Frost64 CPU

Andrew Clark

May 8, 2020

1 Table of Contents

Contents

2 Registers, Main Widths, etc.

- The main width of the processor, hinted at with the name, is 64-bit. Addresses are 64-bit, and some instructions only support 64-bit operations.
- Instructions LARs (ILARs)
  - There are 256 total ILARs, with 128 of them used for interrupt service routines and 128 of them used for user code.
  - The two ILARs named `i0`, which are ILAR number 0 and ILAR number 128 within the ILAR file, have their fields always set to zero. They cannot be changed.
  - ILARs are 256 bytes long, and instructions are 32-bit. This means that one ILAR holds 256 / 4 = 64 instructions.
- Data LARs (DLARs)
  - There are 256 total DLARs, with 128 used for interrupt service routines and 128 of them used for user code.
  - The two DLARs named `dzero`, which are DLAR number 0 and DLAR number 128 within the DLAR file, have their fields always set to zero. They cannot be changed.
  - The user-mode DLAR named `dira`, which is DLAR 1 within the DLAR file, is treated specially, as its address field is the interrupt return address for the `reti` instruction.
  - DLARs are 256 bytes long.
  - DLARs can take on the following type tags (3-bit enum):
    - 8-bit, unsigned
    - 8-bit, signed
    - 16-bit, unsigned
    - 16-bit, signed
    - 32-bit, unsigned
    - 32-bit, signed
    - 64-bit, unsigned
    - 64-bit, signed
  - The base address of a DLAR is 64 - 8 = 56 bits long.
  - The scalar offset of a DLAR is 8 bits long.
- Program Counter (PC)
  - The program counter consists of an ILAR number (7-bit) and an offset into said ILAR (6-bit).
  - Two program counters exist: one for when servicing an interrupt, and one for when not servicing an interrupt (user mode).

- Interrupt Enable (`ie`)
  - This 1-bit register indicates whether or not an interrupt can be serviced.

3 Instructions (CPU's perspective)

3.1 Group 0 Instructions

- Encoding: `000v ooof fffa aaaa aabb bbbb bccc cccc`
  - `v`:
    - when `0`: scalar operation
    - when `1`: vector operation
  - `o`: opcode
  - `f`: functional unit number f
  - `a`: DLAR a
  - `b`: DLAR b
  - `c`: DLAR c
- Instruction List:
  1. `add fF, dA, dB, dC`
  2. `sub fF, dA, dB, dC`
  3. `slt fF, dA, dB, dC`
  4. `mul fF, dA, dB, dC`
  5. `div fF, dA, dB, dC`
     - Note: Perform an unsigned divide if `dB` is unsigned, but perform a signed divide if `dB` is signed.
  6. `and fF, dA, dB, dC`
  7. `orr fF, dA, dB, dC`
  8. `xor fF, dA, dB, dC`

3.2 Group 1 Instructions

- Encoding: `001v ooof fffa aaaa aabb bbbb bccc cccc`
  - `v`:
    - when `0`: scalar operation
    - when `1`: vector operation
  - `o`: opcode
  - `f`: functional unit number f

  - `a`: DLAR a
  - `b`: DLAR b
  - `c`: DLAR c
- Instruction List:
  1. `shl fF, dA, dB, dC`
     - Shift left
  2. `shr fF, dA, dB, dC`
     - Shift right
     - Note: Perform a logical right shift if `dB` is unsigned, but perform an arithmetic right shift if `dB` is signed.
  3. Reserved
  4. Reserved
  5. Reserved
  6. Reserved
  7. Reserved
  8. Reserved

3.3 Group 2 Instructions

- Encoding: `010o oooa aaaa aabb bbbb biii iiii iiii`
  - `o`: opcode
  - `a`: DLAR a
  - `b`: DLAR b
  - `i`: Signed 11-bit immediate
- Notes:
  - These instructions compute the address to load from or store to via the following formula: `u64(dB.scalar_data) + s64(simm11)`
- Instruction List:
  1. `ldiu8 dA, [dB, simm11]`
  2. `ldis8 dA, [dB, simm11]`
  3. `ldiu16 dA, [dB, simm11]`
  4. `ldis16 dA, [dB, simm11]`
  5. `ldiu32 dA, [dB, simm11]`
  6. `ldis32 dA, [dB, simm11]`
  7. `ldiu64 dA, [dB, simm11]`

