Lab 9 : ALU Control and Data Path

Name:

Sign the following statement:

On my honor, as an Aggie, I have neither given nor received unauthorized aid on this academic work.

1 Objective

The main objectives of this lab are

- Experiment with floating point arithmetic;
- Build the basic elements of the MIPS data path that will eventually be used to build a complete processor.

2 Pre-requisite

For this lab you are expected to be familiar with MIPS single cycle processor from sections 4.2, 4.3 and 4.4 of the textbook.

3 Floating point arithmetic in MIPS

MIPS processor has 32 single precision floating point registers $f0$-$f31$. Unlike integer registers, $f0$ can hold non-integer values. Single precision operations work with these registers. MIPS also has hardware for double precision (64 bit) floating point operations. For this, it uses pairs of single precision registers to hold operands. There are 16 pairs, named $f0$, $f2$, .., $f30$. Only the even numbered register is specified in a double precision instruction; the odd numbered register of the pair is included automatically.
Actual hardware has a delay between a load instruction and the time when the data reaches the register. In SPIM there is an option that disables the load delay. For this chapter, disable the load delay. Make sure that Simulator→Settings→MIPS→Enable Delayed Loads is not checked. Loading a single precision value is done with a pseudoinstruction:

1. \texttt{l.s $fd, addr \# load register $fd from addr}
   
   This instruction loads 32 bits of data from address \texttt{addr} into floating point register \texttt{$fd} (where \texttt{d} is 0, 1, 2, ..., 31). Whatever 32 bits are located at \texttt{addr} are copied into \texttt{$fd}.

There is a single precision store pseudoinstruction:

1. \texttt{s.s $fd, addr \# store register $fd to addr}
   
   Whatever 32 bits are in \texttt{$fd} are copied to \texttt{addr}.

There is a floating point load immediate pseudoinstruction. This loads a floating point register with a constant value that is specified in the instruction.

1. \texttt{l.i.s $fd, val \# load register $fd with val}
   
   Here is a code snippet showing this:

1. \texttt{l.i.s $f1, 1.0 \# $f1 = constant 1.0}
2. \texttt{l.i.s $f2, 2.0 \# $f2 = constant 2.0}
3. \texttt{l.i.s $f10, 1.0e-5 \# $f10 = 0.00001}

Here is a program that exchanges (swaps) the floating point values at \texttt{valA} and \texttt{valB}. Notice how the two floating point values are written. The first in the ordinary style; the second in scientific notation.

1. \texttt{## swap.s}
2. \texttt{##}
3. \texttt{## Exchange the values in valA and valB}
4. \texttt{.text}
5. \texttt{.globl main}
6. \texttt{main:}
7. \texttt{l.s $f0, valA \# $f0 <--- valA}
8. \texttt{l.s $f1, valB \# $f1 <--- valB}
9. \texttt{s.s $f0, valB \# $f0 --> valB}
10. \texttt{s.s $f1, valA \# $f1 --> valA}
11. \texttt{l i \$v0, 10 \# code 10 == exit}
12. \texttt{syscall \# Return to OS.}
13. \texttt{.data}
14. \texttt{valA: \ .float 8.32 \# 32 bit floating point value}
15. \texttt{valB: \ .float -0.6234e4 \# 32 bit floating point value}
16. \texttt{# small 'e' only}
1. Can a general purpose register hold a floating point value?

To print a floating point value to the SPIM monitor, use service 2 (for single precision) or service 3 (for double precision). To read a floating point value from the user, use service 6 (for single precision) or service 7 (for double precision).

