Lab 4 - ISA Familiarization and Single Cycle Processor

In lab 3 and 4, you are required to perform the following activities, please check out the respective eval sheet for tasks of each lab and their corresponding demo requirements.

1. Familiarize yourself with the MIPS ISA by creating assembly programs and debugging simple vhdl files before creating a single cycle processor.
2. All block diagrams and schematics must be electronically generated. Hand drawn, photo copied/scanned, or HDL Designer diagrams will NOT be accepted. No exceptions.

> NOTE
>
> 
>
> Any evaluation turned in without block diagram(s) of the design will receive a grade of 0.

3. Design VHDL code for the single cycle CPU specified in chapter 4.1-4.4 of the course text.
4. Your design must be modular. Create a test bench for each component.
5. Verify the functionality of each component using your test benches.
6. Construct the components into a data path for the single cycle processor.
7. Verify the functionality of the synthesized design using the provided test bench and assembly files.

1 Background

The instruction set architecture (ISA) that we will be implementing this semester is the MIPS ISA found in your text book. It is extremely important that you understand how each instruction functions before you design your single cycle processor.

There are a few things that you should know right away. The address space is limited to 16 bits, not 32 (memory size limitations). Also due to memory constraints the very end of memory is 0x7FFC, but this should be plenty of room for any programs you will write in assembly. There is an additional instruction not listed in the text book, this is the Load Linked / Store Conditional (LL/SC) instruction. This instruction allows you to aquire locks for threading later in the semester.

NOTE

Remember the very end of the address space? Well for the Single Cycle processor we must split the memory into instruction and data memory, so take 0x7FFC and divide by two. This gives you the new end of memory for your single cycle, 0x3FFC.

To get a detailed view of the ISA issue this command:

```
> asm -i
```

The single cycle processor is explained in chapter 4 of "Computer Organization and Design." Read these sections (4.1 - 4.4) to get an understanding of the processor and its data path. The design you will implement in lab is very closely modeled off of this design. There are a few minor differences, but these differences are due to the hardware restrictions of the fpga. These differences will be explained in this lab handout.

WARNING

The diagram of the single cycle data path in the book does not show the implementation of every instruction in the MIPS ISA. Some paths are missing.

You are responsible for implementing the ISA that is shown (excluding LL/SC, push, pop, nop, and org/chw/cfw) when you issue the command:

```
> asm -i
```

WARNING

Note: There is NO "add" and "addi" in the "asm -i" list, there are only "addu" and "addiu". You have to use only "addu" and "addiu" to write your assembly code, and only support these two in your processor implementation, Do NOT use "add" and "addi" in your assembly code even if the asm simulator works with them. The .sof file and the grading script will not support them.

These instructions constitute a majority of the MIPS ISA listed on the insert in the front of your text book. The LL/SC instruction has no relevance in this processor, thus it is left out. If you are wondering why NOP is left out, a nop simply maps to shift left logical register zero by zero. The add subtract instructions are unsigned *NOT* signed. This does not mean you are dealing with unsigned numbers. This means that the instructions you implement do not cause exceptions when overflows occur. An immediate operand to these instructions is always sign-extended.

However, the numbers are to be treated as unsigned for the unsigned set less than (SLTIU and SLTU) instructions.

NOTE

The simulator will simulate all the MIPS instructions on the insert of your text book. Even the ones you are not responsible for implementing. If this is a problem please feel free to implement the rest of the ISA.

2 Writing Assembly Code

The first part of this lab is to familiarize yourself with writing assembly code. The following is a short example of a program written in C:

```
/* this pgroam just adds integers
 * A and B and stores the sum in C
 */
int A = 0x000A;
int B = 0x00F0;
int C = 0.0000;

C = A + B;
```

The assembly code equivalent may look like this:

```
#-------------------------------------------------------------
# This assembly program adds two numbers,
# one stored in memory and the other loaded directly
# into a register.  The result is stored in memory,
# one word after the original stored
# variable
#-------------------------------------------------------------

# org <addr> is a start point for the following instructions/data
# in memory

org  0x0000
ori  $15, $zero, 0x80     #memory address of stored variable
ori  $2, $zero, 0xF0      #second number loaded through immediate addressing
lw   $1, 0($15)    #load operand from memory
addu $3,$1,$2   #add
sw   $3, 4($15)   #store result to memory
halt              #end of program

# cfw = constant full word -- not an instruction,
# only used for assembler to initialize data in memory locations
```

```
org 0x0080
cfw 0010    # 0x000A
```

NOTE

You must always make sure you align your memory accesses to addresses which are a multiple of 4. Accessing address 0x0000 accesses byte 0, byte 1, byte 2, and byte 3. If you do a memory access on address 0x0001, you will get data from address 0x0001, 0x0002, 0x0003, 0x0004. This data is bad because it is misaligned.

