

ECE/CS 552: Introduction to Computer Architecture
ASSIGNMENT #2

Due Date: At the beginning of lecture, October 6th, 2010

This homework is to be done individually.

Total 4 Questions, 100 points

1. (10 pts.) Do the following calculations below in single precision floating point representation. Show the floating point binary values for the operands, show the result of the add or subtract, then show the final normalized binary representation.

(1) $0.5 + 0.3125$;

(2) $12 - 3.875$.

Solution: Each of the numbers can be represented in the following single precision floating point format.

Sign (1 bit)	Exponent (8 bits)	(1), Fraction (23 bits)
--------------	-------------------	-------------------------

(1)

0.5:

0	01111110	(1).0000 0000 0000 0000 0000 000
---	----------	----------------------------------

0.3125:

0	01111101	(1). 0100 0000 0000 0000 0000 000
---	----------	-----------------------------------

0.3125 after aligning the exponent:

0	01111110	(0).1010 0000 0000 0000 0000 000
---	----------	----------------------------------

0.3125 + 0.5:

0	01111110	(1). 1010 0000 0000 0000 0000 000
---	----------	-----------------------------------

Normalize the result:

0	01111110	1010 0000 0000 0000 0000 000
---	----------	------------------------------

(2)

12:

0	10000010	(1).1000 0000 0000 0000 0000 000
---	----------	----------------------------------

3.875

0	10000000	(1).1111 0000 0000 0000 0000 000
---	----------	----------------------------------

3.875 after aligning the exponent:

0	10000010	(0).0111 1100 0000 0000 0000 000
---	----------	----------------------------------

12-3.875: use the 2's complement to represent the fractions, and do addition for two operands:

$$\begin{array}{r}
 \mathbf{0} \ 1.100000 \dots \text{ (all zeros)} \\
 + \ \mathbf{1} \ 1.100001 \dots \text{ (all zeros)} \\
 \hline
 \mathbf{0} \ 1.000001 \dots \text{ (all zeros)}
 \end{array}$$

Therefore the result is:

0	10000010	0000 0100 0000 0000 0000 000
---	----------	------------------------------

2. (10 pts.) We wish to add the instruction **jalr** (jump and link register. e.g. jalr \$rs: \$ra = PC+8; PC = \$rs) to the single-cycle datapath described in the book. Add any necessary datapaths and control signals to the single-cycle datapath of Figure 1 below. Draw your datapath neatly (hand writing OK). Specify the settings of control signals in table 1. The third and fourth rows in the table are used to specify values of new control signals you used in your design.

Solution: This is an example solution. Student can come up with their own solutions as long as the instruction, datapath and control signals are consistent.

First, define the instruction format:

Jalr \$rs: R-type

Opcode (00)	Rs = rs	Rt = 0	Rd = ra (31)	shamt = 0	funct
-------------	---------	--------	--------------	-----------	-------

We need to add two paths for our datapath: 1. old PC value => RF; 2. RF => new PC value. (See below, modules in yellow are additional modules)