<table>
<thead>
<tr>
<th>Service</th>
<th>Code in $v0</th>
<th>Arguments</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print integer</td>
<td>1</td>
<td>$a0==$integer</td>
<td></td>
</tr>
<tr>
<td>Print float</td>
<td>2</td>
<td>$f12==$float</td>
<td></td>
</tr>
<tr>
<td>Print double</td>
<td>3</td>
<td>($f12,$f12)==float</td>
<td></td>
</tr>
<tr>
<td>Read float</td>
<td>6</td>
<td>$f0&lt;--$float</td>
<td></td>
</tr>
<tr>
<td>Read double</td>
<td>7</td>
<td>($f0,$f1)&lt;--float</td>
<td></td>
</tr>
</tbody>
</table>

2. Run the following program (print.s) using SPIM:

```assembly
.globl main
main:
    li $f12, val       # use the float as an argument
    li $v0, 2          # code 2 == print float
    syscall            # (correct)
    li $v0, 4          # print
    la $a0, lfeed      # line separator
    syscall
    lw $a0, val        # use the float as a int
    li $v0, 1          # code 2 == print int
    syscall            # (mistake)
    li $v0, 10         # code 10 == exit
    syscall            # Return to OS.
.data
    val: .float -8.32   # floating point data
    lfeed: .asciiz "\n"
```

## End of file
3. Copy the output of the program:

4. Why the first printed number is different from -8.32.

5. Due to a mistake in the program, the second number is not printed correctly. Explain the mistake and show a way to correct it.

Here are some single precision arithmetic instructions. Each of these corresponds to one machine instruction. There is a double precision version of each instruction that has a “d” in place of the “s”. So add.s becomes add.d and corresponds to the machine code that adds two double precision registers.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs.s $fd, $fs</td>
<td>$fd=</td>
</tr>
<tr>
<td>add.s $fd, $fs, $ft</td>
<td>$fd=$fs+$ft</td>
</tr>
<tr>
<td>sub.s $fd, $fs, $ft</td>
<td>$fd=$fs-$ft</td>
</tr>
<tr>
<td>mul.s $fd, $fs, $ft</td>
<td>$fd=$fs×$ft</td>
</tr>
<tr>
<td>div.s $fd, $fs, $ft</td>
<td>$fd=$fs/$ft</td>
</tr>
<tr>
<td>neg.s $fd, $fs</td>
<td>$fd=-$fs</td>
</tr>
</tbody>
</table>

6. How does the abs.s instruction alter the 32-bit pattern of the float?
7. There are three instructions that move data between registers inside the processor:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov.s $fd, $fs</td>
<td>copy 32 bits from $fs to $fd</td>
</tr>
<tr>
<td>mtc1.s $rs, $fd</td>
<td>Copy 32 bits from general register $rs to float register $fd. No data conversion is done.</td>
</tr>
<tr>
<td>mfc1.s $rd, $fs</td>
<td>Copy 32 bits from general register $fs to float register $rd. No data conversion is done.</td>
</tr>
</tbody>
</table>

You will need to write a program that computes the value of $a \cdot x^2 + b \cdot x + c$. Part of the program is given below. You will need to complete missing lines. Run your program in SPIM and verify its correctness.

```assembly
.globl main
main:
    # read input
    la $a0, prompt # prompt user for x
    li $v0, 4 # print string
    syscall

    li $v0, 6 # read single
    syscall # $f0 <-- x

    # evaluate the quadratic
    # sum = a
    # sum = ax
    # get b
    # sum = ax + b
    # sum = (ax+b)x = ax^2 +bx
    # get c
    # sum = ax^2 + bx + c

    # print the result
    mov.s $f12, $f2 # $f12 = argument
    li $v0, 2 # print single
    syscall

    la $a0, newl # new line
    li $v0, 4 # print string
    syscall
```
li $v0,10   # code 10 == exit
syscall     # Return to OS.

## Data Segment
.data
a: .float 1.0
bb: .float 1.0
c: .float 1.0
prompt: .asciiz "Enter x: 
blank: .asciiz " 
newl: .asciiz "\n"

## end of file

8. Run the problem for $x = 0.1$. Show the result and explain why it is not exact.

## Data Memory

You will be implementing the memory components, ALU control block, and next PC logic in this lab. Instruction memory and data memory are the most crucial components since they are used to store instructions and data respectively. Instructions are read from instruction memory that process data stored in the data memory. Under normal circumstances, instructions and data are stored in different parts of the same large memory blocks. For ease in this lab, we will use different memory blocks for data and instructions.