8. `ldis64 dA, [dB, simm11]`
9. `stiu8 dA, [dB, simm11]`
10. `stis8 dA, [dB, simm11]`
11. `stiu16 dA, [dB, simm11]`
12. `stis16 dA, [dB, simm11]`
13. `stiu32 dA, [dB, simm11]`
14. `stis32 dA, [dB, simm11]`
15. `stiu64 dA, [dB, simm11]`
16. `stis64 dA, [dB, simm11]`

3.4 Group 3 Instructions

- Encoding: `011o oooa aaaa aabb bbbb bccc cccc iiii`
  - `o`: opcode
  - `a`: DLAR a
  - `b`: DLAR b
  - `b`: DLAR c
  - `i`: Signed 4-bit immediate
- Notes:
  - These instructions compute the address to load from or store to via the following formula: `u64(dB.scalar_data) + dC.addr + s64(simm4)`
- Instruction List:
  1. `ldu8 dA, [dB, dC, simm4]`
  2. `lds8 dA, [dB, dC, simm4]`
  3. `ldu16 dA, [dB, dC, simm4]`
  4. `lds16 dA, [dB, dC, simm4]`
  5. `ldu32 dA, [dB, dC, simm4]`
  6. `lds32 dA, [dB, dC, simm4]`
  7. `ldu64 dA, [dB, dC, simm4]`
  8. `lds64 dA, [dB, dC, simm4]`
  9. `stu8 dA, [dB, dC, simm4]`
  10. `sts8 dA, [dB, dC, simm4]`
  11. `stu16 dA, [dB, dC, simm4]`
  12. `sts16 dA, [dB, dC, simm4]`
  13. `stu32 dA, [dB, dC, simm4]`
  14. `sts32 dA, [dB, dC, simm4]`
  15. `stu64 dA, [dB, dC, simm4]`
  16. `sts64 dA, [dB, dC, simm4]`

3.5 Group 4 Instructions

- Encoding: `100a aaaa aabb bbbb bccc cccc iiii jjjj`
  - `a`: ILAR a (base ILAR)
  - `b`: DLAR b
  - `c`: ILAR c
  - `i`: Signed 4-bit immediate
  - `j`: Unsigned 4-bit immediate. How many additional, consecutive ILARs, starting with `iA`, will be `fetch`ed into.
    - Note: The destination ILAR number will wrap around to `i0` if incrementing from `i128` to the next ILAR, i.e. a 7-bit counter is used.
- Notes:
  - The one instruction in this group of instructions, `fetch`, computes the starting address, via the following equation: `u64(dB.data) + iC.addr + s64(simm4)`
- Instruction List:
  - `fetch iA, dB, iC, simm4`

3.6 Group 5 Instructions

- Encoding: `101o oooa aaaa aabb bbbb biii iii0 0000`
  - `o`: opcode
  - `a`: DLAR a (mode-specific)
  - `b`: LAR b (instruction-specific, either a DLAR or an ILAR)
  - `i`: 6-bit immediate, `imm6`
- Instruction List:
  1. `gudata dA, dB, imm6`
     - Name: ”get user-mode data”
     - For this instruction, the destination, mode-specific DLARs are treated as vectors of 128-bit data, where each vector element stores the following:
       - The full address (64-bit) of user-mode DLARs (starting with `dB`), stored in vector element bit range `63..0`.
       - The type tag (3-bit) of user-mode DLARs (starting with `dB`), stored in vector element bit range `66..64`
     - Each 128-bit vector element corresponds to one user-mode DLAR’s contents.

- This instruction grabs the full address and type tag of user-mode DLARs, starting with `dB`, and including the following (consecutive) `imm6` user-mode DLARs, storing the 128-bit vector elements in (mode-specific) DLARs, starting with `dA` as the base DLAR. The base mode-specific DLAR is followed by consecutive DLARs if needed to store the user-mode DLAR fields.
- If the last mode-specific DLAR is not completely filled with user-mode DLAR contents, its remaining 128-bit vector elements will be filled with `0x0`.

