

Flare32 CPU

FL4SHK

April 29, 2023

# Table of Contents

|  |          |
|--|----------|
| <b>Table of Contents</b>   | <b>1</b> |
| <b>Introduction</b>  | <b>2</b> |
| 2.1 General Information . . . . .  | 2        |
| 2.2 Registers . . . . .  | 2        |
| <b>Instruction Set</b>   | <b>4</b> |
| 3.1 Instruction Group 0: <b>pre</b> and <b>lpre</b> . . . . .                    | 4        |
| 3.1.1 Handling of <b>pre</b> , <b>lpre</b> , and <b>index</b> . . . . .          | 5        |
| 3.2 Instruction Group 1 . . . . .  | 6        |
| 3.3 Instruction Group 2 . . . . .  | 8        |
| 3.4 Instruction Group 3: Relative Branches . . . . .                             | 10       |
| 3.5 Instruction Group 4 . . . . .  | 12       |
| 3.6 Instruction Group 5: Immediate Indexed Load . . . . .                        | 17       |
| 3.7 Instruction Group 6: Immediate Indexed Store . . . . .                       | 18       |
| 3.8 Instruction Group 7, Subgroup 0b00: Extra 8-bit and 16-bit Ops               | 18       |
| 3.9 Instruction Group 7, Subgroup 0b010: Extra load/store instructions . . . . . | 19       |

# Introduction

## 2.1 General Information

- Addresses are 32-bit.
- Big-endian byte ordering is used.
- Bytes are octets (8 bits).
- Instructions must be aligned to 16 bits, so jump and branch targets must also be aligned to 16 bits.
  - Branch offsets encoded into instructions must be 16-bit aligned, or in other words bit 0 of the branch offset must be 0b0

## 2.2 Registers

There are sixteen general-purpose registers (all of which are 32-bit): `r0`, `r1`, `r2`, ..., `r11`, `r12`, `lr` (link register, the return address of `bl` and `jl`), `fp` (frame pointer), `sp` (stack pointer).

The program counter, `pc`, is 32 bits long, as addresses are 32-bit.  
Here are the special-purpose registers:

- `flags`: arithmetic/logic FLAGS; (reg encoding: 0x0)
- `ids`: Interrupt DeStination: the address to jump to upon an interrupt being serviced (also known as the interrupt vector); (reg encoding: 0x1)
- `ira`: Interrupt Return Address: the address that was jumped from upon an interrupt being serviced; (reg encoding: 0x2)
- `ie`: IRQ Enable flag: flag indicating whether IRQs are disabled (0x0) or enabled (0x1); note that this flag starts with a value of 0x0; (reg encoding: 0x3)
- `ity`: Interrupt TYpe: flag indicating whether the most recently taken interrupt is an IRQ (0x0) or a `swi` (0x1) (reg encoding: 0x4)
- `sty`: Software interrupt TYpe: `swi`'s argument. For `swi rA, #simm`, this is the value `rA + simm` (reg encoding: 0x5)
- Note: All other encodings for special-purpose registers are reserved.

Here are the bits of `flags`:

- Zero (Z): (`flags` bit 0)

- Carry (C): (**flags** bit 1)
- oVerflow (V): (**flags** bit 2)
- Negative (N): (**flags** bit 3)
- Note: All other bit positions of **flags** are reserved.

## Instruction Set

### 3.1 Instruction Group 0: **pre** and **lpre**

For **pre**, the following encoding is used, with each character representing one bit:

0000 *iiii* *iiii* *iiii*, where

- *i* is a 12-bit constant.

For **lpre**, the following encoding is used, with each character representing one bit:

0001 0*iiii* *iiii* *iiii* *iiii* *iiii* *iiii* *iiii*, where

- *i* is a 27-bit constant.

**pre** and **lpre** are mechanisms by which immediates larger than normal can be used, essentially acting like variable width instructions.

There is no mechanism in the assembly language itself to use **pre** or **lpre** as instructions. Instead, it is expected that the assembler or linker will be the one to insert **pre** or **lpre** as needed if an immediate is too large for a particular instruction.

