

Volt32 CPU

Andrew Clark

June 26, 2022

Table of Contents

Table of Contents	1
Registers, Main Widths, etc.	2
Exceptions	5
Instructions	6
4.1 Group 0 Instructions	6
4.2 Group 1 Instructions	9
4.3 Group 2 Instructions	12
4.4 Group 3 Instructions	13
4.5 Group 4 Instructions	14
4.6 Group 5 Instructions	15
4.7 Group 6 Instructions	16
4.8 Group 7 Instructions	18
4.9 Group 8 Instructions	20
4.10 Group 9 Instructions	21

Registers, Main Widths, etc.

- The main width of the processor is 32-bit, and addresses are 32-bit. Some 64-bit operations exist.
- The machine is an implementation of Line Associative Registers (LARs). Both instruction LARs (ILARs) and data LARs (DLARs) are included in the design. There are a grand total of 128 ILARs and 128 DLARs, but they are split between the LARs owned by the supervisor mode and the LARs owned by the user mode. There are 64 supervisor mode ILARs, 64 supervisor mode DLARs, 64 user mode ILARs, and 64 user mode DLARs.
- The machine boots in supervisor mode. The processor jumps to address `0x0` when it enters supervisor mode, which includes when the machine boots.
- ILARs
 - In user mode, ILARs 0 to 63 are referred to as `i0`, `i1`, `i2`, ..., `i61`, `i62`, `ipc`.
 - In supervisor mode, ILARs 64 to 127 are referred to as `i0`, `i1`, `i2`, ..., `i61`, `i62`, `ipc`. Note that supervisor mode ILARs are encoded into instructions without the most significant bit, i.e. supervisor mode `i0` is encoded as `0b000000`.
 - The two ILARs called "`i0`" have all their fields set to zero, and when written to, the contents of the ILAR does not change.
 - The two ILARs called "`ipc`" are the program counters for the two operating modes of the processor. The location within the `ipc` ILARs can be computed by taking the low 6 bits (6 because 128 bytes long ILARs) of their addresses.
 - An ILAR's data field is 128 bytes long. It is composed of 32-bit instructions aligned to 32 bits.
 - An ILAR's scalar offset field is $(7 - 2 = 5)$ -bit due to instructions being 32-bit and the data field being 128 bytes long.
 - The base address field of an ILAR is $(32 - 6 = 26)$ -bit.
 - An ILAR's tag field is 7-bit because there are 128 total ILARs.
- DLARs
 - In user mode, DLARs 0 to 63 are referred to as `d0`, `d1`, `d2`, ..., `d59`, `d60`, `dcp`, `dfp`, `dsp`.
 - In supervisor mode, DLARs 64 to 127 are referred to as `d0`, `d1`, `d2`, ..., `d59`, `d60`, `dcp`, `dfp`, `dsp`. Note that supervisor mode DLARs are encoded into instructions without the most significant bit, i.e. supervisor mode `d0` is encoded as `0b000000`.
 - The two DLARs called "`d0`" have all their fields set to zero, and when written to, the contents of the DLAR does not change.

