**FT68000**

robfinch<remove>@finitron.ca

# Overview

FT68000 is Finitron’s rendition of a 68000 similar core. It features all the instructions, opcode and addressing modes of the 68k with only minor differences. Differences are outlined in this document. The instruction formats are similar enough to the 68k’s to make it possible to leverage existing software for use with the FT68000.

The most significant difference from the 68000 is the use of tasks to handle exceptional conditions. The vector table contains task identifiers rather than program counter vectors.

The FT68000 supports high-speed hardware multi-tasking. The program visible register state is stored in an internal task state memory. Loading or storing the entire register set is a single cycle operation. Task switching may be done in fewer than 10 cycles (exact timing TBD).

## Summary of Differences from the 68k

### Definition of Core Control and Status Registers (CSR’s)

The core uses a reserved address range to implement a number of additional control and status registers beyond the status register normally visible in the programming model. Addition registers include:

|  |  |  |
| --- | --- | --- |
| Address | Register |  |
| $FFFFFFE0 | CORENUM | sometimes called a hartid, identifies the core in a multi-core system |
| $FFFFFFE4 | TICK | tick count, counts up continuously after external reset. |
| $FFFFFFFC | TASK | task register, used to set the running task. |
|  |  |  |
|  |  |  |
|  |  |  |

The additional registers may be updated or read using the move.l instruction. Note that byte and wyde instructions will not access the CSR’s.

### Internal Task State Memory

The core contains an internal memory to store task state. This memory is accessible with the load task state LTSK and store task state STSK instructions. There is enough task state memory to store state for 512 tasks. The visible register set is transferred to and from the task state memory by a task switch operation. Moving a value to the task register with the move.l instruction will cause a task switch. The task may also be switched by specifying a task number as an operand to the jump or jump to subroutine instruction.

### Stack Pointer

There is only a single stack pointer rather than two separate stack pointers for user and supervisor modes. Since there is a separate stack pointer for each task having two stack pointers would be redundant. The task itself may be specified as a user or supervisor task depending on the settings in the flags register for the task. Effectively there are 512 stack pointers. And in fact duplicates of all program visible registers for each task.

On reset the 68k loads the stack pointer from the first long word of memory. The FT68000 does not load the stack pointer. The stack pointer will need to be set in the reset routine.

### Program Counter

At reset the program counter is loaded with the address $FFFC0000 and program execution begins there. This is different from the 68k which loads the PC automatically from the second vector in the vector table.

### Bus

The FT68000 uses 32 bit data and address busses. With a wider data bus some of the instructions accessing 32 bit data execute more quickly than they would for the 68k. The FT68000 uses a WISHBONE bus rather than the asynchronous 68k bus.

### Endian

The FT68000 is a little endian machine. The least significant bytes are stored at the lowest address. This differs from the 68k which a big endian machine. This impacts the order of data stored in the system and specification of immediate constants.

### Branch Displacements

The branch displacement is shifted left once when added to the program counter. This doubles the effective range of branches. The 68k does not shift the displacement. The least significant bit would always be zero since instructions parcels are 16 bits in size. The FT68000 makes use of this extra bit that would otherwise be zero.

### Decrement and Branch Instruction

The decrement and branch instruction decrements the whole data register rather than just the low order word as in the 68k. This means loops longer than 64ki can be formed more easily.

### Immediates

There is an additional immediate addressing mode that allows the size of the immediate constant to be controlled. Immediates may be either sixteen or thirty-two bits regardless of the size of the operation. Sixteen bit constants are sign extended to thirty-two bits when a 32 bit operation is present. It is possible to perform a 32 bit op using just a 16 bit constant. This conserves code space.

### Task Numbers

Task numbers are always odd and vary from 1 to 1023.

# Interrupts, Exceptions and Traps

## Vector Table

This table is a list of task identifiers corresponding to the task to execute when a given exception, interrupt or trap occurs. There are 512, 16 bit word entries in the task vector table. Each entry may point to a different task or several entries may point to the same task. A task may be invoked automatically by hardware for exception processing.

|  |  |  |  |
| --- | --- | --- | --- |
| Vector | Address Offset | Usage |  |
| 002 | 0004 | Bus Error |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Instruction Set Description

Only instructions which differ from the 68k are described here.

## DIVS / DIVU

Description:

This instruction divides a data register by a long word value determined by the source addressing mode. It differs from the 68k divide instructions in that the divisor is a long word rather than a word. It performs a 32/32 divide rather than a 32/16 divide.

Instruction Format: < same as 68k >

## JMP – Jump

Description:

This instruction differs from the 68k in that it also performs task switching if a task number is specified as an operand. Task numbers are always odd and instruction addresses are always even. This allows the processor to differentiate between the two.

Instruction Format: <same as 68k>

## JSR – Jump to Subroutine

Description:

This instruction differs from the 68k in that it also performs task switching if a task number is specified as an operand. Task numbers are always odd and instruction addresses are always even. This allows the processor to differentiate between the two. Using the JSR instruction to activate a task causes the current task number to be stored on the stack of the incoming task. The original task may be returned to using an RTS instruction.

Instruction Format: <same as 68k>

## LTSK – Load Task Memory

Description:

This instruction loads an internal task memory entry with values for each of the program visible registers from a record in main memory. The D0 register identifies which task memory entry to update. This instruction is used primarily during system initialization to setup initial tasks.

Instruction Format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 15 12 | 11 8 | 7 6 | 5 3 | 2 0 |
| A4 | 04 | 02 | M3 | Xn3 |

Task Record Layout in Memory

|  |  |
| --- | --- |
| Table Offset | Value for |
| 00 | d0 |
| 04 | d1 |
| 08 | d2 |
| 0C | d3 |
| 10 | d4 |
| 14 | d5 |
| 18 | d6 |
| 1C | d7 |
| 20 | a0 |
| 24 | a1 |
| 28 | a2 |
| 2C | a3 |
| 30 | a4 |
| 34 | a5 |
| 38 | a6 |
| 3C | sp |
| 40 | flags |
| 44 | pc |
| 48 | reserved |
| 4C | reserved |

## RTS

Description:

This instruction may be used to return from a subroutine or return back to a task that invoked the current task. RTS can tell what operation to perform by looking at the data on the stack. Task numbers are always odd and addresses are always even. So if the instruction detects an odd value on the stack a task return is assumed.

Instruction Format: <same as 68k>

## TRAP

Description:

This instruction performs a function similar to that on the 68k however a task is invoked to handle the exception rather than executing an exception routine. The TRAP instruction differs from the 68k in that it supports identifying up to 512 vectors rather than being limited to 16. Trap vector codes 1 to 15 are quick traps which invoke the tasks identified in the vector table from vector 33 to 47.

Opcode Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | E | | 4 | Vect4 | TRAPQ |
| 4 | E | | 4 | 0 | TRAP |
| ~7 | | Vect9 | | |