EEL4713
Lab 1
Spring 2018

Overview: This EEL-4713 assignment have 2 main sections: non-rtl, rtl. Note that this lab is heavier on the non-rtl than the rtl section. Please use the ENTIRE 2 weeks for this labs as it is long and tedious.

Please help us by reporting inconsistencies or bugs you find in the lab to TAs so we can adjust and improve!

In this assignment, you will perform the following tasks:

1. Non-rtl
    a. Python Assembler and Disassembler
    b. Design strategy: Controllable delay
2. Rtl
    a. ALU
        i. Multiple implementations
    b. Sign Extenders
    c. Counters
        i. Multiple implementations

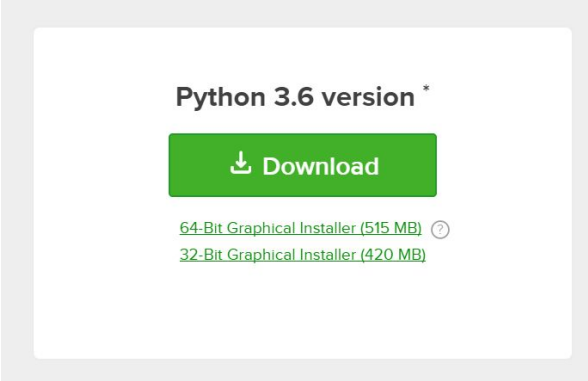
EEL4713
Lab 1
Spring 2018

NON-RTL:

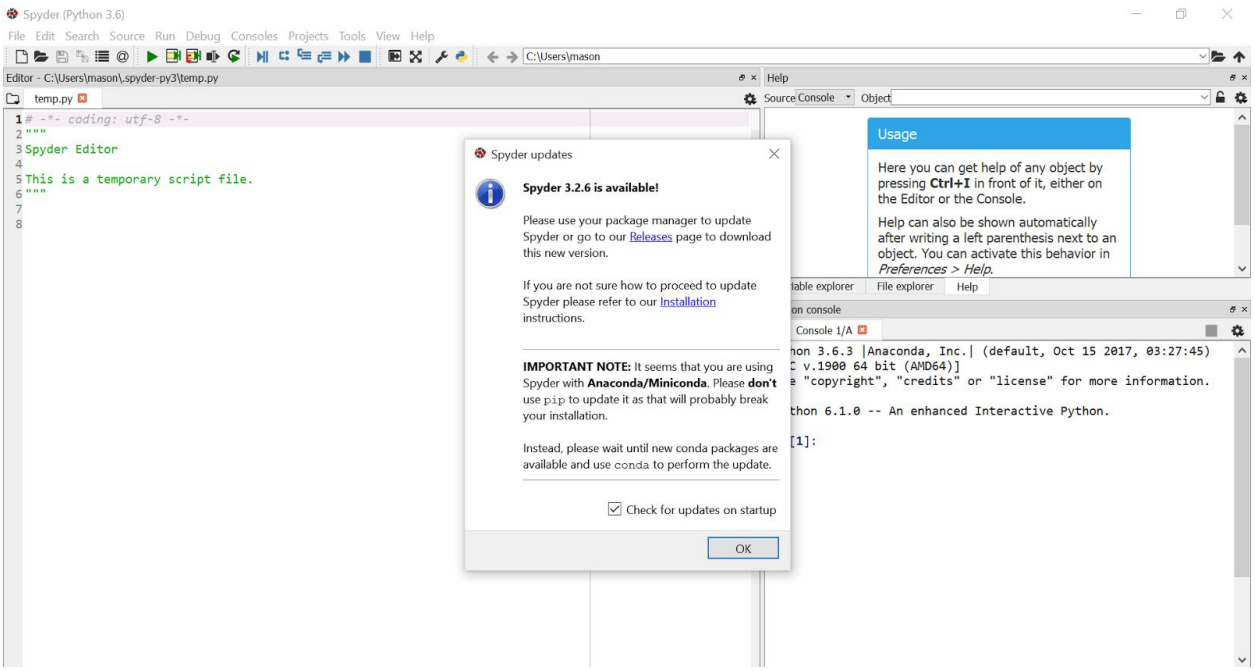
1. Python Assembler and Disassembler

If you are new to python, you will need to familiarize yourself with the basic flow. It is very similar to matlab and used throughout industry. It is open source, free, and very well supported. **If you have never touched python or feel your programming skills are weak, I recommend you start very early as this is only a portion of this lab.**

Get started by downloading and installing Anaconda for Python 3.6:
https://www.anaconda.com/download/



Once Anaconda is installed, you should be able to open Spyder which is a matlab-like python IDE. In windows 10, searching for spyder in cortona brings it up. Should look like this.



