CSE 151A HW 01

Karlo Godfrey Escalona Gregorio
PID: A16536865

Contents

1 Preface

This project explores a custom implementation of a subset of the ARMv7 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

2.1 R-type: Fixed Point

2.1.1 Overview

The R-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.

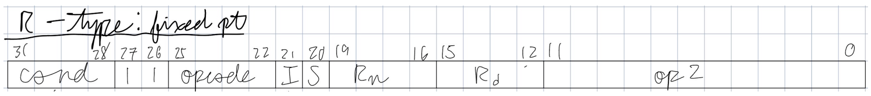


Figure 1: R-type format for fixed-point instructions.

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
opcode	[25:22]	Determines the operation performed on operands
I	21	Determines whether or not op2 is an immediate (I = 0 means op2 is not an immediate, but a shift register)
S	20	Determines whether or not to alter condition codes (S = 0 means do not alter)
R_n	[19:16]	First source register
R_d	[15:12]	Destination register
$op2$	[11:0]	Varying field depending on the instruction

R-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 R-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)-(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**
addx	0000	Adds two fixed-point values
subx	0001	Subtracts two fixed-point values
mulx	0010	Multiplies two fixed-point values
divx	0011	Divides two fixed-point values
notx	0100	Takes the bitwise NOR of two operands)
andx	0101	Takes the bitwise AND of two operands)
orrx	0110	Takes the bitwise OR of two operands)
convx	0111	Convert value to fixed-point format

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
addx.s-al r3, r1, r2
// add 8 to the value stored in r1 and store them into r3
addx.s-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
addx.s-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 3rd operand is stored in a register. For I=1, the $op2$ field operates under the assumption that the 3rd operand is an immediate value.

3rd Operand: Register

When the 3rd 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
addx 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
addx r3, r1, r2, lsl #8
```

The $op2$ field specifications are as follows:

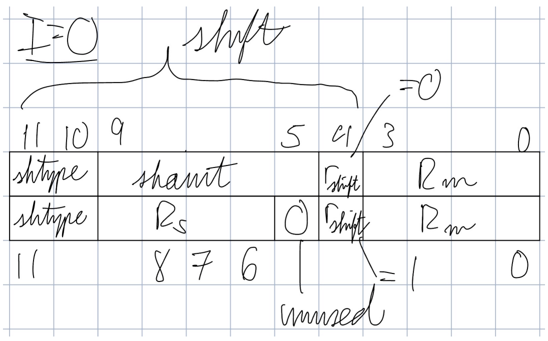


Figure 2: $op2$ field when the 3rd operand is a register. The top field is the format when the 3rd operand is shifted by a constant. The bottom field is the format when the 3rd operand is shifted by an amount specified in a register.

Field List for $op2$ (3rd operand register, shifted by immediate)		
Field	**Bits**	**Description**
shtype	[11:10]	The shift type performed on the 3rd operand
shamt	[9:5]	The amount that the 3rd 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 $op2$ (3rd operand register, shifted by register value)		
Field	**Bits**	**Description**
shtype	[11:10]	The shift type performed on the 3rd operand
R_s	[9:6]	The register that contains the amount that the 3rd 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.

3rd Operand: Immediate When the 3rd operand is an immediate, the values the immediate can take a variety of values.

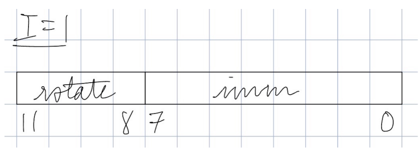


Figure 3: $op2$ format for when the 3rd operand is an immediate.

Field List for $op2$ (3rd operand immediate)		
Field	**Bits**	**Description**
rotate	[11:8]	Number defining how many times the immediate is rotated right in a 16-bit value
imm	[7:0]	Immediate to be encoded

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.

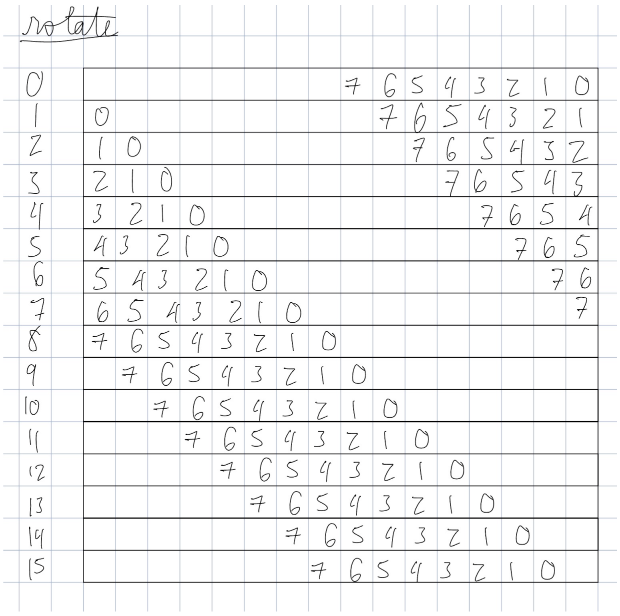


Figure 4: How the value of rotation affects which bits are selected to be affected

One unique difference from ARMv7 is that the rotation encoding from an 8-bit immediate into 16-bits is that this allows for full access to the full range of immediates possible for 16-bit operands, meaning the effective range of encoding immediates directly is from 0 to $2^{16}-1$. This makes the encodable range for the immediate much wider at the cost of higher hardware complexity.

2.1.3 mul/div

The mulx 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
mulx-al r3, r4, r1, r2
// divide the values stored in r1 and r2 and store the
   quotient into r3 and the remainder in r4
divx-al r3, r4, r1, r2
// integer division of r1 and r2 and store the quotient
   into r3
divx-al r3, r1, r2
```