- A DLAR's data field is 128 bytes long. It is composed of the scalar data elements of the 128 byte vectors, where the type of the scalar data elements is determined by the type tag field of the DLAR.
 - A DLAR's scalar offset field is 7-bit due to the data field being 128 bytes long.
 - The base address field of a DLAR is $(32 - 6 = 26)$ -bit.
 - DLARs can take on the following types (3-bit enum):
 - * 8-bit, unsigned (`u8`)
 - * 8-bit, signed (`s8`)
 - * 16-bit, unsigned (`u16`)
 - * 16-bit, signed (`s16`)
 - * 32-bit, unsigned (`u32`)
 - * 32-bit, signed (`s32`)
 - * 64-bit, unsigned (`u64`); only usable for some operations
 - * 64-bit, signed (`s64`); only usable for some operations
 - A DLAR's tag field is 7-bit because there are 128 total DLARs.
 - Similarly, a DLAR's reference count field is 7-bit because there are 128 total DLARs.
 - A DLAR's dirty field is 1-bit.
- The `ie` register
 - "ie" is short for "interrupt enable".
 - This register is 1-bit.
 - This register is a flag indicating whether or not an interrupt can be serviced. It can be read from/written to using `cpy` instructions.
 - The `reti` instruction sets `ie` to `0b1` and returns to user mode from supervisor mode.
 - The `xct` register
 - "exception type"
 - This register is 32-bit.
 - This register is set to the numerical value of an exception's type upon the machine entering supervisor mode. It can be read from/written to using `cpy` instructions.
 - The `swiarg0` register
 - This register is 128 bytes long.
 - This register indicates argument 0 to `swi`. In supervisor mode, it can be read from/written to with `cpy` instructions.

- The `swiarg1` register
 - This register is 128 bytes long.
 - This register indicates argument 1 to `swi`. In supervisor mode, it can be read from/written to with `cpy` instructions.
- The `swiarg2` register
 - This register is 128 bytes long.
 - This register indicates argument 2 to `swi`. In supervisor mode, it can be read from/written to with `cpy` instructions.
- The `swiarg3` register
 - This register is 128 bytes long.
 - This register indicates argument 3 to `swi`. In supervisor mode, it can be read from/written to with `cpy` instructions.

Exceptions

Some instructions may cause an exception to occur, putting the processor in supervisor mode.

The following exceptions may occur during normal execution of a program. `xct` is set to a numerical value representing these upon the processor encountering an exception.

- When `xct == 0x0`: Taking a non-software interrupt (which would also set `ie` to `0b0`).
- When `xct == 0x1`: Division by zero.
- When `xct == 0x2`: Undefined instruction.
- When `xct == 0x3`: Instructions where 64-bit ops are not defined.
- When `xct == 0x4`: `swi`.
- When `xct == 0x5`: `reti` when in user mode.
- When `xct == 0x6`: `retx` when in user mode.
- When `xct == 0x7`: `cpy` that reads from `ie` in user mode.
- When `xct == 0x8`: `cpy` that writes to `ie` in user mode.
- When `xct == 0x9`: `cpy` that reads from `xct` in user mode.
- When `xct == 0xa`: `cpy` that writes to `xct` in user mode.
- When `xct == 0xb`: `cpy` that reads from `swiarg0` when in user mode.
- When `xct == 0xc`: `cpy` that writes to `swiarg0` when in user mode.
- When `xct == 0xd`: `cpy` that reads from `swiarg1` when in user mode.
- When `xct == 0xe`: `cpy` that writes to `swiarg1` when in user mode.
- When `xct == 0xf`: `cpy` that reads from `swiarg2` when in user mode.
- When `xct == 0x10`: `cpy` that writes to `swiarg2` when in user mode.
- When `xct == 0x11`: `cpy` that reads from `swiarg3` when in user mode.
- When `xct == 0x12`: `cpy` that writes to `swiarg3` when in user mode.
- When `xct == 0x13`: Instructions that read from supervisor mode ILARs or DLARs when in user mode.
- When `xct == 0x14`: Instructions that write to supervisor mode ILARs or DLARs when in user mode.
- When `xct == 0x15`: When user mode `ipc`'s next destination is not in any ILARs.

