CSE 151A HW 01

Karlo Godfrey Escalona Gregorio
PID: A16536865

Contents

1 Preface 1

2 ISA Design 1

2.1 RX-type . . . 2
2.1.1 Overview . . . 2
2.1.2 add/sub . . . 4
2.1.3 mul/div . . . 7
2.1.4 mac . . . 8
2.1.5 sqrt . . . 9
2.1.6 convf . . . 9
2.1.7 cmp . . . 9
2.1.8 Miscellaneous Notes: RX-type Instructions . . . 9
2.2 RF-type . . . 10
2.2.1 Overview . . . 10
2.2.2 add/sub . . . 13
2.2.3 mul . . . 13
2.2.4 div . . . 13
2.2.5 mac . . . 13
2.2.6 sqrt . . . 13
2.2.7 convx . . . 13
2.2.8 cmp . . . 13
2.3 D-type . . . 13
2.3.1 Overview . . . 13
2.3.2 ldr . . . 15
2.3.3 str . . . 15
2.4 B-type . . . 15
2.4.1 Overview . . . 15
2.4.2 bx . . . 16
2.4.3 b . . . 16
2.4.4 bl . . . 16
2.5 Miscellaneous Notes . . . 16

3 Assembler 18

4 Instruction Memory 19

5 Program Counter 19

6 Program Counter Adder 19

7 Register File 19

8 Data Memory 19

9 ALU 19

10 ALU Control 19

11 FPU 19

12 FPU Control 19

13 Pipelining and Hazard Control 19

1 Preface

This project explores a custom implementation of a subset of the ARM instruction set. A custom instruction set inspired by the ARM architecture is designed with a custom assembler. The architecture is implemented in hardware as an RTL model, whose functionality is verified.

The assembler is implemented in Python, and the RTL model is implemented using SystemVerilog.

It should be noted that this architecture is an educational project inspired by ARM-style RISC design using the ARM7TDMI-S data sheet as refereence. It is not ARM-compatible and does not use proprietary ARM encoding or IP.

2 ISA Design

All instruction words are designed to be 32 bits wide. Each instruction has 4 condition bits that will determine whether or not the instruction executes based on CPSR condition flags (N, Z, C, V). This makes it simpler to write conditional statements for simple instructions. A list of the condition codes is listed below.

Field List			
Condition Code	**Instruction Suffix**	**Flags Set (NZCV)**	**Explanation**
0000	unused	N/A	unused
0001	AL	flags ignored	Always Executed
0010	LE	Z set OR (N not equal to V)	Less Than or Equal
0011	GT	Z clear AND (N equals V)	Greater Than
0100	LT	N not equal to V	Less Than
0101	GE	N equals V	Greater Or Equal
0110	LS	C clear or Z set	Unsigned Lower or Same
0111	HI	C set and Z clear	Unsigned Higher
1000	VC	V clear	No Overflow
1001	VS	V set	Overflow
1010	PL	N clear	Positive or Zero
1011	MI	N set	Negative
1100	CC	C clear	Unsigned Lower
1101	CS	C set	Unsigned Higher or Equal
1110	NEQ	Z clear	Not Equal
1111	EQ	Z set	Equal

There are a total of 16 16-bit registers in the register file, including link register, stack pointer, program counter, and stack pointer. 16-bit registers were chosen, due to the goal of designing a processor that performs floating point operations, which are too complex to be done in 1 clock cycle for 32-bit operands. 16-bit operands can get very close to IEE-754 compliance. The remaining 12 registers are general-purpose.

2.1 RX-type

2.1.1 Overview

The RX-type instructions are used for fixed-point arithmetic data-processing instructions. A summary of the format can be seen in Figure 1, and explanations of the fields can be seen under the figure.

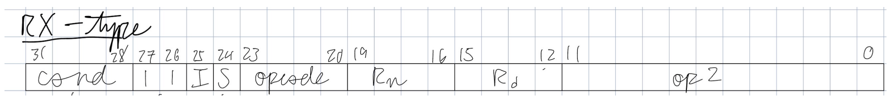

Figure 1: RX instruction type format.

Field List		
Field	**Bits**	**Description**
cond	[31:28]	State of CPSR condition codes (based on NZCV flags)
type	[27:26]	Encoding specific to instruction type
I	25	Determines whether or not op2 is an immediate (I = 0 means op2 is not an immediate, but a shift register)
S	24	Determines whether or not to alter condition codes (S = 0 means do not alter)
opcode	[23:20]	Determines the operation performed on operands
R_n	[19:16]	First source register
R_d	[15:12]	Destination register
$op2$	[11:0]	Varying field depending on the instruction

RX-type instructions have a varying $op2$ field that can be used depending on whether or not the instruction uses an immediate. For each of the RX-type instructions, a closer look will be given in their individual instruction sections.

