Snow64 Instruction Set

- Notes
    - There are no Instruction LARs (the typical instruction fetch of most computer processors is used instead).
    - Addresses are 64-bit.

- Data LARs:

```
typedef struct packed
{
    // Data field
    union packed
    {
        // This is possibly not valid SystemVerilog because of
        // arrays inside a packed struct, but it makes for nice
        // pseudocode
        logic [7:0] data_8[0:31];

        logic [15:0] data_16[0:15];

        struct packed
        {
            // sign bit, 1 means negative
            logic sign;

            // Exponent, +bias
            logic [7:0] exp;

            // Mantissa; normalized implies 1 MSB
            logic [6:0] mant;
        } data_float_16[0:15];

        logic [31:0] data_32[0:7];
        logic [63:0] data_64[0:3];
    } data;

    // Note that this is a 64-bit structure
    union packed
    {
        struct packed
        {
            logic [63 - 5 : 0] base_ptr;

            logic [4:0] offset;
        } addr_8;

        // Used for both 16-bit integers and the half floats
        struct packed
        {
            logic [63 - 4 : 0] base_ptr;
```

```
                logic [3:0] offset;
            } addr_16;

            struct packed
            {
                logic [63 - 3 : 0] base_ptr;

                logic [2:0] offset;
            } addr_32;

            struct packed
            {
                logic [63 - 2 : 0] base_ptr;

                logic [1:0] offset;
            } addr_64;
        } addr;

        // Integer Type (used when "is_float_16" is 1'b0):
        // 2'b00:  8-bit
        // 2'b01:  16-bit
        // 2'b10:  32-bit
        // 2'b11:  64-bit
        logic [1:0] type_of_int;

        // These are actually packed 16-bit floats, the implementing the
        // high 16 bits of 32-bit IEEE float.
        // "type_of_int" is ignored when "is_float_16" is 1'b1.
        logic is_float_16;

        // Unsigned:  1'b0
        // Signed:  1'b1
        // Note:  unsgn_or_sgn is ignored for floats
        logic unsgn_or_sgn;

        // Data should be lazily stored to memory if this is 1'b1
        // Otherwise, when this is 1'b0, data in this LAR is up to date
        // with memory.
        logic dirty;

    } DataLar;
```

* Registers * The DLARs themselves (there are 16, but this may be changed later): * `dzero` (always zero), * `du0`, `du1`, `du2`, `du3` `du4`, `du5`, `du6`, `du7`, `du8`, `du9`, `du10`, `du11` (user registers) * `dlr` (standard link register (hardware does not enforce this)) * `dfp` (standard frame pointer (hardware does not enforce this)) * `dsp` (standard stack pointer (hardware does not enforce this)) * Other registers: * `pc` (the program counter, 64-bit)

Instruction Set

- Note: All invalid instructions are treated as NOPs.
- ALU Instructions: Opcode Group: 0b000
    - Encoding: `000t aaaa bbbb cccc oooo iiii iiii iiii`
        - `t`: operation type: if `0b0`: scalar operation; else: vector operation
        - `a`: dDest
        - `b`: dSrc0
        - `c`: dSrc1
        - `o`: opcode
        - `i`: 12-bit signed immediate
    - Note: For ALU instructions, any result that doesn't fit in the destination will be truncated to fit into the destination. This affects both scalar and vector operations.
    - Note: also, for each of these instructions, the address field is not used as an operand, just the data field.
        - Example: `adds d1`
            - Effect: `d1.sdata <= cast_to_type_of_d1(d2.sdata) + cast_to_type_of_d1(d3.sdata)`
    - Instructions:
        - **add** dDest, dSrc0, dSrc1
            - Opcode: 0x0
            - Scalar Mnemonic: `adds`
            - Vector Mnemonic: `addv`
        - **sub** dDest, dSrc0, dSrc1
            - Opcode: 0x1
            - Scalar Mnemonic: `subs`
            - Vector Mnemonic: `subv`
        - **slt** dDest, dSrc0, dSrc1
            - Opcode: 0x2
            - Scalar Mnemonic: `slts`
            - Vector Mnemonic: `sltv`
            - Note: set less than
            - Note: The signedness of dDest will be used for the operation
        - **mul** dDest, dSrc0, dSrc1
            - Opcode: 0x3
            - Scalar Mnemonic: `muls`
            - Vector Mnemonic: `mulv`
            - Note: If dDest has a larger size than both dSrc0 and dSrc1, then the signedness used for the operation will be that of dDest
            - Note: This operation is not guaranteed to be single cycle, and thus pipeline stalls will be used
        - **div** dDest, dSrc0, dSrc1
            - Opcode: 0x4
            - Scalar Mnemonic: `divs`
            - Vector Mnemonic: `divv`