Now you will practice writing some assembly code. Following are the descriptions of the three assembly programs that you need to write.

2.1 Program 1: Multiply Algorithm

Since our ALU cannot perform multiplication, it would be useful to have a subroutine written in assembly for our processor to multiply two numbers. Download the multiply algorithm template from the website and place it in your `asmFiles` directory. You must construct an algorithm that multiplies two unsigned words by using a shift-add loop to form a four byte product (This could overflow the product). Remember that shifting left by 1 equals multiplying the number by 2. You have add instructions, shift instructions, and branch instructions at your disposal.

To pass data to this routine, we will set up a simple stack. We will be using the stack pointer register ($29 or $sp). $sp will point to the top of the stack. Here is one way to use the stack, this is not the only way. To pop something off the stack, you must:

1. Load the data pointed to by register $sp (LW $t1, 0($sp));
2. Add 4 to $sp.

To push data onto the stack:

1. Subtract 4 from $sp
2. Store the data into the location pointed to by $sp (SW $t1, 0($sp))

You may need to initialize $sp to 0x3FFC. This way, when pushing the first data value, you store to 0x3FF8.

When you need the two values to multiply, you will pop them off the stack and into registers of your choice. When your value is computed, you will push it back onto the stack. For example, operands will be at addresses 0x3FF8 and 0x3FF4. $sp will be set at 0x3FF4. Load both operands, multiply them, then push the result back on the stack. $sp should end up with the value of 0x3FF8. You can check these values in your memdump.hex file.

2.2 Program 2: Multiply Procedure

Download the multiply algorithm template from the website and place it in your `asmFiles` directory. You must write the assembly to fill in the template. Notice that some of the jump operations save the next program counter (PC) into a register before they jump. This is important, because you need to return from the procedure. If the following code is executed:

```
<push data1>
<push data2>
JAL mult
HALT
```

The program should branch to mult, multiply the two numbers, push the value back to the stack, then jump back to HALT.

> NOTE
>
> 
>
> Why use jump instead of branch? Branch allows you to add a 16 bit offset to your PC. Jump on the other hand allows you to change your PC to a new 26 bit value. Jump lets you traverse more of your address space.

2.3 Program 3: Calculate the number of days

This program calculates the number of days from January 1, 2000 to a given date. The date is given in your evaluation sheet. The date is passed to the routine as follows:

```
<push> CurrentYear #example 2008
<push> CurrentMonth #1 – 12
<push> CurrentDay #1 – 31
```

Use the following equation to calculate the number of days:
NumberOfDays = CurrentDay + (30 * (CurrentMonth - 1)) + 365 * (CurrentYear - 2000))
Notice that this will be an inaccurate result. Just implement the above equation. The result should be placed on the stack.

3 Generating and running the assembly program

An instruction simulator and an assembler already exist for the architecture being built for this semester.

Given an assembly language source file, you can create a `meminit.hex` file that is compatible with memory by running the following command:

`asm` *`filename`*

The resulting `meminit.hex` file can be used by the memory model or by the instruction simulator.

The program **sim** simulates an actual processor, and will execute the instructions as if a real processor would. The simulator **sim** will also show you the final value of the program counter (PC), and the values stored in the registers. After executing a program, **sim** will then dump the contents of memory to a file called `memdump.hex`.

`sim [-t] [`*`filename`*`]`

The -t flag allows you to trace your programs execution. The `memdump.hex` is output in Intel hex format so you may want to know the fields.

Intel hex format fields by digits:

1. The first field is the semicolon(;) that starts every line.
2. The second field is the data width,2 digits, in hex. This should always be 0x04, as each word is 4 bytes in length.
3. The third field is the address,4 digits,in hex.
4. The fourth field is the data type, 2 digits, in hex. This should always be 0x00.
5. The fifth field is the data, 8 digits, in hex. This is what is stored at the address, specified in field 3, of memory.
6. The sixth field is the checksum of fields 2 - 5, 2 digits, in hex.