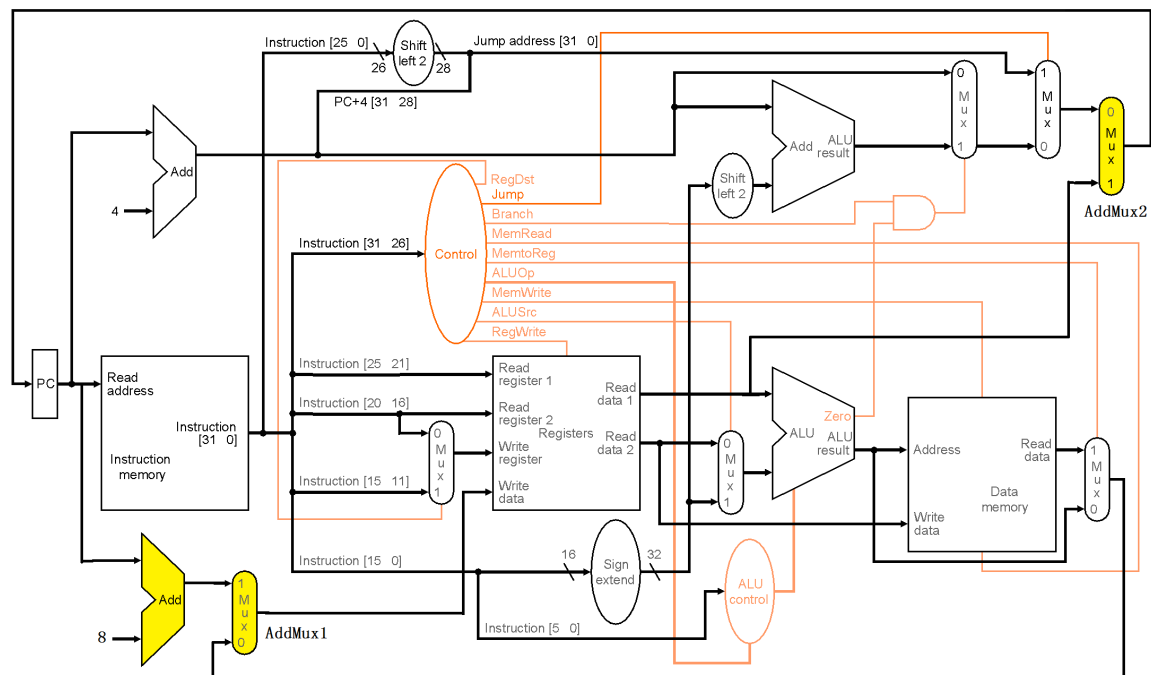


Table 1: Control settings for jalr

Ctrl Signal	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch
Value	1	x	0 (or x)	1	0 (or x)	0	0
Ctrl Signal	AddMux1	AddMux2					
Value	1	1					

3. (40 pts.) Design a 16-bit register file (RF) using Quartus II.

Specifications:

1) Design a Register-File of sixteen 16-bit registers. It has three 16-bit data buses **ReadData1**, **ReadData2** and **WriteData**, three 4-bit address buses **ReadReg1**, **ReadReg2** and **WriteReg**, a write control signal **RegWrite** and **Clock**, **Reset** inputs. The RF is edge triggered (flip flops should be used as your building block). The data on ReadData1 and ReadData2 corresponds to addresses on ReadReg1 and ReadReg2 respectively. The data on WriteData gets written into the register specified by WriteReg at the rising edge of the clock when the RegWrite signal is high. Furthermore, \$0 always reads zero. All registers are reset to x0000 when Reset is low. Figure 1 shows the block diagram of the RF.

2) The RF should support simultaneous reads and writes (2 reads and 1 write in the same cycle). For this project, you do not need to handle read-after-write hazard (when register being written is one of the registers being read from). We assume an external bypass mechanism will be used in the system.

3) You are only allowed to use schematics in your design. No verilog HDL is allowed. Also, you are only allowed to use primitives in Quartus II. That is, no megafunctions or maxplus2 library components can be used.

4) Some tips: Do not gate clock signal. And, multiplexor has poor scalability for large fan-ins. Instead, you may use tri-state buffer (available in Quartus primitives) to control the output register databus.

