Components of Lab 1 report:

1. Introduction.

This should include the name of your architecture, overall philosophy, specific goals strived for and achieved.

The name of our architecture is called first designed architecture (FDA)

Our overall designing philosophy is to brainstorm and come up with different algorithms that are similar to accumulator type and vote on one algorithm to convert specific integers to floating point, and then test to verify the algorithm to see whether it fits the accumulator type ISA or not.

The goal is design a special purpose processor to solve the three problems listed below.

2. Instruction formats.

List all formats and an example of each. (ARM has R, I, and B type instructions, for example.)

We have three types of instructions: R, I, and B instruction, and there is a type bit in the very beginning to indicate the type of each instruction.

I type instruction:

The type bit is 0. The rest of the 8 bits are all used to indicate an immediate. This type is to store the immediate into the accumulator register r0.
Example: 0 1111 1111, (set 255). The first bit is type bit, and the rest 8 bits are the immediate (255), which means the value 255 is stored into accumulator r0.

R-type instruction:

The type bit is 1. The rest of 8 bits which take 4 bits represent as opcode, and the other 4 bits assigned for operand.
Example: 1 0000 0001, (move r1). 0000 is the opcode of “move”, and 0001 is a register which is r1.

Branch instruction:

Jump to the related label
Example: load r1
    assign r2
    load r3

beq r2 label // if r2 == r3, branch to the label

3. Operations.

List all instructions supported and their opcodes/formats.

Operation name	Description	Format	Opcode
set	Set 8 bits immediate into the accumulator register	I	none
move	Move data of the operand register into accumulator r0	R	0000
assign	Get the data of operand register from accumulator r0	R	0001
add	Add the operand register to accumulator r0	R	0010
sub	Subtract data of register from the accumulator r0	R	0011
load	Load data into accumulator r0 from the address of register	R	0100
store	Store data into specified register from the accumulator r0	R	0101
and	And gate operation register with accumulator and store result into accumulator r0	R	0110
lsl	Logic shifts register left by the number of accumulator r0	R	0111
lsr	Logic shifts register right by the number of accumulator r0	R	1000
orr	Or gate operation register with accumulator and store result into accumulator r0	R	1001
b	Branch to the label if accumulator r0 equal to 0	B	1010
beq	Compare if accumulator r0 equal to register, if true then branch to label else continue running	B	1011
blt	Check if accumulator r0 is less than register, if true then branch to label else continue running	B	1100
bge	Check if accumulator r0 is grater than and equal register, if true then branch to label else continue running	B	1101
bgt	Check if accumulator r0 is grater than register, if true	B	1111

	then branch to label else continue running		

4. Internal operands.

How many registers are supported? Is there anything special about any of the registers, or all of them general purpose?

16 registers are supported. All of them are for general purposes

5. Control flow (branches).

What types of branches are supported? How are the target addresses calculated? What is the maximum branch distance supported?

b, beq, blt, bge, bgt, conditional jumps are supported. The target address is calculated relative to the current instruction.
The maximum branch distance is 2^6 instructions are supported.

6. Addressing modes.

What memory addressing modes are supported, e.g. direct, indirect? How are addresses calculated? Give examples.

Memory addressing is supported to memory location 2^6 bytes relative to register's memory address direct.
Example: load [#0]
store [#2]
The address needs to be calculated and loaded into accumulator register. Then it would store to the target memory address

7. Can you classify your machine in any of the classical ways (e.g., stack machine, accumulator, register, load-store)? If so, which? If not, devise a name for your class of machine.

Accumulator

8. Give an example of an "assembly language" instruction in your machine, then translate it into machine code.

Assembly example: load r2
Machine code: 1 0100 0010

9-11. For the three programs listed, give assembly instructions. Make sure your assembly format is either very obvious or well described, and that the code is well commented. If you also want to include machine code, the effort will not be wasted, since you will need it later. We shall not correct/grade the machine code. State any assumptions you make. If you need initial nonzero values in registers or memory, you need to put them there. (Exception -- the test bench will load the incoming operands for you.)

