Little Finger Assembly Manual

Developer Manual

Ezekiel Elin

October 22, 2017

Writing Assembly and LFA Design

High-level components

Little Finger Assembly (LFA) is a simple collection of instructions, directives, and labels, combined to create a basic example of an instruction set architecture.

Instructions make up the functional part of an assembly program, providing the core functionality. LFA includes instructions for arithmetic and logical manipulation of binary numbers, instructions for loading and storing values in main memory, and instructions for control flow.

Directives are solely part of the assembling process and do not remain in the assembly file in their original form). Directives tell the assembler what to do, and support setting configuration options and storing raw data directly into main memory.

Labels are the last high level component of LFA, and provide easy ways to organize the assembly program. Labels are converted into byte addresses during the assembly process, and the human-readable string is not saved.

Instructions

Instructions are the most important part of any LFA assembly program. Instructions include operations like `ADD` or `SUB`, which are fundamental to basic data manipulation. Instructions fall into seven categories, depending on the format they are written in.

Instruction Formats

All instructions start with a capitalized instruction name, followed by 0-3 parameters (depending on the instruction) separated by whitespace and/or commas.

A generic instruction would follow the following format:

```
OPER [params…]
```

Parameters are generally either registers, numerical literals, or labels. However, the syntax may change from format to format, so refer to specific formats for the required syntax.

A-format

A-format instructions are used for some arithmetic operations involving three registers. These instructions take the following format:

```
OPER <REGISTER 1> <REGISTER 2> <REGISTER 3>
```

Generally, these instructions are roughly equated the following pseudocode:

```
<REGISTER 1> = <REGISTER 2> OPER <REGISTER 3>
```

B-format

B-format instructions are used for some arithmetic operations involving two registers and a numerical literal. These instructions take the following format:

```
OPER <REGISTER 1> <REGISTER 2> <LITERAL 1>
OPER <REGISTER 1> <LITERAL 1> <REGISTER 2>
```

Generally, these instructions are roughly equated the following pseudocode (respectively):

```
<REGISTER 1> = <REGISTER 2> OPER <LITERAL 1>
<REGISTER 1> = <LITERAL 1> OPER <REGISTER 2>
```

C-format

C-format instructions are used for instructions with no parameters and take the following format:

```
OPER
```

D-format

D-format instructions are used for instructions manipulating main memory, involving two registers and a numerical literal. These instructions differ from B-format instructions because the literal may not be moved around. They also have a slightly modified syntax:

```
OPER <REGISTER 1> [<REGISTER 2>, <LITERAL 1>]
```

Note that the square brackets are part of the instruction, whereas the angled brackets are indicating an item to be replaced.

E-format

E-format instructions are used for instructions involving a register and a memory address in the form of a label. These instructions are primarily used for some branching operations (although not exclusively) and take the following format:

```
OPER <REGISTER 1> <LABEL>
```

G-format

G-format instructions are used for instructions involving just a memory address in the form of a label. These instructions are used for branching operations, and take the following format:

```
OPER <LABEL>
```

H-format

H-format instructions are used for instructions involving a single register. These instructions are used for stack modification and some branching operations. They take the following format:

```
OPER <REGISTER 1>
```

Instruction List

Instruction Name	Format	Opcode (decimal)	Example	Pseudocode Eq.	Notes
NOP	C	0	`NOP`	`N/A`	
ADD	A	1	`ADD R1, R2, R3`	`R1 = R2 + R3`	
SUB	A	2	`SUB R2, R2, R3`	`R2 = R2 − R3`	
ADDI	B	3	`ADDI R0, R2, #8`	`R0 = R2 + 8`	Literal may be on either side of operand
SUBI	B	4	`SUBI R0, #16, R1`	`R0 = 16 − R1`	Literal may be on either side of operand
ADDS	A	5	`ADD R1, R2, R3`	`R1 = R2 + R3`	Sets the flags based on the result
SUBS	A	6	`SUB R2, R2, R3`	`R2 = R2 − R3`	
ADDIS	B	7	`ADDIS R0, #15, R3`	`R0 = 15 + R3`	
SUBIS	B	8	`SUBIS R1, R1, #3`	`R1 = R1 − 3`	
AND	A	21	`AND R3, R1, R2`	`R3 = R1 & R2`	
ORR	A	22	`ORR R2, R3, R2`	`R2 = R3 | R2`	
EOR	A	23	`EOR R4, R0, R3`	`R4 = R0 ^ R3`	
ANDI	B	24	`ANDI R3, R3, #4`	`R3 = R3 & 4`	Literal may be on either side of operand

