

Flare32 CPU

FL4SHK

April 9, 2023

Table of Contents

Table of Contents	1
Introduction	2
2.1 Registers	2
Instruction Set	3
3.1 Instruction Group 0: pre and lpre	3
3.1.1 Handling of pre , lpre , and index	4
3.2 Instruction Group 1	5
3.3 Instruction Group 2	7
3.4 Instruction Group 3: Relative Branches	9
3.5 Instruction Group 4	11
3.6 Instruction Group 5: Immediate Indexed Load	16
3.7 Instruction Group 6: Immediate Indexed Store	16

Introduction

2.1 Registers

There are sixteen general-purpose registers: **r0**, **r1**, **r2**, ..., **r11**, **r12**, **lr**, **fp**, **sp**. Each register is 32 bits long. For special purpose registers, there are also **pc**, the program counter (which is 32 bits long), and the **flags**. Also there are the interrupts-related registers: **ids** (the destination to go to upon an interrupt happening), **ira** (the program counter value to return to after an interrupt) and **ie** (whether or not interrupts are enabled). Two more registers are **hi** and **lo**, which are used as the high 32 bits and low 32 bits of the result of a 32 by 32 -> 64 multiplication, or as the high 32 bits and low 32 bits of the result of a 64 by 64 -> 64 division. Here are the flags:

Table 1: The Flags

Zero (Z)	Carry (C)	oVerflow (V)	Negative (N)
----------	-----------	--------------	--------------

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, interrupts 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`
- Opcode 0x2: `add rA, sp, #simm`
- Opcode 0x3: `add rA, fp, #simm`
- Opcode 0x4: `cmp rA, #simm`
 - Note: Compare `rA` to `simm`.
 - Affectable flags: Z, C, V, N
- Opcode 0x5: `cpy rA, #simm`
 - Note: Copy an immediate value into `rA`
- Opcode 0x6: `lsl rA, #simm`
 - Note: Logical shift left
- Opcode 0x7: `lsr rA, #simm`
 - Note: Logical shift right
- Opcode 0x8: `asr rA, #simm`
 - Note: Arithmetic shift right
- Opcode 0x9: `and rA, #simm`
 - Note: Bitwise AND
- Opcode 0xa: `orr rA, #simm`
 - Note: Bitwise OR
- Opcode 0xb: `xor rA, #simm`
 - Note: Bitwise XOR
- Opcode 0xc: `zeb rA`
 - Effect: Set `rA[31:8]` to zero.

- Opcode 0xd: **zeh** *rA*
 - Effect: Set *rA*[31:16] to zero.
- Opcode 0xe: **seb** *rA*
 - Effect: Sign-extend *rA*[7:0] to 32 bits, then copy that value to *rA*
- Opcode 0xf: **seh** *rA*
 - Effect: Sign-extend *rA*[15:0] to 32 bits, then copy that value to *rA*

3.3 Instruction Group 2

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

010f oooo cccc aaaa, where

- o is the opcode
- c encodes register *rC*
- 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*, *rC*
 - Mnemonic for when flags not affected: **add**
 - Mnemonic for when flags affected: **add.f**
 - Affectable flags: Z, C, V, N
- Opcode 0x1: **sub** *rA*, *rC*
 - 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*, *rC*
 - Mnemonic for when flags not affected: **add**
 - Mnemonic for when flags affected: **add.f**
 - Affectable flags: Z, C, V, N