Field List		
Instruction	**Bits**	**Description**
shift	[11:4]	Used for instructions using two source registers. The amount to shift the value in R_m
R_m	[3:0]	Used for instructions using two source registers. The second source register
rotate	[11:8]	Used for instructions using one source register and one immediate. Rotates the immediate a specific number of positions
imm	[7:0]	A constant used with another shift register to produce the result

Instructions take the following form:

```
(mneumonic).x-(instruction suffix) (rd), (rn), (rm)
```

where in each parentheses:

- mneumonic - the type of instruction (e.g. add, sub, etc.)
- instruction suffix - the instruction suffix that details the condition that the instruction is executed under
- rd - destination register
- rn - source register 1
- rm - source register 2/immediate

A list of suported instructions is listed below. It should be noted that because of some complex instructions, the ALU is pipelined to [insert how many stages here] stages.

Instructions		
Field	**opcode**	**Description**
add.x	0000	Adds two values
sub.x	0001	Subtracts two values
mul.x	0010	Multiplies two values
div.x	0011	Divides two values
mac.x	0100	Multiply-accumulate
sqrt.x	0101	Takes square root of a value
convf.x	0110	Convert value to IEEE-754 floating-point standard format
cmp.x	0111	Compare two fixed point numbers (automatically sets flags, don't add 's')
and.x	1000	Takes the bitwise AND of two operands)
or.x	1001	Takes the bitwise OR of two operands)
not.x	1010	Takes the bitwise NOR of two operands)
xor.x	1011	Takes the bitwise XOR of two operands)

2.1.2 add/sub

The add and sub instructions add or subtract two numbers and store them into a destination register. The following snippet shows the cases for add, but sub follows a similar format.

```
// add the values stored in r1 and r2 and store them
   into r3
add.x-al r3, r1, r2
// add 8 to the value stored in r1 and store them into r3
add.x-al r3, r1, #8
// add the values stored in r1 and r2 and store them
   into r3, and use the result to set NCZV flags
adds.x-al r3, r1, r2
```

The $op2$ field in the instruction format for add/sub takes on different forms depending on the value of of bit 25 (I). For I=0, the $op2$ field operates under the assumption that the 2nd operand is stored in a register. For I=1, the $op2$ field operates under the assumption that the 2nd operand is an immediate value.

2nd Operand: Register

When the 2nd operand is a register, the value in the register can be manipulated through shifting before carrying out addition or subtraction.

```
// add the values stored in r1 and r2 (whose value is
   shifted logically to the left by a value specified in
   r4) and store them into r3
add r3, r1, r2, lsl r4
// add the values stored in r1 and r2 (whose value is
   shifted logically to the left by 8) and store them
   into r3
add r3, r1, r2, lsl #8
```

The $op2$ field specifications are as follows:

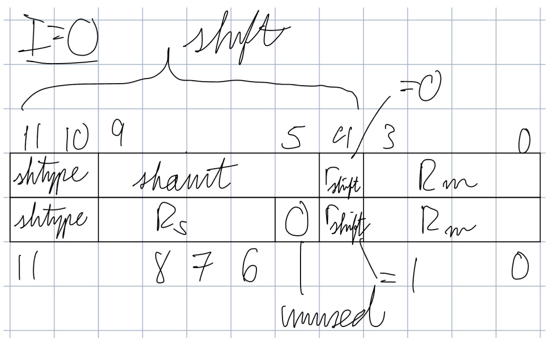

Figure 2: $op2$ field when the 2nd operand is a register. The top field is the format when the 2nd operand is shifted by a constant. The bottom field is the format when the 2nd operand is shifted by an amount specified in a register.