Instructions

4.1 Group 0 Instructions

- Encoding 0: 0000 aaaa aabb bbbb cccc cc00 000v oooo
- Encoding 1: 0000 aaaa aabb bbbb cccc ccdd dddd oooo
 - a: DLAR a
 - b: DLAR b
 - c: DLAR c or ILAR c
 - d: DLAR d
 - v:
 - * when 0b0: scalar operation. The assembly syntax indicating a scalar operation simply adds ".s" to the instruction's name.
 - * when 0b1: vector operation. The assembly syntax indicating a vector operation simply adds ".v" to the instruction's name.
 - o: Opcode
- Most instructions in this group use Encoding 0. The instructions `div.s` and `div.v` use Encoding 1. These two instructions have opcodes of 0b1110 and 0b1111, respectively, such that bit 0 of the instruction specifies the same information as the v bit does for Encoding 0.
- Instruction List:
 1. `add dA, dB, dC`
 2. `sub dA, dB, dC`
 3. `slt dA, dB, dC`
 4. `mul dA, dB, dC`
 - This instruction causes an exception if dB or dC is of the following types: u64, s64.
 5. `and dA, dB, dC`
 6. `or dA, dB, dC`
 7. `xor dA, dB, dC`
 8. `shl dA, dB, dC`
 - Logical shift left.
 - This instruction causes an exception if dA, dB, or dC is of the following types: u64, s64.
 - This instruction casts a temporary copy of dC to the unsigned type that is the same size as dA's type and uses that instead of dC.
 9. `shr dA, dB, dC`

- Logical shift right if `dA` is unsigned, or arithmetic shift right if `dA` is signed.
 - This instruction causes an exception if `dA`, `dB`, or `dC` is of the following types: `u64`, `s64`.
 - This instruction casts a temporary copy of `dC` to the unsigned type that is the same size as `dA`'s type and uses that instead of `dC`.
10. `rol dA, dB, dC`
- Rotate left.
 - This instruction causes an exception if `dA`, `dB`, or `dC` is of the following types: `u64`, `s64`.
 - This instruction casts a temporary copy of `dC` to the unsigned type that is the same size as `dA`'s type and uses that instead of `dC`.
11. `ror dA, dB, dC`
- Rotate right.
 - This instruction causes an exception if `dA`, `dB`, or `dC` is of the following types: `u64`, `s64`.
 - This instruction casts a temporary copy of `dC` to the unsigned type that is the same size as `dA`'s type and uses that instead of `dC`.
12. `add dA, dB.addr, dC`
- This instruction causes an exception if `dA` or `dC` is of the following types: `u64`, `s64`.
 - For `add.v dA, dB.addr, dC`, this instruction duplicates the value of `cast(dA.type, dB.addr)` (a scalar) into a temporary (DLAR data's length number of bytes long) vector of element type `dA.type` for the purposes of this calculation.
13. `shl dA, dB.addr, dC`
- This instruction causes an exception if `dA` or `dC` is of the following types: `u64`, `s64`.
 - For `shl.v dA, dB.addr, dC`, this instruction duplicates `cast(dA.type, dB.addr)` (a scalar) into a temporary (DLAR data's length number of bytes long) vector of element type `dA.type` for the purposes of this calculation.
 - This instruction casts a temporary copy of `dC` to the unsigned type that is the same size as `dA`'s type and uses that instead of `dC`.
14. `add dA, iB.addr, dC`
- For `add.v dA, iB.addr, dC`, this instruction duplicates the value of `cast(dA.type, iB.addr)` (a scalar) into a temporary (DLAR data's length number of bytes long) vector of element type `dA.type` for the purposes of this calculation.
15. `div.s dA, dB, dC, dD`

- This instruction causes an exception if `dC` is of the following types:
`u64, s64`.
- This instruction writes the quotient into `dA`, and the remainder into `dD`.

16. `div.v dA, dB, dC, dD`

- This instruction causes an exception if `dC` is of the following types:
`u64, s64`.
- This instruction writes the quotient into `dA`, and the remainder into `dD`.