- Opcode 0x3: **add rA, rC**
 - Mnemonic for when flags not affected: **add**
 - Mnemonic for when flags affected: **add.f**
 - Affectable flags: Z, C, V, N
- Opcode 0x4: **cmp rA, rC**
 - Note: Compare **rA** to **rC**. **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, rC**
 - Mnemonic for when flags not affected: **cpy**
 - Mnemonic for when flags affected: **cpy.f**
 - Note: Copy **rC** into **rA**
 - Affectable flags: Z, N
- Opcode 0x6: **lsl rA, rC**
 - Mnemonic for when flags not affected: **lsl**
 - Mnemonic for when flags affected: **lsl.f**
 - Note: Logical shift left
 - Affectable flags: Z, N
- Opcode 0x7: **lsr rA, rC**
 - Mnemonic for when flags not affected: **lsr**
 - Mnemonic for when flags affected: **lsr.f**
 - Note: Logical shift right
 - Affectable flags: Z, N
- Opcode 0x8: **asr rA, rC**
 - Mnemonic for when flags not affected: **asr**
 - Mnemonic for when flags affected: **asr.f**
 - Note: Arithmetic shift right
 - Affectable flags: Z, N
- Opcode 0x9: **and rA, rC**
 - Mnemonic for when flags not affected: **and**
 - Mnemonic for when flags affected: **and.f**
 - Note: Bitwise AND

- Affectable flags: Z, N
- Opcode 0xa: **orr rA, rC**
 - Mnemonic for when flags not affected: **orr**
 - Mnemonic for when flags affected: **orr.f**
 - Note: Bitwise OR
 - Affectable flags: Z, N
- Opcode 0xb: **xor rA, rC**
 - Mnemonic for when flags not affected: **xor**
 - Mnemonic for when flags affected: **xor.f**
 - Note: Bitwise XOR
 - Affectable flags: Z, N
- Opcode 0xc: **adc rA, rC**
 - Mnemonic for when flags not affected: **adc**
 - Mnemonic for when flags affected: **adc.f**
 - Note: Add with Carry
 - Affectable flags: Z, C, V, N
- Opcode 0xd: **sbc rA, rC**
 - Mnemonic for when flags not affected: **sbc**
 - Mnemonic for when flags affected: **sbc.f**
 - Note: 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 **simm**
- o is the opcode

Here is a list of instructions from this encoding group.

- Opcode 0x0: **bl simm**

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

- o is the opcode
- c encodes register **rC** or register **sC**, where **sC** is one of **hi**, **lo**, **flags**, **ira**, **ids**, or **ie**
- a encodes register **rA** or register **sA**, where **sA** is one of **hi**, **lo**, **flags**, **ira**, **ids**, or **ie**

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 rA, rC**
 - Effect: $pc \leq (rA + rC)$;
- Opcode 0x3: **jmp ira**
 - Effect: $pc \leq ira$;
- Opcode 0x4: **reti**
 - Effect: enables interrupts (by copying 1 into **ie**) and performs $pc \leq ira$;
- Opcode 0x5: **cpy rA, sC**
 - Effect: $rA \leq sC$;
- Opcode 0x6: **cpy sA, rC**
 - Effect: $sA \leq rC$;
- Opcode 0x7: **ei**
 - Effect: copy 1 into **ie**.
- Opcode 0x8: **di**
 - Effect: copy 0 into **ie**.
- Opcode 0x9: **push rA, rC**
 - Effect: pushes **rA** onto the stack, using **rC** as the stack pointer, post-decrementing **rC**.
 - This instruction does nothing when **rA** is the same register as **rC**.
- Opcode 0xa: **pop rA, rC**
 - Effect: pops **rA** off the stack, using **rC** as the stack pointer, pre-incrementing **rC**.
 - This instruction does nothing when **rA** is the same register as **rC**.
- Opcode 0xb: **push sA, rC**