Instruction Name	Format	Opcode (decimal)	Example	Pseudocode Eq.	Notes
ORRI	B	25	`ORRI R2, R2, #2`	`R2 = R2 | 2`	Literal may be on either side of operand
EORI	B	26	`EORI R0, R1, #9`	`R0 = R1 ^ 9`	Literal may be on either side of operand
LSL	B	27	`LSL R3, R2, #3`	`R3 = R2 << 3`	Fills in with 0
LSR	B	28	`LSR R2, R2, #1`	`R2 = R2 >> 1`	Fills in with 0
LDUR	D	9	`LDUR R0 [R2, #16]`	`R0 = &R2[2]`	Loads and stores 8 byte numbers
STUR	D	10	`STUR R3 [R1, #8]`	`&R1[1] = R3`	
LDURSW	D	11	See LDUR		Loads and stores 4 byte numbers
STURW	D	12	See STUR		
LDURH	D	13	See LDUR		Loads and stores 2 byte numbers
STURH	D	14	See STUR		
LDURB	D	15	See LDUR		Loads and stores single byte numbers
STURB	D	16	See STUR		
CBZ	E	29	`CBZ R3, label`	`if (R3 == 0)` `  goto label`	
CBNZ	E	30	`CBNZ R2, place`	`if (R3 != 0)` `  goto place`	
B	G	31	`B label`	`goto label`	
BR	H	32	`BR R2`	`goto R2`	Branches to the location specified by the value of the register
BL	G	33	`BL label2`	`goto label2`	Branches, and sets `LINK` register to first byte of next instruction prior to branching
B.EQ	G	34	`B.EQ label`	`if (Z == 1)` `  goto label`	See the flags section, which specifies what the different letters refer to
B.NE	G	35	`B.NE label`	`if (Z == 0)` `  goto label`	
B.LT	G	36	`B.LT label`	`if (N != V)` `  goto label`	

Instruction Name	Format	Opcode (decimal)	Example	Pseudocode Eq.	Notes
B.LE	G	37	`B.LE label`	`if (!(Z == 0 & N == V)) goto label`	
B.GT	G	38	`B.GT label`	`if (Z == 0 & N == V) goto label`	
B.GE	G	39	`B.GE label`	`if (N == V) goto label`	
B.MI	G	40	`B.MI label`	`if (N == 1) goto label`	
B.PL	G	41	`B.PL label`	`if (N == 0) goto label`	
B.VS	G	42	`B.VS label`	`if (V == 1) goto label`	
B.VC	G	43	`B.VC label`	`if (V == 0) goto label`	
PUSH	H	44	`PUSH R0`	`R0 = &SP[0]` `SP = SP + 8`	SP refers to the stack pointer. Refer to the stack section for more info.
POP	H	45	`POP R1`	`SP = SP - 8` `R1 = &SP[0]`	
MOVZ	E	46	`MOVZ R3 label`	`R3 = label`	Stores the address of the label in the register
HALT	C	47	`HALT`	`N/A`	Ends the simulation

Registers

Registers may either refer to a general-use register or to one of the two special registers.

Caveat: The two special purpose registers replace `R30` and `R31`, and will make those registers unaccessible even if your `.regcnt` directive specifies that you have that many registers.

General Use Registers

General use registers start at zero, counting up. The number of general use registers is configured by the `.regcnt` directive. The first letter of the sequence is discarded, however convention is to use an upper-case `R`. The second two examples work equally as well, however.

```
R0
R9
X4
x1
```

Zero Register

The zero register, specified with `ZERO`, always has the numerical value `0`, and will not be modified even if an instruction attempt to write to that register. The zero register is recommended as a location to write out unwanted numbers (for example, in a flag-setting operation, the result is usually not needed and can be stored in `ZERO` safely).

