# FT64

# Overview

FT64 is a two-way superscalar processing core capable of executing up to two instructions per clock cycle. The core features register renaming to avoid data hazards.

## Goals

One of the primary goals for the development of this core was the implementation of a register renaming mechanism. The author also wanted a stream-lined core as a starting place.

## Register Set

There are 32 general purpose registers in the architecture. R0 always has the value zero.

|  |  |  |
| --- | --- | --- |
| Register | Description / Suggested Usage | Saver |
| r0 | always reads as zero |  |
| r1-r2 | return values / exception | caller |
| r3-r10 | temporaries | caller |
| r11-r17 | register variables | callee |
| r18-r23 | function arguments | caller |
| r24 | type number / function argument | caller |
| r25 | class pointer / function argument | caller |
| r26 | thread pointer | callee |
| r27 | global pointer |  |
| r28 | exception link register | caller |
| r29 | return address / link register | caller |
| r30 | base / frame pointer | callee |
| r31 | stack pointer | callee |

## Caches

The core has both instruction and data caches in order to improve performance.

The instruction cache is a two level cache (L1, L2) allowing better performance. The first level cache is fully associative, the second level cache is four-way set associative. L1 is 2kB in size and made from distributed ram in order to get single cycle performance. L1 is organized as 64 lines of 32 bytes. L2 is 16kB in size implemented with block ram. L2 is organized as 512 lines of 32 bytes. The instruction cache is dual ported to allow two instructions to be fetched at one time.

The data cache is organized as 512 lines of 32 bytes (16kB) and implemented with block ram. Access to the data cache is multicycle. The data cache has three read ports allowing three load operations to be in progress at the same time. Stores write through to memory. There is only a single write port on the data cache.

## Branch Predictor

The branch predictor is a (2,2) correlating predictor. The branch history is maintained in a 512 entry history table. It has four read ports for predicting branch outcomes, one port for each instruction in the fetch buffer.

## Operating Levels

The core has eight operating levels. The operating level corresponds to the interrupt level. The highest operating level is operating level zero which is called the machine operating level. It corresponds to interrupt level seven. When an interrupt occurs the operating level is set to the level corresponding to the interrupt. As can be seen in the table below the operating level is the complement of the interrupt level. A given operating level may redirect the interrupt to a lower level using the REX instruction.

|  |  |  |  |
| --- | --- | --- | --- |
| IRQ Level | Operating Level | Privilege Level | Moniker |
| 0 | 7 | 7 to 255 | user |
| 1 | 6 | 6 | supervisor |
| 2 | 5 | 5 | supervisor |
| 3 | 4 | 4 | supervisor |
| 4 | 3 | 3 | supervisor |
| 5 | 2 | 2 | supervisor |
| 6 | 1 | 1 | hypervisor |
| 7 | 0 | 0 | machine |

## Privilege Levels

The core supports a 256 level privilege level system. Privilege level zero is assigned to operating mode zero. Privilege level one is assigned to operating level one. Privilege levels 2 to 6 are assigned to their corresponding operating level. The remaining privilege levels are assigned to operating level seven.

## Control and Status Registers

### HARTID (0x001)

This register contains a number that is externally supplied on the hartid\_i input bus to represent the hardware thread id or the core number.

### TICK (0x002)

This register contains a tick count of the number of clock cycles that have passed since the last reset.

### CAUSE (0x006)

This register contains a code indicating the cause of an exception or interrupt. The break handler will examine this code in order to determine what to do. Only the low order 16 bits are implemented. The high order bits read as zero and are not updateable.

### TVEC (0x030 to 0x037)

These registers contain the address of the exception handling routine for a given operating level.

### EPC (0x040)

This register contains the address of the interrupted or exceptioned code.

### STATUSL (0x044)

This register contains the interrupt mask, operating level, and privilege level stack. When an exception or interrupt occurs this register is shifted to the left and the current status copied to the low order bits, when an RTI instruction is executed this register is shifted to the right and the status bits copied from the low order bits of the register.

|  |  |  |  |
| --- | --- | --- | --- |
| 63 14 | 13 6 | 5 3 | 2 0 |
| Stack area | PL8 | OL3 | IM3 |

### STATUSH (0x045)

This register contains the interrupt mask, operating level, and privilege level stack. When an exception or interrupt occurs this register is shifted to the left when an RTI instruction is executed this register is shifted to the right.

|  |  |
| --- | --- |
| 63 62 | 61 0 |
| ~2 | Stack area |

### CODEBUF (0x080 to 0x0BF)

This register range is for access to 64 adaptable code buffers. The code buffers are used by the EXEC instruction in order to execute code which may change at run-time.