A great place to start to learn python is here: https://www.codecademy.com/learn/learn-python

EEL4713
Lab 1
Spring 2018

And here: https://developers.google.com/edu/python/introduction
And here: https://www.datacamp.com/courses/intro-to-python-for-data-science
And here: https://www.udemy.com/pythonforbeginnersintro/
And here: http://introtopython.org/

Python is very popular and there are many, many free tutorials online. Use them.

For you reference,
http://www2.engr.arizona.edu/~ece369/Resources/spim/MIPSReference.pdf
http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html

NOTE (this quite a few typos for instance in immediate instructions, should be instruction $t, $s, C)
https://github.com/MIPT-ILab/mipt-mips/wiki/MIPS-Instruction-Set#instruction-format

Note you do not need to implement/code/anything the following instructions: MIPS (IV), Likely Conditional Branches, Conditional Moves, Atomics, Prefetch, Memory Order, Operating System.

Note the differences between 1 and 2 pass assemblers.
http://www.chegg.com/homework-help/difference-one-pass-assembler-two-pass-assembler-chapter-b-problem-7rq-solution-9780134102061-exc

NOTE I am okay with multiple passes!

Q:
Create a table that details the structure of the MIPS instructions. You will not be implementing all of these instructions. Include the following columns:
Type (R, J, I), Opcode, function bits (for R-type only), ALU function code (the one used by the ALU controller), ALU function performed, flags set (C, Z, S, V).
Your ALU controller and Assembler/Disassembler should match the table you make. The portion of the table relating to ALU control signals should be similar to Figure 4.12 in your textbook. **(note: Figure 4.12 is based on a 2-bit encoding of ALUop, but you will need to extend the encoding of ALUop to more bits to accommodate the extra instructions)**

Create an mipsAssembler.py which takes an input assembly file and converts it to MIPS machine code file. Your code's starting location for this assignment is 0x00400000.

Branch and Jump details:
How the exact addressing for the instruction memory comes down to the exact memory implementation.

There are two ways of thinking about this. You can assume that the instruction memory is not byte addressable. i.e. A set of 32 bit words where the PC counter only increments by 1. This

should make some sense because we never byte address an instruction. Then the assembler can view each instruction as a single address.

OR you can assume that memory is byte addressable, each instruction is 4 simultaneous reads, and the PC counts by 4. Then when using immediate addressing, there is a left shift (<<2). In this case the assembler can still view each instruction as a single address.

The key is that at the end of the day, it should be impossible to accidentally address a non word aligned instruction.

Data will of course be byte addressable. This will be useful so we can squeeze 4 characters into one 32 bit word in memory.

See chapter 2.10 of your textbook to understand how the location of labels in your code will be addressed in your code assembly.

Recommendations:
Open files with open example: open('./assembly_0.txt','r')
Use file.readlines to read all lines
Use line.split to split a line into a string array
Use a dictionary/case/if else
string.replace is a super useful method
You can use

```
if “a” in string:
    #’a’ char is in the string
```

Create a mipsDisassembler.py which takes a machineCode file and converts it to an assembly file. Note: There will be no labels, only address, in the assembly text which you generate and that is fine.

Use the files in the lab1 resources to test your code. Place your code in src/python/ for your lab.

2. Design strategy: Controllable delay

In terms of designing and building systems, timing outputs correctly and pipelining is generally difficult to introduce to a design which is already implemented. To give an example, you will first be making a single cycle processor and later a pipelined processor. A solid solution to this is to send outputs through a delay entity at the end of the module. Thus you can pass a DELAY generic to the port. For instance, consider doing an add. If the bitwidths are small the maximum clock speeds might be for 1 clk. But for larger bitwidths, more clocks would be ideal to maximize clock speed.

EEL4713
Lab 1
Spring 2018

For here on, include a delay module in all modules before outputs. Pass the DELAY generic through your modules. Default the Delay to an appropriate number which will almost always be zero. Zero delay will cause some warnings because Quartus realizes the module is synthesized down to a wire.

RTL:

1. **ALU**

Make an ALU which supports the MIPs execution instructions. I.e. Add/subtract, Mult, Shifts, and Logical Operations. Do not do division (might be added later).