4.2 Group 1 Instructions

- Encoding: 0001 aaaa aabb bbbb cccc cc00 000o oooo
 - a: DLAR a
 - b: DLAR b
 - c: DLAR c
 - o: Opcode
- For the first 8 instructions with names of the form "ld<type>", the dB DLAR's scalar data field (temporarily casted to the u32 type) is added to the dC DLAR's scalar data field (temporarily casted to the u32 type), to calculate the address being loaded from. Also, for these same instructions, the type dA is set to is indicated in the instruction name, with, for example, ldu8 setting dA's type to u8.
- For the second 8 instructions with names of the form "ld<type>", the dB DLAR's address field (temporarily casted to the u32 type) is added to the dC DLAR's scalar data field (temporarily casted to the u32 type), to calculate the address being loaded from. Also, for these same instructions, the type dA is set to is indicated in the instruction name, with, for example, lds32 setting dA's type to s32.
- For the instructions with names of the form "st<type>", the dB DLAR's scalar data field (temporarily casted to the u32 type) is added to the dC DLAR's scalar data field (temporarily casted to the u32 type), to calculate the address being stored to. Also, for these same instructions, the type dA is set to is indicated in the instruction name, with, for example, sts64 setting dA's type to s64.
- Instruction List:
 1. add.r dA, dB
 - This instruction casts a temporary copy of dB to the dA's type and performs a sum of all the scalar data elements of the temporary copy of dB, then stores the result in dA's scalar data.
 2. mul.r dA, dB
 - This instruction casts a temporary copy of dB to the dA's type and performs a product of all the scalar data elements of the temporary copy of dB, then stores the result in dA's scalar data.
 - This instruction causes an exception if dA is of the following types: u64, s64.
 3. max.r dA, dB
 - This instruction casts a temporary copy of dB to the dA's type and finds the scalar data element of the temporary copy of dB that is the maximum, then stores the result in dA's scalar data.

4. `min.r dA, dB`
 - This instruction casts a temporary copy of `dB` to the `dA`'s type and finds the scalar data element of the temporary copy of `dB` that is the minimum, then stores the result in `dA`'s scalar data.
5. `and.r dA, dB`
 - This instruction casts a temporary copy of `dB` to the `dA`'s type and performs a bitwise AND reduction of all the scalar data elements of the temporary copy of `dB`, then stores the result in `dA`'s scalar data.
6. `or.r dA, dB`
 - This instruction casts a temporary copy of `dB` to the `dA`'s type and performs a bitwise OR reduction of all the scalar data elements of the temporary copy of `dB`, then stores the result in `dA`'s scalar data.
7. `xor.r dA, dB`
 - This instruction casts a temporary copy of `dB` to the `dA`'s type and performs a bitwise XOR reduction of all the scalar data elements of the temporary copy of `dB`, then stores the result in `dA`'s scalar data.
8. *Reserved for future expansion.*
9. `ldu8 dA, dB, dC`
10. `lds8 dA, dB, dC`
11. `ldu16 dA, dB, dC`
12. `lds16 dA, dB, dC`
13. `ldu32 dA, dB, dC`
14. `lds32 dA, dB, dC`
15. `ldu64 dA, dB, dC`
16. `lds64 dA, dB, dC`
17. `ldu8 dA, dB.addr, dC`
18. `lds8 dA, dB.addr, dC`
19. `ldu16 dA, dB.addr, dC`
20. `lds16 dA, dB.addr, dC`
21. `ldu32 dA, dB.addr, dC`
22. `lds32 dA, dB.addr, dC`
23. `ldu64 dA, dB.addr, dC`
24. `lds64 dA, dB.addr, dC`
25. `stu8 dA, dB, dC`
26. `sts8 dA, dB, dC`
27. `stu16 dA, dB, dC`
28. `sts16 dA, dB, dC`

- 29. stu32 dA, dB, dC
- 30. sts32 dA, dB, dC
- 31. stu64 dA, dB, dC
- 32. sts64 dA, dB, dC