# Exceptions

## External Interrupts

There is very little difference between an externally generated exception and an internally generated one. An externally caused exception will force a BRK instruction into the instruction stream. The BRK instruction contains a cause code identifying the external interrupt source.

## Effect on Machine Status

The operating mode is always switched to the machine mode on exception. It’s up to the machine mode code to redirect the exception to a lower operating mode when desired. Further exceptions at the same or lower interrupt level are disabled automatically. Machine mode code must enable interrupts at some point. This can be done automatically when the exception is redirected to a lower level by the REX instruction. The RTI instruction will also automatically enable further machine level exceptions.

## Exception Stack

The program counter and status bits are pushed onto an internal stack when an exception occurs. This stack is only eight entries deep as that is the maximum amount of nesting that can occur. Further nesting of exceptions can be achieved by saving the state contained in the exception registers.

## Exception Vectoring

Exceptions are handled through a single vector. An exception handler at the machine level may redirect exceptions to a lower lever handler identified in one of the vector registers. More specific exception information is supplied in the cause register.

## Reset

The core begins executing instructions at address $FFFC0010. All registers are in an undefined state.

# Instruction Set Description

## Formats

Instructions have a fixed 32 bit format. Immediate constants may be extended using prefix instructions. There are only a handful of different instruction formats.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Immed16 | | | | | | Rt5 | | Ra5 | Opcode6 | RI |
| Funct6 | | | ~5 | Rt5 | | Rb5 | | Ra5 | Opcode6 | RR |
| Immed16 | | | | | P2 | | Cond3 | Ra5 | 01h6 | B |
| Op2 | OL3 | Regno11 | | | Rt5 | | | Ra5 | 0Eh6 | CSR |

## Arithmetic Operations

Arithmetic operations include addition, subtraction and comparison.

## Logical Operations

Logical operations include bitwise and, or, and exclusive or.

## Memory Operations

Memory operations include loads and stores of bytes, words or half-words. There isn’t yet a full complement of memory operations in order to keep the size of the core smaller. Notably missing are instructions to load / store 16 bit quantities. The core can perform loads using indexed addressing but not stores. Indexed addressing with stores is not allowed because it would require too many read ports.

## Control Flow Instructions

Control flow instructions include jumps and branches, breakpoint and return instructions.

# ADD

Description:

Add two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 04h6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 046 | ~5 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

# AND – Bitwise And

Description:

Perform a bitwise and operation between two operands. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 08h6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 086 | ~5 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

# ASR – Arithmetic Shift Right

Description:

Bits from the source register Ra are shifted right by the amount in register Rb or an immediate value. The sign bit is shifted into the most significant bits.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 246 | ~5 | Rt5 | Rb5 | Ra5 | 02h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 256 | ~4 | I1 | Rt5 | Imm5 | Ra5 | 02h6 |

Clock Cycles: 1

# BEQ/BNE/BMI/BPL – Conditional Branch

Description:

If the branch condition is true, a sixteen bit sign extended value is added to the program counter. The branch is relative to the address of the instruction directly following the branch. The immediate value may not be extended with a prefix instruction.

Instruction Format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Immed16 | P2 | Cond3 | Ra5 | 01h6 |

|  |  |  |
| --- | --- | --- |
| Cond3 | Mne. |  |
| 0 | BEQ | register Ra = 0 |
| 1 | BNE | register Ra <> 0 |
| 2 | BMI | register Ra < 0 (bit 63 is set) |
| 3 | BPL | register Ra >=0 (bit 63 is clear) |
| 4-7 |  | reserved |

The P2 field is reserved for branch prediction hints.

|  |  |
| --- | --- |
| P2 | Prediction Type |
| 0 | no static prediction (use branch history) |
| 1 | reserved |
| 2 | always predict as not-taken |
| 3 | always predict as taken |

If a branch prediction is supplied, then the branch instruction doesn’t occupy room in the history tables.

# BFCHG – Bit-Field Change

Description:

The bit-field change instruction inverts all the bits in a bit-field.

