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

080 tools:

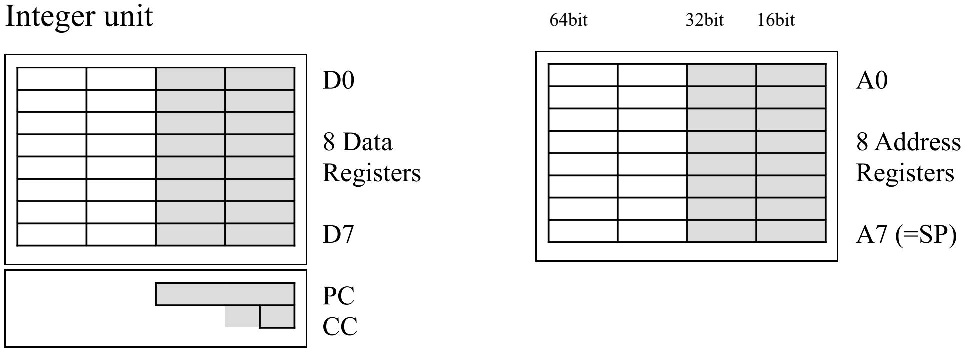
assembler	VASM , vasmm68k_mot_os3
debugger	Devpac's "Vamped" MonAm 3.09
docs	Developer Documentation

aug 2024

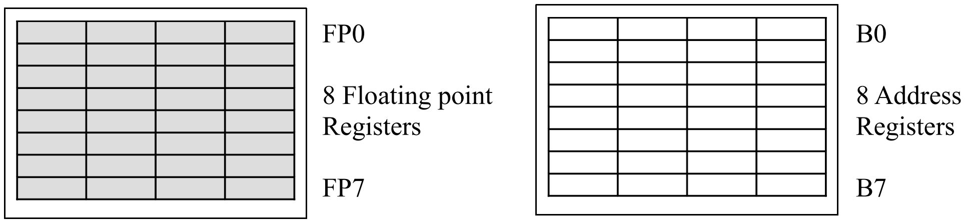
68080 user programming model.

Legacy model in gray.

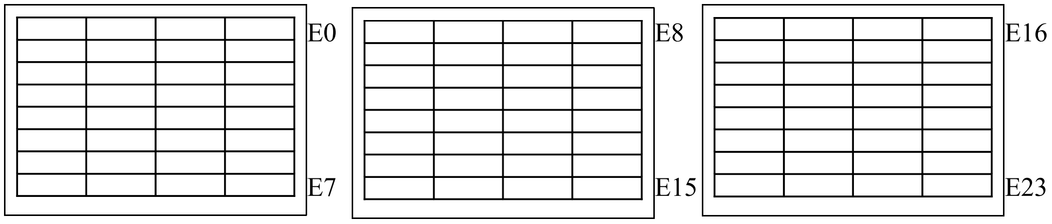
Integer unit



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 curently 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.
Except bitfield, that oddly extends to 64 bit for now.

Integer instructions that use e-registers:
move(q),addi, subi eor(i),ori, andi, not, neg(x), (b)tst, (b)clr, bset, swap, scc
bf(clr/set/tst/chg), ext(u), cmpi, as(r/l), ls(r/l), ro(x)(r/l), moviw.l, perm
handling of banking not correct yet:
pack, unpk, mul, div, abcd, sbcd, movex, add, sub,and, or, cmp, addx, subx
exg, bf(extu/exts/ffo/ins)

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 + 13 nibbles fraction. It uses the 64bit double-real format.

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. (mostly 6 ticks or more) The user does not have to keep track of anything.
(picture page 85)

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

<ea> can be Double format from/to data-register. Single was always possible.

Note:
The 080 has no implementation of packed bcd real size. (same on 040 & 060)
The extended format has a different layout, see page 81 fmovem.
B-address registers can not be used. (yet?)
Newer V4 cores uses all 64 bits with calculation & fmove.x
Older V4 cores use 11 nibbles fraction.
V2 core 2.17 use 8 nibbles fraction (more does not fit in the fpga)
So the accuraccy of the V2 is 1 / 4.000 million.