4.3 Group 2 Instructions

- Encoding: 0010 aaaa aabb bbbb iiii iiii iiii iooo
 - a: DLAR a
 - b: DLAR b
 - i: `simm13` (sign-extended 13-bit immediate)
 - o: Opcode
- For these instructions, the `dB` DLAR's scalar data field (temporarily casted to the `u32` type) is added to `simm13` (sign-extended to 32-bit), to calculate the address being loaded from.
- Also, the type `dA` is set to is indicated in the instruction name, with, for example, `ldu8` setting `dA`'s type to `u8`.
- Instruction List:
 1. `ldu8 dA, dB, simm13`
 2. `lds8 dA, dB, simm13`
 3. `ldu16 dA, dB, simm13`
 4. `lds16 dA, dB, simm13`
 5. `ldu32 dA, dB, simm13`
 6. `lds32 dA, dB, simm13`
 7. `ldu64 dA, dB, simm13`
 8. `lds64 dA, dB, simm13`

4.4 Group 3 Instructions

- Encoding: 0011 aaaa aabb bbbb iiii iiii iiii iooo
 - a: DLAR a
 - b: DLAR b
 - i: `simm13` (sign-extended 13-bit immediate)
 - o: Opcode
- For these instructions, the `dB` DLAR's scalar data field (temporarily casted to the `u32` type) is added to `simm13` (sign-extended to 32-bit), to calculate the address being stored to.
- Also, the type `dA` is set to is indicated in the instruction name, with, for example, `stu8` setting `dA`'s type to `u8`.
- Instruction List:
 1. `stu8 dA, dB, simm13`
 2. `sts8 dA, dB, simm13`
 3. `stu16 dA, dB, simm13`
 4. `sts16 dA, dB, simm13`
 5. `stu32 dA, dB, simm13`
 6. `sts32 dA, dB, simm13`
 7. `stu64 dA, dB, simm13`
 8. `sts64 dA, dB, simm13`

4.5 Group 4 Instructions

- Encoding: 0100 aaaa aabb bbbb 0000 0000 0000 0000
 - a: DLAR a
 - b: DLAR b
 - o: Opcode
- For these instructions, the dB register's scalar data field is used.
- Instruction List:
 1. dpu8 dA, dB
 - This instruction casts (a temporary copy of) the scalar data of dB to the u8 type. The casted scalar data is then stored into every u8 vector element of dA. The type of dA is then changed to u8.
 2. dps8 dA, dB
 - This instruction casts (a temporary copy of) the scalar data of dB to the s8 type. The casted scalar data is then stored into every s8 vector element of dA. The type of dA is then changed to s8.
 3. dpu16 dA, dB
 - This instruction casts (a temporary copy of) the scalar data of dB to the u16 type. The casted scalar data is then stored into every u16 vector element of dA. The type of dA is then changed to u16.
 4. dps16 dA, dB
 - This instruction casts (a temporary copy of) the scalar data of dB to the s16 type. The casted scalar data is then stored into every s16 vector element of dA. The type of dA is then changed to s16.
 5. dpu32 dA, dB
 - This instruction casts (a temporary copy of) the scalar data of dB to the u32 type. The casted scalar data is then stored into every u32 vector element of dA. The type of dA is then changed to u32.
 6. dps32 dA, dB
 - This instruction casts (a temporary copy of) the scalar data of dB to the s32 type. The casted scalar data is then stored into every s32 vector element of dA. The type of dA is then changed to s32.
 7. dpu64 dA, dB
 - This instruction casts (a temporary copy of) the scalar data of dB to the u64 type. The casted scalar data is then stored into every u64 vector element of dA. The type of dA is then changed to u64.
 8. dps64 dA, dB
 - This instruction casts (a temporary copy of) the scalar data of dB to the s64 type. The casted scalar data is then stored into every s64 vector element of dA. The type of dA is then changed to s64.

