compare.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	1	0	CC		1	

CC:

eq = 000 (ne)

hi = 001 (ls) unsigned

ge = 110 (lt) signed

gt = 111 (le) signed

hs (lo) unsigned is not implemented

pcmphsw e0,e1,e2 calculation:

pcmpeqw e0,e1,e3 ;  $e1 == e0 ? \rightarrow e3$

pcmphiw e0,e1,e2 ;  $e1 > e0 ? \rightarrow e2$  (unsigned)

por e3,e2,e2 ;  $(e1 == e0) \text{ or } (e1 > e0) \rightarrow e1 \geq e0$

# PEOR

# PEOR

## Vector eor

**Operation:** 64x a eor b → d

**Syntax:** POR <vea>,b,d

**Condition Codes:** not affected

**Short:** Bitwise logical operation.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	0	0	1	0	1	0

Example:

peor d0,d1,d2

vea d0

1	2	F	F	1	2	F	F	0	0	F	F	0	0	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

b d1

1	2	1	2	F	F	F	F	0	0	0	0	F	F	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d d2

0	0	E	D	E	D	0	0	0	0	F	F	F	F	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# PMAXxB

# PMAXxB

## Vector maximum

**Operation:**  $8 \times \max(a, b) \rightarrow d$

**Syntax:** PMAXSb <vea>,b,d

PMAxUb <vea>,b,d

**Condition Codes:** not affected

**Short:** Byte-by-byte vector compare and obtain biggest.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	1	1	0	1	U	S

S=Size 0=byte, 1=word.

U 1=unsigned, 0=signed

Example:

pmaxub d0,d1,d2

vea d0

0	1	2	3	4	5	6	7	4	0	5	0	6	0	7	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

b d1

0	0	5	3	6	5	E	8	4	1	6	2	8	2	A	3
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d d2

0	1	5	3	6	5	E	8	4	1	6	2	8	2	A	3
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result: pmaxsb d0,d1,d2

d d2

0	1	5	3	6	5	6	7	4	1	6	2	6	0	7	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



# PMAXxW

# PMAXxW

## Vector maximum

**Operation:**  $4 \times \max(a, b) \rightarrow d$

**Syntax:** PMAWSW <vea>,b,d

PMAUW <vea>,b,d

**Condition Codes:** not affected

**Short:** Word-by-word vector compare and obtain biggest.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	1	1	0	1	U	S

S=Size 0=byte, 1=word.

U 1=unsigned , 0=signed

# PMINxB

# PMINxB

## Vector minimum

**Operation:**  $8 \times \min(a, b) \rightarrow d$

**Syntax:** PMINSB <vea>,b,d

PMINUB <vea>,b,d

**Condition Codes:** not affected

**Short:** Byte-by-byte vector compare and obtain smaller.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	1	1	0	0	U	S

S=Size 0=byte, 1=word.

U 1=unsigned, 0=signed

Example:

pminub d0,d1,d2

vea d0

0	1	2	3	4	5	6	7	4	0	5	0	6	0	7	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

b d1

0	0	5	3	6	5	E	8	4	1	6	2	8	2	A	3
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d d2

0	0	2	3	4	5	6	7	4	0	5	0	6	0	7	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result: pminsb d0,d1,d2

d d2

0	0	2	3	4	5	E	8	4	0	5	0	8	2	A	3
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# PMIN<sub>x</sub>W

# PMIN<sub>x</sub>W

## Vector minimum

**Operation:**  $4 \times \min(a, b) \rightarrow d$

**Syntax:** PMINSW <vea>,b,d

PMINUW <vea>,b,d

**Condition Codes:** not affected

**Short:** Word-by-word vector compare and obtain smaller.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	1	1	0	0	U	S

S=Size 0=byte, 1=word.

U 1=unsigned , 0=signed

# PMUL

# PMUL

## Vector multiply

**Operation:**  $a \times b \rightarrow d$   
**Syntax:** PMULH <vea>,b,d  
 PMULL <vea>,b,d  
 PMUL88 <vea>,b,d

**Condition Codes:** not affected

**Short:** Vector multiply short

**Description:** Pmul is 4 word vector signed multiply. Pmulh keeps upper 16 bits (31..16). Pmull keeps lower 16 bits (15..0). Pmul88 keeps the middle part (23..8).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	A	B	D	Mode			Register			
b				d				0	0	0	1	1	0	T		

T type

0=pmul88

1=pmula (next page)