For non-branch instructions:

- If **pre** is used, the immediate field of the **pre** instruction will form bits [16:5] of the immediate of the next non-**index** instruction. The 17-bit immediate will then be sign-extended to 32 bits.
- On the other hand, if **lpre** is used, the immediate field of the **lpre** instruction will form bits [31:5] of the immediate of the next non-**index** instruction.

For branch instructions (group 3):

- if **pre** is used, the immediate field of the **pre** instruction will form bits [20:9] of the immediate of the next non-**index** instruction. The 21-bit immediate will then be sign-extended to 32 bits.
- On the other hand, if **lpre** is used, the immediate field of the **lpre** instruction will form bits [31:9] of the immediate of the next non-**index** instruction.

Instructions that use an immediate value, other than **pre**, **lpre**, and relative branches normally have 5-bit immediates. Relative branches normally have 9-bit branch offsets. if **pre** or **lpre** is used with a relative branch, bits [9:5] of the branch offset that is encoded directly into a relative branch instruction will be ignored, and the immediate field of the **pre** or **lpre** instruction will be used to determine bits [9:5] of the branch offset..

### 3.1.1 Handling of pre, lpre, and index

When a **pre** or **lpre** instruction is found, **pre** or **lpre** will be considered to be "in effect". This condition lasts for one or two instructions after the **pre** or **lpre** instruction, depending on whether or not **index** was in effect.

**index** is an instruction (defined later) that allows a subsequent load or store instruction to perform **base\_reg + index\_reg** indexing. When an **index** instruction is found, it is considered to be in effect. Like **pre** and **lpre**, **index** is inserted automatically by the assembler.

**index** can be combined with **pre** or **lpre**, and it does not matter whether **index** or the **pre/lpre** instruction came first.

When **pre**, **lpre**, or **index** is in effect, IRQs will not be serviced.

Pseudo code for handling the how **pre**, **lpre**, and **index** are to be handled regarding whether or not they are "in effect" is as follows:

```
function handle_pre_lpre_index(input in, output out) {
    if (in.instruction.is_pre()) {
        if (in.state.pre.have || in.state.lpre.have) {
            out.state.can_service_interrupts = false;

            // Instruction was a NOP
            out.state.pre.have = false;
            out.state.lpre.have = false;
            out.state.index.have = false;
            out.instruction = nop;
        } else {
            if (in.state.index.have) {
                out.state.can_service_interrupts = false;
            } else {
                out.state.can_service_interrupts = true;
            }
            out.state.pre.set_have(true);
            out.instruction = in.instruction;
        }
    } else if (in.instruction.is_lpre()) {
        if (in.state.pre.have || in.state.lpre.have) {
            out.state.can_service_interrupts = false;

            // Instruction was a NOP
            out.state.pre.have = false;
            out.state.lpre.have = false;
            out.state.index.have = false;
            out.instruction = nop;
        } else {
            if (in.state.index.have) {
                out.state.can_service_interrupts = false;
            }
        }
    }
}
```

```

        } else {
            out.state.can_service_interrupts = true;
        }
        out.state.lpre.set_have(true);
        out.instruction = in.instruction;
    }
} else if (in.instruction.is_index()) {
    if (in.state.index.have) {
        out.state.can_service_interrupts = false;

        // Instruction was a NOP
        out.state.pre.have = false;
        out.state.lpre.have = false;
        out.state.index.have = false;
        out.instruction = nop;
    } else {
        if (in.state.pre.have || in.state.lpre.have) {
            out.state.can_service_interrupts = false;
        } else {
            out.state.can_service_interrupts = true;
        }
        out.state.index.have = true;
        out.instruction = in.instruction;
    }
} else {
    if (
        in.state.pre.have || in.state.lpre.have || in.state.index.have
    ) {
        out.state.can_service_interrupts = false;
    } else {
        out.state.can_service_interrupts = true;
    }
    out.instruction = instruction;

    // Whenever we see an instruction other than pre, lpre, or index,
    // that means that those instructions stop being "in effect".
    out.state.pre.have = false;
    out.state.lpre.have = false;
    out.state.index.have = false;
}
}

```

### 3.2 Instruction Group 1

The following encoding is used, with each character representing one bit:  
001i iiii oooo aaaa, where

- `i` is a 5-bit sign-extended immediate, and is denoted `simm`
- `a` encodes register `rA`
- `o` is the opcode

Here is a list of instructions from this encoding group.