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.

This division of 64bits into lets say 4 x 32 or 8 x 8 parts is called vector.

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

Ammx does not change the condition-codes at all. Pcmp sets the condition in the destination register. Bsel can be used to make conditional changes with that.

Apollo bit.

Multitask & aditional line-a instructions.

ApolloOS always saves the new registers on task switching.
A program that sets SR bit 11 keeps e-registers intact when switching task on an rom patched OS like amiga or coffin. To make this easier the 080 allows it to be set even in user-mode !
Other bits will of course give a privilage exception as expected.

In the near future
LineA-instructions become valid when the apollo-bit in SR is set.
They do not work at all for now. (beta testers: sa_10082 does)
LineA $a000..$a00f will not be used to be atari-os compatible.

“LineA” instructions are: clr.q , mov3q , moviw.l , mov(s/z)

```
    Ori #$800,sr    sets the apollo-bit.
    Andi #$f7ff,sr clears the apollo-bit.
```

The apollo-bit affects only the program that sets it. Setting it and expecting another program to behave if the apollo-bit is set does not work.
So restoring the bit on exit is not needed.

B registers.

Registers B are handicaped versus its brother A
Bn is restricted to manipulate long data or ammx data

The integer unit supports:

```
movea.l <ea>,Bn
move.l Bn,<ea>
lea      <ea>,Bn
lea      (Bn),An          = move.l Bn,An

addq.l #n,Bn
subq.l #n,Bn
cmp.l  Bn,Dn
```

<ea> supports E but not B
B can not be moved to another B
B can not be exchanged.

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
clr.l (an)+	clr.l (an)+	quad clr ! in new cores
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 exept 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
fmove.x fpn,fpm	fmul.x fpx,fpm	fmul fpn,fpx,fpm
fmove.x fpn,fpm	fadd.x fpx,fpm	fadd fpn,fpx,fpm

Ammx vector handling by type, source or destination match:

Vector Bit (64x bit)

bsel	pand	peor
minterm	pandn	por

Vector Byte (8x byte)

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

Vector Word (4x word)

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

Vector Long (2x long)

pack3216	storem3	unpack1632
pmula		

Vector Quad (1 quad)

c2p	store	storem
load	storec	storem3
loadi	storei	vperm
lsdq	storeilm	

INTEGER INSTRUCTIONS

AMMX INSTRUCTIONS

FLOATING POINT INSTRUCTIONS

Note:
!la – LineA instructions become valid when the apollo-bit in SR is set. (bit 11)
They do not work at all for now (beta testers use: sa_10082), its a future upgrade.
!p – reading is not Privilaged on 080.

ADDIW ADDIW

Add Immediate Word extended to Long

Operation: data + <ea> → <ea>
Syntax: ADDIW.L #<data>,<ea>
Short: Add data to destination.
Description: Sign extend immediate word data to long and add to destination. 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	0	0	0	0	1	1	0	1	1	Mode			Register		
16-bit word data															

Example:
addiw.l #$8001,d0

ea d0

								0	0	1	2	3	4	5	6

Result:
ea d0

								0	0	1	1	B	4	5	7

Note:
Replaces CALLM
There is no subiw.l , just use a negative value with addiw.l

Add Quick

Operation: data + Bn → Bn
Syntax: ADDQ #<data>,Bn
Short: Add data to destination.
Description: Adds an immediate value of one to eight to 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			0	0	0	0	0	1	Register		

Example:

```
addq.l    #8,b1
```

ea b1

								0	0	1	2	3	4	5	6

Result:
ea b1

								0	0	1	2	3	4	5	E

BANK BANK

Bank

Operation: Inform the next instruction apollo-registers are used.
Syntax: - none -