2=pmulh

3=pmull

Example:

pmulh d0,d1,d2

vea d0

0	0	0	2	0	0	2	0	0	2	0	0	F	F	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

b d1

1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d d2

0	0	0	0	0	0	0	2	0	0	2	4	F	F	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result pmull d0,d1,d2

2	4	6	8	4	6	8	0	6	8	0	0	E	D	C	C
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result pmul88 d0,d1,d2

0	0	2	4	0	2	4	6	2	4	6	8	F	F	E	D
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# PMULA

# PMULA

## Vector multiply

**Operation:**  $\text{alfa} < 100\% ? a + \text{alfa} \times b \rightarrow d$   
 $\text{alfa} = 100\% ? b \rightarrow d$

**Syntax:** PMULA <vea>,b,d

**Condition Codes:** not affected

**Short:** 32bit color vector multiply and add.

**Description:** Fade b-colors by alfa then add a-colors to it. Resulting colors are unsigned saturated bytes.

0%=<alfa<100%      ( alfa x b ) + a → d  
 alfa=100%=255      100% b → d (When alfa is 100% (255) there is no addition done.)

1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	0	1	1	0	0	1

Long: 8bit alpha = 0..100%, 8bit red 0..255, 8bit green 0..255, 8bit blue 0..255

vea

Alfa 8bit	Red 8bit	Green 8bit	Blue 8 bit	Src a
-----------	----------	------------	------------	-------

b

	Red 8bit	Green 8bit	Blue 8 bit	Src b
--	----------	------------	------------	-------

Result:

d

0	Red 8bit	Green 8bit	Blue 8 bit	Dest d
---	----------	------------	------------	--------

Example:

vea

\$40	\$10	\$62	\$dc	Sprite
------	------	------	------	--------

b

	\$ff	\$80	\$b0	Background
--	------	------	------	------------

Result:

d

0	\$3f+\$10=\$4f	\$20+\$62=\$82	\$2c+\$dc=\$ff	Faded background + sprite
---	----------------	----------------	----------------	---------------------------

# POR

# POR

## Vector or

**Operation:** 64x a or b → d

**Syntax:** POR <vea>,b,d

**Condition Codes:** not affected

**Short:** Bitwise logical operation.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	0	0	1	0	0	1

Example:

por d0,d1,d2

vea d0

1	2	F	F	1	2	F	F	0	0	F	F	0	0	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

b d1

1	2	1	2	F	F	F	F	0	0	0	0	F	F	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d d2

1	2	F	F	F	F	F	F	0	0	F	F	F	F	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# PSUB

# PSUB

## Vector subtract

**Operation:**  $b - a \rightarrow d$   
**Syntax:** PSUBB <vea>,b,d  
 PSUBW <vea>,b,d  
 PSUBUSB <vea>,b,d  
 PSUBUSW <vea>,b,d

**Condition Codes:** not affected

**Short:** Vector subtract.

**Description:** Psubb is 8byte vector, It calculates 8 substractions. Psubw is 4 word vector. This is for 4 substractions. Unsigned Saturated has a lower limit and an upper limit. When the result is below zero it is clipped to be zero.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d				0	0	0	1	0	U	1	S

S=Size 0=byte, 1=word.

U=1 Unsigned Saturated else signed & no limiting.

Example:

psubb d0,d1,d2

vea d0

0	1	2	3	4	5	6	7	8	9	A	B	0	4	1	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

b d1

0	4	1	2	0	1	0	2	F	F	0	5	0	1	2	3
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d d2

0	3	E	F	B	C	9	B	7	6	5	A	F	D	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result psubusb d0,d1,d2

d d2

0	3	0	0	0	0	0	0	7	6	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result psubusw d0,d1,d2

d d2

0	3	E	F	0	0	0	0	7	6	5	A	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# STORE

# STORE

## Store register into memory

**Operation:** b → <vea>

**Syntax:** STORE b,<vea>

**Condition Codes:** not affected

**Short:** Store 64 bit from source register in memory

**Description:** Store is the AMMX equivalent of move dn,<ea>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	0	Mode			Register		
b				0	0	0	0	0	0	0	0	0	1	0	0

Not allowed <vea> : #imm





# STOREI

# STOREI

## Store indirect register into memory

**Operation:** (d) → <vea>

**Syntax:** STOREI b,<vea>

**Condition Codes:** not affected

**Short:** Store 64 bit from indirect source register in memory

**Description:** Store is the AMMX equivalent of move dn,<ea>.

For many cases, the normal store instruction is more appropriate and convenient.

