68k2 ISA 64 Bit Design

Introduction

A 64 bit processor ISA based on the 68000. Similar in some ways to the “coldfire”, but extended to have a more friendly and general implementation. Not all “old” opcodes are supported, and some instructions are recoded to better fit with a consistent 64 bit extension of the 68000 ISA. The is a specification and a speculation document. The processor may not be available. Be free to produce FPGA implementations of the ISA. The assembler level is a subset, and some instructions would have to be emulated or inlined as sequences of simpler instructions.

Addressing Modes

The following changes are made. The **XXX.W** mode is replaced by a **(An, d32)** mode for a 32 bit displacement. The **111 101** special register mode gets meaning being a **(PC, d32)** mode. The **111 110** mode accesses the **SR**, and the **111 111** mode accesses the **CCR**. That completes all changes to the addressing modes. The immediate mode is still special when used as a destination. Sometimes it does not make sense to use direct **An**, **Dn**, **SR** and **CCR** modes when an address is required. Sometimes the direct **An** is not used so as to free instruction space for other instructions.

The most important change to addressing is the operation of the d8 displacement modes, and a different full extension word format. A potential 24 bit displacement option is available by setting **d24** and appending another low word to the high byte displacement. The **Dn** register is pre-multiplied by the operand size, which would be 1, 2, 4 or 8. This sets the alignment.

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
SHS		DS		Dn			d24	Displacement d8							

The resulting indexing is **(An, Dn<<{size-16*SHS}.{DS<<size}, d8/24)** and is flexible for arrays of primary data types. Where there is some sub-select potential to limiting the width of the D register used in the calculation. The SHS essentially performs a D register shift right by a size, to allow splitting D registers into a number of indexes. This is better written **(An, Dn>SHS.DS, dX)** to get rid of confusion. The data transfer **size** comes from the instruction, not the addressing mode.

Operand and Transfer Sizes

Sometimes not all four sizes are available. Byte (00), word (01), long (10) and quad (11). If the size field is one bit, only long and quad are available. The general **MOVE** instruction format remains almost the same, but does not have a byte transfer size for example, and it never did. The workload of moving data keeps this an important large part of the opcode space. All other opcodes are more limited in addressing. See later for **MOVE.B**.

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	0	size		Src reg			Src mode			Dest mode			Dest reg		

The rest of the lower quarter of the opcode space is mainly short immediate mode dyadic opcodes. There is no sense in immediate mode destinations, and **BTST** is the exception needing a source only. The general **MOVE** instruction also has no sense in an immediate mode destination. The operand order was changed to assist in hardware decode of nonsense destination modes, for fast alternate instruction tracking, not for fast write pre-selection. A sweep through code, skipping unreachable, perhaps PC relative embedded data, would make 68000 **MOVE** codes easy to transform.

The rest of the lower quarter instructions will also accept **An** as a register to operate on, unlike the 68000. They have the following general format. I(0)/B(1) selects either immediate or bit operation.

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	0	0	0	Op1/Dn			I/B	Size/op2		mode			reg		

For B(1), op2 selects either **BTST(00)**, **BCHG(01)**, **BCLR(10)** or **BSET(11)** as the operation, and **Dn** that has the bit count in. If the immediate mode **111 100** is set, and I/B is set, and size is not zero, then something can be done to the register **Dn**. These are assigned **EXT** as a from to byte sign extend to word, word to long, and long to quad. **EXT.W/L/Q** as needed. A useful operation set when using register part packing.

For I(0), there are a number of sized immediate data operations.