Instruction Format:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ~10 | | ~2 | | me6 | | ~2 | mb6 | | 1Ah6 |
| 026 | 25 | | Rt5 | | ~5 | | | Ra5 | 02h6 |

Clock Cycles: 1

ALU Support: ALU #0 Only

# BFCLR – Bit-Field Clear

Description:

The bit-field clear instruction zeros out all the bits in a bit-field.

Instruction Format:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ~10 | | ~2 | | me6 | | ~2 | mb6 | | 1Ah6 |
| 026 | 15 | | Rt5 | | ~5 | | | Ra5 | 02h6 |

Clock Cycles: 1

ALU Support: ALU #0 Only

# BFEXT – Bit-Field Extract

Description:

Extracts a bitfield from register Ra located between the mask begin (mb) and mask end (me) bits and places the sign extended result into the target register. This instruction may be used to sign extend a value beginning at any bit.

Instruction Format:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ~10 | | ~2 | | me6 | | ~2 | mb6 | | 1Ah6 |
| 026 | 55 | | Rt5 | | ~5 | | | Ra5 | 02h6 |

Clock Cycles: 1

ALU Support: ALU #0 Only

# BFEXTU – Bit-Field Extract Unsigned

Description:

Extracts a bitfield from register Ra located between the mask begin (mb) and mask end (me) bits and places the zero extended result into the target register.

Instruction Format:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ~10 | | ~2 | | me6 | | ~2 | mb6 | | 1Ah6 |
| 026 | 65 | | Rt5 | | ~5 | | | Ra5 | 02h6 |

Clock Cycles: 1

ALU Support: ALU #0 Only

# BFINS – Bit-Field Insert

Description:

The bit-field insert instruction inserts the contents of Rb into a bit-field.

Instruction Format:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ~10 | | ~2 | | me6 | | ~2 | mb6 | | 1Ah6 |
| 026 | 35 | | Rt5 | | Rb5 | | | Ra5 | 02h6 |

Clock Cycles: 1

ALU Support: ALU #0 Only

# BFINSI – Bit-Field Insert Immediate

Description:

The bit-field insert immediate instruction inserts an immediate value into a bit-field. The constant is a maximum of fifteen bits and is zero extended to the width of the bit-field.

Instruction Format:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Imm[14..5] | | ~2 | | me6 | | ~2 | mb6 | | 1Ah6 |
| 026 | 45 | | Rt5 | | Imm[4..0] | | | Ra5 | 02h6 |

Clock Cycles: 1

ALU Support: ALU #0 Only

# BFSET – Bit-Field Set

Description:

Sets the bits to one of the bitfield in Ra located between the mask begin (mb) and mask end (me) bits and stores the result in the target register.

Instruction Format:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ~10 | | ~2 | | me6 | | ~2 | mb6 | | 1Ah6 |
| 026 | 05 | | Rt5 | | ~5 | | | Ra5 | 02h6 |

Clock Cycles: 1

ALU Support: ALU #0 Only

# BRK – Hardware / Software Breakpoint

Description:

Invoke the break handler routine. The break handler routine handles all the hardware and software exceptions in the core. A cause code is loaded into the CAUSE CSR register. The break handler should read the CAUSE code to determine what to do. The break handler is located at $FFFC0000. This address should contain a jump to the break handler. Note the reset address is $FFFC0010.

Instruction Format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 19 | 18 16 | 15 | 14 6 | 5 0 |
| Immed13 | L3 | S | Cause Code9 | 00h6 |

S = 1 = software interrupt – return address is next instruction

S = 0 = hardware interrupt – return address is current instruction

L3 = the priority level of the hardware interrupt, the priority level at time of interrupt is recorded in the instruction, the interrupt mask will be set to this level when the instruction commits. This field is not used for software interrupts and should be zero.

Cause Code = numeric code associated with the cause of the interrupt.

# CALL – Call Subroutine

Description:

Call subroutine. This instruction first decrements the stack pointer then stores the address of the next instruction on the stack.