Link Register

The link register, specified with `LINK`, has the value of the first byte of the next instruction after a BL instruction. Therefor, after executing a `BL` instruction, at any point the `BR` instruction can be used to return and execute the next instruction after the `BL`. Here is an example, with the order of execution marked with the numbers on the left-hand side:

```
[1]> BL somewhere
[3]> HALT
[2]> somewhere: BR LINK
```

The Stack, Push, and Pop

The PUSH and POP instructions are aliases to other instructions, and use the stack pointer as the location around which those instructions are performed.

If a `stack:` label is specified, then it will be used as the top of the stack. If a stack label is not specified, then the top of the stack will start at the end of the last byte inserted into main memory before it is filled and saved.

`PUSH R0` functions as:

```
STUR R0, [stack:, #0]
ADDI stack:, stack:, #8
```

`POP R0` functions as:

```
SUBI stack:, stack:, #8
LDUR R0, [stack: #0]
```

Note: In this example, the stack pointer is indicated using the label syntax (`stack:`). Note that labels are discarded after running the assembler, and the implementation of this uses a special register that is not otherwise accessible.

Directives

Directives are two-piece components. The directive name and the numerical value. The directive name is case-sensitive and starts with a period. The directive value is always numerical and can be specified in base 16, base 10, base 8, or base 2.

Example directive values:

- hex: `0xF923AB`
- decimal: `12394`
- octal: `0o012735`
- binary: `0b01010111`

Word Size Directive

```
.wordsize <word size>
```

The word size directive specifies the number of bits in a register. For example, if the word size is set to 16, then a register will hold 16 bits. If more than 16 bits are stored in the register, the most significant bits will be removed so that the register fits within the specified size.

Register Count Directive

```
.regcnt <register count>
```

The register count specifies the number of general purpose registers. It is important to note that there are some caveats to register counts above 30, specified in the register section.

Max Memory Directive

`.maxmem <max memory>`

The max memory directive specifies the number of bytes in the memory image.

Warning: The memory image outputted by the assembler will have this many bytes, defaulting to zero.

Warning: Bytes are stored in Java ArrayLists, which are indexed with *integers*. Numbers above the size of the Java Integer max value may behave unexpectedly.

Align Directive

`.align <alignment>`

Pads the memory with zeros until the current location is evenly divisible by the alignment number.

Position Directive

`.pos <position>`

Pads the memory with zeros until the current location is equal to or greater than the position number.

Data Directives

There are four data directives, each allowing a certain width (number of bytes) number to be stored directly into main memory, at that location.

Byte

`.byte <number>`

Stores a 1-byte number in main memory.

Half

`.half <number>`

Stores a 2-byte number in main memory.

Single

`.single <number>`

Stores a 4-byte number in main memory.

Double

`.double <number>`

Stores a 8-byte number in main memory.

Labels

`labelName:`

Labels are used to mark locations in memory. They can then be used in conjunction with certain instructions, like `MOVZ` and `B`.

Labels are specified with the label name (user specified), then a colon. Labels may not start with a period or contain commas or whitespace.

To use a label, the name without the colon should be specified in an instruction which refers to a label. The instruction list table uses “label” to refer to any label.

Unused labels will be discarded and do not impact the memory image.

Flags

While flags are not part of the assembler, they are important for understanding some Assembler features. Some operations (usually specified with a trailing `S` in the instruction name, for example `SUBS`) will set CPU flags based on the result.

N: Negative

The negative flag will be set to *true* if the result of the operation is negative.

Z: Zero

The zero flag will be set to *true* if the result of the operation is `0`.

C: Carry

The carry flag will be set to *true* if a carry occurred out of the most significant bit, or if a borrow occurs on the most significant bit.

V: Overflow

The overflow flag will be set to *true* if the operation overflowed.

The distinction between carry and overflow is a fine one, however the basics are that a carry is not inherently bad, whereas an overflow is. If your number overflows, the sign changes, for example. In contrast, a carry is supposed to happen in some operations, and the resulting number is still correct.

Flags impact conditional branch operations. See the instruction list for some examples.