3.1 Procedures to generate .hex file and use it in sim

1. Place your .asm files in your `asmFiles` directory.
2. Invoke this command in the project1 directory in a LINUX terminal. **asm asmFiles/xxx.asm** where `xxx.asm` is the name of the assembly file you wish to compile.
3. Now you are ready to run your assembly code by invoking the simulator **sim** using the following command:

```
> sim meminit.hex
```

NOTE

By default, **sim** executes the file `meminit.hex`. Just typing **sim** is the same as typing **sim meminit.hex**.

3.2 Verifying the correct behavior of assembly programs

Examine the `memdump.hex` file. The correct answer is always pushed back to the stack so the result should be at the very end of the dump file.

NOTE

You will notice the addresses of the dump file do not match those of the program exactly. The processor addresses have to be translated to use sequential addresses in the boards memory, this is done by dividing by 4. processor address 0x0004 maps to board address 0x0001.

Remember popping values off the stack does not actually remove the data from memory, it just moves the stack pointer.

WARNING

Remember to only use instructions listed in "asm -i". Unsupported instructions will function in the simulator but they will most likely NOT work in hardware. Another common issue causes the simulator to print the "illegal r-type instruction" message. It most likely trying to execute data. Did you forget halt? Did you overwrite instructions with data using a badly place org statement?

4 Hardware Testing

Included on the web site is a file called `singlecycle.sof`. Download this file and place it in a directory called `singlecycle` inside your `project1` directory.

This is a single cycle processor that can be downloaded to the fpga. To use this processor you must open quartus with the command:

```
> quartus
```

Wait patiently for the program to load. When it does finish loading click "Tools" on the menu-bar. In the "Tools" menu select "In-System Memory Content Editor". When the in system memory content editor appears be sure "USB-Blaster" is selected as hardware.

Now in the JTAG Chain Configuration panel select the singlecycle.sof file and program the board by clicking the icon with the arrow pointing at a chip (this is right by the word "File"). The file should begin downloading to the board. The left panel called instance manager should now detect the two instances of ram.

The instance manager allows you to edit the value at memory addresses. If you select an instance and right click you may "Import" a file to memory. These files are the ones produced by the assembler program. If quartus complains about the depth or number of addresses that is fine, it will just fill the unspecified addresses with zeros.

When you run the processor by pressing KEY3 the contents of data memory will change. You must click the refresh button to get an updated view of memory. You may now test your assembly files on a processor enjoy.

NOTE

The end of memory is 0x3FFC for the single cycle processor. Please adjust your stack pointers before you upload your assembly files to the hardware.

Use the assembler to make a `meminit.hex` file. When you program the board be sure to hold down the nReset (KEY03) while you upload the meminit.hex file to the board.

Here is an ordered list of how to program your fpga.

1. Select the ramI instance and import your meminit.hex file It will prompt you twice. It imports the file for both ramI and ramD.
2. Hold down the nReset buttom (KEY03).
3. Write to ramI (the paper with the arrow pointing down).

4. Write to ramD.
5. Release the nReset button.
6. Select ramD.
7. Read the contents of memory (paper with arrow pointing up).
8. Look for the desired memory location. Remember the address space is multiplied by 4.

5 Commonly Encountered Software Coding Bugs

There are usually three classes of bugs you might face when coding your projects. These are some examples of them.

The first are syntax errors. These are the easiest to detect and debug. The first two programs contain these errors.

The second type of bug is similar to logical errors. Programs belonging to this class may compile without error but their behavior is incorrect. The next two programs are examples of this type.

The last program belongs to the last class of bugs. Here, code usually simulates well before synthesis but either does not synthesize, or synthesizes into incorrect logic.

Download `debug1.vhd`, `debug2.vhd`, `debug3.vhd`, `debug4.vhd`, and `debug5.vhd`. Save these files to your `source` directory. Correct the errors in these programs and explain the changes on the eval sheet.

6 Provided VHDL Files for the Single Cycle Processor

There are five provided VHDL files on the web page. These files are there to provide an interface to our grading script. Remember your makefile and your entities must be correct (as we specify) in order to pass the grading script.

WARNING

It is very important that the test bench filename (`tb_cpu.vhd`) and the entity names inside `tb_cpu.vhd` and `cpu.vhd` do not change. Any changes to these files that are not approved by the TAs will cause the grading script to fail in interfacing with your processor.