11	**10**	**9**	**Operation**	**“111 100”**	**7**	**6**
0	0	0	**ORI**	?	size	
0	0	1	**ANDI**	?		
0	1	0	**SUBI**	**BIGN**		
0	1	1	**ADDI**	**PSHC**		
1	0	0	**BTST/BCHG/BCLR/BSET** immediate not register	¾ **EXT**	op2	
1	0	1	**EORI**	?	size	
1	1	0	**CMPI**	**TRAPV**		
1	1	1	**MOVEC** not immediate	½ **RTE/ILL**	op3	

The op3 is defined as **ADDR(00)** and **DATA(01)** as write, and **STATUS(10)** and **DATA(11)** as read. The **ADDR** and **DATA** actions have no issue with an immediate via **111 100** source, but the other two do, as they expect a destination. They are fully defined bit patterns. Yes, that’s right **MOVEC** as there is no MOVES on this design. To place two other supervisor only instructions here is nice, and cleans up the later opcode space. So the match for **STATUS** using **111 100** is **RTE**. The match for **DATA** using **111 100** is **ILL** or **ILLEGAL**. The data and the address of the control registers are supplied using different opcodes, and the address auto-increments on data access.

Return from subroutine **RTS** and the related **RTR** are assigned new opcodes on **111 100** destinations to remove some overcrowding later on in the opcode space. A new instruction **PSHC** pushes the **CCR** on to the stack. It is joined by **BIGN** which places the **PC** on the stack but continues in execution. This is useful for looping constructs. **RTD** and **TRAPV** also make an appearance.

0100 Miscellaneous

This section has been juggled significantly, and also has many omissions. The most general of not omitted is **LEA** as it has an instruction format like the following.

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	1	0	0	Destination reg			op			mode			reg		

There is no invalid meaning in the immediate mode, as it just fetches the **PC+2** as that is the address of this immediate data. There is however issue with the **An** and **Dn** addressing modes as they have no conceptual effective address. Bit **8** controls a lot. When zero the destination reg is replaced by a secondary op, and bits **7** and **6** become size as per usual. The **MOVEM** use pointers and not effective addressing. The pointer is fetched before the pattern of masked register bits. It's all very pointer **PC++** for assembling.

11	**10**	**9**	**8**	**7**	**6**	**Operation**	**111 100**	**An/Dn(bit3)**
0	0	0	0	size		**NEGX**	**RST**	
0	0	1	0	size		**CLR**	**STOP**	
0	1	0	0	size		**NEG**	?	
0	1	1	0	size		**NOT**	?	
1	0	0	0	size		**MOVEM** load (64 bit)	Uses operand as pointer indirect	
1	0	1	0	size		**TST**		
1	1	0	0	size		**MOVEM** save (64 bit)	Uses operand as pointer indirect	
1	1	1	0	0	0	**PEA**	See later	**Below!**
1	1	1	0	0	0	**LINK** uses **An** not **Dn**		**Dn**(0)
1	1	1	0	0	0	**UNLK**		**An**(1)
1	1	1	0	0	1	**JTR**	Uses operand as pointer indirect	
1	1	1	0	1	0	**JMP**	Uses operand as pointer indirect	
1	1	1	0	1	1	**JSR**	Uses operand as pointer indirect	
Dn			1	0	0	**MULQ** (64*64 bit)		
Dn			1	0	1	**DIVQ** (64/64 bit)		
Dn			1	1	0	**CHK**		
An			1	1	1	**LEA**	See later	**Below!**
An			1	1	1	**EXG**		**Dn**(0)
An			1	1	1	**EXG**		**An**(1)

The new opcode **JTR** is "jump threaded routine" and it performs an extra level of indirection before the **JSR** implicit and forced to happen after. The **EXG Dn,Dm** is covered later. **CHK** has byte mode and can use an **An** as the bound. **DIVQ** and **MULQ** are by default and design unsigned. The modification of the **RTE** instruction to have state information can accelerate the exception exit. There is not much use for this bit pattern otherwise.

Please note well: Uses operand as pointer indirect, means exactly that. **JMP #$xxxxxxxx** is valid. The 68000 would use effective address and not fetched content. **LINK** is d16 version only.