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

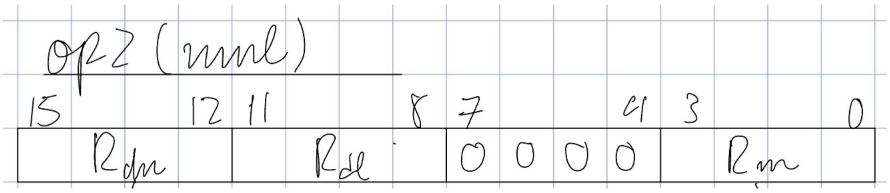


Figure 5: $op2$ encoding for the mul instruction

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.

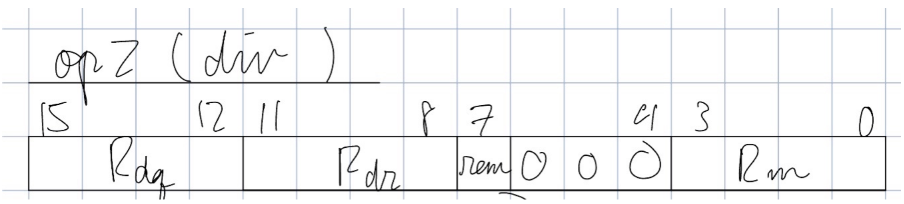


Figure 6: $op2$ encoding for the div/divi instructions.

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, 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. The 32-bit instructions doesn't have the capacity to use immediates. This means I is always set to 0
- NCZV flags cannot be set with mul and div. Allowing for this increases the complexity of the hardware by too much. This means S is always set to 0
-

2.1.4 not

The notx instruction can take the bitwise not of what is stored in the source register.

```
// take the bitwise not of r1 and store into r2
notx-al r2, r1
```

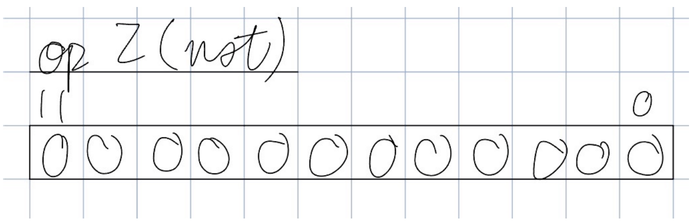


Figure 7: $op2$ encoding for the not instruction

Uniquely, $op2$ is set to all 0s, since not is a unary operator. This also means that immediates have no purpose for this instruction, as well as setting NCZV flags (I = 0, S = 0).