Short: Prefix for legacy instructions.
Description: Bank gives older instuctions 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 1st bankable source operand from 3bit to 5bit. (From 8 to 32 options)
BB extends 1st bankable destination operand.
CCccc is xored to BBbbb to create a thirth operand.

```
If CCccc <> 0  then  BBbbb xor CCccc → DDddd
   instr a ? b → d
else
   instr a ? b → b
endif
```

AA & BB & DD:

Data & Float	ea mode (Address & Index) *
00 = original	00 = original
01 = E0 - E7	01 = B0-7
10 = E8 - E15	10 = Xn → E0-7 or B0-7
11 = E16- E23	11 = B0-7 & Xn → E0-7 or B0-7

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

Note:
* Addressed ea mode is not implemententent yet.
Size SS is not needed anymore. Will be used to expand instruction options.

Bcc Bcc

Branch Conditional

Operation: If cc then PC + dn → PC
Syntax: Bcc.S+ <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 “b2” “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 .. –132 & 128 .. 254

Note:
Variant on bcc.s
Offset -130 can not be short, $ff conflict with bcc.l

BRA BRA

Branch

Operation: PC + dn → PC
Syntax: BRA.S+ <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 “b2” “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 .. –132 & 128 .. 254

Note:
Variant on bra.s
Offset -130 can not be short, $ff conflict with bra.l

BSR BSR

Branch to SubRoutine

Operation: SP – 4 → SP ; PC → (SP) ; PC + dn → PC
Syntax: BSR.S+ <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 “b2” “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 .. –132 & 128 .. 254

Note:
Variant on bsr.s
Offset -130 can not be short, $ff conflict with bsr.l

CLR CLR

Clear

Operation: 0 → <ea> *LineA
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:
Clr.q is a LineA instruction, set apollo-bit in SR to become active. (ori #$800,sr)

CMP CMP

Compare

Operation: Dn – Bn → cc
Syntax: CMP Bn,Dn
Short: Substract & use only the condition.
Description: Substracts 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		

Compare Immediate Word extended Long

Operation: <ea> – data → cc
Syntax: CMPIW.L #<data>,<ea>
Short: Substract & use only the condition.
Description: Sign extend immediate word data to long, substract 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 & **c**onditiononal 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 ironicly about the only one used of the group.

EXTUB EXTUW

Extend Unsigned Byte/Word

Operation: Dn → Dn.L
Syntax: EXTUB.L Dn
EXTUW.L Dn
Short: Extend with zeros to long.
Description: Unsigned extend. Extend data register with zeros to long. Source size can be byte or word.

Condition Codes:

X	N	Z	V	C
–	0	*	0	0

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

Size 01=byte to long, 10=word to long

Example:
extub.l d0

reg d0

														E	3

Result:
reg d0

								0	0	0	0	0	0	E	3

LEA LEA

Load Effective Address

Operation: <ea> → An
Syntax: LEA <ea>,Bn
LEA (Bn),An
Description: Loads the effective address into an address register.
Condition Codes: not affected
Constraints: No LEA (Bn),Bm or MOVEA.L Bn,Bm

LEA <ea>,Bn

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

LEA (Bn),An

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

Example:

```
   lea    1(a0),b1
```

ea a0

								0	0	1	2	3	4	5	6

Result:
bn b1

								0	0	1	2	3	4	5	7

MOV3Q MOV3Q

Move 3-Bit Data Quick

Operation: 3-bit Immediate Data → <ea> *LineA
Syntax: MOV3Q #<data>,<ea>
Description: Move immediate -1..7 to destination. Size is long.

Condition Codes:

X	N	Z	V	C
–	*	0	0	0

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

Data=0 represents -1

Example:
```
   mov3q.l   #7,d0
```

Result:
d0

								0	0	0	0	0	0	0	7

Note:
This is a LineA instruction, set apollo-bit in SR to become active. (ori #$800,sr)

MOVE MOVE

Move

Operation: Bn → <ea>

Syntax: MOVE Bn,<ea>

MOVE <ea>,Bn (next page)

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

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.

