# FT64v3 ISA

## Overview

The FT64v3 ISA is in part organized around the idea of a simple compiler at the expense of some hardware cost. The ISA makes use of a unified register file for integer and floating-point registers. This makes it a bit easier to manage register usage by the compiler. Having a large number of registers directly available means the compiler doesn’t have to be as sophisticated in it’s allocation of registers and good performance is still possible. However, increasing the number of bits required to represent registers leads to an increase in the size of instructions.

This instruction set uses a fixed 36-bit format instruction as its base because 32 bits isn’t quite enough. 36 bits may seem like an odd number, but the number of bits in an instruction is somewhat irrelevant as long as all instructions can fit on a cache line. Instruction caches are loaded in terms of cache lines. Sixteen 36-bit instructions will fit into a 576-bit cache line with no wasted space. 576 bits is 72 bytes, in range with typical cache line sizes. It is also a multiple of 64 bits, the width of the data bus.

## Design Considerations

### Compression

A number of contemporary processors make use of compressed instructions sets to improve code density as a way to compete with other designs using byte codes. Narrower instructions make better use of memory and caches. They have benefits in the form of performance and power consumption. They are so important to contemporary designs that RiSC-V for instance reserves three quarters of the opcode space for compressed instructions. Other processors have mode switching instructions to enable compressed instruction sets. This design directly supports compressed instructions; one half of the instruction space is allocated for compressed instructions. Half of the compressed instruction space is reserved for memory operations, the other half for other operations.

Typically, a compressed version of the instruction set relates to the uncompressed version in terms of registers used and field specifications. The register and field specifications map more or less directly to fields in the expanded instruction. The mapping is designed to take a minimal amount of logic to convert into expanded instructions so that the decompression doesn’t significantly impact the cycle time of the processor. A subset of registers may be supported along with a subset of operations that the processor provides.

Significant analysis has gone into which instructions should be available in a compressed form. It is often the same subset of instructions that are best compressed. These instructions typically include an add of a small amount to a register, and frame pointer or stack pointer relative loading / store instructions.

For this design, rather than develop a format for and completely specify fields for compressed instructions, compressed instructions are simply expanded from look-up tables. An application profiler may be used to select the set of instructions to compress. The compressed instruction set in use is definable at run-time, similar to a micro-code update capability. Using lookup tables may reduce the number of compressed instructions available, but the best selection for a given app(s) can be chosen. Decompressing instructions becomes a simple indexed table lookup rather than logic.

A parcel size one half of the chosen base instruction size or 18-bit parcels offer good compression. Compressed instructions use a 8k entry lookup table to decompress the instructions. It is envisioned that decompressing these instructions may require an extra clock cycle for access to the lookup table. Often the instruction will be able to queue in the pipeline without knowing the exact details of the instruction. Looking up the instruction is similar to having to fetch values from the register file. The exact opcode can be filled in a cycle later after lookup takes place.

A consequence of the delay in decompressing an instruction is that a compressed branch type instruction may have an additional clock cycle of delay causing a reduction in performance.

# Instruction Set Description

## Overview

This instruction set uses a fixed 36 bit format instruction as its base. 36 bits may seem like an odd number, but the number of bits in an instruction is somewhat irrelevant as long as all instructions can fit on a cache line. Instruction caches are loaded in terms of cache lines. Sixteen 36-bit instructions will fit into a 576-bit (72 byte) cache line with no wasted space and it’s a multiple of 64 bits. Part of the reason for a 36-bit base instruction is that 32 bits isn’t quite enough. This is brought about because of the desire for a base of 64 available general-purpose registers in the instruction set and the desire to support a full instruction set. 64 registers were chosen because it makes the compiler simpler to implement. There is no distinguishing between integer and floating-point register sets.