While this indexed variant requires to preload the index register, it helps for example at places where the source register is to be changed conditionally. Also, you may think of storing a list of AMMX registers in a loop instead of in a row to keep code size small (where appropriate). See also loadi.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	0	Mode			Register		
b				0	0	0	1	0	0	0	0	0	1	0	0

Not allowed <vea> : #imm

d: value => register

00 - 07 = D0 - D7

08 - 15 = A0 - A7

16 - 23 = B0 - B7

40 - 47 = E0 - E7

48 - 55 = E8 - E15

56 - 63 = E16 - E23

Example:

if d0=47 then

storei d0,(a1)

would do the same as

store e7,(a1)

# STOREILM

# STOREILM

Store inverted long masked register into memory

**Operation:**  $b \rightarrow \langle vea \rangle$  depending on mask.

**Syntax:** STOREILM b,mask,<vea>

Where 8 lsb bits are used as mask to write (0) or not (1)

**Condition Codes:** not affected

**Short:** Store 8 byte from register-b in memory when its mask=0

**Description:** Store is a conditional write of 8 bytes. The selection is made by the lsb of the 8 bytes from register d.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d (mask)				0	0	1	0	0	1	0	1

Not allowed <vea> : #imm

No masking is done when destination is a register.

Example:

storeilm e10,e11,(a2)

b e10

1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

d e11

														7	C
--	--	--	--	--	--	--	--	--	--	--	--	--	--	---	---

(\$7c = binair %0111 1100)

Result:

vea (a2) memory content where a0 points to

1	1													7	7	8	8
---	---	--	--	--	--	--	--	--	--	--	--	--	--	---	---	---	---

Note:

Also called storem2

Storem is similar

# STOREM

# STOREM

## Store masked register into memory

**Operation:**  $b \rightarrow \langle \text{vea} \rangle$  depending on mask.

**Syntax:** STOREM b,mask,<vea>

Where the lower 8 bits of d is used as mask to write (1) or not (0)

**Condition Codes:** not affected

**Short:** Store 8 byte from register-b in memory when its mask=1

**Description:** Store is a conditional write of 8 bytes. The selection is made by the last 8 bit of register d.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	Mode			Register		
b				d (mask)				0	0	0	0	0	1	0	1

Not allowed <vea> : #imm

No masking is done when destination is a register.

Storeilm is similar

Example:

storem d0,d1,(a2)

b d0

1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

d d1

0	1	0	1	0	0	0	1	0	1	0	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

vea (a2) memory content where a0 points to

1	1	2	2			4	4	5	5					8	8
---	---	---	---	--	--	---	---	---	---	--	--	--	--	---	---

# STOREM3

# STOREM3

## Store gfx-masked register into memory

**Operation:**  $b \rightarrow \langle \text{vea} \rangle$  depending on mode.

**Syntax:** STOREM3 b,#mode,<vea>

Where the <vea> content is used as mask to write or not.

**Condition Codes:** not affected

**Short:** Store bytes from b into destination, depending on mode

**Description:** Specialised memory write to preform fast grafical cookie cut.

Storem3 is a conditional write of 8 bytes. The selection depends on the source and mask\_mode.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	0	Mode			Register		
b				d (mask_mode)				0	0	0	0	0	1	0	1

mask\_mode: writing is done when:

0 - Long: 2x 32bit color when msb=1

1 - Byte: 8x 8bit color-index when <>0

2 - word: 4x Sixteen bit color when color<>\$f81f (= max red & blue = purple)

3 - Word: 4x 15bit color when msb=0

Not allowed <vea> : #imm

No masking is done when destination is a register.

bit 11 & 10 second operand are ignored.

Example:

b d0

F	8	1	F	0	0	3	4	1	2	0	0	8	7	6	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

storem3 d0,#0,(a0) L result:

storem3 d0,#1,(a0) B result:

storem3 d0,#2,(a0) S result:

storem3 d0,#3,(a0) W result:

F	8	1	F	0	0	3	4								
F	8	1	F			3	4	1	2			8	7	6	5
				0	0	3	4	1	2	0	0	8	7	6	5
				0	0	3	4	1	2	0	0				

Note:

**Vasm** syntax:

“storem3 d0,#3,(a0)” must be written as “storem3 d0,d3,(a0)”

Debugging with monam shows what it does “storem3 d0,w,(a0)” so the function.