- Opcode 0x0: `add rA, #simm`
- Opcode 0x1: `add rA, pc, #simm`
  - Effect: `rA <= pc + simm + 2;`
- Opcode 0x2: `add rA, sp, #simm`
- Opcode 0x3: `add rA, fp, #simm`
- Opcode 0x4: `cmp rA, #simm`
  - Effect: Compare `rA` to `simm`.
  - Affectable flags: Z, C, V, N
- Opcode 0x5: `cpy rA, #simm`
  - Effect: Copy an immediate value into `rA`
- Opcode 0x6: `lsl rA, #simm`
  - Effect: Logical shift left
- Opcode 0x7: `lsr rA, #simm`
  - Effect: Logical shift right
- Opcode 0x8: `asr rA, #simm`
  - Effect: Arithmetic shift right
- Opcode 0x9: `and rA, #simm`
  - Effect: Bitwise AND
- Opcode 0xa: `orr rA, #simm`
  - Effect: Bitwise OR
- Opcode 0xb: `xor rA, #simm`
  - Effect: Bitwise XOR
- Opcode 0xc: `ze rA, #simm`



- Effect: Set `rA[31:simm]` to zero.
- Opcode 0xd: **se** `rA, #simm`
  - Effect: Sign-extend `rA[simm:0]` to 32 bits, then copy that value to `rA`.
- Opcode 0xe: **swi** `rA, #simm`
  - Effect: Call software interrupt number `rA + simm`.
- Opcode 0xf: **swi** `#simm`
  - Effect: Call software interrupt number `simm`.

### 3.3 Instruction Group 2

The following encoding is used, with each character representing one bit:

010f oooo bbbb aaaa, where

- o is the opcode
- b encodes register `rB`
- a encodes register `rA`
- f is encoded as 0 if this instruction cannot affect flags and encoded 1 if this instruction is permitted to affect flags. Note that `cmp` is permitted to affect flags regardless of this bit.

Here is a list of instructions from this encoding group.

- Opcode 0x0: **add** `rA, rB`
  - Mnemonic for when flags not affected: **add**
  - Mnemonic for when flags affected: **add.f**
  - Affectable flags: Z, C, V, N
- Opcode 0x1: **sub** `rA, rB`
  - Mnemonic for when flags not affected: **sub**
  - Mnemonic for when flags affected: **sub.f**
  - Affectable flags: Z, C, V, N
- Opcode 0x2: **add** `rA, sp, rB`
  - Mnemonic for when flags not affected: **add**

- Mnemonic for when flags affected: **add.f**
  - Affectable flags: Z, C, V, N
- Opcode 0x3: **add rA, fp, rB**
  - Mnemonic for when flags not affected: **add**
  - Mnemonic for when flags affected: **add.f**
  - Affectable flags: Z, C, V, N
- Opcode 0x4: **cmp rA, rB**
  - Effect Compare **rA** to **rB**. **cmp** is *always* able to affect flags, independent of the encoded **f** bit of the instruction.
  - Affectable flags: Z, C, V, N
- Opcode 0x5: **cpy rA, rB**
  - Mnemonic for when flags not affected: **cpy**
  - Mnemonic for when flags affected: **cpy.f**
  - Effect Copy **rB** into **rA**
  - Affectable flags: Z, N
- Opcode 0x6: **lsl rA, rB**
  - Mnemonic for when flags not affected: **lsl**
  - Mnemonic for when flags affected: **lsl.f**
  - Effect Logical shift left
  - Affectable flags: Z, N
- Opcode 0x7: **lsr rA, rB**
  - Mnemonic for when flags not affected: **lsr**
  - Mnemonic for when flags affected: **lsr.f**
  - Effect Logical shift right
  - Affectable flags: Z, N
- Opcode 0x8: **asr rA, rB**
  - Mnemonic for when flags not affected: **asr**
  - Mnemonic for when flags affected: **asr.f**
  - Effect Arithmetic shift right
  - Affectable flags: Z, N
- Opcode 0x9: **and rA, rB**
  - Mnemonic for when flags not affected: **and**