## Instruction Addresses

Instructions are addressed in 36-bit parcels as if they were composed of two eighteen-bit parcels. An 18-bit parcel size is to allow for the possibility of a compressed instruction set. As a consequence, instruction addresses and data addresses are not the same. The physical (data) address of the instruction can easily be calculated as 18/8 or 9/4 of the instruction address. That is (Phys Addr = Instr. (((Add << 3)+ Instr. Add) >> 2). Example: instruction address 16 = ((16 << 3) + 16) >> 2) = data address 36. The address translation is important only when loading cache lines and can be done very quickly using just a single addition and shifting.

## Formats

Instructions have a fixed 36 bit format. There are only a handful of different instruction formats. The opcode, register read Ra, Rb, and Rc fields always occur in the same place in an instruction to simplify decoding and keep the register read address which is needed prior to enqueue at a fixed decoding location. The Rt field is allowed to float around to make the instruction encoding easier. In a pipelined processor there is usually at least one clock cycle before Rt is used meaning it has time to be shifted around before it’s use.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Fields | | | | | | | | | | | |  |
|  | Immed16 | | | | | | | | | Rt6 | Ra6 | Opcode8 | RI |
|  | Funct6 | | | | ~ | Sz3 | Rt6 | | | Rb6 | Ra6 | Opcode8 | R2 |
|  | 016 | | | | ~ | Sz3 | Funct6 | | | Rt6 | Ra6 | Opcode8 | R1 |
|  | Funct6 | | | | I | Sz3 | Rt6 | | | Rb6 | Ra6 | Opcode8 | SR |
|  | Funct6 | | | | I | Sz3 | Immed6 | | | Rt6 | Ra6 | Opcode8 | SI |
|  | Funct4 | | Me6 | | | | Mb6 | | | Rt6 | Ra6 | Opcode8 | BF |
|  | Disp11 | | | | | | | P2 | Cn3 | Rb6 | Ra6 | Opcode8 | BD |
|  | Disp11 | | | | | | | P2 | Cn3 | Bitno6 | Ra6 | Opcode8 | BB |
|  | Disp11 | | | | | | | P2 | Immed9 | | Ra6 | Opcode8 | BI |
|  | ~4 | | P2 | | Cnd4 | | Rc6 | | | Rb6 | Ra6 | Opcode8 | BR |
|  | Funct5 | | | ar2 | | Sc3 | Rt5/Rc6 | | | Rb6 | Ra6 | Opcode8 | MX |
|  | Op2 | OL2 | Regno12 | | | | | | | Rt6 | Ra6 | Opcode8 | CS |
|  | Address28 | | | | | | | | | | | Opcode8 | JC |
|  | Funct5 | | | P2 | | Rm3 | Rt6 | | | Rb6 | Ra6 | Opcode8 | FP |

There are a handful of additional formats primarily for control type instructions. See the particular instruction for the exact format used and additional information.

|  |  |
| --- | --- |
| Format | Instruction Group |
| RI | register-immediate and load / store with displacement |
| RR | register-register, two source registers |
| R1 | single source register |
| SR | shift register-register |
| SI | shift register-immediate |
| BF | bitfield |
| BD | branch with displacement |
| BB | branch on bit set / clear, decrement and branch |
| BI | branch equal immediate |
| BR | branch to register |
| MX | memory indexed |
| CS | control and status register access |
| JC | jump and call |
| FP | floating-point |

There are quite a few instructions operating on memory. Once volatile and non-volatile loads, stores, read-modify-write, and different sizes of operations are taken into consideration there are about 50 memory ops.

A single bit (bit 6 of the opcode) in the instruction determines if the instruction is a memory instruction or some other type of instruction. Memory instructions are further broken down into three groups – loads, stores, and read-modify-write instructions. Also, easily discernible by looking at the next two bits (bits 5 and 4) of the opcode.

## Compressed Instruction Formats

Suggested Compressed Instruction Formats

The branch instruction formats are fixed as they need to be decoded by the fetch stage of the processor.

