ECE 270

Lab Experiment 12: Instructions and the ALU

IMPORTANT! You must complete this experiment during your scheduled lab period. All work for this experiment must be demonstrated to and verified by your lab instructor *before the end* of your scheduled lab period.

STEP	DESCRIPTION	MAX
Prelab	Pre-lab questions	30
1	Implement result generation in the ALU	25
2	Implement flag value generation in the ALU	25
3	Verify the ALU with more instructions	20
4	Post-lab code submission	*
	TOTAL	100

*** All lab points are contingent on this step.**

Instructions and the ALU

Instructional Objectives:

- To learn about and practice implementing an ALU
- To learn about and practice using machine instructions

Pre-lab Preparation:

- Read this document in its entirety
- Answer the prelab questions
- Review the material presented in Module 4

Experiment Description:

In this lab experiment, you will implement the details of a two-input, single output arithmetic logic unit. You will implement not only the result generation for each operation type, but the flag values as well. The operation types are provided for you and the ALU has been embedded in an "instruction trainer." You can use this system to type in instructions, execute them, and observe their results.

Immediate Instructions (format XrNN)

opcode	mnemonic	description	flags affected
0	ORI Rr,#NN	Rr = (Rr) | NN	N,Z
1	ANDI Rr,#NN	Rr = (Rr) & NN	N,Z
2	BICI Rr,#NN	Rr = (Rr) & ~NN	N,Z
3	ADDI Rr,#NN	Rr = (Rr) + NN	N,Z,C,V
4	SUBI Rr,#NN	Rr = (Rr) - NN	N,Z,C,V
5	CMPI Rr,#NN	(Rr) – NN // only set flags	N,Z,C,V
6	LDIBU Rr,#NN	Rr = [zero-extended] NN	(none)
7	(shift/rotate)		

Two-Register Arithmetic Instructions (format XrTs)

opcode	type	mnemonic	description	flags affected
8	0	CMP Rr,Rs	(Rr) – (Rs) // only set flags	N,Z,C,V
8	1	CPY Rr,Rs	Rr = (Rs)	N,Z
8	2	ADD Rr,Rs	Rr = (Rr) + (Rs)	N,Z,C,V
8	3	ADC Rr,Rs	Rr = (Rr) + (Rs) + C	N,Z,C,V
8	4	SUB Rr,Rs	Rr = (Rr) - (Rs)	N,Z,C,V
8	5	SBC Rr,Rs	Rr = (Rr) – (Rs) + C	N,Z,C,V
8	6	NEG Rr,Rs	Rr = - (Rs)	N,Z,C,V
8	7	MULU Rr,Rs	Rr = (Rr) * (Rs) // unsigned	N,Z
8	8	MULS Rr,Rs	Rr = (Rr) * (Rs) // signed	N,Z
8	9	DIVU Rr,Rs	Rr = (Rr) / (Rs) // unsigned	N,Z
8	a	DIVS Rr,Rs	Rr = (Rr) / (Rs) // signed	N,Z
8	b	MODU Rr,Rs	Rr = (Rr) % (Rs) // unsigned	N,Z
8	c	MODS Rr,Rs	Rr = (Rr) % (Rs) // signed	N,Z

Two-Register Logical Instructions (format XrTs)

opcode	type	mnemonic	description	flags affected
9	0	TST Rr,Rs	(Rr) & (Rs) // only set flags	N,Z
9	1	AND Rr,Rs	Rr = (Rr) & (Rs)	N,Z
9	2	OR Rr,Rs	Rr = (Rr) | (Rs)	N,Z
9	3	BIC Rr,Rs	Rr = (Rr) & ~(Rs)	N,Z
9	4	XOR Rr,Rs	Rr = (Rr) ^ (Rs)	N,Z
9	5	NOT Rr,Rs	Rr = ~(Rs)	N,Z
9	6	EXTBU Rr,Rs	Rr = [zero-extend byte] (Rs)	N,Z
9	7	EXTBS Rr,Rs	Rr = [sign-extend byte] (Rs)	N,Z
9	8	EXTWU Rr,Rs	Rr = [zero-extend word] (Rs)	N,Z
9	9	EXTWS Rr,Rs	Rr = [sign-extend word] (Rs)	N,Z

Load/Store Instructions (format XrTT aaaa)

opcode	mnemonic	description	flags affected
a	LDL Rr, addr	Rr = (mem[aaaa])	(none)
c	STL Rr,addr	mem[aaaa] = (Rr)	(none)

Experiment Step (1): Implement result generation in the ALU

A template file, lab12.v, is provided for you. The first thing you should do is write familiar modules from previous labs (like **scankey** and **ssdec**) that will be used in this lab. If you did not succeed in getting those modules working correctly in previous labs, you should continue trying.

