CSE 151A HW 01

Karlo Godfrey Escalona Gregorio
PID: A16536865

Contents

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. The remaining 12 registers are general-purpose.

2.1 RX-type

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.

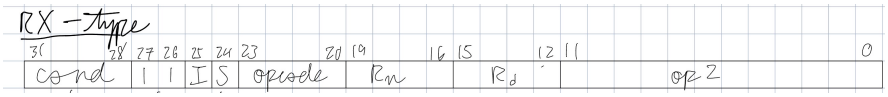

Figure 1: RX instruction type format.

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 value of opcode

RX-type instructions have a varying $op2$ field that can be used depending on whether or not the instruction uses an immediate. A summary of the $op2$ fields is shown below.

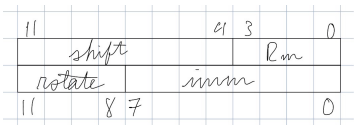

Figure 2: RX instruction type format.

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
convx2f.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')

- To add or subtract with an immediate, simply append an 'i' after "add" or "subtract" (i.e. addi/subi). Immediate values cannot be used with other instructions besides add.x and sub.x.
- To update NCZV flags after add, addi, sub, subi, append an 's' after the mneumonic (i.e. adds, addis, subs, subis). NCZV flags cannot be set with other instructions besides add.x and sub.x

2.2 RF-type

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.

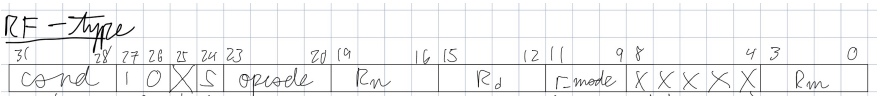

Figure 3: RF instruction type format.

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
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

- 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 $-\infty$).
- 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 D-type

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

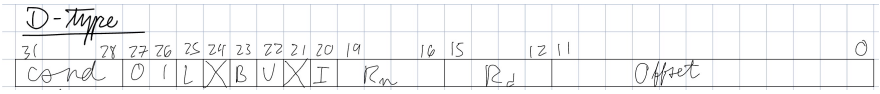

Figure 4: D instruction type format.

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).

2.4 B-type

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

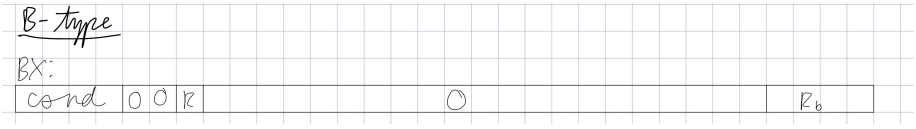

Figure 5: B instruction type format for BX instruction

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

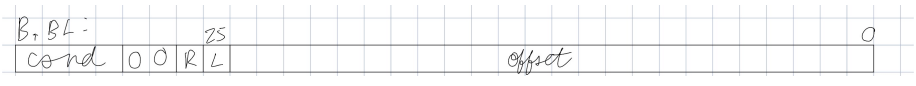

Figure 6: B instruction type format for B and BL instruction

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

2.5 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
```

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
```

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