The only other thing at fixed positions in the compressed instruction format is the register read numbers. The register numbers for read are required as the processor needs them in order to queue the instruction. Any variety of operations may be chosen with the given register read numbers. Reading three ports in the same instruction is not supported with compressed instructions. This excludes indexed store operations, and floating-point multiply and add instructions from being compressed.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | |  | |  |  | |  |
| Format: | 00?? | | ?????? | | 10 | ?????? | | ANY |
| Suggested Op’s | 0000 | | 006 | | 10 | 06 | | NOP |
|  | 0000 | | Amt[8..3] | | 10 | 316 | | ADDISP |
|  | 0000 | | Amt[5..0] | | 10 | Rt6 | | ADDI |
|  | 0001 | | Amt[5..0] | | 10 | Rt6 | | LDI |
|  | 0010 | | ?????? | | 10 | ?????? | | reserved |
|  | 0011 | | Amt[5..0] | | 10 | Rt6 | | SHLI |
|  | 0100 | | Amt[5..0] | | 10 | 00 | Rt’4 | SHRI |
|  | 0100 | | Amt[5..0] | | 10 | 01 | Rt’4 | ASRI |
| Format: | 0100 | | ?? | ???? | 10 | 10 | Rb’3..0 | 1 Partial Read Port |
| Suggested Op’s | 0100 | | 00 | Ra/Rt’4 | 10 | 10 | Rb’3..0 | SUB |
|  | 0100 | | 01 | Ra/Rt’4 | 10 | 10 | Rb’3..0 | AND |
|  | 0100 | | 10 | Ra/Rt’4 | 10 | 10 | Rb’3..0 | OR |
|  | 0100 | | 11 | Ra/Rt’4 | 10 | 10 | Rb’3..0 | XOR |
|  | 0101 | |  | |  |  | | reserved |
|  | 0110 | |  | |  |  | | reserved |
|  | 0111 | | Disp[7..2] | | 10 | D2 | Ra’3..0 | BRA |
|  | 10 | Disp8 | | | 10 | 11 | Ra’3..0 | BEQZ |
|  | 11 | Disp8 | | | 10 | 11 | Ra’3..0 | BNEZ |
|  |  | |  | |  |  | |  |
| Format: | 00?? | | ?????? | | 11 | Ra5..0 | | 1 Full Read Port |
| Suggested Op’s | 0000 | | Rt5..0 | | 11 | Ra5..0 | | MOV |
|  | 0001 | | Rt5..0 | | 11 | Ra5..0 | | ADD |
|  | 0010 | | Rt5..0 | | 11 | Ra5..0 | | JALR |
|  | 0011 | | ?????? | | 11 | Ra5..0 | | reserved |
|  | The following two instructions have SP as an implied register read | | | | | | | |
| Format: | 010? | | ?????? | | 11 | Rt5..0 | | SP implied |
|  | 0100 | | ?????? | | 11 | Rt5..0 | | [SP] implied |
|  | 0101 | | ?????? | | 11 | Rt5..0 | | [SP] implied |
| Example: | 0100 | | Disp8..3 | | 11 | Rt5..0 | | LW Rt,d[SP] |
|  | The following two instructions have FP as an implied register read | | | | | | | |
| Format: | 011? | | ?????? | | 11 | Rt5..0 | | FP implied |
|  | 0110 | | ?????? | | 11 | Rt5..0 | | [FP] implied |
|  | 0111 | | ?????? | | 11 | Rt5..0 | | [FP] implied |
| Example: | 0110 | | Disp8..3 | | 11 | Rt5..0 | | LW Rt,d[FP] |
|  | The following two instructions have SP as an implied register read | | | | | | | |
| Format: | 100? | | ?????? | | 11 | Rb5..0 | | SP implied, 2nd Read port |
|  | 1000 | | ?????? | | 11 | Rb5..0 | | Rb,d[SP] |
|  | 1001 | | ?????? | | 11 | Rb5..0 | | Rb,d[SP] |
| Example: | 1000 | | Disp8..3 | | 11 | Rb5..0 | | SW Rb,d[SP] |
|  | The following two instructions have FP as an implied register read | | | | | | | |
| Format: | 101? | | ?????? | | 11 | Rb5..0 | | FP implied, 2nd read port |
|  | 1010 | | ?????? | | 11 | Rb5..0 | | Rb,d[FP] |
|  | 1011 | | ?????? | | 11 | Rb5..0 | | Rb,d[FP] |
|  |  | |  | |  |  | |  |
| Format: | 110? | | ?????? | | 11 | ?? | Ra’3..0 | 1 read port |
| Example: | 1100 | | d6..5 | Rt’3..0 | 11 | d4..3 | Ra’3..0 | LW Rt,d[Ra] |
|  | 1101 | | ?????? | | 11 | ?? | Ra’3..0 |  |
| Format: | 111? | | ?? | Rb’3..0 | 11 | ?? | Ra’3..0 |  |
|  | 1111 | | ?? | Rb’3..0 | 11 | ?? | Ra’3..0 |  |
|  |  | |  | |  |  | |  |