Field List for $op2$ (shifted by immediate)		
Field	**Bits**	**Description**
shtype	[11:10]	The shift type performed on the 2nd operand
shamt	[9:5]	The amount that the 2nd operand is shifted by
r_{shift}	4	The bit that specifies whether the shifting operand is a register or an immediate (value after lsl)
R_m	[3:0]	The register holding the second operand

Field List for $op2$ (shifted by register value)		
Field	**Bits**	**Description**
shtype	[11:10]	The shift type performed on the 2nd operand
R_s	[9:6]	The register that contains the amount that the 2nd operand is shifted by
unused	5	unused
r_{shift}	4	The bit that specifies whether the shifting operand is a register or an immediate (value after lsl)
R_m	[3:0]	The register holding the second operand

The shift type (shtype) determines what kind of shift the second operand goes through. The specifications for the shift type are as follows:

Description of Shift Types		
Shift Type	**Encode**	**Description**
ror	00	Rotate right
asr	01	Arithmetic shift right
lsr	10	Logical shift right
lsl	11	Logical shift left

For carrying out the operation without any shifting, it is sufficient to just not include a mention of the shift. It will assume lsl #0, which will not perform any shift.

2nd Operand: Immediate When the 2nd operand is an immediate, the values the immediate can take a variety of values.

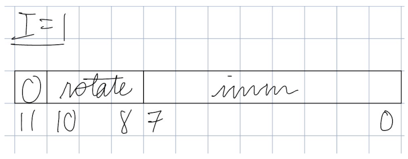

Figure 3: $op2$ format for when the 2nd operand is an immediate.

The 8-bit immediate field can be used to values from 0 to 255. Since each register is 16-bits, an 8-bit immediate isn't enough to reach the full value range that can be held by the register. With the rotate field, the rest of the bits in each register can be set, and a wider range of immediates can be used.

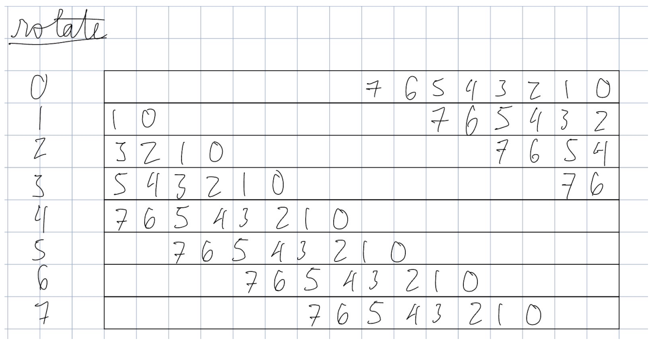

Figure 4: How the value of rotation affects which bits are selected to be affected

Not all immediates can be represented in one instruction, but a register can take any immediate value by setting the upper byte and lower byte individually.

```
// the following sets r2 to 0xffff
add.x-al r2, r1, #0x00ff
add.x-al r2, r1, #0xff00
```

2.1.3 mul/div

The mul instruction can multiply two numbers and store them into a destination register.

```
// multiply the values stored in r1 and r2 and store the
   product into r3 and r4
mul.x-al r3, r4, r1, r2
// divide the values stored in r1 and r2 and store the
   quotient into r3 and the remainder in r4
div.x-al r3, r4, r1, r2
// integer division of r1 and r2 and store the quotient
   into r3
divi.x-al r3, r1, r2
```

The $op2$ format for mul is shown below. For full context, part of the rest of the instruction encoding is also shown.

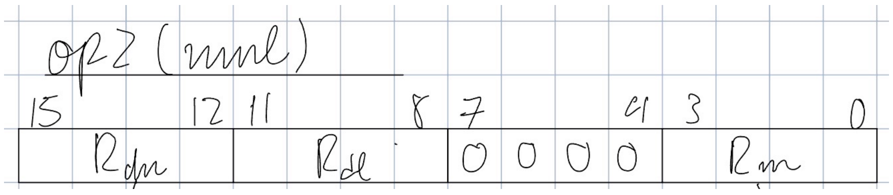

Figure 5: $op2$ encoding for the mul instruction

Field List for $op2$ (mul)		
Field	**Bits**	**Description**
R_{du}	[15:12]	Register to hold the upper byte of the product (technically not part of $op2$
R_{dl}	[11:8]	Register to hold the lower byte of the product
unused	[7:4]	unused
R_m	[3:0]	The register holding the second operand

Because the product of 2 16-bit numbers is 32-bit, two registers are necessary to hold the entire product.

The $op2$ format for div is shown below. For full context, part of the rest of the instruction encoding is also shown.

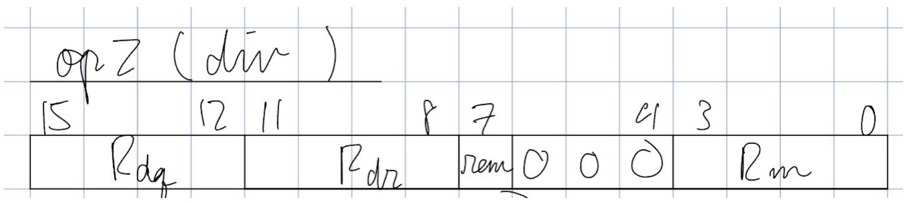

Figure 6: $op2$ encoding for the div/divi instructions.