- Effect: pushes **sA** onto the stack, using **rC** as the stack pointer, post-decrementing **rC**.
 - Note that **sA** is considered to be 32-bit for the purpose of the store to memory and decrementing **rC**, even if **sA** is **flags** or **ie**.
- Opcode 0xc: **pop sA, rC**
 - Effect: pops **sA** off the stack, using **rC** as the stack pointer, pre-incrementing **rC**.
 - Note that **sA** is considered to be 32-bit for the purpose of the load from memory and incrementing **rC**, even if **sA** is **flags** or **ie**.
- Opcode 0xd: **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.
 - Note: Any time **index** stops being in effect, **pre** will stop being in effect as well.
- Opcode 0xe: **mul rA, rC**
 - Effect: `rA <= rA * rC;`
- Opcode 0xf: **udiv rA, rC**
 - Effect: `rA <= u32(rA) / u32(rC);`
- Opcode 0x10: **sdiv rA, rC**
 - Effect: `rA <= s32(rA) / s32(rC);`
- Opcode 0x11: **umod rA, rC**
 - Effect: `rA <= u32(rA) % u32(rC);`
- Opcode 0x12: **smod rA, rC**
 - Effect: `rA <= s32(rA) % s32(rC);`

- Opcode 0x13: **lumul rA, rC**
 - Effect: This instruction multiplies **rA** by **rC**, performing an unsigned 32-bit by 32-bit -> 64-bit multiply, storing result in {hi, lo}.
- Opcode 0x14: **lsmul rA, rC**
 - Effect: This instruction multiplies **rA** by **rC**, performing a signed 32-bit by 32-bit -> 64-bit multiply, storing result in {hi, lo}.
- Opcode 0x15: **ludiv rA, rC**
 - Effect: performs a 64-bit by 64-bit unsigned division of {hi, lo} by {rA, rC}, storing 64-bit result in {hi, lo}.
 - 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 -> 64-bit unsigned divide.
- Opcode 0x16: **lsdiv rA, rC**
 - Effect: performs a 64-bit by 64-bit signed division of {hi, lo} by {rA, rC}, storing 64-bit result in {hi, lo}.
 - Note: This instruction executes more quickly if **rA** is equal to bits [63:32] of **sign_extend_to_64(rC)**.
- Opcode 0x17: **lumod rA, rC**
 - Effect: performs a 64-bit by 64-bit unsigned modulo of {hi, lo} by {rA, rC}, storing 64-bit result in {hi, lo}.
 - 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 -> 64-bit unsigned modulo.
- Opcode 0x18: **lsmod rA, rC**
 - Effect: performs a 64-bit by 64-bit signed modulo of {hi, lo} by {rA, rC}, storing 64-bit result in {hi, lo}.
 - Note: This instruction executes more quickly if **rA** is equal to bits [63:32] of **sign_extend_to_64(rC)**.
- Opcode 0x19: **ldub rA, [rC]**
 - Effect: Load an 8-bit value from memory at address computed as **rC** + <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 rB** instruction before this one: **ldub rA, [rC, rB]**

- Opcode 0x1a: **ldsb** rA, [rC]
 - Effect: Load an 8-bit value from memory at address computed as rC + <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** rB instruction before this one: **ldsb** rA, [rC, rB]
- Opcode 0x1b: **lduh** rA, [rC]
 - Effect: Load a 16-bit value from memory at address computed as rC + <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** rB instruction before this one: **lduh** rA, [rC, rB]
- Opcode 0x1c: **ldsh** rA, [rC]
 - Effect: Load a 16-bit value from memory at address computed as rC + <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** rB instruction before this one: **ldsh** rA, [rC, rB]
- Opcode 0x1d: **stb** rA, [rC]
 - Effect: Store rA[7:0] to memory at the address computed as rC + <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** rB instruction before this one: **stb** rA, [rC, rB]
- Opcode 0x1e: **sth** rA, [rC]
 - Effect: Store rA[15:0] to memory at the address computed as rC + <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** rB instruction before this one: **sth** rA, [rC, rB]

3.6 Instruction Group 5: Immediate Indexed Load

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

101i iiii cccc aaaa, where

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

The one instruction from this encoding group is `ldr rA, [rC, #simm]`. This is a 32-bit load into `rA`, where the effective address to load from is computed as `rC + <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 rB` instruction before this one: `ldr rA, [rC, rB, #simm]`

3.7 Instruction Group 6: Immediate Indexed Store

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

110i iiii cccc aaaa, where

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

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