    - Note: This operation is not guaranteed to be single cycle, and thus pipeline stalls will be used
- **and** dDest, dSrc0, dSrc1
    - Opcode: 0x5
    - Scalar Mnemonic: `ands`
    - Vector Mnemonic: `andv`
    - Note: For floats, this operation treats all operands as 16-bit signed integers.
- **orr** dDest, dSrc0, dSrc1
    - Opcode: 0x6
    - Scalar Mnemonic: `orrs`
    - Vector Mnemonic: `orrv`
    - Note: For floats, this operation treats all operands as 16-bit signed integers.
- **xor** dDest, dSrc0, dSrc1
    - Opcode: 0x7
    - Scalar Mnemonic: `xors`
    - Vector Mnemonic: `xorv`
    - Note: For floats, this operation treats all operands as 16-bit signed integers.
- **shl** dDest, dSrc0, dSrc1
    - Opcode: 0x8
    - Scalar Mnemonic: `shls`
    - Vector Mnemonic: `shlv`
    - Note: Shift left
    - Note: For floats, this operation treats all operands as 16-bit signed integers.
- **shr** dDest, dSrc0, dSrc1
    - Opcode: 0x9
    - Scalar Mnemonic: `shrs`
    - Vector Mnemonic: `shrv`
    - Note: Shift right
    - Note: dSrc0's signedness is used to determine the type of right shift:
        - If dSrc0 is unsigned, a logical right shift is performed
        - If dSrc0 is signed, an arithmetic right shift is performed
    - Note: dSrc1 is always treated as unsigned (due to being a number of bits to shift by)
    - Note: For floats, this operation treats all operands as 16-bit signed integers.
- **inv** dDest, dSrc0
    - Opcode: 0xa
    - Scalar Mnemonic: `invs`
    - Vector Mnemonic: `invv`
    - Note: Bitwise invert
    - Note: For floats, this operation treats all operands as 16-bit signed integers.
- **not** dDest, dSrc0
    - Opcode: 0xb
    - Scalar Mnemonic: `nots`

        - Vector Mnemonic: `notv`
        - Note: Logical not
    - **add** dDest, pc, simm12
        - Opcode: 0xc
        - Scalar Mnemonic: `adds`
        - Vector Mnemonic: `addv`
        - Note: This is useful for pc-relative loads, relative branches, and for getting the return address of a subroutine call into a LAR before jumping to a subroutine.

- Instructions for interacting with special-purpose registers: Opcode Group: 0b001
    - Encoding: `0010 aaaa oooo iiii iiii iiii iiii iiii`
        - `a`: dA
        - `o`: opcode
        - `i`: 20-bit signed immediate
    - Note: all instructions in group 0b001 are scalar operations.
    - Note: `dX.sdata` is simply the current scalar portion of the data LAR called `dX`
    - Instructions:
        - **btru** dA, simm20
            - Opcode: 0x0
            - Effect: `if (dA.sdata != 0) pc <= pc + sign_extend_to_64(simm20);`
        - **bfal** dA, simm20
            - Opcode: 0x1
            - Effect: `if (dA.sdata == 0) pc <= pc + sign_extend_to_64(simm20);`
        - **jmp** dA
            - Opcode: 0x2
            - Effect: `pc <= dA.sdata;`
            - Note: It is suggested to have dA.sdata be at least as large as the largest memory address (which might not be 64-bit if there isn't enough physical memory for that)

- Load Instructions: Opcode Group: 0b010
    - Encoding: `0100 aaaa bbbb cccc oooo iiii iiii iiii`
        - `a`: dDest
        - `b`: dSrc0
        - `c`: dSrc1
        - `o`: opcode
        - `i`: 12-bit signed immediate
    - Effect:
        - Load LAR-sized data from 64-bit address computed as follows: `(dB.address + extend_to_64(dC.sdata) + (sign_extend_to_64(simm12)))`
            - This 64-bit address is referred to as the "effective address".