Field List for $op2$ (div)		
Field	**Bits**	**Description**
R_{dq}	[15:12]	Register to hold the quotient (technically not part of $op2$
R_{dr}	[11:8]	Register to hold the remainder
rem	7	Bit to decide whether or not to keep the remainder (rem = 1 means keep the remainder)
unused	[6:4]	unused
R_m	[3:0]	The register holding the second operand

One register is used to store the quotient, and 1 register is used to store the remainder of the division. For integer division (divi), the rem bit is set to 1, and the bit values of R_{dr} are all set to 1.

Some things to note about mul/div:

- Immediates cannot be used.
- NCZV flags can be set with mul and div.

2.1.4 mac

The mac instruction performs a multiply-accumulate on 3 operands. The first two operands designate the two numbers being multiplied, and the 3rd operand designates the number being accumulated.

$$Y = A \cdot B + C \tag{1}$$

```
// multiply the values stored in r1 and r2, add r3, and
   store it into r4 and r5
mul.x-al r4, r5, r1, r2, r3
```

The mac instruction can The $op2$ format for mac is shown below. For full context, part of the rest of the instruction coding is also shown.

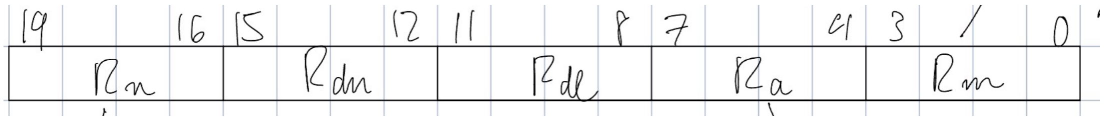

Figure 7: $op2$ encoding for mac instruction

Field List for $op2$ (mul)		
Field	**Bits**	**Description**
R_n	[19:16]	Register to hold the first operand (A)
R_{du}	[15:12]	Register to hold the upper byte of the product (technically not part of $op2$
R_{dl}	[11:8]	Register to hold the lower byte of the product
R_a	[7:4]	Register to hold the accumulating 2nd operand (B)
R_m	[3:0]	Register holding the multiplying 2nd operand (C)

Important things to note:

- mac cannot be used with immediates

2.1.5 sqrt

Important things to note:

- sqrt cannot be used to set NCZV flags

2.1.6 convf

Important things to note:

- convf cannot be used to set NCZV flags
- convf cannot be used with immediates

2.1.7 cmp

2.1.8 Miscellaneous Notes: RX-type Instructions

A few things to note about RX-type instructions:

- The assembler does not check for the validity of the immediate used for instructions that use immediates.
- Immediates must be done for the 2nd operand, so ensure that you do not use them for the first operand. convx2f, mac does not take immediate inputs.
- To update NCZV flags after add, sub, append an ’s’ after the mneumonic (i.e. adds, subs). NCZV flags cannot be set with other operations besides add.x and sub.x, and cmp
- mac, sqrt, convf, and cmp are undefined for ’s’ additions. Do not use them.
- and, or, not, and xor instructions can also be used on floating point data.

2.2 RF-type

Note: This section will be implemented if time allows for it.

2.2.1 Overview

The RF-type instructions are used for floating-point arithmetic data-processing instructions, using the IEEE-754 floating-point standard format. A summary of the format can be seen in Figure 3, and explanations of the fields can be seen under the figure.

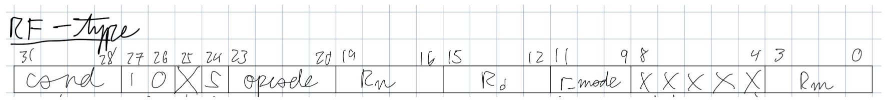

Figure 8: RF instruction type format.