2.1.5 and/or

The andx instruction can take the bitwise and of what is stored in the source register. The orrx instruction can take the bitwise or of what is stored in the source register.

```
// take the bitwise and of r1 and r2 and store into r3
andx-al r3, r1, r2
// take the bitwise and of r1 and #0x00ff and store into
   r3
andx-al r2, r1, #9
// take the bitwise or of r1 and r2 and store into r3
orrx-al r3, r1, r2
// take the bitwise or of r1 and #0x00ff and store into
   r3
orrx-al r2, r1, #9
```

The encoding done for $op2$ is identical to that of addx/subx instructions, so shifting operations can be applied to the 3rd operand (given that it is a register), if desired (see section 2.1.2 for more details).

2.1.6 convx

Will be updated if I get to floating-point instructions

2.1.7 Miscellaneous Notes: R-type Fixed-Point Instructions

A few things to note about R-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 3rd operand, so ensure that you do not use them for the first operand. convx
- To update NCZV flags after addx, subx, append an 's' after the mneumonic (i.e. adds, subs).
- and, or, not instructions can also be used on floating point data.

2.2 R-type: Floating Point

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.

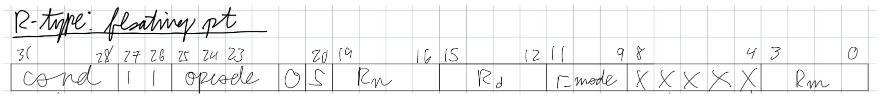


Figure 8: RF instruction type format.

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
opcode	[25:22]	Determines the operation performed on operands
unused	21	unused
S	20	Determines whether or not to alter condition codes (S = 0 means do not alter)
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

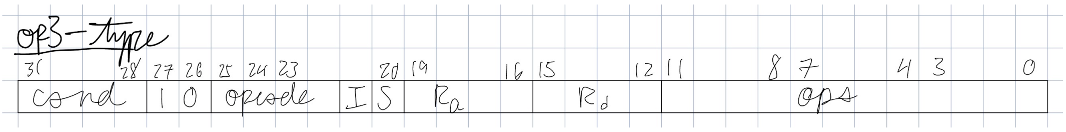
A list of suported instructions is listed below.

Instructions		
Field	**opcode**	**Description**
addf	1000	Adds two floating-point values
subf	1001	Subtracts two floating-point values
mulf	1010	Multiplies two floating-point values
divf	1011	Divides two floating-point values
cmpf	1100	Compares two floating-point values)
cnvf	1101	Convert value to IEEE-754 floating-point standard format)
sqrf	1110	Takes square root of a floating-point value)
recf	1111	Takes reciprocal of a floating-point value

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.

2.3 OP3-type



Instructions	
Instruction	**Description**
vmac	Multiply-accumulate 3 registers
vadd	Add two registers in accordance to the bits set in the 3rd register
vsub	Subtract two registers in accordance to the bits set in the 3rd register
vsel	Select bits between two registers in accordance to the bits set in the 3rd register

2.4 D-type

2.4.1 Overview

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

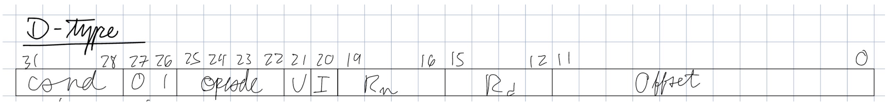


Figure 9: D instruction type format.

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
opcode	[25:22]	Encoding specific to instruction
U	21	Determines whether the offset is added or subtracted (U = 1 means that the offset is added, U = 0 means that the offset is subtracted)
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)-(instruction suffix) (rd), [(rn), (offset)]
```

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 - 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		
Instruction	**opcode**	**Description**
ldw	000	Loads a 16-bit word from data memory into a register in the register file
ldb2l	001	Loads a byte from data memory into the lower byte of a register in the register file
ldb2h	010	Loads a byte from data memory into the upper byte of a register in the register file
stw	011	Stores a 16-bit word from a register in the register file into data memory
stb2l	100	Stores a byte from a register in the register file into data memory
stb2h	101	Stores a byte from a register in the register file into data memory

2.4.2 ld

Load instructions are used to load data from data memory into the register file. Users have the option of loading an entire 16-bit word or just a byte, which can be written into the upper or lower byte of a register. Additionally, it is possible to choose whether or not the offset is defined by an immediate or by a register.