0101 ADDQ and SUBQ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	literal			A/S	size		mode			reg		

The A/S bit selects add or subtract. These instructions work as advertised in the 68000 ISA reference. The only difference being a quad word mode, and the fact that using address registers may be part used, if the quad word size is not specified, and byte size is possible. This allows packing data in address registers if required.

When the **111 100** immediate addressing mode is used, the following instruction is executed instead. The lack of a version using address registers or mixed type is not too big a problem. If register pressure is that high, you likely need a wide vector unit. It is more for getting the registers in line with calling highly optimized subroutines with specific register assignments after a long sequence of calculation. Compilers do not usually use it. Also it was covered earlier.

11	10	9	8	7	6	Operation
Dn			Dm			**EXG** exchange 64 bit data registers.

0110 Branching Out

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	condition				offset							
16 bit if offset = $0															
32 bit if offset = $ff															

There are no changes here. It’s 68000 all the way, including **BRA** and **BSR**.

0111 MOVEQ

General MOVEQ instructions can only use immediate bytes. It would be possible to set a d24 immediate on MOVEQ. The d24 bit in this case loads in another immediate word to form the low two bytes of a 24 bit immediate. The immediate 8 or 24 bit is sign extended to a 32 bit long. The rationale being the high long in the register would possibly be a packed long for “quick” code more often than not. **EXT.Q** can always be used on the off chance.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	Dn			d24	Immediate d8							

That completes half the instruction opcode space. There are some omissions (TAS, MOVE USP, BKPT and NOP) which are still options on the table.

1000 OR and DIV

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Dn			L/A	size		mode			reg		

The L/A bit selects logical **OR** or **DIV** in various forms. There is no memory modification form of **OR**, such that it is. This is alright for the added flexibility of **DIV**, and not BCD instructions here.

When the **DIV** is selected by bit 8 being 1, the size bits are used differently. Bit 6 selects between **L(0)** and **W(1)** forms, and bit 7 selects **S(0)** and **U(1)** choices.

L	W
64 / 32 → 32 r 32	32 / 16 → 16 r 16

1001 SUB

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	Dn			X	size		mode			reg		

The full featured subtraction. The X bit when set switches on the subtraction of the extend bit. There is no write back to memory version as size can be **SUB.Q** for the general case, including **An**.

1010 Customizations

The ISA does not have a trap vector for A. As a decision to depart from exact encoding was made, there seems little point in reserving such a large opcode space when the instructions are getting dense. It is not a compatible processor, but a similar processor, easing emulation and extending ability. The simple float processor is placed on the A line. The **An** register direct addressing mode is set aside to map onto **Fn** for efficient use of an instruction decode structure. The **Dn** register direct addressing mode does integer to floating point conversions. It is the main way of loading information into the FPU.

The FPU does not necessarily use an IEEE exact algorithm. The aim is to speed up general floating point processing, and not to replace an advanced vector unit.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	Fn			operation			mode			Reg		

8	7	6	Operation	111 100 immediate addressing
0	0	0	**FLD** (load **Fn**)	
0	0	1	**FSV** (save **Fn)**	**FIRT** (inverse root)
0	1	0	**FSUB**	
0	1	1	**FADD**	
1	0	0	**FMUL**	
1	0	1	**FDIV**	
1	1	0	**FATN2** (divider <ea>)	
1	1	1	**FPOW** (base <ea>, -ve is **FLOG**)	

1011 EOR and CMP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	Dn			L/A	size		mode			reg		

The L/A bit selects logical **EOR** or **CMP** in various forms. There is no memory modification form of **EOR**, such that it is.

1100 AND and MUL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Dn			L/A	size		mode			reg		

The L/A bit selects logical **AND** or **MUL** in various forms. There is no memory modification form of **AND**, such that it is. This is alright for the added flexibility of **MUL**, and not BCD instructions here. When the **MUL** is selected by bit 8 being 1, the size bits are used differently. Bit 6 selects between **L(0)** and **W(1)** forms, and bit 7 selects **S(0)** and **U(1)** choices.