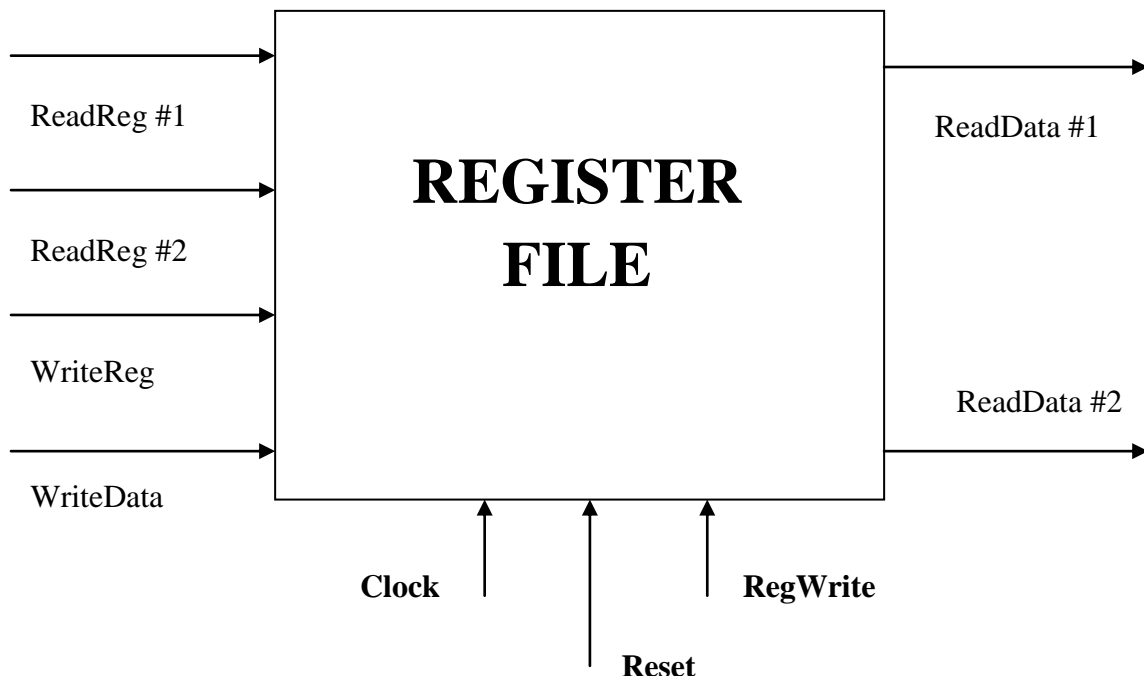


Figure 2. Register File

You should turn in:

1) A schematic of your register file. If your top level design contains low level functional blocks (e.g. you generate symbols and use them in the top level), you should also provide the schematic of those functional blocks.

2) A testbench (vector waveform file .vwf) for your design. You should provide enough test cases in your testbench in order to test the full functionality of your design. The test bench must contain and can be longer than the following code segment (“Rx <= value” here means write value into Rx. Each line indicates one clock cycle):

Expected values are in blue:

```
# !Reset
# R1 <= 0x00FF; Read R0, R2; # R0 = R2 = 0x0000
# Read R0, R1; # R0= 0x0000, R1 = 0x00FF
# R2 <= 0xFF00; Read R0, R1; # R0= 0x0000, R1 = 0x00FF
# R8 <= 0x0001; Read R1, R2; # R1= 0x00FF, R2 = 0xFF00
# Read R2, R8; # R2= 0xFF00, R8 = 0x0001
# R15 <= 0x8000; Read R1, R8; # R1= 0x00FF, R8 = 0x0001
# R0 <= 0xFFFF; Read R2, R8; # R2= 0xFF00, R8 = 0x0001
# Read R0, R15; # R0= 0x0000, R15 = 0x8000
# !Reset;
# Read R1, R2; # R1 = R2 = 0x0000
# Read R8, R15; # R8 = R15 = 0x0000
.....
```

3) A simulation result of your design. Functional simulation is required. In your simulation result, please highlight and annotate the input values **ReadReg1**, **ReadReg2**, **WriteReg**, **WriteData** and the output values **ReadData1** and **ReadData2** in the waveform. Represent the 4-bit and 16-bit waveform values in hexadecimal.

4. (40 pts.) Design a 16-bit ALU using Quartus II.

Specifications:

1) Design a 16-bit Carry Look-ahead Adder (CLA). You may start from 4-bit CLA with four full adders (FA) and a 4-bit Look-ahead Carry Unit (LCU). The LCU is used to take the propagate (**p**) and generate (**g**) signal from FAs and generates the carry signal c4 and group propagate (**PG**) and group generate (**GG**). A sample diagram of 4-bit CLA is shown in figure 3.