# TEX

# TEX

## Texture

**Operation:**  $(A_n, (A_u, A_v)) \rightarrow d$   
**Syntax:** TEX8.512  $(A_n, (A_u, A_v)), D_n$   
 TEX16.256  $(A_n, (A_u, A_v)), D_n$   
 TEX24.64  $(A_n, (A_u, A_v)) * D_0, D_n$

**Condition Codes:** not affected

**Short:** Gets from picture array  $A_n$  ( position  $A_u, A_v$  ) a byte, word or 3byte.

**Description:** Gets a color from a texture position  $u, v$ . shifts the destination up and inserts the color there.  $A_n$  points to the texture.  $A_u$  &  $A_v$  are 16bit integer 16 bit fraction longs.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	0	D	1	1	0	A <sub>n</sub>		
0	A <sub>u</sub>			d				0	0	1	1	1	1	1	0
1	A <sub>v</sub>			1	S	S	0	S	T			0	0	S	S

S size destination:

8 byte            00 0 00  
 16 word          01 0 01  
 24 24bit        11 1 10 \*

Texture size

64 x 64          000  
 128 x 128       011  
 256 x 256       101  
 512 x 512       110

\* The texture for 24bit must be nvidia dxt1 compressed.  
 This is also the case for Maggie rendering.

Note:

The third word seem to be a specialised brief extension word.  
 Tex is fully supported by sa core 7.4 but seem to be broken in current cores.

# TRANS

# TRANS

## Transpose

**Operation:** takes bytes from 4 sources and places them in 2 destinations.

**Syntax:** TRANSHI a0-a3,d:d2  
TRANSLO a0-a3,d:d2

**Condition Codes:** not affected

**Short:** Matrix word transpose.

**Description:** Transpose a 4x4 block with 16 bit per element from row to column order and vice versa.

**Constraints:** This instruction does not support memory operands. The four inputs and the destination must be consecutive registers. The first source a is constrained to a multiple of 4 (i.e. D0-D3,D4-D7,E0-E3,...,E20-E23). The destination register index pair (d:d2) are restricted to a multiple of two (i.e. D0:D1,D2:D3 etc.)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	0	D	0	0	a	a	0	0
0	0	0	0	d			0	0	0	0	0	0	0	1	L

L=1 translo else transhi

Example:

translo d0-d3,d6:d7

a d0-d3

								0	0	1	1	2	2	3	3
								4	4	5	5	6	6	7	7
								8	8	9	9	A	A	B	B
								C	C	D	D	E	E	F	F

Result:

d d6:d7

0	0	1	1	4	4	5	5	8	8	9	9	C	C	D	D
2	2	3	3	6	6	7	7	A	A	B	B	E	E	F	F

# UNPACK1632

# UNPACK1632

## Unpack 16 bitcolor to 32 bitcolor

**Operation:** <vea> convert → d:d2

**Syntax:** UNPACK1632 <vea>,d:d2

**Condition Codes:** not affected

**Short:** Unpack 16 Bit RGB565 data into 32 Bit ARGB.

**Description:** Convert gfx mode. Expand 4 16 bit color into 2 x 2 32 bit color.

**Constraints:** The destination register index pair are restricted to a multiple of two (i.e. D0:D1,D2:D3 etc.)

1	1	1	1	1	1	1	A	0	D	Mode			Register		
0	0	0	0	d			0	0	0	0	1	1	1	1	0

Example of red, green, purple & blue:

unpack1632 d0,d2:d3

vea d0

F	8	0	0	0	7	E	0	F	8	1	F	0	0	1	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d d2:d3

0	0	F	F	0	0	0	0	0	0	0	0	F	F	0	0
0	0	F	F	0	0	F	F	0	0	0	0	0	0	F	F



# VPERM

# VPERM

## Vector Permute

**Operation:** Pick bytes from a & b → d

**Syntax:** VPERM #n,a,b,d

where #n contains the picking order from a & b

**Condition Codes:** not affected

**Short:** Permute the contents of two registers into destination register.

**Constraints:** The operands a, b & d must be data registers.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	A	B	D	1	1	1	1	1	1
b				d				0	0	0	0	a			
s0	pos0			s1	pos1			s2	pos2			s3	pos3		
s4	pos4			s5	pos5			s6	pos6			s7	pos7		

S=0 takes pos from a else from b.

Example:

vperm #3210AB78,d0,e1,e6

a d0

0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

b e1

8	8	9	9	A	A	B	B	C	C	D	D	E	E	F	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Result:

d e6

3	3	2	2	1	1	0	0	A	A	B	B	7	7	8	8
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# FLOATING POINT INSTRUCTIONS

fdbcc.l                      fmove(u)rz  
fmove fstorei floadi      fmovem

# FDBcc

# FDBcc

## Floating-point Test Decrement & Branch Conditional

**Operation:** If not cc then ( PC - 1 → Dn ; if Dn <> - 1 then PC + dn → PC )

**Syntax:** FDBcc.L Dn,<label>

**Short:** Decrement Dn & conditional jump to label.

**Description:** Controls a loop of instructions. If condition is true the loop ends and the program continues with the next instruction.