L	W
32 * 32 → 64	16 * 16 → 32

1101 ADD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Dn			X	size		mode			reg		

The full featured addition. The X bit when set switches on the addition of the extend bit. There is no write back to memory version as size can be **ADD.Q** for the general case, including **An**.

1110 Rotations

Operation	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ASL/ASR	1	1	1	0	Count/Dn			L/R	size		I/R	0	0	reg		
LSL/LSR	1	1	1	0	Count/Dn			L/R	size		I/R	0	1	reg		
ROXL/ROXR	1	1	1	0	Count/Dn			L/R	size		I/R	1	0	reg		
ROL/ROR	1	1	1	0	Count/Dn			L/R	size		I/R	1	1	reg		

There are no size restrictions, and the instructions follow the 68000 ones.

1111 A F* Colliderthon

The while of the opcode space up to this part all has effect. There are some race conditions (or override selections), and this will generate some open opcodes. The main culprits are things like idempotent moves, but more of an option for **NOP** with an understanding the synchronization mechanism is different than read modify write instruction atomicity. The more important ones are double write ordering compatibility. These can be given unambiguous meaning, via instruction assignment of benefit, or side note of curiosity.

They ALL occur with pre and post decrement and increment instructions. As the order is read before write, double operands using the same **-(An),(An)+** structure can have definite meaning. Similar for **+(An),(An)-** even though they are both low quality code. **-(An),An** and **(An)+,An** are special in the sense the first is fully defined, and the second is error.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	Size3/4		An			0	0	1	0	1	1	An		
0	1	0	0	An			1	1	1	0	1	1	An		
0	1	0	0	1	0	0	0	size		0	1	1	An		

These are the free opcodes so far as defined by the consistency criterion. So 64 opcodes. So from the above some more operation codes can be created and defined.

14	8	Size	An bit #	Operation
1	1	<Null>	<Null>	**SWAP** use Dn not An
0	0	00	<Null>	*See later*
0	0	01	<Null>	**BKPT** An is breakpoint number
0	0	10	<Null>	**MOVE to USP** use Dn not An
0	0	11	<Null>	**MOVE from USP** use Dn not An
1	0	<Any>	1	**SYNC**

The thing about the last one with bit 14 set is that the exception will not occur before the literal immediate has been fetched. This means the instruction has already passed decode, and has finished some bus cycles. Specifically it has loaded some data from an address **PC+2** (or where the PC is may have changed, and the load has been committed to cache. It also seems good for a lock wait mechanism, as the thread can be identified by the **An** in the instruction, and bit shift operations can be mutually used by a number of threads to rotate the “lock” into an exception. No requesting task would want to prematurely unlock another thread. Hence **SYNC** as an instruction.

1111 The F Slot Really

The F line has a 3 bit coprocessor field in the hope that there would be 8. As I understand it the MMU hangs about on 000, and the IEEE FPU hangs about on 001. Cache management appears on 010. Everything else is somewhat “non standard” by origin.

ID	Standard Use
000	**MMU** do not assume fitted
001	**FPU (IEEE)** do not assume fitted
010	**CACHE** do not assume fitted
011	**MOVE16** do not assume fitted

In light of the openness of the **68k2** ISA, those four coprocessor IDs are left vacant. With interesting angles on cache design for low area and high speed, there is a need to analyse the required cache management instructions. The only instruction which readily stands out as needed, is one such that the data model of a program can be converted into a programmatic instance, which in turn is capable of being loaded into L1 cache. **FLUSH** is covered later.

With a cache which invalidates on other core writes, some of the issue is sorted if another core loads in the code. With that type of cache there is no mechanism implicitly for flushing a core’s own I cache. The restriction of having to snoop and invalidate on the I cache is lifted if the CPU write bus end of the I cache is retasked in hardware to respond to opcodes.