    - The type of extension of the `extend_to_64(dC.sdata)` expression is based upon the type of `dC`.
      - If `dC` is tagged as an unsigned integer, zero-extension is performed.
      - If `dC` is tagged as a signed integer, sign-extension is performed.
      - If `dC` is tagged as a BFloat16, `dC.sdata` is casted to a 64-bit signed integer. (This one is weird... normally, addressing isn't done with floating point numbers!).
    - Due to associativity of the LARs, these instructions will not actually load from memory if the effective address's data already loaded into a LAR.
  - Instructions:
    - **ldu8** dA, dB, dC, simm12
      - Opcode: 0x0
      - Note: unsigned 8-bit integer(s)
    - **lds8** dA, dB, dC, simm12
      - Opcode: 0x1
      - Note: signed 8-bit integer(s)
    - **ldu16** dA, dB, dC, simm12
      - Opcode: 0x2
      - Note: unsigned 16-bit integer(s)
    - **lds16** dA, dB, dC, simm12
      - Opcode: 0x3
      - Note: signed 16-bit integer(s)
    - **ldu32** dA, dB, dC, simm12
      - Opcode: 0x4
      - Note: unsigned 32-bit integer(s)
    - **lds32** dA, dB, dC, simm12
      - Opcode: 0x5
      - Note: signed 32-bit integer(s)
    - **ldu64** dA, dB, dC, simm12
      - Opcode: 0x6
      - Note: unsigned 64-bit integer(s)
    - **lds64** dA, dB, dC, simm12
      - Opcode: 0x7
      - Note: signed 64-bit integer(s)
    - **ldf16** dA, dB, dC, simm12
      - Opcode: 0x8
      - Note: BFloat16 format floating point number.

- Store Instructions: Opcode Group: 0b011
  - Encoding: `0110 aaaa bbbb cccc oooo iiii iiii iiii`
    - `a`: dA
    - `b`: dB
    - `c`: dC
    - `o`: opcode
    - `i`: 12-bit signed immediate
  - Note: These are actually type conversion instructions as actual writes to memory are done lazily

- Effect:
  - These instructions mark `dA` as dirty, change its address to the effective address (see next bullet), and sets its type.
  - The 64-bit effective address is computed as follows: `(dB.address + extend_to_64(dC.sdata) + (sign_extend_to_64(simm12)))`
  - The type of extension of the `extend_to_64(dC.sdata)` expression is based upon the type of `dC`.
    - If `dC` is tagged as an unsigned integer, zero-extension is performed.
    - If `dC` is tagged as a signed integer, sign-extension is performed.
    - If `dC` is tagged as a BFloat16, `dC.sdata` is casted to a 64-bit signed integer. (This one is weird... normally, addressing isn't done with floating point numbers!).
- Instructions:
  - **stu8** dA, dB, dC, simm12
    - Opcode: 0x0
    - Note: unsigned 8-bit integer(s)
  - **sts8** dA, dB, dC, simm12
    - Opcode: 0x1
    - Note: signed 8-bit integer(s)
  - **stu16** dA, dB, dC, simm12
    - Opcode: 0x2
    - Note: unsigned 16-bit integer(s)
  - **sts16** dA, dB, dC, simm12
    - Opcode: 0x3
    - Note: signed 16-bit integer(s)
  - **stu32** dA, dB, dC, simm12
    - Opcode: 0x4
    - Note: unsigned 32-bit integer(s)
  - **sts32** dA, dB, dC, simm12
    - Opcode: 0x5
    - Note: signed 32-bit integer(s)
  - **stu64** dA, dB, dC, simm12
    - Opcode: 0x6
    - Note: unsigned 64-bit integer(s)
  - **sts64** dA, dB, dC, simm12
    - Opcode: 0x7
    - Note: signed 64-bit integer(s)
  - **stf16** dA, dB, dC, simm12
    - Opcode: 0x8
    - Note: BFloat16 format floating point number.