It is assumed that registers r32 to r63 will be used primarily for floating point and registers r0 to r31 for integer values.

There is no specification of the Rt register. The value for RT is not required at instruction queue time. How Rt is encoded is left to the compressor.

## Major Opcode (inst. bits 0 to 6)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x |  | {VECTOR} | {R2} |  | ADDI | {Bitfield} | CMPI | CMPUI | ANDI | ORI | XORI | QOPI |  | {FLOAT} | CSR | FMA |
| 1x |  |  |  |  |  |  |  |  | MULI | MULUI | MULSUI | FNMS | DIVI | DIVUI | DIVSUI | FMS |
| 2x | SEQI | SNEI | SLTI | SGEI | SLEI | SGTI | SLTUI | SGEUI | SLEUI | SGTUI |  |  | MODI | MODUI | MODSUI | FNMA |
| 3x | JMP | CALL | RET | JAL | SYS | REX |  |  | Bcc | BccR |  |  | BBc | BEQ# | FBcc | FBccR |
| 4x | LB | LBU | LBO | LC | LCU | LCO | LH | LHU | LHO | LW | LWU | LWO | LQ | {Indexed Load} | LV |  |
| 5x | LVB | LVBU | LVBO | LVC | LVCU | LVCO | LVH | LVHU | LVHO | LVW | LVWU | LVWO | LVQ | LVWR | LVV |  |
| 6x | SB | SC | SH | SW | SQ | SWC | SV | {Indexed} |  |  |  |  |  |  |  |  |
| 7x | ASWAP | AADD | AAND | AOR | AXOR | AMIN | AMAX | AMINU | AMAXU | ASHL | ASHR | INC |  |  |  |  |

## {R2} Major Func (inst. bits 30 to 35)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | {BCD} | {R1} |  |  | ADD | SUB | CMP | CMPU | AND | OR | XOR |  | NAND | NOR | XNOR |  |
| 1x | SHL | ASL | SHR | ASR | ROL | ROR |  |  | MUL | MULU | MULSU |  | DIV | DIVU | DIVSU |  |
| 2x | SEQ | SNE | SLT | SGE | SLE | SGT | SLTU | SGEU | SLEU | SGTU |  |  | MOD | MODU | MODSU |  |
| 3x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

## {Indexed Load} Major Func (inst. bits 31 to 35)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | LBX | LBUX | LBOX | LCX | LCUX | LCOX | LHX | LHUX | LHOX | LWX | LWUX | LWOX | LQX |  | LVX |  |
| 1x | LVBX | LVBUX | LVBOX | LVCX | LVCUX | LVCOX | LVHX | LVHUX | LVHOX | LVWX | LVWUX | LVWOX | LVQX | LVWRX | LVVX |  |

## {Indexed} Major Func (inst. bits 31 to 35)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | s8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | SBX | SCX | SHX | SWX | SQX | SWCX | SVX |  |  |  |  |  |  |  |  |  |
| 1x | ASWAPX | AADDX | AANDX | AORX | AXORX | AMINX | AMAXX | AMINUX | AMAXUX | ASHLX | ASHRX | INCX |  |  |  |  |