The hard work of implementing things like the IDMS, register file, and debug mechanisms are already done for you, and built in to the simulator. It is instantiated into the **top** module as **support12**. You need only complete the **alu**, and add the **scankey** and **ssdec** modules to have a working instruction set.

You will make your changes in the **alu** module in the lab12.v file. A ***case*** statement in that module is used to select the operation type that is set up by the IDMS. You should set up an entry for each of the following symbolic constants (such as the example, ALU_ADD). For each one, implement the appropriate expression to produce a correct result. Each operation may produce changes to one or more flags. For each operation type, you should update only the flags that are supposed to change. For instance, the ALU_ADD example updates all four of the flag values by setting **fout** to `{N,Z,C,V}`. The ***default*** case copies input operand 1 to the output and assigns the original value of the flags `{Nin, Zin, Cin, Vin}` to **fout**. In this way, it copies the output from the input and does not change any of the flags.

The operations to implement are as follows: (implement only the ones specified)

Operation Type	Result should be	Flags updated
ALU_ADD	in1 plus in2	N,Z,C,V
ALU_ADC	in1 plus in2 plus the input carry flag	N,Z,C,V
ALU_SUB	in1 minus in2	N,Z,C,V
ALU_SBC	in1 minus in2 plus the input carry flag	N,Z,C,V
ALU_NEG	0 - **in2**	N,Z,C,V
ALU_OR	in1 OR in2	N,Z
ALU_AND	in1 AND in2	N,Z
ALU_BIC	in1 AND the bitwise complement of in2	N,Z
ALU_XOR	in1 XOR in2	N,Z
ALU_NOT	bitwise complement of **in2**	N,Z
ALU_ZXB	24 zero bits followed by the lower 8 bits of **in2**	N,Z
ALU_IN1	in1	N,Z
ALU_IN2	in2	N,Z

Make sure that your ALU remains purely combinational logic. There is no need for latches or flip-flops, non-blocking assignments, etc.

Using the instruction trainer:

The instruction trainer starts with a prompt that looks like "**`INSt    0`**".

It is waiting for you to enter a 4-digit instruction on the keypad. Excess digits will be shifted off to the left, so if you make a mistake in entering an instruction, you can simply keep pressing digits until the correct instruction appears. The instructions recognized are the single-register and two-register instructions of the simple computer described in lecture. For instance, the instruction **`6945`** should load the hexadecimal value **`00000045`** into register **`9`**.

- Enter **`6945`** with the keypad and press **`W`** to *execute* the instruction. By default, the instruction is not cleared, and you may press **`W`** again to repeat the execution of the instruction.

- In instruction entry mode, you may eXamine the flags by pressing the **`X`** key, press it again to go back to instruction entry.

In addition to instruction entry, two other modes can be used.

- Press **Z** to switch to register debug mode. It shows a prompt that looks like "`rEg     0`".
  - Press the digit 0 – F for the register you want to inspect.
  - Press **X** to eXamine the register. Press **X** again to select a different register.
  - Press **W** to modify the register. Enter an 8-digit number, and press **W** to store it.

- Press **Z** again to switch to memory debug mode. It shows a prompt like "**`Addr    0`**".
  - Enter a 4-digit memory address and press **X** to eXamine it, and **W** to modify it.

- Press **Z** once more to switch back to instruction entry mode.

Enter instructions that will exercise the ALU operations you implemented. At this point, the four flags will always be on. Making those correct will be the next step. For now, make sure that the correct values are placed in the registers for the instructions you enter. To do so, use the following instructions:

```
Initial values   Instruction    Final value       check with instructions...
R0=12, R1=49     ADD R0,R1      R0 = 5b           6012  6149  8021
R0=49, R1=12     SUB R0,R1      R0 = 37           6049  6112  8041
R1=1             NEG R0,R1      R0 = ffffffff     6101  8061
R0=11, R1=22     OR R0,R1       R0 = 33           6011  6122  9021
R0=a9, R1=9a     AND R0,R1      R0 = 88           60a9  619a  9011
R0=76, R1=11     BIC R0,R1      R0 = 66           6076  6111  9031
R0=76, R1=11     XOR R0,R1      R0 = 67           6076  6111  9041
R1=1             NOT R0,R1      R0 = fffffffe     6101  9051
R0=ffffff55      LDIBS R0,#ff   R0 = 000000ff     60aa  9050  60ff
```

Try each test by executing the "check" instructions, and then eXamining register 0. For instance, to test the "ADD R0,R1" instruction, press: **`6012 W 6149 W 8021 W Z X`**
and you will view the result in R0. Press **`Z Z`** to get back to instruction entry.

Experiment Step (2): Implement flag generation in the ALU

In the **alu** module, four wires named **N**, **Z**, **C**, and **V** are used to generate the component flag values. They are presently all assigned the value 1. Replace them with expressions that will generate the correct values as described below. Each flag will depend on the generated result (**out**) and may also depend on the input values in1 and in2.