```
// load a 16-bit word from data memory, 2 bytes upstream
ldw r2, [r1, #2]
// Load a 16-bit word from memory, 2 bytes downstream
ldw r2, [r1, #-2]
// Load a 16-bit word from memory, offset according to
   the value stored in r3
ldw r2, [r1, r3]
// Load the byte stored at address r1 into the lower
   byte of the register r2
ldb2l r2, [r1, #0]
// Load the byte stored at address r1 into the upper
   byte of the register r2
```

The value of the offset can be determined by a register or by an immediate value. These follow the same format as the 3rd operand in R-type instructions like add and sub (See Section 2.1.2).

2.4.3 st

Store instructions are used to store data from the register file into data memory. Users have the option of storing an entire 16-bit word or just a byte, which can be read from the upper or lower byte of a register. Additionally, it is possible to choose whether or not the offset is defined by an immediate or by a register.

```
// load a 16-bit word from data memory, 2 bytes upstream
stw r2, [r1, #2]
// Load a 16-bit word from memory, 2 bytes downstream
stw r2, [r1, #-2]
// Load a 16-bit word from memory, offset according to
   the value stored in r3
stw r2, [r1, r3]
// Load the byte stored at address r1 into the lower
   byte of the register r2
stb2l r2, [r1, #0]
// Load the byte stored at address r1 into the upper
   byte of the register r2
```

The value of the offset can be determined by a register or by an immediate value. These follow the same format as the 3rd operand in R-type instructions like add and sub (See Section 2.1.2).

2.4.4 Miscellaneous Notes about D-type Instructions

- 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).
- The hardware uses little-endian formatting.

2.5 B-type

2.5.1 Overview

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

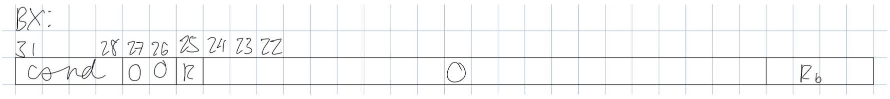


Figure 10: B instruction type format for BX instruction

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	25	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

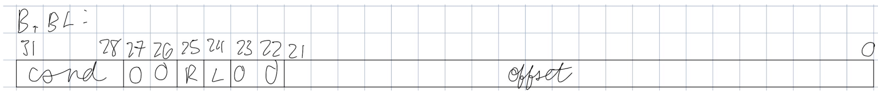


Figure 11: B instruction type format for B and BL instruction

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	25	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	24	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)
unused	[23:22]	unused
offset	[21:0]	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

2.5.2 bx

Branch and exchange is a branching instruction that branches to an address stored in a register. It is commonly used to return from a procedure using the link register (r14).

```
// Return from procedure
bx lr
```

2.5.3 b

The general branch instruction branches to an address stored in a label. For conditional branching, NCZV flags must be set by a previous instruction.

```
// go to label
b label
// go to label if registers r1 and r2 are equal
subs r0, r1, r2
beq label
```

2.5.4 bl

The branch and link instruction stores the address of the next instruction before branching to a label.

```
// go to label and save the location of the instruction
   after the label
bl label
```

2.6 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

Instruction memory is composed of 256 possible locations, each location holding 32 bits.

5 Program Counter

6 Program Counter Adder

7 Register File

There are a total of 16 16-bit registers in the register file, including link register, program counter, and zero/discard register. 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.

Register File	
Register	**Purpose**
r0	Zero Register/Discard Register
r1-r13	General Purpose
r14	Link Register
r15	Program Counter (PC)

8 ALU

9 ALU Control

10 Data Memory

Data memory is composed of 256 possible locations, each location holding 8 bits.

11 Branch Prediction

12 Multiple Operands

13 FPU

14 FPU Control

15 Pipelining and Hazard Control