4.6 Group 5 Instructions

- Encoding 0: 0101 aaaa aabb bbbb cccc cc00 000j jjjo
- Encoding 1: 0101 aaaa aabb bbbb iiii iiii iiij jjjo
 - a: ILAR a
 - b: ILAR b
 - c: DLAR c
 - i: isimm11 (11-bit immediate, left shifted by 2, then sign-extended)
 - j: jimm4, the number of consecutive ILARs past iA to fetch into.
 - o: Opcode
- For instructions using Encoding 0, the address to `fetch` from is computed by adding the address field of `iB` to the scalar data field (temporarily casted to the `u32` type) of the `dC` DLAR.
- For instructions using Encoding 1, the address to `fetch` from is computed by adding the address field of `iB` to the value `cast(s32, (isimm11 << 2))`.
- Additionally, this and any other instructions with "fetch" in their names are the only way to fetch instructions from memory (or other ILARs) in this CPU's design. The instruction pipeline only automatically fetches instructions from `ipc`, with straight line code that overflows outside of `ipc` causing `ipc` to set its data field to that of the other ILARs that have the data from the destination. If no ILARs have the data from the destination, an exception is thrown.
- Instruction List:
 1. `fetch iA, iB, dC, jimm4`
 - This instruction uses Encoding 0.
 2. `fetch iA, iB, isimm11, jimm4`
 - This instruction uses Encoding 1.

4.7 Group 6 Instructions

- Encoding: 0110 aaaa aabb bbbb iiii ijvv jjvo oooo
 - a: DLAR a
 - b: ILAR b
 - i: iimm5 (5-bit immediate, left shifted by 2, then zero-extended)
 - j: jimm5 (5-bit immediate, left shifted by 2, then zero-extended)
 - v:
 - * when 0b0: scalar operation (uses the scalar data of dA). The assembly syntax indicating a scalar operation simply adds ".s" to the instruction's name.
 - * when 0b1: vector operation (uses the vector data of dA). The assembly syntax indicating a vector operation simply adds ".v" to the instruction's name.
 - o: Opcode
- These instructions use the scalar or vector data field of dA.
- Instruction List:
 1. sel dA, iB, iimm5, jimm5
 - This instruction jumps to iB[iimm5 << 2] if the particular data field of dA is non-zero, otherwise to the address iB[jimm5 << 2].
 2. jz dA, iB, iimm5
 - This instruction jumps to iB[iimm5 << 2] if the particular data field of dA is zero.
 3. jnz dA, iB, iimm5
 - This instruction jumps to iB[iimm5 << 2] if the particular data field of dA is non-zero.
 4. reti dA
 - This instruction returns from an interrupt if dA is non-zero, setting ie to 0b1.
 - This instruction causes an exception if the processor is in user mode.
 5. retx dA
 - This instruction returns from supervisor mode to user mode if dA is non-zero.
 - This instruction causes an exception if the processor is in user mode.
 6. *Reserved for future expansion.*
 7. *Reserved for future expansion.*
 8. *Reserved for future expansion.*

9. *Reserved for future expansion.*
10. *Reserved for future expansion.*
11. *Reserved for future expansion.*
12. *Reserved for future expansion.*
13. *Reserved for future expansion.*
14. *Reserved for future expansion.*
15. *Reserved for future expansion.*
16. *Reserved for future expansion.*
17. *Reserved for future expansion.*
18. *Reserved for future expansion.*
19. *Reserved for future expansion.*
20. *Reserved for future expansion.*
21. *Reserved for future expansion.*
22. *Reserved for future expansion.*
23. *Reserved for future expansion.*
24. *Reserved for future expansion.*
25. *Reserved for future expansion.*
26. *Reserved for future expansion.*
27. *Reserved for future expansion.*
28. *Reserved for future expansion.*
29. *Reserved for future expansion.*
30. *Reserved for future expansion.*
31. *Reserved for future expansion.*
32. *Reserved for future expansion.*