Move from status register
(& Set/clear bit 11)

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

Also **apollo-bit 11** is the only one that can be set/cleared in user mode.
Ori #$800,sr & andi #$f7ff,sr will not cause a privilage exception on the 080.

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** have 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 is not correct on V2 (2.17).

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	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.
move2 is not correct on V2 (2.17).

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

Move Control register

808 PCR Procesor 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	1=v4 , 0=v2	(1 on 060)
7	edebug	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:
PRIVILEDGED INSTRUCTION.
On 68080 reading a control register may be done in user mode.

MOVEX MOVEX

Move convert source to destination

Operation: Re-ordered source → destination
Syntax: MOVEX <ea>,b
MOVEX b,<ea>
Short: Endian convert and move.
Description: Changes byte order from the source and places it in the destination.

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	0	1	1	1	0	Size		Mode			Register		
B				d	0	0	0	0	0	0	1	0	0	0	0

Size 1=word, 2= long
B=data or address-register.
direction d=0: <ea>,b else b,<ea>

Example:

```
   movex.l    a0,a1
```

a a0

								0	0	1	1	2	2	3	3

Result:
b a1

								3	3	2	2	1	1	0	0

MOVIW MOVIW

Move Immediate Word extended to Long

Operation: data → <ea> *LineA
Syntax: MOVIW.L #<data>,<ea>
Short: Move data to destination.
Description: Sign extend immediate word data to long and move to destination.

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	0	0	1	0	0	0	Mode			Register		
16-bit word data															

Example:
```
moviw.l    #$8123,d0
```

Result:
ea d0

								F	F	F	F	8	1	2	3

Note:
This is a LineA instruction, set apollo-bit in SR to become active. (ori #$800,sr)

Move with Sign extend

Operation: <ea> → Dn *LineA

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

Note:
This is a LineA instruction, set apollo-bit in SR to become active. (ori #$800,sr)

MOVZ MOVZ

Move with Zero fill

Operation: <ea> → Dn *LineA

Syntax: MOVZ.B <ea>,Dn

MOVZ.W <ea>,Dn

Description: Move the source operand to data register and zero fill to long.

Condition Codes:

X	N	Z	V	C
–	0	*	0	0

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

S=Size 0=byte else word.

Example:

```
movz.b    d0,d1
```

ea d0

														C	4

Result:

dn d1

								0	0	0	0	0	0	C	4

Note:

This is a LineA instruction, set apollo-bit in SR to become active. (ori #$800,sr)

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
data-registers can be banked so a & b includes En.

Example:
perm #@3320,a0,e1

a a0

								0	0	1	1	2	2	3	3

Result:
b e1

								3	3	3	3	2	2	0	0

Note:
Banked address registers are not supported, they show up as En when banked.

SUBQ SUBQ

Sub Quick

Operation: Bn – data → Bn
Syntax: SUBQ #<data>,Bn
Short: Substracts data from destination.
Description: Substracts 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

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 **A**pply **M**ore **M**agic e**X**tension.

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				M	Opcode						

B is msb of b. D is msb of d.
b & d are data-registers. d0..7 / e0..23
*M=1 write to memory (b,d,vea) else default mode 0= to register (vea,b,d)

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)+
Addresss 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.
* new

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

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	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	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

Logical Shift Quad

Operation: b << a → d
b >> a → 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 → d
Syntax: MINTERM a0-a3,d
Condition Codes: not affected

Short: Diverse bitwise logical operantion on 3 operands.
Description: Acts similair 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 *

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: PACKUSBW 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 → d
Syntax: PADDB <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 operantion.