FLUSH is a supervisor only instruction. **PC** relative writes use the I cache, and a “cache line flush via reload” from L2 is less silicon area. It prevents the need for I cache write back buffers to L2. All **PC** relative write is thus performed into the D cache.

The F line slot remains free for system expansion. This is good. The whole immediate **111 100** instruction space has been filled in, with two exceptions covered later when the large table of future allocatable 32 and 48 bit instruction extension space is drawn up.

More on Address Modes

It was pointed out earlier that **JMP**, **JSR**, **JTR** and **MOVEM** use and extra level of indirection beyond the 68000. There is less “control mode addressing” issues, and that is made purely **LEA** and **PEA** which both just get a reference. There’s my two references joke on the horizon! The direct **An** and **Dn** modes of these two instructions were also covered. There is however an expansion possibility realised by the implementation of the following addressing mode, which although looking complete includes some redundancy, or more particular some valid combinations which produce no extra effect beyond that produced by some other combinations.

The **(PC,d8)** mode still has 3 unused slots. Here is the d8 word again. The reason is because a word shift right removes some high bits, and so wider specifiers only add leading “zeros”, or more exactly sign extends.

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	0	1	1	**PC#d12** or **An#d12** a displaced PC or An immediate mode											
1	1	1	0	**Dn**			d24	**((PC,Dn)d8) ((PC,Dn)d24) ((An,Dn)d8) ((An,Dn)d24)**							
1	1	1	1	**Dn**			d24	**((PC,d8)Dn) ((PC,d24)Dn) ((An,d8)Dn) ((An,d24)Dn)**							

And if the **PC#d12** or **An#d12** mode is a destination? Treat the same as immediate, supplying 2 extra opcode possibilities. Without using any of the F line.

The **((PC,Dn)d8)**, **((PC,Dn)d24)**, **((An,Dn)d8)** and **((An,Dn)d24)** modes add in a displacement, after one indirect. The modes **-(An)** and **(An)+** are valid in the 68k2 as they have an address generated, just not recalled, but the addressing arithmetic is done, even on instructions they were not valid on before.

So now seems a good time to build a table of all the remaining opcode space. As this will be useful when considering extensions to the instruction set.

MOVE.B Slow Form and Open Instruction Space

Adding back in this instruction for cases where it’s needed, usually IO, is as easy as assigning it to the **PC#d12** mode on **PEA**, and using the **d12** as the source and destination, with the modified order of destination specification is in the low six bits, and source in the higher six bits of **d12** to simplify reassembling code using **MOVE.B**.

For consistency it should now be considered what the **#d16** (or **#d32** or **#d64** with **MOVE.L** or **MOVE.Q**) mode should mean when it is a **MOVE** destination, as this was not covered earlier.

Under	Mode	Becomes	Features Extras	Length (bits)
PEA	**PC#d12**	?	12 bits	32
PEA	**An#d12**	?	A register spec, 12 bits	32
PEA	**#d16**	**RESET**	*None*	16
LEA	**PC#d12**	?	A register spec, 12 bits	32
LEA	**An#d12**	?	Two register specs, 12 bits	32
LEA	**#d16**	**FLUSH An**	*Register contains address to I cache flush*	16
MOVE	**#d16**	See later	*A complex issue with 0000 group*	32 / 16
FSV	**PC#d12**	**FSCC**	*Some FPU decode connect*	32
FSV	**An#d12**	**FMCC**	*Some FPU decode connect*	32

As can be seen there is plenty of ISA expansion space in the 32 bit opcodes. All currently unassigned raise an exception. An “unresolved instruction exception” so that the instructions set can be expanded without using up coprocessor IDs on the F line. This replaces the A line exception.

Well those were some complexities of fitting a mostly translatable ISA into a 16 bit word format. Quite a lot of the 68020 instructions were removed in favour of 64 bit consistency without expanding the instruction width. There is even good reason to perform various simple mappings between the old and new ISA as automated tools tend to be better, the closer the domains of input and output. The biggest problem I can see is with calculated jump vectors. The automated tool can’t know if the address will remain the same if the instructions differ in size.