- The **N** flag simply indicates that the most significant (leftmost) bit of the result is set.
- The **Z** flag indicates that no bits are set in the result. You may generate this flag by comparing the result to zero, or, perhaps, by using a Verilog ***reduction*** operator. For instance, the statement " **`x = | bus[23:4];`** " uses a prefix OR to generate the bitwise OR between `bus[23]|bus[22]|bus[21]|...|bus[4]`. This is often convenient for circumstances where you want to find ranges in a vector that are all zeros or all ones (using a reduction AND). You can invert the output of the reduction OR operator to generate the **Z** flag.
- The **C** flag indicates that an addition caused the highest significant bit (the leftmost bit) to carry out. Symmetrically, it represents the inverse borrow flag for a subtract operation. One way to generate the **C** flag would be to implement addition and subtraction as 33-bit (e.g., [32:0]) operations, and then use vector element 32 of the result to indicate the value of the carry. This makes a mess of sub-vector references when assigning values.
  An easier way is to remember propagate and generate tricks from carry-lookahead adders to determine what the **C** flag must be. In particular:
  - If in1[31] and in2[31] are both 1, then we know they must *generate* a carry-out.
  - If in1[31] is 1, but out[31] is 0, it must be because a carry was generated or propagated.
  - If in2[31] is 1, but out[31] is 0, it must be because a carry was generated or propagated.

  By ORing those circumstances together, you can generate the **C** flag for addition.
  For subtraction, use the same logic, but complement in2[31] in each case.
- The **V** flag (oVerflow) is also difficult to generate without adding a 33rd bit to the adder. Once again, you can use a trick to determine it. Remember that overflow is the situation where, for instance, the addition of two positive numbers produces a negative result. In other words, the input operands have the same sign bit, but the sign bit of the result is different. (And overflow is not possible when the input operands have different signs.) For subtraction, the same check can be made by complementing the sign bit of the subtrahend. Write an expression for the **V** bit that is true when:
  - for addition (either ADD or ADC), the sign bits of **in1** and **in2** are both 1 and the sign bit of **out** is 0 OR the sign bits of **in1** and **in2** are both 0 and the sign bit of **out** is 1.
  - for subtraction, the sign bits of **in1** and ~**in2** are both 1 and the sign bit of **out** is 0 OR the sign bits of **in1** and ~**in2** are both 0 and the sign bit of **out** is 1.

Once you implement your flag generation expressions, test them with various instructions. For instance, if you initialize two registers to each contain the values `80000000`, and you add them together, the result should set the **Z**, **C**, and **V** flags simultaneously.

Experiment Step (3): Verify the ALU with more instructions

Use instructions to test each of the ALU operations at least once. Use this time to verify that they are all working correctly. For instance:

LDIBU R0,#00 (Instruction 6000)

When you enter 6000, press 'W' to execute it, and then press 'X' to examine the flags, you should see: FLAg _ Z _ _

First, preset four registers:

R1 = 00000001
R2 = 00000002
R8 = 80000000
R9 = FFFFFFFF

Then, verify the operation of your ALU flags with the following operations. Enter and execute the instruction encoding on each line, and press 'X' to examine the flags. Make sure that each line that has an entry in the "Expected Flags" column matches what you see. The groups of instructions separated by a blank line can be restarted independently.

Instruction	Encoding	Expected Flags	Result stored in R0
CPY R0,R8	8018		80000000
ADD R0,R0	8020	_ZCV	00000000
ADD R0,R1	8021	____	00000001
SUB R0,R1	8041	_ZC_	00000000
SUB R0,R1	8041	N___	FFFFFFFF
NEG R0,R1	8061	N___	FFFFFFFF
ADD R0,R1	8021	_ZC_	00000000
CPY R0,R1	8011		00000001
AND R0,R2	9012	_ZC_	00000000
ADD R0,R0	8020	_Z__	00000000
NEG R0,R0	8060	_ZC_	00000000
OR R0,R0	9020	_ZC_	00000000
OR R0,R1	9021	__C_	00000001
OR R0,R8	9028	N_C_	80000001
AND R0,R8	9018	N_C_	80000000
AND R0,R1	9011	_ZC_	00000000
XOR R0,R0	9040		00000000
SUB R0,R1	8041	N___	FFFFFFFF
CMP R0,R9	8009	_ZC_	FFFFFFFF
XOR R0,R0	9040		00000000
ADD R0,R0	8020	_Z__	00000000
XOR R0,R8	9048	N___	80000000
ADD R0,R0	8020	_ZCV	00000000
ADC R0,R9	8039	_ZC_	00000000
SBC R0,R1	8051	_ZC_	00000000

Experiment Step (4): Post-lab submission: Submit your code.

Upload the contents of your completed and tested lab12.v file from Step (3) in the post-lab submission. For this and all subsequent labs, you must submit your Verilog code for testing in order to get credit for the lab.