1. int2float --Write a program that converts a 16-bit two's complement integer into 16-bit IEEE floating point format, i.e.,

shortX;

floatY = X; *// actually, need a special float_binary_16 format command*

The operand, X, is found in memory locations 0 (most significant word of X) and 1 (least significant word of X). The result, Y, shall be written into locations 2 (MSW of Y) and 3 (LSW of Y).

```
load [#1]       // load lsw from memory location 1
assign r1       // put into register r1
load [#0]       // load upper 8 bits from memory location 0
assign r2       // put into register r2
set #0x80
and r2
assign r3       // isolate sign bit into r3
set #0
beq r1, Check_zero    // check of lower bits are all zero

L_1:
set #29
assign r4       // set biased E into r4
set #0x7F
and r2          // assign the upper 7 bits (excludes the sign bit) into r5
assign r5

Loop:
set #0x40
and r5          // isolate 15th bit
assign r6
set #0x40
beq r6, Carry_out      // check if 15th bit is 1, if true branch to done label

set #1
lsl r5          // if r6 == 0 then left shift upper 8bits one left and decrement r4
assign r5
set #0x80
and r1          // extract 8th bot of lower 8 bits
assign r6       // put inti r6
set #7
lsr r6          // shift right 7
assign r6
move r5
orr r6
```

```
assign r5
set #1
lsl r1          // shift lower 8 bit one to left
set #1
sub r3          // decrement exponent by 1
assign r3
b Loop          // branch to loop

Carry_out:
set #0x10      // case 1, if 5th bit is 0 and 4th bit is 1
and r1
assign r6       // extract 5th bit
set #0
beq r6, Case1            // if 5th bit is 1, branch to case 1

End_case1:
set #0x10
beq r6, Case2            // if 5th bit is 0 branch to case 2

End_case2:
set #0x80
beq r3, Overflow         // check overflow

End_overflow:            // get the mantissa
set #2
lsl r4
assign r4
move r3
orr r4
assign r3                // orr with the sign bit
set #15
assign r7
set #11
assign r8

Mantissa_loop:
set #1
lsr r1          // shift lower 8 bit one to the right
assign r1
set #1
and r5
assign r6       // extract the 1th of upper 8 bits
set #7
lsl r6          // shift left 7 bit in order put the extracted bit in 8th
assign r6
move r1
orr r6
assign r1
set #1
lsr r5
assign r5       // shift upper 8 bits one to right
set #1
sub r7
assign r7
```

```
move r8
beq r7, Output // check if reach the 11th bit
b Mantissa_loop

Output:
set #0xFF
and r1
assign r1
set #0x3
and r5
assign r5
move r1
store [#3]      // LSW result store into memory location 3
move r5
store [#2]      // MSW result store into memory location 2
b END           // end program

Case1:
set #0x8
and r1
assign r6       // extract 4th bit
set #0x8
beq r6, Carry_out1 // if 4th bit is 1, branch to carry out case b
End_case1

Carry_out1:
set #0xF8
bge r1, Add_bit1

End_add1:
set #8
add r1
assign r1
b End_case1

Case2:
set #0x8
and r1
assign r6       // extract 4th bit
set #0x8
beq r6, Carry_out2
b End_case2

Carry_out2:
set #0x7
and r1
assign r6
set #0
beq r6, End_case2 // if 3rd bit is 0, then end else continue
set #0xF8
bge r1, Add_bit2

End_add2:
set #8
```

```
add r1
assign r1
b End_case 2

Add_bit1:
set #1
add r5
assign r5
b End_add1

Add_bit2:
set #1
add r5
assign r5
b End_add2

Overflow:
set #1
lsr r1          // shift lower 8 bit one to the right
assign r1
set #0x1
and r5          // extract the first bit of upper 8bits
assign r6
set #7
lsl r6          // left shift 7 bits in order put extracted bit in 8th bit
assign r6
move r1
orr r6
assign r1
set #1
lsr r5
assign r5
set #1
add r4
assign r4
b End_overflow


Check_zero:
set #0x7F
and r1
assign r1
move r3
beq r1, END
b L_1

END
```

2. **float2int**-- Write a program that converts a 16-bit IEEE format floating point number to fixed point, i.e.:

   floatX;

   shortY = X;