Having fewer size differences helps as the instruction byte counting will remain closer to the original, and so have less code to check via other methods. Any code block not containing certain instructions (using certain addressing modes) can be ruled out for advanced processing. It maybe later optimized after profiling to use the more advanced addressing mode options and fewer restrictions on addressing modes.

Relocation Addressing

From earlier it was noted that **JMP #$xxxxxxxx** to assist in code conversion of JMP (xxx).L but loaders tend to use **PC** relative code for compilation for good reasons of relocation of code. In a non MMU context this is the best way of making position independent code. Not using effective addresses has the consequence of leaving just the mode **PC#d12** for 4 kB bounding.

When using the **(PC,d32)** mode for example to replace an old jump pointer, the indirection can be altered to within a relocation table due to the indirection, placed at the end of the code. This costs three times the memory in most cases, but the differing instruction coding lengths can be taken up by having an altered number, and provides a fast translation “up” time.

A **(An,d32)** would work for indirect JMP (An) conversion, used in calculation of jump tables, if it were (getting the subjunctive) not for the indirection. Of course a straight **JMP An** would not be sufficient as some opcodes are of differing implementation length. Having the extra indirection step help in building the “instruction length matcher” jump table, as well as making the instruction set more consistent.

As the 68k2 is 64 bit a 64 bit jump should be done via **MOVE.Q #xxxxxxxxxxxxxxxx, PC#0** perhaps. It would work. A 64 bit indirect jump would be done perhaps by **MOVE.Q (PC, Dn>m.B, d24), PC#x** as there is not many others having the compact table and range to the table. This also puts the calculation in **Dn** there from the start. The constant **x** can be used to select from an entry table. Stacking a 64 bit **PC** would have some unexpected effects for non normalized code. A 32 bit JSR would here be useful. Thankful for stack frames?

Waffle “The Language?”

What’s the difference between an inner function in an extended outer function and a class OOP design. Losing one **An** for tracking the class parent. Of course outer extended functions could only be allocated stacked order by call. Does this make Java’s “new” keyword a task instantiation future? Given encapsulation is supposed to give a lot of the things needed for multi-threading … you decide. Even pointers to tasks are easy, just no self reference rings.

This should be easy. How about an intrinsic data type for circular reference. A list node which can reference anything, but can’t be extended as it’s “final”. Reference counted, and all other types not for a data saving. Always try to sort themselves on a sort chain to precede what they refer to, get marked sorted unless repointed. Any sorted not able to precede marked circular, and never resorted unless repointed. Well not brilliant, but definite improvement in the GC search size.

If the methods of “list” class/function are all written in a specific way, stack bound provability is also possible. Especially if an “inner final” class/function extends list and marks an invisible node, such that consistent lists are always made. Each list as it were has an “ID”, and no cross joining. Similar constraints for tree objects would be good. This allows referring to any list node, and any list node to be (contain) any object. That’s what lists do. Of course, list processing becomes slower, as each list element must also hidden point to its owner inner final, and that in turn to its “inner inner final” unification group. The “inner final” keeps the reference counting done.

Why this? Well, for checking out possible instruction optimizations for OOP. **(An, Dn, d8/d24)**.

Implicit Displacement Size

In **(An, Dn, d8/d24)** the displacement always being a byte is crazy. The size of the transfer should have an effect on all transfers. This effectively increases the range for **.Q** to an 11 bit offset aligned, and a 27 bit offset respectively. The area overhead is minimal, and would allow multiple extra fast access slots in objects.

The question of the relevance to d24 transfers is moot in the 16.7 million bytes. Even the d8 offset at 256 bytes is a lot of “locals”, and although an 8 bit shifter is minimal area, it’s in the lower bits and carry look ahead would be slowed for timing closure. For consistency with other modes it is not used. This maybe considered a bad idea later, but it does improve the ease of mentally getting an address when coding.