- Mnemonic for when flags affected: **and.f**
- Effect Bitwise AND
- Affectable flags: Z, N
- Opcode 0xa: **orr rA, rB**
  - Mnemonic for when flags not affected: **orr**
  - Mnemonic for when flags affected: **orr.f**
  - Effect Bitwise OR
  - Affectable flags: Z, N
- Opcode 0xb: **xor rA, rB**
  - Mnemonic for when flags not affected: **xor**
  - Mnemonic for when flags affected: **xor.f**
  - Effect Bitwise XOR
  - Affectable flags: Z, N
- Opcode 0xc: **adc rA, rB**
  - Mnemonic for when flags not affected: **adc**
  - Mnemonic for when flags affected: **adc.f**
  - Effect Add with Carry
  - Affectable flags: Z, C, V, N
- Opcode 0xd: **sbc rA, rB**
  - Mnemonic for when flags not affected: **sbc**
  - Mnemonic for when flags affected: **sbc.f**
  - Effect Subtract with Borrow
  - Affectable flags: Z, C, V, N

### 3.4 Instruction Group 3: Relative Branches

The following encoding is used, with each character representing one bit:

011i iiii iiii oooo, where

- i is a 9-bit sign-extended immediate, and is denoted **sim**
- o is the opcode

Here is a list of instructions from this encoding group.

- Opcode 0x0: **bl** *sim*m
  - Name: Branch and Link
  - Description: Relative call
  - Effect:  $lr \leq pc + 2$ ;  $pc \leq pc + simm + 2$ ;
- Opcode 0x1: **bra** *sim*m
  - Name: BRanch Always
  - Description: Unconditional relative branch
  - Effect:  $pc \leq pc + simm + 2$ ;
- Opcode 0x2: **beq** *sim*m
  - Name: Branch if EQual
  - Effect: if (flags.Z)  $pc \leq pc + simm + 2$ ;
- Opcode 0x3: **bne** *sim*m
  - Name: Branch if Not Equal
  - Effect: if (!flags.Z)  $pc \leq pc + simm + 2$ ;
- Opcode 0x4: **bmi** *sim*m
  - Name: Branch if MInus
  - Effect: if (flags.N)  $pc \leq pc + simm + 2$ ;
- Opcode 0x5: **bpl** *sim*m
  - Name: Branch if PLus
  - Effect: if (!flags.N)  $pc \leq pc + simm + 2$ ;
- Opcode 0x6: **bvs** *sim*m
  - Name: Branch if oVerflow Set
  - Effect: if (flags.V)  $pc \leq pc + simm + 2$ ;
- Opcode 0x7: **bvc** *sim*m
  - Name: Branch if oVerflow Clear
  - Effect: if (!flags.V)  $pc \leq pc + simm + 2$ ;
- Opcode 0x8: **bgeu** *sim*m
  - Name: Branch if Greater than or Equal Unsigned
  - Effect: if (flags.C)  $pc \leq pc + simm + 2$ ;
- Opcode 0x9: **bltu** *sim*m

- Name: Branch if Less Than Unsigned
- Effect: if ( $\neg \text{flags.C}$ )  $\text{pc} \leq \text{pc} + \text{sim} + 2$ ;
- Opcode 0xa: **bgtu** *sim*
  - Name: Branch if Greater Than Unsigned
  - Effect: if ( $\text{flags.C AND } \neg \text{flags.Z}$ )  $\text{pc} \leq \text{pc} + \text{sim} + 2$ ;
- Opcode 0xb: **bleu** *sim*
  - Name: Branch if Less than or Equal Unsigned
  - Effect: if ( $\neg \text{flags.C OR flags.Z}$ )  $\text{pc} \leq \text{pc} + \text{sim} + 2$ ;
- Opcode 0xc: **bges** *sim*
  - Name: Branch if Greater than or Equal Signed
  - Effect: if ( $\neg (\text{flags.N XOR flags.V})$ )  $\text{pc} \leq \text{pc} + \text{sim} + 2$ ;
- Opcode 0xd: **blts** *sim*
  - Name: Branch if Less Than Signed
  - Effect: if ( $\text{flags.N XOR flags.V}$ )  $\text{pc} \leq \text{pc} + \text{sim} + 2$ ;
- Opcode 0xe: **bgts** *sim*
  - Name: Branch if Greater Than Signed
  - Effect: if ( $\neg (\text{flags.N XOR flags.V}) \text{ AND } \neg \text{flags.Z}$ )  $\text{pc} \leq \text{pc} + \text{sim} + 2$ ;
- Opcode 0xf: **bles** *sim*
  - Name: Branch if Less than or Equal Signed
  - Effect: if ( $(\text{flags.N XOR flags.V}) \text{ OR flags.Z}$ )  $\text{pc} \leq \text{pc} + \text{sim} + 2$ ;