Instruction Format:

The short format first shifts the address field of the instruction by two bits to the left then sign extends the address to 64 bits. This allows accessing s subroutine within the first or last 128MB region of memory.

|  |  |
| --- | --- |
| Address[27..2] | 19h6 |

The long format for the instruction does not shift the address field. Instead the field is extended using immediate constants.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Address[41..16] | | | | 1Ah6 |
| ~4 | Address[63..42] | | | 1Bh6 |
| Address[15..0] | | ~5 | ~5 | 19h6 |

Clock Cycles: 0.5

# CLI – Clear Interrupt Mask

Description:

The interrupt level mask is set to zero enabling all interrupts. This is an alternate mnemonic for the SEI instruction where the mask level to set is set to zero by the assembler.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 306 | ~5 | ~5 | 05 | 05 | 02h6 |

Clock Cycles: 0.5

# CMOVEQ – Conditional Move Equal

Description:

The conditional move if equal instruction moves the contents of register Rb to the target register Rt if Ra is zero. Otherwise the contents of register Rc are moved to the target register.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 28h6 | Rt5 | Rc5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

# CMOVNE – Conditional Move Not Equal

Description:

The conditional move if not equal instruction moves the contents of register Rb to the target register Rt if Ra is non-zero. Otherwise the contents of register Rc are moved to the target register.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 29h6 | Rt5 | Rc5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

# CMP – Signed Comparison

Description:

The compare instruction places a 1, 0 or -1 in the target register based on the relationship between the two source operands. If they are equal a zero is placed in the target register, if register Ra is less than the second operand then a -1 is placed in the target register, otherwise a 1 is placed in the target register. The values are treated as signed operands.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 06h6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 066 | ~5 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

# CMPU – Unsigned Comparison

Description:

The compare instruction places a 1, 0 or -1 in the target register based on the relationship between the two source operands. If they are equal a zero is placed in the target register, if register Ra is less than the second operand then a -1 is placed in the target register, otherwise a 1 is placed in the target register. The values are treated as unsigned operands.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 07h6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 076 | ~5 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

# CSR – Control and Status Access

Description:

The CSR instruction group provides access to control and status registers in the core. For the read-write operation the current value of the CSR is placed in the target register Rt then the CSR is updated from register Ra.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Op2 | OL3 | Regno11 | Rt5 | Ra5 | 0Eh6 |

|  |  |  |
| --- | --- | --- |
| Op2 |  | Operation |
| 0 | CSRRD | Only read the CSR, no update takes place, Ra should be 0. |
| 1 | CSRRW | Both read and write the CSR |
| 2 | CSRRS | Read CSR then set CSR bits |
| 3 | CSRRC | Read CSR then clear CSR bits |

CSRRS and CSRRC operations are only valid on registers that support the capability.

The OL3 field is reserved to specify the operating level. Note that registers cannot be accessed by a lower operating level.

|  |  |  |  |
| --- | --- | --- | --- |
| Regno12 |  | Access | Description |
| 001 | HARTID | R | hardware thread identifier (core number) |
| 002 | TICK | R | tick count, counts every cycle from reset |
| 040 | EPC | RW | exceptioned pc, pc value at point of exception |
| 044 | STATUS | RWSC | status register, contains interrupt mask, operating level |
| 080-0BF | CODE | RW | code buffers |

Clock Cycles: 0.5

# EXEC – Execute Code Buffer

Description:

Execute code from code buffer. The N6 field specifies the code buffer to use. Code buffers allow code to be adapted at run-time. This is useful as an alternative to self-modifying code when code has to change.

Instruction Format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ~10 | N6 | ~5 | ~5 | 0Fh6 |

Clock Cycles: Minimum 0.5 – depends on the instruction in the code buffer

# IMM – Immediate Prefix

Description:

The immediate prefix instruction extends the immediate constant of the following instruction. Immediate constants up to 64 bits may be formed by using two immediate constant prefixes in succession. If using two prefixes the low order prefix should appear first in the instruction stream. The assembler will automatically emit prefix instructions where needed.

Instruction Format:

|  |  |
| --- | --- |
| Immediate[41..16] | 1Ah6 |

|  |  |  |
| --- | --- | --- |
| ~4 | Immediate[63..42] | 1Bh6 |

ADD with 64 bit constant

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Immediate[41..16] | | | | 1Ah6 |
| ~4 | Immediate[63..42] | | | 1Bh6 |
| Immed16 | | Rt5 | Ra5 | 04h6 |

Clock Cycles: 0.5

# JAL – Jump-And-Link

Description:

This instruction loads the program counter with the sum of a register and a constant value specified in the instruction. In addition the address of the instruction following the JAL is stored in the specified target register. This instruction may be used to implement subroutine calls and returns.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 18h6 |

Clock Cycles:

# LB – Load Byte

Description:

This instruction loads a byte (8 bit) value from memory. The value is sign extended to 64 bits when placed in the target register. To load an unsigned byte load a signed byte then mask off the high order bits using an AND instruction.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 13h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 13h6 | ~3 | Sc2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

# LB – Load Byte with Increment

Description:

Loads a byte (8 bit) value from memory and increments the pointer register. If the increment amount is positive it takes place after the load operation. If the amount is negative it takes place before the load operation. The value is sign extended to 64 bits when placed in the target register. The increment amount should be +1, 0, or -1.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1Ah6 | 1h5 | Ra5 | Rt5 | Ra5 | 02h6 |

Assembler:

lb r1,[r2]+

Clock Cycles: 0.5

# LH – Load Half-Word

Description:

This instruction loads a half-word (32 bit) value from memory. The memory address must be half-word aligned. The value is sign extended to 64 bits when placed in the target register.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 10h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 10h6 | ~3 | Sc2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

# LHU – Load Half-Word

Description:

This instruction loads a half-word (32 bit) value from memory. The memory address must be half-word aligned. The value is zero extended to 64 bits when placed in the target register.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 11h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 11h6 | ~3 | Sc2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

# LW – Load Word

Description:

This instruction loads a word (64 bit) value from memory. The memory address must be word aligned.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 12h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 12h6 | ~3 | Sc2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

# LW – Load Word with Increment

Description:

Loads a word (64 bit) value to memory and increments the pointer register. The increment of the pointer register takes place after the load operation. If the increment amount is positive it takes place after the load operation. If the amount is negative it takes place before the load operation.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1Ah6 | 8h5 | Ra5 | Rt5 | Ra5 | 02h6 |

Clock Cycles: 0.5

# MEMDB –Memory Data Barrier

Description:

All memory instructions before the MEMDB are completed and committed to the architectural state before memory instructions after the MEMDB are issued. This instruction is used to ensure that the memory state is valid before subsequent instructions are executed.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 346 | ~5 | ~5 | ~5 | ~5 | 02h6 |

Clock Cycles: varies depending on queue contents

# MEMSB –Memory Synchronization Barrier

Description:

This instruction is similar to the SYNC instruction except that it applies only to memory operations. All instructions before the MEMSB are completed and committed to the architectural state before memory instructions after the MEMSB are issued. This instruction is used to ensure that the memory state is valid before subsequent instructions are executed.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 356 | ~5 | ~5 | ~5 | ~5 | 02h6 |

# MUL – Signed Multiply

Description:

Multiply two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. Both the operands are treated as signed values, the result is a signed result.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 3Ah6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3A6 | ~5 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

# MULSU – Signed-Unsigned Multiply

Description:

Multiply two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. The first operand is treated as a signed value. The second operand is treated as an unsigned value. The result is a signed result.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 39h6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 396 | ~5 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

# MULU – Unsigned Multiply

Description:

Multiply two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. Both the operands are treated as unsigned values. The result is an unsigned result.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 38h6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 386 | ~5 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

# MUX – Multiplex

Description:

The MUX instruction performs a bit-by-bit copy of a bit of Rb to the target register if the corresponding bit in Ra is set, or a copy of a bit from Rc if the corresponding bit in Ra is clear.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2Ah6 | Rt5 | Rc5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

# NEG - Negate

Description:

This is an alternate mnemonic for the SUB instruction where the first register operand is R0.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 056 | ~5 | Rt5 | Rb5 | 05 | 02h6 |

Clock Cycles: 0.5

# NOP – No Operation

Description:

The NOP instruction doesn’t perform any operation. NOP’s are detected in the instruction fetch stage of the core and are not enqueued by the core. They do not occupy queue slots. Because NOPs don’t occupy queue slots they may not be used to synchronize operations between instructions.

Instruction Format:

|  |  |
| --- | --- |
| Immediate26 | 1Ch6 |

# OR – Bitwise Or

Description:

Perform a bitwise or operation between two operands. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 09h6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 096 | ~5 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

# POP – Pop Register from Stack

Description:

Loads a word (64 bit) value to memory and increments the stack pointer register. The increment of the stack pointer register takes place after the load operation. This is an alternate mnemonic for the load word with increment / decrement instruction.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1Ah6 | 8h5 | 1Fh5 | Rt5 | 1Fh5 | 02h6 |

Clock Cycles: 0.5

# PUSH – Push Register on Stack

Description:

Stores a word (64 bit) value to memory and decrements the stack pointer register. The decrement of the stack pointer register takes place before the store operation. This is an alternate mnemonic for the store word with increment / decrement instruction.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 19h6 | 18h5 | 1Fh5 | Rb5 | 1Fh5 | 02h6 |

Clock Cycles: 0.5

# RET – Return

Description:

Return from subroutine. The return address is popped from the stack and the stack pointer adjusted by the amount specified in the instruction. This amount must include eight for the return address and must be a multiple of eight.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | 1Fh5 | 1Fh5 | 18h6 |

Clock Cycles:

# REX – Redirect Exception

Description:

This instruction redirects an exception from an operating level to a lower operating level and privilege level. If the target operating level is hypervisor then the hypervisor privilege level (1) is set. If the target operating level is supervisor then one of the supervisor privilege levels must be chosen (2 to 6). This instruction if successful jumps to the target exception handler and does not return. If this instruction fails execution will continue with the next instruction.

This instruction may fail if exceptions are not enabled at the target level.

When redirecting the target privilege level is set to the bitwise ‘or’ of an immediate constant specified in the instruction and register Ra. One of these two values should be zero. The result should be a value in the range 0 to 255.

The location of the target exception handler is found in the trap vector register for that operating level (tvec[xx]).

The cause (cause) and bad address (badaddr) registers of the originating level are copied to the corresponding registers in the target level.

The REX instruction automatically enables exceptions for operating levels higher (numerically lower than) than the target level.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 24 | 23 16 | 1514 | 13 11 | 10 6 | 5 0 |
| ~8 | PL8 | ~2 | Tgt3 | Ra5 | 0Dh6 |

|  |  |
| --- | --- |
| Tgt2 |  |
| 0 | not used |
| 1 | redirect to hypervisor level |
| 2 | redirect to supervisor level |
| 3 | redirect to supervisor level |
| 4 | redirect to supervisor level |
| 5 | redirect to supervisor level |
| 6 | redirect to supervisor level |
| 7 | not used |

Clock Cycles: 3

Example:

|  |
| --- |
| REX 5,12,r0 ; redirect to supervisor handler, privilege level two  ; If the redirection failed, exceptions were likely disabled at the target level.  ; Continue processing so the target level may complete it’s operation.  RTI ; redirection failed (exceptions disabled ?) |

Notes:

Since all exceptions are initially handled at the machine level the machine level handler must check for disabled lower level exceptions.

# SB – Store Byte

Description:

This instruction stores a byte (8 bit) value to memory.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rb5 | Ra5 | 15h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 15h6 | ~3 | Sc2 | Rc5 | Rb5 | Ra5 | 02h6 |

Operation:

Memory8[Ra + immediate] = Rb

Clock Cycles: 4 minimum depending on memory access time

# SEI – Set Interrupt Mask

SEI #3

Description:

The interrupt level mask is set to the value specified by the instruction. The value used is the bitwise or of the contents of register Ra and an immediate (M3) supplied in the instruction. The assembler assumes a mask value of seven, masking all interrupts, if no mask value is specified. Usually either M3 or Ra should be zero.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 306 | ~5 | ~5 | ~2 | M3 | Ra5 | 02h6 |

Operation:

im = M3 | Ra

# SH – Store Half-Word

Description:

This instruction stores a half-word (32 bit) value to memory. The memory address must be half-word aligned.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rb5 | Ra5 | 14h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 14h6 | ~3 | Sc2 | Rc5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

# SHL – Shift Left

Description:

Bits from the source register Ra are shifted left by the amount in register Rb or an immediate value. Zeros are shifted into the least significant bits.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 206 | ~5 | Rt5 | Rb5 | Ra5 | 02h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 226 | ~4 | I1 | Rt5 | Imm5 | Ra5 | 02h6 |

Clock Cycles: 1

# SHR – Shift Right

Description:

Bits from the source register Ra are shifted right by the amount in register Rb or an immediate value. Zeros are shifted into the most significant bits.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 216 | ~5 | Rt5 | Rb5 | Ra5 | 02h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 236 | ~4 | I1 | Rt5 | Imm5 | Ra5 | 02h6 |

Clock Cycles: 1

# SUB - Subtract

Description:

Subtract two values. Both operands must be in a register.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 056 | ~5 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

# SW – Store Word

Description:

This instruction stores a word (64 bit) value to memory. The memory address must be word aligned.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rb5 | Ra5 | 16h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 16h6 | ~3 | Sc2 | Rc5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