The **((An, d8)Dn)** and similar double indirect modes, are very useful that way round for array indexing. This is far superior to the addressing ((An, Dn)d8) suggested in earlier versions of this document. In fact a fabulous mode. Full on super, smashing, great. The mode **((An, Dn)d8)** and

similar may not be as useful. They do allow dynamic table indexing in a different way, with static element index recall of multiple tables, instead of dynamic recall of a single statically selected table.

Put that way it does sound a bit like and array of object iterated over a fixed instance variable.

A little FPU

The two extensions of the ISA under **FSV** are now allocated. They are **FSCC** and **FMCC** which are FPU set condition code, and FPU mask condition code. The **d12** is the condition code bits, and the **An** in the **FSV** coding **An#d12**, becomes a **Dn** to read the condition code into. It will be noted that there is an **Fn** register spec too. This is register to which the condition code applies. So each register has a 12 bit code, which can be altered to perhaps clear errors or set result precision.

The mask feature on **FMCC** has the effect of zeroing out any bits using “or” which are not of interest, and only returning the wanted bits for a quick test. The FPU will stall if information is in flight. A small instruction queue would be good for better issue order of addressing operands without a stall of the main CPU execution pipeline. The synchronization point and perhaps wait stall occurs when information is needed back from the FPU.

The FPU does either of the threads on the core, so a stall and thread switch is not necessarily going to stop the FPU, so the FPU has 16 registers in real terms!

11	10	9	8	7	6	5	4	3	2	1	0
N	**NaN**	**LP**					**V**	**Z**	**BR**		**DN**

Where **N** is negative, **NaN** is not a number, **LP** is loss of precision when two close numbers are subtracted and such, it is a count of the number of lost bits, and saturates at 31. Then **V** is overflow, and **Z** is underflow. Note that **N** and **Z** can both be true. A register can be quickly zeroed or voided by setting **Z** or **NaN**, or set to infinity by setting **V**. **BR** is the would have been best rounding mode, covered later. **DN** is set if the register is denormalized.

When setting the register some bits have a different function such as **D** for direct output to bypass integer to float and float to integer. **AP** if not zero sets the acceptable **LP** before the register goes **NaN** all automatically like. **R** is the rounding mode to set. The **I** bit is a convenience bit to store an integer within the register for use as fast store for the CPU. The FPU operations treat it as **NaN** paradoxically, as there is no auto conversion coercion. The **M** bit sets the register as Gaussian noise Monte-Carlo style, SD write, and stats read.

11	10	9	8	7	6	5	4	3	2	1	0
D	**I**	**AP**					**V**	**Z**	**R**		**M**

Of ORI and Friends

The need to access and change some global context, without loading it in? There location in a quite almost perfect move anywhere structure is strange. But bytes moving versus immediate action on data using 1 instruction? Thinking about pipelines gives the IN to OUT, versus IN to ALU to OUT paradigm. A pipeline match says put those at some other place.

So if them ops are moved to the MOVE.B current assignment full orthogonal MOVE is done. Detecting an immediate on the MOVE destination opens the possibility, of NOT placing them at the current MOVE.B assignment, but on the **#d16** word. One advantage of this is the pre-decoded source register and addressing mode specification, which can start the fetch if it itself is not **#d16** (or other depending on size) immediate. This “frees” 8 bits (including size), and adds another 2 bits.

The bit 8 at zero forms are the most useful with **ADDI** and others, along with the non register forms of the bit operations including **BTST**. The registered form of the bit tests are not that useful for global flags. The indexing is pitiful too. So this then gives the following table.

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Q	I/B	Size/op2		Op1/Dn			d24	d8							

The **Q** bit essentially is best purposed to assist in the counting of trailing literal words.