### 3.5 Instruction Group 4

The following encoding is used, with each character representing one bit:

100o oooo bbbb aaaa, where

- o is the opcode
- b encodes register **rB** or register **sC**, where **sC** is one of the special-purpose registers
- a encodes register **rA** or register **sA**, where **sA** is one of the special-purpose registers

Here is a list of instructions from this encoding group.

- Opcode 0x0: **j1 rA**
  - Effect:  $lr \leq pc + 2$ ;  $pc \leq rA$ ;
- Opcode 0x1: **jmp rA**
  - Effect:  $pc \leq rA$ ;
- Opcode 0x2: **jmp ira**
  - Effect:  $pc \leq ira$ ;
- Opcode 0x3: **reti**
  - Effect: enables IRQs (by copying 0x1 into **ie**) and performs  $pc \leq ira$ ;
- Opcode 0x4: **ei**
  - Effect: copy 1 into **ie**.
- Opcode 0x5: **di**
  - Effect: copy 0 into **ie**.
- Opcode 0x6: **push rA, rB**
  - Effect: pushes **rA** onto the stack, using **rB** as the stack pointer, post-decrementing **rB**.
  - This instruction does nothing when **rA** is the same register as **rB**.
  - Note: As a pseudo instruction, omitting ", **rB**" will automatically select **sp** as the particular stack pointer.
- Opcode 0x7: **push sA, rB**
  - Effect: pushes **sA** onto the stack, using **rB** as the stack pointer, post-decrementing **rB**.
  - Note that **sA** is considered to be 32-bit for the purpose of the store to memory and decrementing **rB**, even if **sA** is **flags** or **ie**.
  - Note: As a pseudo instruction, omitting ", **rB**" will automatically select **sp** as the particular stack pointer.
- Opcode 0x8: **pop rA, rB**
  - Effect: pops **rA** off the stack, using **rB** as the stack pointer, pre-incrementing **rB**.
  - This instruction does nothing when **rA** is the same register as **rB**.

- Note: As a pseudo instruction, omitting ", rB" will automatically select **sp** as the particular stack pointer.
- Opcode 0x9: **pop sA, rB**
  - Pseudo instruction with same effect:
    - \* **ldrib sA, rB**
    - \* Name explanation: load register, increment base before
  - Effect: pops **sA** off the stack, using **rB** as the stack pointer, pre-incrementing **rB**.
  - Note that **sA** is considered to be 32-bit for the purpose of the load from memory and incrementing **rB**, even if **sA** is **flags** or **ie**.
  - Note: As a pseudo instruction, omitting ", rB" will automatically select **sp** as the particular stack pointer.
- Opcode 0xa: **index rA**
  - Effect: Performs `<index_reg> <= rA;` and stores that **index** is in effect.
  - Note: If **index** is in effect and the current instruction is **index**, the current instruction will be treated as a NOP, and **index** will stop being in effect.
  - Note: **pre** and **index** can be combined with one another (though this is only useful for **ldr** and **str**).
  - Note: A non-**pre** instruction following **index** will store that that **index** is not in effect any more. (It will also store that **pre** is not in effect any more).
  - Note: If **index** is in effect, the current instruction cannot be interrupted by an IRQ.
  - Note: Any time **index** stops being in effect, **pre** will stop being in effect as well.
- Opcode 0xb: **mul rA, rB**
  - Effect: `rA <= rA * rB;`
- Opcode 0xc: **udiv rA, rB**
  - Effect: `rA <= u32(rA) / u32(rB);`
- Opcode 0xd: **sdiv rA, rB**
  - Effect: `rA <= s32(rA) / s32(rB);`
- Opcode 0xe: **umod rA, rB**
  - Effect: `rA <= u32(rA) % u32(rB);`