NOTE

Note that the top level signals in cpu.vhd and tb_cpu.vhd (halt, imemAddr, imemData, dmemAddr, dmemDataRead, dmemDataWrite) are outputs. They are used as probes for testbench to read the corresponding signals in your system. Connect them to the respective ramd and rami signals in your design (details about halt and dumpAddr are in the next section and figure 1). For example, your address line of rami in your toplevel (mycpu.vhd) should connect to imemAddr in cpu.vhd, and that signal is connected to the same signal in tb_cpu.vhd; Therefore, your address line will be output from mycpu to cpu and then to tb_cpu, that is how testbench can monitor the status of your system. You can also use those signals for your debugging (i.e. see if pc is incremented, instruction is correctly fetched, or if correct data is stored to the correct address of your data memory.)

6.1 tb_cpu.vhd

This test bench will reset your design to start it and generate a clock. After your design is finished running it will output the content of your data ram to a file called `memout.hex`.

WARNING

It is very important that your design halt and stay halted. If it does not you will not get a dump of memory and you will not pass *ANY* of the grading scripts tests.

NOTE

You need to have meminit.hex in your project root directory to run your source simulation. If you want to change a testcase, just generate new meminit.hex by asm to the same place. Having meminit.hex in a different place like asmFiles or source folder will result in an error when you try to simulate your design.

This test bench is not synthesizable.

WARNING

For your source simulation, you need to comment out six lines with ”altera_reserved” in the test bench. For your mapped simulation, you need to uncomment those six lines.

Please keep in mind that when you do source simulation, you can change the meminit.hex without recompiling. However, when you compile the mapped version, the current meminit.hex will be hard-coded into the mapped cpu.vhd. This means if you want to change the test assembly in synthsis simulation, you need to regenerate a new meminit.hex file as well as re-synthesize your design. This will change after lab 4.

6.2 cpu.vhd

This is the wrapper for your CPU design. The entity of this file interfaces with `tb_cpu.vhd`, do not change it. The component inside this file is the top block of your CPU. This where you add your mycpu component and map its signals to the CPU signals.

> WARNING
>
> 
>
> From this lab onwards, you want to keep your very top level design entity name as ”mycpu” and your top level file name as ”mycpu.vhd”, in which you map your singals to cpu.vhd wrapper as described above. You can name all other sub-modules as you like except the provided ones on the website.

6.3 cpuTest.vhd

This file is used to test your design on hardware. You will need the file `cpuTest.pins`. This will map the push buttons and switches on the fpga to signals from your design.

> NOTE
>
> 
>
> You will need to add the 7segment decoder given to you in lab 2 to this files dependency list in the makefile as well.

6.4 rami.vhd and ramd.vhd

These files constitute your instruction ram and data ram for the single cycle design.

> WARNING
>
> 
>
> Do not edit these files, except for modifications approved by the TAs.

7 General Differences

7.1 Branches

Branches in this design are not followed by a delay slot. Thus the instruction after the branch will not be executed before the outcome of the branch. Keep this in mind when coding your design.

7.2 ALU

You will need to modify your ALU to meet the specifications of this lab, but it will remain the same from this point onward.

7.3 Memory

The ram inside the fpga is synchronous, meaning that reads/writes occur on a clock edge. If an address is presented to the memory before a clock edge, the data is available immediately after the edge. To accommodate this ram in your design, you need to consider the following points.

To have the instruction ram generate an instruction each clock cycle, the instruction ram can be clocked on the falling edge. Then the program counter can be updated on the rising edge with a new address, and on the immediately following falling edge, the instruction will be available on the instruction memory output. So, every cycle we get a new instruction from the instruction memory. The rami.vhd and ramd.vhd files are already set up to operate on the falling edge. Do not change this in the provided files.

Even with a data memory clocked on the falling edge, there is a difficulty - a load still takes two cycles to complete. This is because we need one cycle to get the data out of the memory, and another cycle to write the data back to the register file. Recall that writes to the register file are also synchoronous. This can be handled by using a simple two state finite state machine that generates a `memwait` signal. This `memwait` signal should be tied logically to your `pcwe` to stop the the program counter from advancing until the load operation completes.

7.4 The Halt Signal

When ever your processor receives a "halt" instruction (0xFFFFFFFF) you must assert a halt signal that stops your program counter. This effectively stops your processor from executing any more instructions. This signal is important as the test bench will only dump the contents of your memory when this signal is high and REMAINS high.

In order to dump the contents of your data memory, the test bench needs to supply addresses to the memory. Hence, the test-bench needs direct access to the address port of your memory. This means you must provide a method to transfer the control of the data memory address input from your processor to the test bench. This can be accomplished by inserting a multiplexer between your processor and data memory address input. When the halt signal goes high the multiplexer will let the test bench access the address input. Otherwise, the processor will supply the address to the memory. Figure 1 shows a diagram of this.