4.8 Group 7 Instructions

- Encoding: 0111 aaaa aabb bbbb cccc ccii iiii Sooo
 - a: ILAR a or DLAR a
 - b: ILAR b or DLAR b
 - c: DLAR c
 - i: imm6 (zero-extended 6-bit immediate), amount of LARs to use
 - S:
 - * when 0b0: Destination LARs (the ones starting with iA or dA) or source LARs (the ones starting with iB or dB) are user mode LARs. An example of the syntax for `getaddrs` in this case is `getaddrs.U`. The ".U" suffix indicates this instruction will have the S bit set to 0b0.
 - * when 0b1: Destination LARs (the ones starting with iA or dA) or source LARs (the ones starting with iB or dB) are supervisor mode LARs. An example of the syntax for `getaddrs` in this case is `getaddrs.S`. The ".S" suffix indicates this instruction will have the S bit set to 0b1.
 - o: Opcode
- Instruction List:
 1. `getaddrs dA, dB, imm6`
 - This instruction uses the S bit to indicate which mode the source DLARs belong to.
 - This instruction grabs the addresses of source DLARs starting with dB and then also the following `imm6 - 1` source DLARs. The grabbed addresses are then written into consecutive scalar data elements of destination DLARs (starting with dA and continuing into the following destination DLARs as necessary).
 - When supervisor mode LARs are used for the source(s), this instruction causes an exception if used in user mode.
 2. `getaddrs dA, iB, imm6`
 - This instruction uses the S bit to indicate which mode the source ILARs belong to.
 - This instruction grabs the addresses of source ILARs starting with dB and then also the following `imm6 - 1` source DLARs. The grabbed addresses are then written into consecutive scalar data elements of destination DLARs (starting with dA and continuing into the following destination DLARs as necessary).
 - When supervisor mode LARs are used for the source(s), this instruction causes an exception if used in user mode.
 3. `gettypes dA, dB, imm6`
 - This instruction uses the S bit to indicate which mode the source DLARs belong to.

- This instruction grabs the types of source DLARs starting with `dB` and then also the following `imm6 - 1` source DLARs. The grabbed types are then written into consecutive scalar data elements of destination DLARs (starting with `dA` and continuing into the following destination DLARs as necessary).
 - When supervisor mode LARs are used for the source(s), this instruction causes an exception if used in user mode.
4. `ldm dA, dB, dC, imm6`
- This instruction's name is short for "load multiple".
 - This instruction uses the `S` bit to indicate to which mode the destination DLARs belong.
 - This instruction uses addresses stored in the `imm6` scalar data elements of consecutive DLARs (starting with `dB`) and types stored in the `imm6` scalar data elements of consecutive DLARs (starting with `dC`). Multiple loads from memory are performed into the `imm6` destination DLARs (starting with `dA`).
 - When supervisor mode LARs are used for the destination(s), this instruction causes an exception if used in user mode.
5. `fetchm iA, dB, imm6`
- This instruction's name is short for "fetch multiple".
 - This instruction uses the `S` bit to indicate to which mode the destination DLARs belong.
 - This instruction uses addresses stored in the `imm6` scalar data elements of consecutive DLARs (starting with `dB`). Multiple fetches from memory are performed into the `imm6` destination ILARs (starting with `iA`).
 - When supervisor mode LARs are used for the destination(s), this instruction causes an exception if used in user mode.
6. *Reserved for future expansion.*
7. *Reserved for future expansion.*
8. *Reserved for future expansion.*

4.9 Group 8 Instructions

- Encoding: 1000 aaaa aabb bbbb cccc ccdd dddd oooo
 - a: DLAR a
 - b: DLAR b
 - c: DLAR c
 - d: DLAR d
 - o: Opcode
- Instruction List:
 1. cpy dA, ie
 2. cpy ie, dA
 3. cpy dA, xct
 4. cpy xct, dA
 5. cpy dA, swiarg0
 6. cpy swiarg0, dA
 7. cpy dA, swiarg1
 8. cpy swiarg1, dA
 9. cpy dA, swiarg2
 10. cpy swiarg2, dA
 11. cpy dA, swiarg3
 12. cpy swiarg3, dA
 13. swi dA, dB, dC, dD
 - Note that this instruction always causes an exception to occur.
 - This instruction stores dA's vector data to into swiarg0.
 - This instruction stores dB's vector data to into swiarg1.
 - This instruction stores dC's vector data to into swiarg2.
 - This instruction stores dD's vector data to into swiarg3.
 14. *Reserved for future expansion.*
 15. *Reserved for future expansion.*
 16. *Reserved for future expansion.*