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 operantion, 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 x (a + b +1) >>1 → 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

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

Vector compare

Operation:	4 x b – a → condition → 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 ? → e3
   pcmphiw    e0,e1,e2        ; e1  >  e0 ? → e2 (unsigned)
   por        e3,e2,e2        ; ( e1 == e0 ) or ( e1  > e0 ) →  e1 >= e0
```

PEOR PEOR

Vector eor

Operation: 64x a eor b → d
Syntax: POR <vea>,b,d
Condition Codes: not affected

Short: Bitwise logical operantion.

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

Vector maximum

Operation: 8 x max (a , b) → 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 x max (a , b) → d

Syntax: PMAXSW <vea>,b,d

PMAXUW <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

Vector minimum

Operation: 8 x min (a , b) → 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

PMINxW PMINxW

Vector minimum

Operation: 4 x min (a , b) → 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 x b → 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: alfa<100% ? a + alpha x b → d
alfa=100% ? b → 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

Vector or

Operation: 64x a or b → d
Syntax: POR <vea>,b,d
Condition Codes: not affected

Short: Bitwise logical operantion.

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 substract

Operation: b – a → 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

STOREC STOREC

Store counted bytes from register into memory

Operation: b → <vea> with a maximum of count bytes
Syntax: STOREC b,count,<vea>
Condition Codes: not affected

Short: Store at most "count" bytes from b into destination
Description: It stores between 0 and 8 bytes of b depending on the count value.

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 (count)				0	0	1	0	0	1	0	0

Count is a long, when negative there is no writing to memory.
Count is ignored when destination is a register.
Not allowed <vea> : #imm

Example:

```
    storec    d0,d1,(a2)
```

b d0

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

d d1

								0	0	0	0	0	0	0	3

Result:
vea (a2) memory content where a0 points to

1	1	2	2	3	3										

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 → <vea> 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	0	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 similair

STOREM STOREM

Store masked register into memory

Operation: b → <vea> 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	1	0	0	1	0	1

Not allowed <vea> : #imm
No masking is done when destination is a register.
Storeilm is similair

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 → <vea> 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	1	0	0	1	0	1

mask_mode: writing is done when:
0 - **L**ong: 2x 32bit color when msb=1
1 - **B**yte: 8x 8bit color-index when <>0
2 - word: 4x **S**ixteen bit color when color<>$f81f (= max red & blue = purple)
3 - **W**ord: 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:	F	8	1	F	0	0	3	4								
storem3 d0,#**1**,(a0)	B	result:	F	8	1	F			3	4	1	2			8	7	6	5
storem3 d0,#**2**,(a0)	S	result:					0	0	3	4	1	2	0	0	8	7	6	5
storem3 d0,#**3**,(a0)	W	result:					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: (An,(Av,Au) → d
Syntax: TEX8.512 (An,(Av,Au)),Dn
TEX16.256 (An,(Av,Au)),Dn
TEX24.64 (An,(Av,Au))*D0,Dn
TEX.b (An,Av*Dm,Au),Dn (next page)
Condition Codes: not affected

Short: Gets from picture array An (position Au,Av) a byte,word or 3byte.
Description: Gets a color from a texture position u,v. shifts the destination up and inserts the color there. An points to the texture. Au & Av are 16bit integer 16 bit fraction longs. Au & Av are seen as modulair, so always point inside the map.

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	An		
0	Au			d				0	0	1	1	1	1	1	0
1	Av			**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 thirth 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.

TEX TEX

Texture, sizeable without modulair

Operation: (An,Av*Dm,Au) → d
Syntax: TEX.b (An,Av*Dm,Au)),Dn
Condition Codes: not affected

Short: Gets from picture array An (position Au,Av) a byte.
Description: This TEX can have any mapsize, but there is no checking if Au & Av are in the map.
Gets a color from a texture position u,v. An points to the texture. Au & Av are 16bit integer 16 bit fraction longs. Dm is the vertical step in the texture.

Dn = (An + Av.h*Dm + Au.h)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	0	1	1	0	An		
0	Au			d				0	0	1	1	1	1	1	0
1	Av			**0**	0	0	0	Dm				0	0	0	0

Texture size: use mulipier Dm

Note:
This TEX is a concept supported from core 10084

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, only data-registers. 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 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 & **c**onditiononal 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

Note:
Too recent to be used on V2 (2.17).

FMOVEM FMOVEM

Move muliple 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, execept for the fact that the format in memory is a tiny bit different. For calculated results it makes no differens 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	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**

Note:
Newer cores will use the 'normal' eXtended format.

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.

Is there a second pipe ?