Example Program

```
.wordsize  64              ; set word size to 64
.regcnt    8               ; set register count to 8
.maxmem    1024            ; set max memory to 1024

start:
MOVZ R0, someData          ; store someData into R0
LDUR R1 [R0, #0]           ; load the 8-byte value at R0 (someData)
                           ; into R1

LSL R1, R1, #1             ; shift R1 left 1, and store back
                           ; (same as multiplying by 2)

STUR R1 [R0, #0]           ; store the modified value back into R0

SUBIS ZERO R1, #7544       ; do R1-7544 and set flags, discard result

B.LT start                 ; branch back to start if R1 < 7544
HALT                       ; stop execution

someData:
.double 943                ; 943 as 8-byte number
```

Memory Image File Binary Format Specification

The memory image file saved by the assembler and by the simulator is a binary file storing the bytes from the memory object.

When grouped bytes (for example, a 4 byte instruction) are written to the memory, the most significant byte of the instruction is stored at the high byte index the least significant byte is stored at the low byte index. This means that `0xABCD` would be stored as

```
0x0 0xCD
0x1 0xAB
```

When a collection of bytes is read from the memory image, the “address” of the collection is the low byte index, which points to the least significant byte in the collection (for example, the above example is a two-byte collection at location `0x0`, and is pulled out as `0xABCD`).

Designing the byte collections in this way allows bit indexing to be intuitive. The first bit (bit zero) is the least significant bit of the first byte in the collection, as well as the least significant bit of the collection of bytes that the first byte belongs to. The 9th bit (bit eight) is the least significant bit of the second byte, and the 9th byte (from the right) of the collection of bytes in (at minimum) the first and second byte.

While unintuitive to read directly, this system allows easy indexing of both bits and bytes. Using the included memory image reader program, they are rearranged to an easily readable format.

Program Configuration

The first four bytes of the file are the 32 configuration bits and are used to store the register count, word size, and max memory directives, as well the stack pointer.

Total: 32 bits

- $\log_2$ of the word size: 3 bits
- $\log_2$ of the register count: 3 bits
- $\log_2$ of the max memory: 6 bits
- Stack pointer: 20 bits

In the following spec:

- `A`: Word size bits
- `B`: Register count bits
- `C`: Max memory bits
- `D`: Stack pointer bits

`Configuration-format`

```
AAAB BBCC CCCC DDDD DDDD DDDD DDDD DDDD
```

Note: alternating segments are underlined for ease of reading

Instructions

Instructions, like the configuration, are also stored as 32 bits

Total: 32 bits

- Opcode: 7 bits
- Registers: 5 bits each
- Literals: 14 bits each
- Pointer Literals: 20 bits each

In the following specs:

- A: OPCODE bits
- B: Destination register bits
- C: Source register a bits
- D: Source register b bits
- E: Source literal
- F: Source literal order flag
- P: Pointer literal
- 0: Discarded bits

A-format

```
AAAA AAAB BBBB CCCC CDDD DD00 0000 0000
```

B-format

```
AAAA AAAB BBBB FCCC CCEE EEEE EEEE EEEE
```

C-format

```
AAAA AAA0 0000 0000 0000 0000 0000 0000
```

D-format

```
AAAA AAAB BBBB CCCC CEEE EEEE EEEE EEE0
```

E-format

```
AAAA AAAB BBBB PPPP PPPP PPPP PPPP PPPP
```

G-format

```
AAAA AAAP PPPP PPPP PPPP PPPP PPP0 0000
```

H-format

```
AAAA AAAB BBBB 0000 0000 0000 0000 0000
```

Instruction Format Rationale

The instruction format is deigned to be consistent, while still providing all the functionality needed for LFA. By making the first 7 bits the Opcode, space is provided for 128 different operations (currently 47 are in use). The remaining bits can then be parsed in accordance with the opcode found.

Registers occupy 5 bits to allow addressing up to 32 different registers.

Immediate values occupy 14 bits as a result of the left over bits in the B-format instruction format. The F flag, seen at the 13th bit (bit twelve) of the B-format, is used to swap the literal and register. Including this flag means only 14 bits are left over for the immediate value.