4.10 Group 9 Instructions

- Encoding: 1001 aaaa aabb bbbb cccc cc00 000v oooo
 - a: DLAR a
 - b: DLAR b
 - c: DLAR c
 - v:
 - * when 0b0: scalar operation. The assembly syntax indicating a scalar operation simply adds ".s" to the instruction's name.
 - * when 0b1: vector operation. The assembly syntax indicating a vector operation simply adds ".v" to the instruction's name.
 - o: Opcode
- These instructions perform a read from/write to the IO address calculated by adding the scalar data of dB and dC, or (in other words) $dB + dC$. The types of dB and dC are temporarily casted to u32 for this calculation.
- When a scalar operation is being performed, only the scalar data of dA is read into from/written out to IO space.
- When a vector operation is being performed, the entire vector data of dA is read into from/written out to IO space.
- Instruction List:
 1. inu8 dA, dB, dC
 - This instruction sets the type of dA to u8 before performing anything else of the operation.
 - This instruction reads from IO space and writes to dA.
 2. ins8 dA, dB, dC
 - This instruction sets the type of dA to s8 before performing anything else of the operation.
 - This instruction reads from IO space and writes to dA.
 3. inu16 dA, dB, dC
 - This instruction sets the type of dA to u16 before performing anything else of the operation.
 - This instruction reads from IO space and writes to dA.
 4. ins16 dA, dB, dC
 - This instruction sets the type of dA to s16 before performing anything else of the operation.
 - This instruction reads from IO space and writes to dA.
 5. inu32 dA, dB, dC

- This instruction sets the type of dA to u32 before performing anything else of the operation.
 - This instruction reads from IO space and writes to dA.
6. ins32 dA, dB, dC
- This instruction sets the type of dA to s32 before performing anything else of the operation.
 - This instruction reads from IO space and writes to dA.
7. inu64 dA, dB, dC
- This instruction sets the type of dA to u64 before performing anything else of the operation.
 - This instruction reads from IO space and writes to dA.
8. ins64 dA, dB, dC
- This instruction sets the type of dA to s64 before performing anything else of the operation.
 - This instruction reads from IO space and writes to dA.
9. outu8 dA, dB, dC
- This instruction sets the type of dA to u8 before performing anything else of the operation.
 - This instruction writes to IO space and reads from dA.
10. outs8 dA, dB, dC
- This instruction sets the type of dA to s8 before performing anything else of the operation.
 - This instruction writes to IO space and reads from dA.
11. outu16 dA, dB, dC
- This instruction sets the type of dA to u16 before performing anything else of the operation.
 - This instruction writes to IO space and reads from dA.
12. outs16 dA, dB, dC
- This instruction sets the type of dA to s16 before performing anything else of the operation.
 - This instruction writes to IO space and reads from dA.
13. outu32 dA, dB, dC
- This instruction sets the type of dA to u32 before performing anything else of the operation.
 - This instruction writes to IO space and reads from dA.
14. outs32 dA, dB, dC
- This instruction sets the type of dA to s32 before performing anything else of the operation.

- This instruction writes to IO space and reads from dA.
15. `outu64 dA, dB, dC`
- This instruction sets the type of dA to u64 before performing anything else of the operation.
 - This instruction writes to IO space and reads from dA.
16. `outs64 dA, dB, dC`
- This instruction sets the type of dA to s64 before performing anything else of the operation.
 - This instruction writes to IO space and reads from dA.