NOTE

One of the tools you are free to use is the script `test_asm`. You can run this in your project directory to test asm files on your processor. For example, `test_asm test.rtype.asm` will run test.rtype.asm on the source version of your processor and print a pass/fail message. Enter `test_asm -h` for more options.

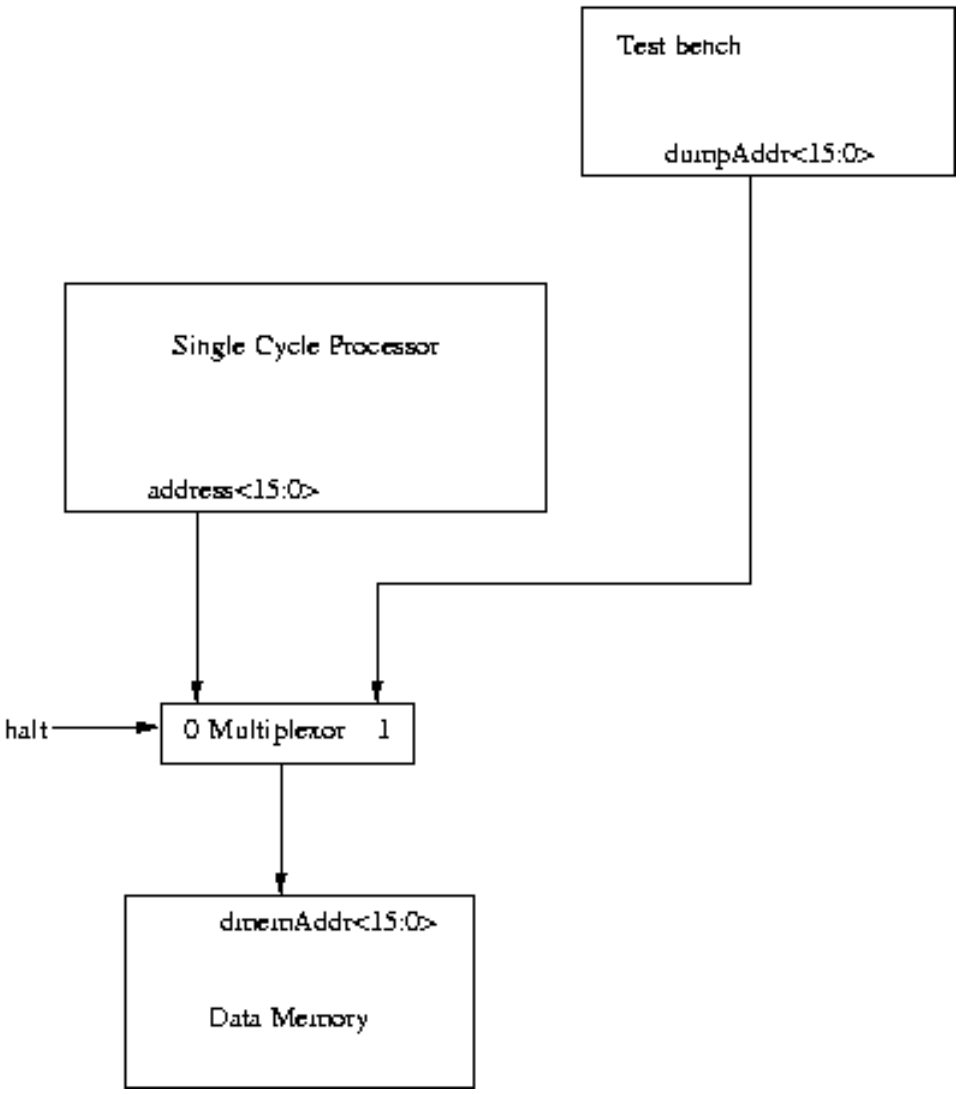
8 Electronic Submission

8.1 The Makefile

Update your Makefile so that it compiles your design when the following command is executed:

```
> make lab4
```

Figure 1 Transfer Address Control



WARNING

Incomplete compilation will cause ALL tests to fail and you will get ZERO credit for the lab. It is your duty to ensure that all files compile with NO warnings.

8.2 Submission

WARNING

Please make sure your cpu.vhd and tb_cpu.vhd are the same as given on the website. Please make sure that your own top level file is called mycpu.vhd before you do your submission.

Run the following command from your mg account:

```
> submit -p lab4
```

You can run this command as many times as you want up until the TAs disallow submissions.