The operand, X, is found in memory locations 4 (MSW) and 5 (LSW). The result is stored in locations 6 (MSW) and 7 (LSW).

```
load [#5]        // LSW memory location
assign r1
load [#4]        // MSW memory location
assign r2
set #0x80
and r2           // isolate sign bit into r3
assign r3
set #0x7C
and r2
assign r3
set #2
lsr r3
assign r3
set #0x3
and r1
assign r4
set #0x3
and r2
assign r5
set #0
beq r3, Done
b L_1

Done:
set #29
bge r3, Overflow
set #26
bge r3, Lsl_mantissa
set #25
beq r3, Mantissa
set #14
bgt r3, Lsr_mantissa
set #0x80
beq r2, Underflow_negative
set #0x00
assign r4
set #0x00
assign r5
move r4
```

```
store [#7]
move r5
store [#6]
b END

L_1:
set #0x4
orr r5
assign r5
b Done
Overflow:
set #0x80
assign r6
beq r2, Overflow_positive
set #0xFF
assign r4
set #0x7F
assign r5
move r4
store [#7]
move r5
store [#6]
b END

Overflow_positive:
set #0xFF
assign r4
set #0xFF
assign r5
move r4
store [#7]
move r5
store [#6]
b END

Lsl_mantissa
set #15
sub r3
assign r3
set #0

Loop1:
beq r3, F_1
set #1
assign r5
set #0x80
and r4
assign r6
```

```
set #7
lsr r6
assign r6
move r6
orr r5
assign r5
set #1
lsl r4
assign r4
set #1
sub r3
assign r3
b Loop1

F_1:
move r4
store [#7]
move r5
store [#6]
b END

Mantissa:
move r4
store [#7]
move r5
store [#6]
b END

Underflow_negative:
set #0x10
store [#7]
set #0x00
store [#6]
b END

Lsr_mantissa
set #15
sub r3
assign r3
set #0
assign r6
assign r7
assign r8

Loop2:
move r3
beq r6, F_2
move r8
```

```
orr r7
assign r7
set #0x1
and r5
assign r9
set #7
lsl r9
assign r9
set #0x1
lsr r4
assign r4
set #0x1
lsr r5
assign r5
move r9
orr r4
assign r4
set #1
sub r3
assign r3
b Loop2

F_2:
set #0x1
and r4
assign r6
set #1
beq r6, Check_round
set #0
beq r8, Output
set #0
beq r7, Output
set #1
add r4
assign r4
b Output

Check_round:
set #0
beq r9, Output
set #1
add r4
assign r4

Output:
move r4
store [#7]
move r5
```

```
store [#6]

END
```

3. **float_add**-- Write a program that adds two 16-bit floating point numbers.

   floatX;

   floatY;

   floatZ = X+Y;

One 16-bit floating point operand will occupy data memory locations 8 (MSW) and 9 (LSW), whereas the other will occupy locations 10 and 11. Write the 16-bit floating point sum into locations 12 and 13 (LSW).

```
 load [#8]
assign r1
load [#9]
assign r2
load [#10]
assign r3
load [#11]
assign r4
set #0x80
add r1
assign r6
set #0x80
add r3
assign r7
move r6
beq r7, Continue // check the sign bits
b END

Continue:
set #0x7C
and r1
assign r5
set #0x7C
```

```
and r3
assign r6
set #0x3
and r1
assign r1
set #0x3
and r3
assign r3
move r6
bgt r5, Greater_than
blt r5, Less_than

L_1:
move r3
add r1
assign r1
move r1
add r4
assign r4
set #1
and r4
assign r10
set #0
beq r9, Output
set #1
beq r10, Addup
beq r8, Addup
b Output

Greater_than:
move r6
sub r5
assign r6
set #2
lsr r6
assign r6
set #0x80
orr r3
assign r3
set #0
assign r8
set #0
assign r9
set #0
assign r10

Loop1:
move r6
```

```
beq r10, F_1
move r9
orr r8
assign r8
set #0x1
and r4
assign r9
set #7
lsl r11
assign r11
set #0x1
lsr r4
assign r4
set #0x1
lsr r3
assign r3
move r11
orr r3
assign r3
set #1
sub r6
assign r6
b Loop1

F_1:
move r5
assign r6
b L_1

Less_than:
move r6
sub r5
assign r5
set #2
lsr r5
assign r5
set #0x80
orr r1
assign r1

Loop2:
move r6
beq r10, F_2
move r9
orr r8
assign r8
set #0x1
and r2
```

```
assign r9
set #0x1
and r1
assign r11
set #7
lsl r11
assign r11
set #0x1
lsr r2
assign r2
set #0x1
lsr r1
assign r1
move r11
orr r1
assign r1
set #1
sub r5
assign r5
b Loop2

F_2:
move r6
assign r5
b L_1

Output:
move r4
Store[#13]
move r6
orr r1
assign r1
move r7
orr r1
assign r1
move r1
store [#12]
b L_1

Addup:
set #1
add r4
assign r4
move r4
store [#13]
move r6
orr r1
assign r1
```

```
move r7
orr r1
assign r1
move r1
store [#12]
b L_1
END
```