Else count register Dn is decremented by 1.

If Dn = - 1 the loop also ends and the program continues with the next instruction.

If Dn <> - 1 the loop continues and the program execution continues at location (PC) + displacement.

The displacement is always even. When it appears as odd, then the counter is a long, not a word.

**FP Condition Codes:** not affected

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	1	0	0	1	Count Register		
0	0	0	0	0	0	0	0	0	0	Conditional predicate					
16 bit Displacement														1	

Note:

Variant on fdbcc. Here Dn is a long counter, not a word counter.

# FMOVE

# FMOVE

## Floating point convert and Move

**Operation:** FPn → Dn

**Syntax:** FMOVE.s Dn,FPn  
FMOVE.s FPn,Dn

**Description:** Move in Double format from/to data-register.  
Move in Single format was always possible, now double (& extended) too.

**FP Condition Codes:**

N	Z	I	nan
*	*	*	*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	1	D	Src			fpn			Opmode						

Mode=0=data register.

Source Specifier 110=b 100=w 000=l 001=s 101=d 010=x

Direction d=0: <ea>,fpn else fpn,<ea>

<ea>,fpn : opmode = R000P00 where RP = Rounding Precision.

00=default 10=single 11=double.

fpn,<ea> : opmode = zero (or k-factor for unsupported packed source-format)

## FSTOREI & FLOADI

e-registers can be both float & data register.

fmove.w e2,e3 can mean fp → d but also d → fp

Vasm uses fstorei for fp → d & floadi for d → fp in that case.

Up to vasm 1.9f fmove defaults to fp → d. Force d → fp direction by using fdmove.

Note:

Apollo\_eXtende\_format layout is on fmovem page.

Packed is not supported. (like 040 & 060)

# FMOVERZ

# FMOVEURZ

Floating point Round to Zero, convert & Move (as Unsigned)

**Operation:** FPn → Dn

**Syntax:** FMOVERZ.s Fpn,<ea>

FMOVEURZ.s FPn,<ea>

**Description:** Move to ea (un)signed byte, word or long.

**FP Condition Codes:**

N	Z	I	nan
*	*	*	*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode		Register			
0	1	D	Src			fpn			Opmode						

Source Specifier 110=b 100=w 000=l

Direction d=1: fpn,<ea>

opmode 1=Round Zero , 3=Unsigned Round Zero

# FMOVEM

# FMOVEM

## Move multiple float registers from/to memory

**Operation:** list → <ea>

**Syntax:** FMOVE <list>,<ea>

FMOVE <ea>,<list>

**Description:** Fmovem on the 68080 is the same as previous generations, except for the fact that the format in memory is a tiny bit different. For calculated results it makes no difference when a push and later a pull is done.

To get all the bits of the float you need a fmove.d fpn,<ea>

Programs that use the expected extended layout in memory may be affected.

**FP Condition Codes:** not affected

Motorola eXtende format:

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
S	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E																		
M	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31		
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	

Apollo eXtende format:

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
S	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	
M	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31		

Double format:

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	E	E	E	E	E	E	E	E	E	E	E	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52



# SECOND PIPE

## What does the second pipe smoke?

Fusing = yes (except quad move)

AMMX = no (except store)

FPU = no

No:

to from ccr/cr/usp  
subx addx negx  
mul div  
abcd nbcd sbcd  
xshift shift ,(ea)  
bitfield  
exg  
cmp2 cmpm chk(2) cas(2)  
moves movep movem movec  
(un)link un(pack)  
perm bank  
nop  
rte rts jsr bsr bcc

Yes:

bra bcc dbra jmp  
(e)or(i) and(i) clr neg not tst  
cmpi cmp(a)  
subi subq sub(a) addi addq add(a)  
btst/bchg/bclr/bset  
move(a) moveq  
lea pea swap ext  
shift ,Dn  
touch  
  
store

Restrictions:

1 READ from & 1 WRITE to Data Cache (memory) allowed for both pipes.

### Pipeline stages:

- 1 Icache Fetch
- 2 Decoding
- 3 Register fetch
- 4 EA calculation
- 5 Dcache Fetch
- 6 ALU calculation
- 7 Write back