Field List		
Field	**Bits**	**Description**
cond	[31:28]	State of CPSR condition codes (based on NZCV flags)
type	[27:26]	Encoding specific to instruction type
unused	25	unused
S	24	Determines whether or not to alter condition codes (S = 0 means do not alter)
opcode	[23:20]	Determines the operation performed on operands
R_n	[19:16]	First source register
R_d	[15:12]	Destination register
r_{mode}	[11:9]	Specifies the rounding mode of the floating point operation. See the underlying table for details.
unused	[8:4]	unused (might do flags for invalid operations)
R_m	[3:0]	Varying field depending on the value of opcode

r_{mode}	
r_{mode} **value**	**Description**
000	Operation rounds toward 0
001	Operation rounds toward nearest, ties away from 0
010	Operation rounds toward nearest, ties to even
011	Operation rounds toward $+\infty$
100	Operation rounds toward $-\infty$

Instructions take the following form:

```
(mneumonic).f-(instruction suffix) (rd), (rn), (rm),
   #(r_mode)
```

where in each parentheses:

- mneumonic - the type of instruction (e.g. add, sub, etc.)
- instruction suffix - the instruction suffix that details the condition that the instruction is executed under
- rd - destination register
- rn - source register 1
- rm - source register 2

- r_{mode} - the rounding mode for the floating point operation

  Instructions take the following form:

```
(mneumonic).f-(instruction suffix) (rd), (rn), (rm),
   #(r_mode)
```

where in each parentheses:

- mneumonic - the type of instruction (e.g. add, sub, etc.)
- instruction suffix - the instruction suffix that details the condition that the instruction is executed under
- rd - destination register
- rn - source register 1
- rm - source register 2
- r_{mode} - the rounding mode for the floating point operation

  A list of suported instructions is listed below.

Instructions		
Field	**opcode**	**Description**
add.f	0000	Adds two values
sub.f	0001	Subtracts two values
mul.f	0010	Multiplies two values
div.f	0011	Divides two values
mac.f	0100	Multiply-accumulate
sqrt.f	0101	Takes square root of a value
convf2x.f	0110	Convert value to fixed-point format
cmp.f	0111	Compare two fixed point numbers

A few things to note about RF-type instructions:

- The instructions cannot be used to set CPSR condition codes, and are undefined for immediate type instructions.
- To choose the rounding mode for the floating point operations, after the '.f' market, use '#' followed by the value of r_{mode} to specify the rounding operation (e.g. add.f-al r1, r2, r3, #4 to round toward $-\infty$).

- Rounding mode is underfined for cmp instruction. Just only use the two operands being compared
- Note the lack of immediate operations. To use immediate values, use fixed-point representation to create the immediate value with addi, and then convx2f.

2.2.2 add/sub

2.2.3 mul

2.2.4 div

2.2.5 mac

2.2.6 sqrt

2.2.7 convx

2.2.8 cmp

2.3 D-type

2.3.1 Overview

The D-type instructions are used for loading and storing data from and into memory.

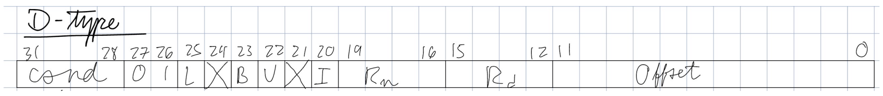

Figure 9: D instruction type format.

Field List		
Field	**Bits**	**Description**
cond	[31:28]	State of CPSR condition codes (based on NZCV flags)
type	[27:26]	Encoding specific to instruction type
L	25	Determines whether it is a load or store operation (L = 1 means load, L = 0 means store)
unused	24	unused
B	23	Determines whether or a word (16 bits) or a byte (8 bits) is loaded/stored (B = 1 means a byte is used, B = 0 means a word is used)
U	22	Determines whether the offset is added or subtracted (U = 1 means that the offset is added, U = 0 means that the offset is subtracted)
unused	21	unused
I	20	Determines whether or not the the offset is an immediate value or a register (I = 1 means that it is an immediate value, I = 0 means that the offset is stored in a register).
R_n	[19:16]	Address register used to interact with memory
R_d	[15:12]	Destination register
offset	[11:0]	Offset used to calculate where to load/store data. For a register offset, the register would be the least significant 4 bits

Instructions take the following form:

```
(mneumonic)(B)-(instruction suffix) (rd), [(rn), (offset)]
```

where in each parentheses:

- mneumonic - the type of instruction (e.g. add, sub, etc.)
- B - presence of B dictates whether a byte or word is loaded in (more under instruction table)
- instruction suffix - the instruction suffix that details the condition that the instruction is executed under
- rd - Register in register file to load or store to
- rn - Register holding the address to interact with in data memory
- offset - offset used to calculate where to load/store data

A list of suported instructions is listed below.