Pointers occupy 20 bits for similar reasons. In the E-format instruction, only 20 bits are available when 12 are used for the opcode and register. 20 bits is used in the remaining instructions, even when more space is available, to preserve consistency.

Using Java Assembler and Simulator

Compiling

All three parts of the project can be compiled at the same time, and reside together. The project can be compiled by running any of the following while at the top of the project directory:

```
make compile
```

```
javac -d build src/*/*.java
```

The `make` will clear the build directory, then perform the same `javac` command listed here. The compiled Java class files will be stored in the build directory.

Running the Assembler

The assembler is not interactive. Refer to the Writing Assembly section for information on how to format the input file, including an example.

The assembler will output any errors to the console. If verbose is set to true (false by default) then more information will be logged about the assembly process.

The assembler can be run using either of the following options while at the top of the project directory:

```
make assemble
     assembly-file=<assembly path>
     memory-image=<output path>
     [verbose=<true | false>]
java -classpath build assembler.Assembler
     <assembly path>
     <output path>
     [true | false]
```

Note: `<angled brackets>` indicate required parameters, `[square brackets]` indicate optional parameters that may be omitted.

Note: newlines are used in the above examples for readability, and should not be included when executing the commands.

The Simulator

The simulator will display it's status in the GUI as well as in the console, if verbose is set to true (false by default). Additional logging will also output to the console when verbose is true, that is not visible in the GUI.

The main window of the simulator shows the simulator state. This can be saved to a text file by pressing the Save button.

The simulator will also open a viewer window, which shows the state of the memory and updates after every step occurs.

The simulator can be manually stepped by pressing the Step button. The step will occur immediately and update both the reader and the simulator windows. The simulator can also automatically step by checking the checkbox labeled Auto-step. The time can be adjusted with the slider (in seconds).

Note: The auto-step option will wait 2 seconds before activating, and can be disabled during this time.

Note: The 0 second option runs at 20 Hz

The Halt button will perform the same action as a `HALT` instruction.

Running

The simulator can be run using either of the following options while at the top of the project directory:

```
make simulator
     [verbose=<true | false>]
java –classpath build simulator.SimulatorController
     [true | false]
```

Note: `<angled brackets>` indicate required parameters, `[square brackets]` indicate optional parameters that may be omitted.

Note: newlines are used in the above examples for readability, and should not be included when executing the commands.

The Viewer

The viewer shows a memory image.

When run as a standalone program, press the reload button and choose a file. Subsequent presses to reload will read from the same file.

When run attached to a simulator instance, the reload button will force a refresh. This is usually not needed, however can be used to show the initial state of the simulator, as it does not update until after a step has occurred. The save button can be used to save the modified state of the simulator after instructions have been executed.

The simulator shows bytes grouped in groups of four, with the address of the least significant byte displayed on the leftmost side.

Warning: If `.half` or `.byte` directives are used, instructions can become misaligned, and the least significant byte of the instruction will no longer be the one displayed. Use the `.align` directive to resolve this.

The user can scroll to view a certain area of the memory, referring to the indices shown on the leftmost column.

The user can enter a byte address (least significant byte) of an instruction to view the parsed value of that instruction. This is useful for debugging.

Note: the simulator will show labels as addresses, not as the original text.

Running

The viewer can be run using either of the following options while at the top of the project directory:

```
make reader
```

```
java –classpath build reader.Reader
```

Modifying and Understanding Java Source

The project is divided up into 5 java packages, represented as folders on disk.

The `src/assembler/` directory contains the files most directly relevant to the assembler, including the entry point and main assembly loop.

The `src/common/` directory contains most general tools and utilities. This includes classes which can perform manual binary arithmetic, as well as tools to perform String and Byte manipulation.

The `src/instructions/` directory contains the classes for all the instruction formats as well as all the instruction implementations.

The `src/reader/` directory contains the code for the reader.

The `src/simulator/` directory contains the code for the simulator.

Entry Points

The entry points for the three main parts of the project are:

Assembler	`src/assembler/Assembler.java`.
Viewer	`src/reader/Reader.java`.
Simulator	`src/simulator/SimulatorController.java`.