- Opcode 0xf: **smod** rA, rB
  - Effect:  $rA \leftarrow s32(rA) \% s32(rB);$
- Opcode 0x10: **lumul** rA, rB
  - Effect: This instruction multiplies rA by rB, performing an unsigned 32-bit by 32-bit  $\rightarrow$  64-bit multiply, storing result in {r0, r1}.
- Opcode 0x11: **lsmul** rA, rB
  - Effect: This instruction multiplies rA by rB, performing a signed 32-bit by 32-bit  $\rightarrow$  64-bit multiply, storing result in {r0, r1}.
- Opcode 0x12: **ludiv** rA, rB
  - Effect: performs a 64-bit by 64-bit unsigned division of {r0, r1} by {rA, rB}, storing 64-bit result in {r0, r1}.
  - Note: This instruction executes more quickly if rA's value is 0x00000000, i.e. if the operation is actually a 64-bit by 32-bit  $\rightarrow$  64-bit unsigned divide.
- Opcode 0x13: **lsdiv** rA, rB
  - Effect: performs a 64-bit by 64-bit signed division of {r0, r1} by {rA, rB}, storing 64-bit result in {r0, r1}.
  - Note: This instruction executes more quickly if rA is equal to bits [63:32] of `sign_extend_to_64(rB)`.
- Opcode 0x14: **lumod** rA, rB
  - Effect: performs a 64-bit by 64-bit unsigned modulo of {r0, r1} by {rA, rB}, storing 64-bit result in {r0, r1}.
  - Note: This instruction executes more quickly if rA's value is 0x00000000, i.e. if the operation is actually a 64-bit by 32-bit  $\rightarrow$  64-bit unsigned modulo.
- Opcode 0x15: **lsmod** rA, rB
  - Effect: performs a 64-bit by 64-bit signed modulo of {r0, r1} by {rA, rB}, storing 64-bit result in {r0, r1}.
  - Note: This instruction executes more quickly if rA is equal to bits [63:32] of `sign_extend_to_64(rB)`.
- Opcode 0x16: **ldub** rA, [rB]
  - Effect: Load an 8-bit value from memory at address computed as rB + <index\_reg>, zero-extend the 8-bit value to 32 bits, then put the zero-extended 32-bit value into rA.



- The `<index_reg>` value is guaranteed to be zero unless an `index` is in effect.
- Shorthand for having the assembler insert an `index rC` instruction before this one: `ldub rA, [rB, rC]`
- Opcode 0x17: `ldsb rA, [rB]`
  - Effect: Load an 8-bit value from memory at address computed as `rB + <index_reg>`, sign-extend the 8-bit value to 32 bits, then put the sign-extended 32-bit value into `rA`.
  - The `<index_reg>` value is guaranteed to be zero unless an `index` is in effect.
  - Shorthand for having the assembler insert an `index rC` instruction before this one: `ldsb rA, [rB, rC]`
- Opcode 0x18: `lduh rA, [rB]`
  - Effect: Load a 16-bit value from memory at address computed as `rB + <index_reg>`, zero-extend the 16-bit value to 32 bits, then put the zero-extended 32-bit value into `rA`.
  - The `<index_reg>` value is guaranteed to be zero unless an `index` is in effect.
  - Shorthand for having the assembler insert an `index rC` instruction before this one: `lduh rA, [rB, rC]`
- Opcode 0x19: `ldsh rA, [rB]`
  - Effect: Load a 16-bit value from memory at address computed as `rB + <index_reg>`, sign-extend the 16-bit value to 32 bits, then put the zero-extended 32-bit value into `rA`.
  - The `<index_reg>` value is guaranteed to be zero unless an `index` is in effect.
  - Shorthand for having the assembler insert an `index rC` instruction before this one: `ldsh rA, [rB, rC]`
- Opcode 0x1a: `stb rA, [rB]`
  - Effect: Store `rA[7:0]` to memory at the address computed as `rB + <index_reg>`.
  - The `<index_reg>` value is guaranteed to be zero unless an `index` is in effect.
  - Shorthand for having the assembler insert an `index rC` instruction before this one: `stb rA, [rB, rC]`
- Opcode 0x1b: `sth rA, [rB]`