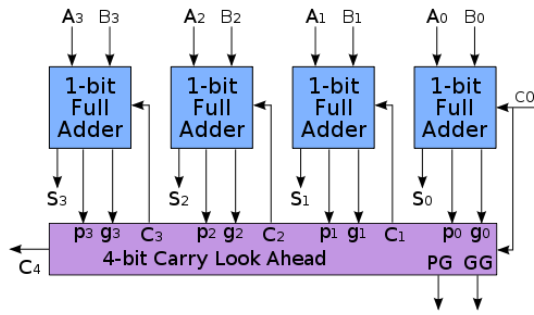


Figure 3. 4-bit CLA

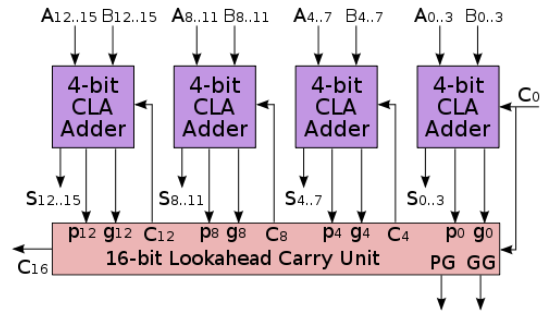


Figure 4. 16-bit CLA

You can use four 4-bit CLA to create a 16-bit CLA. The structure is similar and is shown in figure 4.

2) Extend the CLA into a 16-bit Arithmetic Logic Unit (ALU) that can perform 2's complement addition, subtraction, bit-wise AND and bit-wise OR operation. A control bus $opr[1:0]$ is used to decide which operation should be performed. Your ALU should be able to detect overflow when performing addition and subtraction. A simple block diagram is shown below.

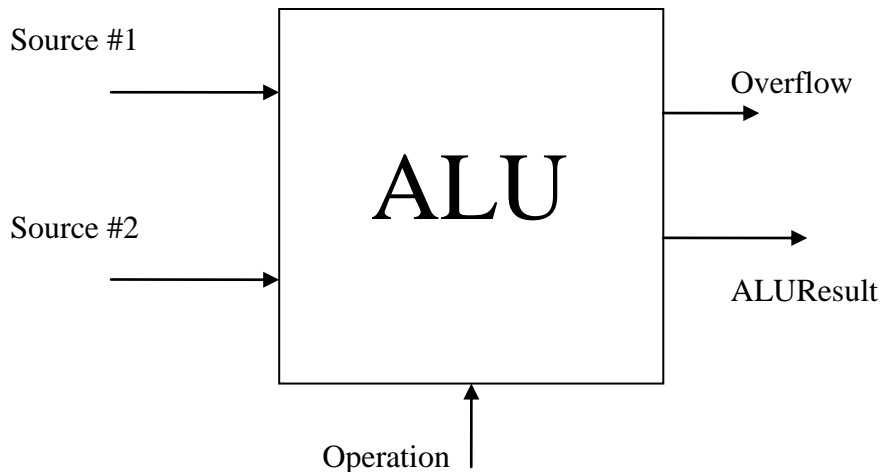


Figure 5. ALU

3) You are only allowed to use schematic or structural Verilog HDL in your design. No behavior verilog HDL is allowed. Also, for schematic design, only primitives in Quartus II are allowed.

You should turn in:

1) A schematic or verilog HDL code of your design. If your top level design contains low level functional blocks (e.g. you generate symbols and use them in the top level), you should also provide the schematic or verilog HDL code of those functional blocks.

2) A testbench (vector waveform file .vwf) for your design. You should have at least fifteen cases to show correct functionality of your design, including the following ten cases:

Expected values are in blue:

- a. $0x00FF + 0x0F00$; # Result = $0x0FFF$
- b. $0x0FFF + 0x0001$; # Result = $0x1000$
- c. $0x10FF + 0xFF01$; # Result = $0x1000$
- d. $0x1000 + 0xFFFF$; # Result = $0x0FFF$
- e. $0xFFFF + 0x0001$; # Result = $0x0000$
- f. $0x7FFF - 0x0FFF$; # Result = $0x7000$
- g. $0x1000 - 0x0001$; # Result = $0x0FFF$
- h. $0x7FFF - 0xFFFF$; # Result = $0x8000$, overflow
- i. $0xAAAA \text{ AND } 0x5555$; # Result = $0x0000$
- j. $0xAAAA \text{ OR } 0x5555$. # Result = $0xFFFF$

3) A simulation result of your design. Functional simulation is required. In your simulation result, please highlight and annotate the two input values **Source1**, **Source2**, and the output value **ALUResult** as well as **Overflow** in the waveform. Represent the 16-bit waveform values in hexadecimal. You are not required but encouraged to put on the simulation result of your 4-bit CLA (not graded).