# SW – Store Word with Decrement

Description:

Stores a word (64 bit) value to memory and increments or decrements the pointer register Ra. This instruction may be used to implement a stack push operation. The amount should be +8, 0, or -8. If the increment amount is positive it takes place after the store operation. If the amount is negative it takes place before the store operation.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 19h6 | Amt5 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

# SYNC -Synchronize

Description:

All instructions before the SYNC are completed and committed to the architectural state before instructions after the SYNC are issued. This instruction is used to ensure that the machine state is valid before subsequent instructions are executed.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 366 | ~5 | ~5 | ~5 | ~5 | 02h6 |

Clock Cycles: varies depending on queue contents

# XOR – Bitwise Exclusive Or

Description:

Perform a bitwise exclusive or operation between two operands. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 0Ah6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0A6 | ~5 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

# Opcode Tables

## Major Opcode (inst. bits 0 to 5)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | BRK | Bxx | {RR} |  | ADDI |  | CMPI | CMPUI | ANDI | ORI | XORI |  |  | REX | CSR | EXEC |
| 1x | LH | LHU | LW | LB | SH | SB | SW |  | JAL | CALL | IMM | IMM | NOP |  |  |  |
| 2x |  |  |  |  |  |  |  |  |  | RET |  |  | MODUI | MODUI | MODI |  |
| 3x |  |  |  |  |  |  |  |  | MULUI | MULSUI | MULI |  | DIVUI | DIVSUI | DIVI |  |

## Major Funct (inst. bits 26 to 31)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x |  |  | {Bitfield} |  | ADD | SUB | CMP | CMPU | AND | OR | XOR |  | NAND | NOR | XNOR |  |
| 1x | LHX | LHUX | LWX | LBX | SHX | SBX | SWX |  |  | PUSH | POP |  |  |  |  |  |
| 2x | SHL | SHR | SHLI | SHRI | ASR | ASRI |  |  | CMOVEQ | CMOVNE | MUX |  | MODU | MODSU | MOD |  |
| 3x | SEI / CLI |  | RTI |  | MEMDB | MEMSB | SYNC |  | MULU | MULSU | MUL |  | DIVU | DIVSU | DIV |  |

# Appendix

## Reducing the size of the core.

Register renaming adds considerably to the size of the core. It uses approximately 30,000 LUTs to implement register renaming. The core may be built without register renaming by setting the RENAME parameter to zero.

Architectural Register vs Physical Registers

Architectural registers are the registers visible to the programmer as part of the programming model. Physical registers are the registers physically present in the machine’s hardware. There are substantially more physical registers than there are architectural ones. For FT64 there are 32 registers visible to be programmed which are supported by 64 physical registers.

Register Renaming

The core maintains an eight entry deep history file for register rename mappings and register in use flags. The depth of the history file corresponds to the number of entries in the re-order buffer. At most a new map will be needed for each re-order buffer entry. Typically the history file is cycled through at half or less the rate of the instruction queue as approximately 50% of instructions don’t have target registers.

The core can allocate up to two registers as target registers for every pair of instructions queued. If there are no target registers available the core stalls until previous instructions have made more target registers available.

Instruction Cache Miss

During a cache miss the core streams NOP operations to the instruction fetch unit while the core is waiting for the instruction cache to load. The program counters are not incremented however, and they remain at the value when the cache miss occurred.

## Instructions Supported Only on ALU #0

The following less frequently used instructions are only supported on ALU #0 in order to reduce the size of the core.

* + division and remainder instructions (DIV,DIVSU,DIVU,MOD,MODSU, MODU)
  + bit-field instructions (BFCLR, BFSET, BFCHG, BFINS, BFINSI, BFEXT, BFEXTU)
    - these are rarely used instructions
  + shift instructions (ASR, SHL, SHR)
    - The shift instructions use barrel shifters to shift by any amount in a single clock cycle and so are relatively resource expensive compared to how often they are used.
  + indexed memory loads / stores (LBX, LHX, LHUX, LWX, SBX, SHX, SWX)
    - since indexed memory instructions are infrequently used they are supported only on alu #0.
  + CSR instruction
    - CSR instructions are rarely used. They often also have synchronization issues as there is no bypassing for the CSR registers. Since they typically require synchronization operations there is no benefit to having multiple CSR instructions executing at the same time.