Write a behavioral Verilog code to implement the data memory of size 64 words, where a word is 32 bits in size. Remember that your code must be synthesizable. A memory write only commits to the memory on the negative edge of a clock, where read operations occur on the positive edge of the clock. In addition, memory reads and writes must have a delay of 20. Be sure to use non-blocking assignments.

**The data memory must have the following inputs:**

**Address** The 32-bit address where the data is either written or read.

**Write Data** The 32-bit data that is to be written at the address specified by the Address input above.

**MemRead** A single bit signal which should be set when you wish to read data from the memory.
MemWrite A single bit signal which should be set when you wish to write data to the memory.

Clock The clock signal to synchronize data writes.

The data memory must have the following outputs:

Read Data The 32-bit data that is read from the address specified by the Address input when a memory read operation is performed.

Note: MemRead and MemWrite must not be active at the same time.

Your Data memory module must have the port definition.

```verilog
module DataMemory(ReadData, Address, WriteData, MemoryRead, MemoryWrite, Clock);
input [31:0] Address, WriteData;
input MemoryRead, MemoryWrite, Clock;
output [31:0] ReadData;
/* write your code here */
endmodule
```

Use the Verilog code `DataMemoryTest.v` for testing your code. Run the synthesize tool to make sure your Verilog can be translated into logic.

Demonstrate your program to the TA.

5 ALU Control

As described in the previous lab, the ALU control block, takes the following 2 as inputs:

- ALUOP(4 bits) from the main control unit
• Function field in an R-type instruction (bits [5:0] of the instruction).

The output of the ALU control unit is a 4-bit signal that directly controls the ALU block.

Use the following form as the module interface:

    module ALUControl (ALUCtrl, ALUop, FuncCode);
    input [3:0] ALUop;
    input [5:0] FuncCode;
    output ALUCtrl[3:0];
    /* write your code here */
endmodule

Note: The ALUop you will use here is different from previous labs. For an R-type instruction, ALUop should be 1111. For I-type instructions, ALUop should be the value of ALUCtrl for that instruction. Your output should have a delay of 2.

Write a test bench to test your code with the following cases. Fill in the expected ALUCtrl for the given inputs before you proceed. Remember that your code must be synthesizable.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>ALUop</th>
<th>FuncCode</th>
<th>Expected ALUCtrl</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0010</td>
<td>XXXXX</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0110</td>
<td>XXXXX</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1111</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1111</td>
<td>000010</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1111</td>
<td>100000</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1111</td>
<td>100010</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1111</td>
<td>100100</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1111</td>
<td>100101</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1111</td>
<td>101010</td>
<td></td>
</tr>
</tbody>
</table>

Use $monitor or $display commands to print the inputs (ALUop, FuncCode) and the output (ALUCtrl) values for each test case. Run the synthesize tool to make sure your Verilog can be translated into logic. Copy and paste the testing result:
6 Next PC Logic

Write a behavior model for calculating the next PC for an instruction. It will use information from the processor control module and the ALU to determine the destination for the next PC.
Use the following module interface:

```verilog
module NextPCLogic (NextPC, CurrentPC, JumpField, SignExtImm32, Branch, ALUZero, Jump);
input [31:0] CurrentPC, SignExtImm32;
input [25:0] JumpField;
input Branch, ALUZero, Jump;
output [31:0] NextPC;

/* write your code here */
endmodule
```

Where `JumpField` is the jump field from the current instruction and `SignExtImm32` is the sign extended lower 16 bits of the current instruction. `Branch` is true if the current instruction is a branch instruction, `Jump` is true if the current instruction is a jump, and `ALUZero` is the Zero output of the ALU.

Any additions with a constant should have a delay of 1, general addition should have a delay of 2, and any multiplexers should have a delay of 1 (This includes statements inside if/else statements). Write a test module to test your module's correct operation. In your test code, use `$display` command to print the inputs (CurrentPC, JumpField, SignExtImm32, Branch, ALUZero, Jump) and the output (NextPC). Copy and paste the testing result:
7 Deliverables

Please turn-in the following:

- Your MIPS code.
- Your Verilog code along with the test code. Ensure that your program is clearly commented.