15	**8**	**Literal**
0	0	d8
0	1	d24
1	0	d40
1	1	d56

For the register mode **BTST** and other bit set and clear ones, the **Q** is free, along with **d8** and **d24** for other possible use. For optimality **RTS** and other return instructions would have to be first in line for the double immediate **MOVE** case, and not trailing literal. If the immediate forms of the bit instructions were combined into the registered forms by retasking so, with register enable.

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
BE	1	op2		Dn			En	W		bit					

The flag **BE** would indicate the bit end to start at 0 for 0, 1 for 63. **W** would be a word shift similar in application to **Dn**>>16*__W__ for effect. The extra **xxxI** instruction to add? Something for a **PC#d12** destination perhaps with the **d12** subtracted off? **NTAI** is a nice negative test then “not” and add if so immediate else do nothing. **MOVEC** also has to move? The **d8** provides the address number? A write after saving the read? An exchange for task switching? Yes. All the **ANDI #d16, #d16** have little use when strength reduction happens. There are 4 of 16 bit opcodes that can be mapped **RTS**, **RTR**, **RTD** and **RTE** for some consistency.

Using offset arithmetic on **PC#d15** and **An#15** modes in reverse, gets the displacement subtracted (but added is better to do) when going in reverse. This adds lots of useful displacements and jump summation targets. That would be good, so yes.

There is an extra **F*** address source destination overlap created by aligning the **MOVE.B** with the others. This is allocated to **TRAP** with the **An** used as the trap number. This supplies 8 traps. This is a lower number than on the 68k, which has 16. Usually OSes have more than 16 calls, and so use a splitter in general, and base the split on the needed system state to spin up. I think 8 is enough.

There a just a few odds and ends left for filling in the **111 100** or **#d16** as destination mode. Not all instructions have destinations with this mode. The reduced availability table follows, with the main opcode blocks, and no reference to the odd small groups.

Under	Mode	Becomes	Features Extras	Length (bits)
PEA	**PC#d12**	**GTHR**	*See later*	32
PEA	**An#d12**	?	A register spec, 12 bits	32
LEA	**PC#d12**	?	A register spec, 12 bits	32
LEA	**An#d12**	**PSO**	*Uses Dm not An*	32 / 48
PSO	**#d16**	*See later*	*Immediate 3 operand*	96 / 112

The 2nd and 3rd row as a pair cry out for an A/D reg selector, a reg and 12 bits perhaps for 2 more operands. But that would not leave any room for the operation selector. To cut down on bus cycles row 4 could be a 2 input 1 output op, with 6 bits to spare. Such that the expansion word is as below. As a pseudo op **PSO** for ease until opcodes have been decoded upon.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	op			msk	size		mode			reg		

The table of ops should include ALU things like the **xxxI** but in **xxx3** form. The msk bit selects a further “and” mask loaded from the instruction stream. The full quad is processed, but carry can be suppressed between sizes by **SUBS3** and **ADDS3** for some kind of vector extension. The size argument sets the “split” of where the condition codes come out of the lower bits, and also suppress the carry (if requested) over the split.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Dn .B selectors								**Dm .B selectors**							

There are also **111 100** ops which work on 2 registers. In this case the immediate quad operand, coming before any mask word, becomes exchanged with **Dn** before the operation, and the result goes to **Dm** for immediate, with **Dn** not actually changing in value.

11	10	9	Operation	“111 100”
0	0	0	**OR3**	Excellent!
0	0	1	**AND3**	
0	1	0	**SUBS3**	
0	1	1	**ADDS3**	
1	0	0	**NTA3**	
1	0	1	**EOR3**	
1	1	0	**MULH3**	
1	1	1	**MULL3**	

A **GTHR** gather instruction takes the following, with size deciding how many times to “iterate the source mode reg to fetch” enough size arguments to fill **Dn**. If the size field specifies a quad transfer, this is remapped to provide bit to byte expansion. The L/B flag does either little or big endian.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	Dn			L/B	size		mode			reg		

This makes a reasonable vector processor for small scale use.