- Effect: Store `rA[15:0]` to memory at the address computed as `rB + <index_reg>`.
- The `<index_reg>` value is guaranteed to be zero unless an `index` is in effect.
- Shorthand for having the assembler insert an `index rC` instruction before this one: `sth rA, [rB, rC]`
- Opcode 0x1c: `cpy rA, sC`
  - Effect: `rA <= sC`;
- Opcode 0x1d: `cpy sA, rB`
  - Effect: `sA <= rB`;
- Opcode 0x1e: `cpy sA, sB`
  - Effect: `sA <= sB`;

### 3.6 Instruction Group 5: Immediate Indexed Load

The following encoding is used, with each character representing one bit:

101i iiii bbbb aaaa, where

- `i` is a 5-bit sign-extended immediate, which can be expanded by `pre`, and is denoted `simm`
- `b` encodes register `rB`
- `a` encodes register `rA`

The one instruction from this encoding group is `ldr rA, [rB, #simm]`. This is a 32-bit load into `rA`, where the effective address to load from is computed as `rB + <index_reg> + simm`, using the sign-extended form of `simm`.

The `<index_reg>` value is guaranteed to be zero unless an `index` is in effect.

Shorthand for having the assembler insert an `index rC` instruction before this one: `ldr rA, [rB, rC, #simm]`

### 3.7 Instruction Group 6: Immediate Indexed Store

The following encoding is used, with each character representing one bit:

110i iiii bbbb aaaa, where

- i is a 5-bit sign-extended immediate, which can be expanded by `pre`
- b encodes register `rB`
- a encodes register `rA`

The one instruction from this encoding group is `str rA, [rB, #simm]`. This is a 32-bit store of `rA`, where the effective address to store to is computed as `<index_reg> + rB + simm`, using the sign-extended form of `simm`.

The `<index_reg>` value is guaranteed to be zero unless an `index` is in effect.

Shorthand for having the assembler insert an `index rC` instruction before this one: `str rA, [rB, rC, #simm]`

### 3.8 Instruction Group 7, Subgroup 0b00: Extra 8-bit and 16-bit Ops

The following encoding is used, with each character representing one bit:

1110 0woo bbbb aaaa, where

- w is the operation width
  - When 0b0: 8-bit operation
  - When 0b1: 16-bit operation
- o is the opcode
- b encodes register `rB`
- a encodes register `rA`

Here is a list of instructions from this encoding group.

- Opcode 0x0:
  - w value 0b0: `cmpb rA, rB`

- \* Effect: Compare **rA[7:0]** to **rB[7:0]**
  - \* Affectable flags: Z, C, V, N
- **w** value 0b1: **cmph rA, rB**
  - \* Effect: Compare **rA[15:0]** to **rB[15:0]**
  - \* Affectable flags: Z, C, V, N
- Opcode 0x1:
  - **w** value 0b0: **lsrb rA, rB**
    - \* Effect: Logical shift right **rA[7:0]** by **rB**
  - **w** value 0b1: **lsrh rA, rB**
    - \* Effect: Logical shift right **rA[15:0]** by **rB**
- Opcode 0x2:
  - **w** value 0b0: **asrb rA, rB**
    - \* Effect: Arithmetic shift right **rA[7:0]** by **rB**
  - **w** value 0b1: **asrh rA, rB**
    - \* Effect: Arithmetic shift right **rA[15:0]** by **rB**

### 3.9 Instruction Group 7, Subgroup 0b010: Extra load/store instructions

The following encoding is used, with each character representing one bit:

1110 10oo bbbb aaaa, where

- o is the opcode
- b encodes register **sA**
- a encodes register **rB** or **sB**

Here is a list of instructions from this encoding group.

- Opcode 0x0: **ldr sA, [rB]**
  - Effect: 32-bit load of **sA** from memory at address held in **rB**.
- Opcode 0x1: **ldr sA, [sB]**
  - Effect: 32-bit load of **sA** from memory at address held in **sB**.
- Opcode 0x2: **str sA, [rB]**

- Effect: 32-bit store of **sA** to memory at address held in **rB**.
- Opcode 0x3: **str sA, [sB]**
  - Effect: 32-bit store of **sA** to memory at address held in **sB**.