

ECS 154B – Spring 2010 – Lab 1

Due by noon Wednesday 4/28

Objectives

- Build and test a single cycle MIPS CPU that implements a subset of the MIPS instruction set
- Design a combinational logic control unit

Description

In this lab you will use Quartus to build a single cycle CPU to understand the MIPS control and datapath signals. To test your CPU, you will run assembly language programs that you write on it and simulate the processor operation in Quartus. You will be given several functional blocks to help you out and an assembler that will generate a file to initialize your program memory. You may find figures 4.12 and 4.17 from your text helpful. You must implement the control signals as combinational logic. During interactive grading you will be given an initialization file to run that will use all the required instructions.

Details

For the **main part (90%)** of this lab it is important that you select "APEX II" as the device family when creating your project and logical blocks. You can change this setting by selecting Assignments→Device. Any generated logical blocks must also use the APEX II device family. You can safely ignore the message that APEX II is not a recommended family.

In a **second small part (10%)** you are asked to use a more modern device. Choose either the "Arria", or "Stratix II" family for the project. Change the device for the project and make sure all created and used logic blocks use the same device. Update all used blocks and update the MegaFunctions used. You will be given functional blocks configured for a "Stratix II" device, that should replace the corresponding blocks used for the "APEX II" device.

When generating your instruction memory, make it a word addressable memory with a depth of 128 words. Thus, no program may be longer than 128 instructions. Use `lab1.mif` as your instruction memory initialization file name. Also, you must use a 32-bit **byte** addressable program counter. For the second section you will be given a program counter in an additional logic block called "IC" that you need to use on the modern device types.

You will be given three logical blocks to start your lab: an implementation of the MIPS ALU, a register file, and a RAM module to use as data memory. These files are available on the course website. Simply download the tar file, extract the files into your project directory, and insert them like any other logic block. They will be listed under the Project folder in the Insert Symbol dialog box. The blocks have the following I/O:

- ALU
 - A,B: The 32-bit data inputs to the ALU.
 - ALUCt1: The 4-bit control input as described on page 316 of the text. `ALUCt1[3]` is the left most bit in the table.

- **ALUResult**: The 32-bit result of the ALU operation.
- **Zero**: A flag bit that is set when the ALU result equals zero (all bits are low).
- **Register File**
 - **ReadReg1,ReadReg2**: The 5-bit register addresses to read.
 - **ReadData1,ReadData2**: The 32-bit data values from the read registers.
 - **WriteReg**: The 5-bit register address to write.
 - **WriteData**: The 32-bit data to store in the register specified by **WriteReg** if **RegWrite** is set.
 - **RegWrite**: A control signal that causes the register file to be written to when set. If **RegWrite** is low, no register values will change.
 - **CLK**: A clock signal.
- **RAM**
 - **Address**: A 7-bit address used to read or write the memory. Note that the RAM size is limited to 128 words and it is **word** addressable.
 - **ReadData**: The 32-bit data located at the address specified by **Address**.
 - **WriteData**: The 32-bit data written to the location specified by **Address** when **MemWrite** is set.
 - **MemWrite**: A control signal that causes the memory to be written when set.
 - **MemRead**: A control signal that causes the memory to be read when set.
 - **CLK**: A clock signal.
- **IC (second section only)**
 - **PC_IN**: The "new" 32-bit value of the program counter.
 - **PC**: The "old" 32-bit value of the program counter.
 - **PC1_Inst**: The "old" 32-bit value of the program counter clocked differently for connection to the instruction memory.
 - **CLKF**: A clock signal (fast clock). The instruction changes only every second clock cycle.

Your CPU must execute the following instructions:

- add, sub, and, or, nor, slt, addi, lw, sw, beq, j

To test your CPU you can generate a `mif` file to initialize your instruction memory by executing `/home/cs154b/bin/mips2mif` on the instructional machines (you may want to add this directory to your `PATH`). This program takes a MIPS assembly language program as an argument, for example `lab1.mips`, and generates a `mif` file with the extension `mif`. When you create your instruction memory block, specify `lab1.mif` as the file that will initialize its contents. Note that you must resynthesize your project whenever you change the initialization file. You can alternatively select to "Update Memory Initialization File" from the Processing menu.

For this lab, you must use only combinational logic to implement the control signals. Do not use a mux with constant inputs to generate your control signals. You may optimize the functions to only work for the required instructions. Include the equations for all control signals you generate in your lab report.

Grading

To grade your lab, provide the TA with I/O pins for (at least) the following signals:

- **CLK**: The clock for your CPU.
- **PC**: The current PC before it is incremented.
- **ALUOut**: The output of the ALU.
- **WriteData**: The register file data input for writing.

Hints

- Test and debug in steps. Start with a subset of the lab requirements, implement it, test it, and then add other requirements. Performing the testing and debugging in steps will ease your efforts. For example, you could implement the R-type and addi instructions, then add the branch instruction, and finally add the memory access instructions.
- Think about the hardware you are creating before trying it out. The text is necessarily vague and leaves out details, so do not simply copy the figures and expect your CPU to work.
- In the first section use the LPM MegaFunctions whenever you need more than a simple logic gate. Some LPM blocks you may find useful in this lab include: `lpm_add_sub`, `lpm_dff`, `lpm_rom`, `lpm_mux`, `lpm_and`, and `lpm_or`.
- For the second section use `altsyncram` instead of `lpm_rom`.
- Espresso is a very handy program for minimizing large, multiple-input multiple-output boolean equations.