2. `rudata dA, dB, imm6`
   - Name: ”restore user-mode data”
   - This instruction sort of does the reverse of the `gudata` instruction, performing the equivalent of load instructions, loading into user-mode DLARs based upon the metadata stored in mode-specific DLARs. As such, this instruction can read from memory, just like load instructions.
3. `guinst dA, iB, imm6`
   - Name: ”get user-mode instructions”
   - For this instruction, the destination, mode-specific DLARs are treated as vectors of 64-bit data, where each vector element stores the address field of a single user-mode ILAR.
   - Each 64-bit vector element corresponds to one user-mode ILAR’s address.
   - This instruction grabs the full address of user-mode ILARs, starting with `dB` and including the following (consecutive) `imm6` user-mode ILARs, storing the 64-bit vector elements in mode-specific DLARs, starting with `dA` as the base DLAR. The base mode-specific DLAR is followed by consecutive DLARs if needed to store the user-mode ILAR addresses.
   - If the last mode-specific DLAR is not completely filled with user-mode ILAR addresses, its remaining 64-bit vector elements will be filled with `0x0`.
4. `ruinst dA, iB, imm6`
   - Name: ”restore user-mode instructions”
   - This instruction performs `imm6 + 1` consecutive `fetch` operations, starting with (mode-specific) `dA` as the base register that is the source of addresses to `fetch` instructions from. User-mode `iB` is the first ILAR that instructions will be `fetch`ed into.
   - Basically, this instruction takes the results from a `guinst` instruction and re-`fetch`es the instructions back into user-mode ILARs.
5. `gupc dA`
   - Name: ”get user-mode pc”

- This instruction grabs the user-mode program counter value, encoded into a scalar value as
  - `concat(ilar_number, ilar_scalar_offset)`,

  and stores that value into DLAR `dA`.
- Note: the type tag of mode-specific DLAR `dA` is respected during this operation.
- Note: This instruction is useful for implementing user-mode jump tables that are (at least partially) stored in a bunch of ILARs.

6. `rupc dA`
   - Name: ”restore user-mode pc”
   - This instruction simply does the reverse operation of the `gupc` instruction.
7. `gidata dA, dB, imm6`
   - Name: ”get interrupts-mode data”
   - For this instruction, the destination, mode-specific DLARs are treated as vectors of 128-bit data, where each vector element stores the following:
     - The full address (64-bit) of interrupts-mode DLARs (starting with `dB`), stored in vector element bit range `63..0`.
     - The type tag (3-bit) of interrupts-mode DLARs (starting with `dB`), stored in vector element bit range `66..64`
   - Each 128-bit vector element corresponds to one interrupts-mode DLAR’s contents.
   - This instruction grabs the full address and type tag of interrupts-mode DLARs, starting with `dB`, and including the following (consecutive) `imm6` interrupts-mode DLARs, storing the 128-bit vector elements in (mode-specific) DLARs, starting with `dA` as the base DLAR. The base mode-specific DLAR is followed by consecutive DLARs if needed to store the interrupts-mode DLAR fields.
   - If the last mode-specific DLAR is not completely filled with interrupts-mode DLAR contents, its remaining 128-bit vector elements will be filled with `0x0`.
8. `ridata dA, dB, imm6`
   - Name: ”restore interrupts-mode data”
   - This instruction sort of does the reverse of the `gidata` instruction, performing the equivalent of load instructions, loading into interrupts-mode DLARs based upon the metadata stored in mode-specific DLARs. As such, this instruction can read from memory, just like load instructions.
9. `giinst dA, iB, imm6`
   - Name: ”get interrupts-mode instructions”

- For this instruction, the destination, mode-specific DLARs are treated as vectors of 64-bit data, where each vector element stores the address field of a single interrupts-mode ILAR.
- Each 64-bit vector element corresponds to one interrupts-mode ILAR's address.
- This instruction grabs the full address of interrupts-mode ILARs, starting with `dB` and including the following (consecutive) `imm6` interrupts-mode ILARs, storing the 64-bit vector elements in mode-specific DLARs, starting with `dA` as the base DLAR. The base mode-specific DLAR is followed by consecutive DLARs if needed to store the interrupts-mode ILAR addresses.
- If the last mode-specific DLAR is not completely filled with interrupts-mode ILAR addresses, its remaining 64-bit vector elements will be filled with `0x0`.