```
  movec pcr,d0 ; bit 0=enable super scalar  (second pipe)
  btst #0,d0
  bne second_pipe_active
```

PIPELINE STAGE

1 Icache Fetch	
2 Decoding	
3 Register fetch	
4 EA calculation	(owns address-registers)
5 Dcache Fetch	
6 ALU calculation	(owns data-registers)
7 Write back	

These instruction can be executed in the EA Unit:

```
  ADDQ #,An  ADDA #im,An  ADDA Reg,An
  SUBQ #,An  SUBA #im,An  SUBA Reg,An
  MOVE.L #im,An  MOVE.L Reg,An  LEA (ea),An
```

All other instructions are executed in the ALU.

EA result to EA input = no LATENCY
ALU result to ALU input = no LATENCY
ALU result to EA input = 2 cycle bubble

INSTRUCTION TIMING

CPU instructions normally are 1 cycle.
MUL=2 or 3, DIV=<18
MOVE16=4
MOVEM=n
JMP, JSR=1 except with a calculated ea, then its 4   { like JSR -6(a6) }

FPU instructions FNEG, FABS, FMOVE are 1 cycle
FADD, FCMP, FSUB, FMUL=6
FDIV=10, FSQRT=22

FMUL, FADD, FSUB, FDIV, SQRT = are all fully pipelined.
Other calculations (FSINCOS FTWOTOX etc) use a kind of micro-code that take about 100..200+ cycles.

Integer ea calculations costs
Free.
Excep exotic ones with indirect memory cost about 4 cycle.
([d16,An],Dn.s*n,d16)

Float converting costs
Free: Dn.s Dn.d #.s #.d #.x (single double extended)
Integer convert adds 1 cycle. (byte word long)

FPU Pipeline Explaned:
FADD = 6cycle
You only need 1 cycle to issue it. And you can also in this cycle issue a (second pipe) integer instruction with it. But if you NEED the result and want to use it, then the CPU will wait until the FADD is done.

This waiting will waste cycles. In which you could have executed instructions (5 cycles for free). You can issue integer or FPU instructions - this does not matter.

cycles	mnemonic	0	1	2	3	4	5	6	7	
0	FADD fp0,fp2			░						
1	FMUL #2.51.s,fp3			░	░					
2	FMOVE #7.s,fp4			░	░	░				
3				░	░	▓				fp4 finished
4				░	░					
5				░	░					
6	FMOVE fp2,fp0	░		▓	░					fp2 finished
7		▓			▓					fp0 & fp3 finished

OPTIMISING

Use the Clock Cycle Counter to see how many cycles your code took.

```
  movec ccc,a6              ; curent counter
  <your code>
  movec ccc,d7
  sub.l a6,d7               ; cycles used
```

Mis-aligned memory reads (cache hit) cost no extra.
Align writes for fastest result, so quad write to quad bound address.

Conditional instruction, a faster way to execute a single following instruction after bcc.s.

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

Touch can preload the data cache.

SPECS

– 64bit memorie Data-bus.
– 16kb ICache , 1cycle=16byte to CPU every cycle.
– 128kb DCache 3ported.
  1cycle=8byte read AND 8 byte write to/from CPU AND talk to mem.
– mem burst=32byte.(4x8) latency is around 12 CPU cyle.

The CPU itself detect continous memory access and will automatically prefetch the memory.