Instructions	
Field	**Description**
ldr	Loads a data value from data memory into the register file
str	Stores a data value from the register file into data memory

- To specify loading a byte, add a 'b' after the mneumonic (ldrb, strb), otherwise it will default to loading/storing a word.
- To specify whether an offset is added or subtracted, use positive offset values for adding, and negative offset values for subtracting (e.g. ldr r0, [r1, #8] for the address r1 + 8, ldr r0, [r1, #-8] for the address r1 - 8).
- To specify whether an offset is an immediate value or a register, use '#' to specify the offset, or 'r' to specify a register (e.g. ldr r0, [r1, #8] for an offset or ldr r0, [r1, r2] for a register).

2.3.2 ldr

2.3.3 str

2.4 B-type

2.4.1 Overview

B-type instructions are used for procedure calls. The ISA uses relative branching.

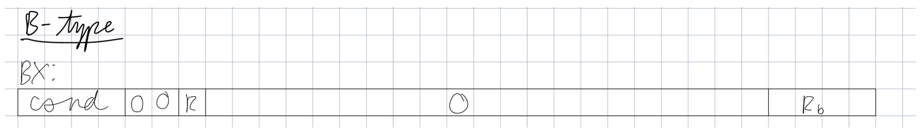

Figure 10: B instruction type format for BX instruction

Field List (BX)		
Field	**Bits**	**Description**
cond	[31:28]	State of CPSR condition codes (based on NZCV flags)
type	[27:26]	Encoding specific to instruction type
R	26	Determines whether the instruction is a BX instruction vs B or BL instructions (R = 0 means that it is a BX instruction, while R = 1 means that it is either a B or a BL instruction)
R_b	[3:0]	Address of the register containing the address to branch to

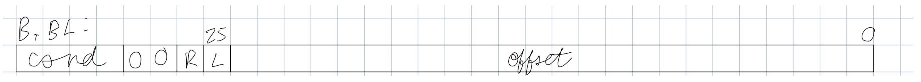

Figure 11: B instruction type format for B and BL instruction

Field List (B or BL)		
Field	**Bits**	**Description**
cond	[31:28]	State of CPSR condition codes (based on NZCV flags)
type	[27:26]	Encoding specific to instruction type
R	26	Determines whether the instruction is a BX instruction vs B or BL instructions (R = 0 means that it is a BX instruction, while R = 1 means that it is either a B or a BL instruction)
L	25	Determines whether the instruction is a B instruction vs a BL instruction (L = 0 means that it is a B instruction, while L = 1 means that it is a BL instruction)
offset	[24:0]	Relative address of the label to branch to

Instructions take the following form:

```
(mneumonic)-(instruction suffix) (label)
```

where in each parentheses:

- mneumonic - the type of instruction (e.g. add, sub, etc.)
- instruction suffix - the instruction suffix that details the condition that the instruction is executed under
- label - the label or register containing program counter value to branch to

Instructions	
Field	**Description**
bx	Branches to an address specified by a register
b	Branch to a label
bl	Branch and link

2.4.2 bx

2.4.3 b

2.4.4 bl

2.5 Miscellaneous Notes

- Labels must be alone on its own line. In other words, this is allowed:

```
label:
add.x-al r1, r2, r3
```

But this is not:

```
label: add.x-al r1, r2, r3
```

- Labels don't have a specific syntax defined. As long as the label is before a ':', it is a valid label. Using multiple colons for a label will cause some undefined behavior.

3 Assembler

The assembler is implemented as a two-pass assembler in Python. In the first pass, labels are assigned location counter (LC) values to represent where they will be stored in instruction memory. For an instruction memory of 256 addresses, 8 bits are used to represent the addresses. These values are stored in a symbol table implemented as a hash table. In the second pass, all instructions are put into their machine code counterpart in the following format (similar to .bin files):

```
0x##: ## ## ## ##
```

The number before the colon is a hexadecimal representation of the LC value, and the numbers after it are the hexadecimal representation of the instruction encoding. A binary version of this is also produced. Consider the following example instruction:

```
    addi.x-al r0, #9
```

A few things to note about the assembler:

- Multiple labels of the same have undefined behavior. Since the symbol table was implemented as a Python dictionary, the most recent definition of the label will probably be what defines the label.
- There is nothing to check for invalid syntax. The programmer takes responsibility for making sure everything is correct.

4 Instruction Memory

5 Program Counter

6 Program Counter Adder

7 Register File

8 Data Memory

9 ALU

10 ALU Control

11 FPU

12 FPU Control

13 Pipelining and Hazard Control