10. `riinst dA, iB, imm6`
    - Name: "restore interrupts-mode instructions"
    - This instruction performs `imm6 + 1` consecutive `fetch` operations, starting with (mode-specific) `dA` as the base register that is the source of addresses to `fetch` instructions from. User-mode `iB` is the first ILAR that instructions will be `fetch`ed into.
    - Basically, this instruction takes the results from a `giinst` instruction and re-`fetch`es the instructions back into interrupts-mode ILARs.
11. `gipc dA`
    - Name: "get interrupts-mode pc"
    - This instruction grabs the interrupts-mode program counter value, encoded into a scalar value as
      * `concat(ilar_number, ilar_scalar_offset)`,

      and stores that value into DLAR `dA`.
    - Note: the type tag of mode-specific DLAR `dA` is respected during this operation.
    - Note: This instruction is useful for implementing interrupts-mode jump tables that are (at least partially) stored in a bunch of ILARs.
12. `ripc dA`
    - Name: "restore interrupts-mode pc"
    - This instruction simply does the reverse operation of the `gipc` instruction.
13. `ei`
    - This instruction sets the interrupt enable register, `ie`, to `0b1`, which enables interrupts.
14. `di`
    - This instruction sets the interrupt enable register, `ie`, to `0b0`, which disables interrupts.

15. `reti dA`
    - Name: ”return from interrupt”
    - This instruction does the following:
        * Sets `ie` to `0b1`, enabling interrupts.
        * Switches to user-mode from interrupts-mode.
        * Sets the user-mode `pc` to the scalar data of user-mode DLAR `dA`
16. Reserved for future expansion

3.7 Group 6 Instructions

- Encoding: `110o ooaa aaaa abbb bbbb iiii iijj jjjj`
    - `o`: opcode
    - `a`: DLAR a, `dA`
    - `b`: ILAR b (`iB`), or DLAR b (`dB`)
    - `i`: `i_imm6`, field `i` 6-bit unsigned immediate
    - `j`: `j_imm6`, field `j` 6-bit unsigned immediate
- Instruction List:
- `jzo.s dA, iB, i_imm6`
    - Name: ”jump if zero, scalar”
    - If `dA`’s scalar data is zero, this instruction jumps to the instruction in ILAR `iB` at index `i_imm6`.
- `jzo.v dA, iB, i_imm6`
    - Name: ”jump if zero, vector”
    - If `dA`’s entire (vector) data is zero, this instruction jumps to the instruction in ILAR `iB` at index `i_imm6`.
- `jnz.s dA, iB, i_imm6`
    - Name: ”jump if non-zero, scalar”
    - If `dA`’s scalar data is non-zero, this instruction jumps to the instruction in ILAR `iB` at index `i_imm6`.
- `jnz.v dA, iB, i_imm6`
    - Name: ”jump if non-zero, vector”
    - If `dA`’s entire (vector) data is non-zero, this instruction jumps to the instruction in ILAR `iB` at index `i_imm6`.
- `sel.s dA, iB, i_imm6, j_imm6`

  - Name: ”select, scalar”
  - If `dA`’s scalar data is non-zero, jump to the instruction at ILAR `iB`, offset `i_imm6`. Otherwise, if `dA`’s scalar data is zero, jump to the instruction at ILAR `iB`, offset `j_imm6`.

- `sel.v dA, iB, i_imm6, j_imm6`
  - Name: ”select, vector”
  - If `dA`’s entire (vector) data is non-zero, jump to the instruction at ILAR `iB`, offset `i_imm6`. Otherwise, if `dA`’s entire (vector) data is zero, jump to the instruction at ILAR `iB`, offset `j_imm6`.

- Reserved for future expansion.

- Reserved for future expansion.