There are around 15 ALU configurations/outputs, varying slightly on your exact implementation. Note that while branch uses the ALU, it does not do an address calculation in the ALU. You can implement shift (like sll) and shift value (sllv) the same in the alu and mux the input to the ALU in the next lab. For the HI and LO, I recommend doing it in the ALU module. Alternatively you can make a mips_mult module which implements HI and LO.

Remember that the ALU does the computation, not the interpretation. For instance add and addu are the same to the ALU. The main difference in these instructions is the interpretation of the output value and looking at the flags for possible overflow or carry.

Match the ALU ctrl to the table you made at the beginning of the lab.

Make a module for add/subtract. It is okay to implement the add/sub in logic (i.e. not using an embedded DSP).

Make a separate module for mult. It is okay to implement the mult in logic (i.e. not using an embedded DSP).

Logic and shifts can be done in the ALU (i.e. you don't need to make modules for logic and shifts).

Make sure the ALU has a controllable delay.

The ALU will only ever have 2 inputs vectors. Make sure to account for setting flags C,Z,V,S in the add module (since you might use this elsewhere) and in the alu module.
See https://en.wikipedia.org/wiki/Status_register

I recommend making a separate adder and subtractor in the ALU to start.

Extra Credit (+10): Support a second implementation which instantiates Quartus IP for the multiplier and adder (i.e. use embedded DSPs). Show comparisons of timing analysis.

Tools->megawizard->...->**checkmark** the Instantiation template file->finish

Extra Credit (+3): Support an additional implementation that only uses a single adder for the add and subtraction.

See https://github.com/MIPT-ILab/mipt-mips/wiki/MIPS-Instruction-Set#instruction-format
Also See the green reference sheet in lab resources.

```
module alu #(
        parameter DATA_WIDTH = 32,
        parameter CTRL_WIDTH = 3,
        parameter STATUS_WIDTH = 4,
        parameter SHAMT_WIDTH = 5,
        parameter DELAY = 0
)(
        input clk,
        input rst,
        input [DATA_WIDTH*2-1:0] dataIn,
        input [CTRL_WIDTH-1:0] ctrl,
        input [SHAMT_WIDTH-1:0] shamt,
        output [DATA_WIDTH-1:0] dataOut,
        output [STATUS_WIDTH-1:0] status
);
```

2. **Sign Extenders**

Sign extension is done in hardware for various parts of the MIPS. Make and test a hardware sign extension unit. Make this unit as generic as possible. Input should definitely be passed an array, so you can sign extend multiple values at once.

3. **Counters**

Counters are also made it hardware in the MIPS. Make and test a generic counter. You should be able to load, pause, and reset the counter. Output the count value. The max count value should be a parameter (at least for one implementation). Use my clk_div module for help.

Extra Credit (+2): For the add, call your add module from earlier and configure it to use an embedded DSP.

DELIVERABLES:

For this lab, you will submit a zipped file of your verilog code and a document with your answers to lab questions on them. Be prepared to talk about code and lab questions in the lab section. I.e. zip up src/ and sim/

EEL4713
Lab 1
Spring 2018

Deliverables	Grade
Questions throughout lab doc (table)	10%
Assembler and Disassembler	40%
ALU	25%
Sign Extenders, Bit Shifters, Counters	25%

Extra Credit	15%

Extra (good to know, not required)
Mathematical Induction Applied to RTL
In the field I have come across many situations where a straightforward rtl solution doesn't exist. You can draw out the circuit, but a clean generic solution for it is non-trivial. In the Discrete Math course in CISE they teach a fundamental math proof technique known as induction which can be very useful in computer science. Mathematical Induction can be a great solution to non-trivial solutions.

Gain a basic functional understanding of induction.
Recommendations:
https://www.mathsisfun.com/algebra/mathematical-induction.html
http://www.bma.upatras.gr/staff/stamatiou/polypl_samos/20.pdf
See ppt in resources doc

Induction should look somewhat similar. The software analog of induction is recursion. Induction is the proof a recursive solution. A truth of FPGA designing is that Verilog and VHDL fail to make recursion easy.

The following is the most basic example for how you can use induction/recursion to write RTL.

Make a new directory and file /src/induct/induct.v
Induct module should take the same parameters as delay module. Instantiate delay module in a for generate; generate a generic DELAY number of delays. Set the delay of these modules to one. Tie the inputs of one delay to the outputs of another. Pass the output to dataOut via an assign. Use only one for generate statement. Use induction to tie the inputs to the first delay in the pipe.