Part II Team 6
March 28 2013
CSE 230
Mat Reny, Phil Ringsmuth, Steve Brown

Outline

This document consists of four parts: Design, testing, testing with VSIM, and results of our testing. Design explains each stage of the processor and the components needed for the stage. Testing provides a short overview of the problems and issues we encountered while getting the processor to compile as well as an overview of how we conducted our testing. Testing with VSIM is an in-depth look at the specific commands and results that we manufactured to ensure the processor works. Results of our testing shows the wave data generated from VSIM. It also explains how we read the image.

Design

Control Unit: The control unit is the brain of the processor. The control unit is fed op codes which dictate the correct values (1 or 0) for each of its outputs. Every multiplexer that selects which data to input in to the components necessary for each stage managed by this component. The control unit was provided by the professor.

Stage 1 and 5: Stage 1 retrieves the instruction, sends the components to the control unit and the 16 bit registry file. The registry we used was provided to us in Lab 9. Our registry holds sixteen sixteen-bit [sic] values. Register Zero is usually given the value zero but for testing we changed the registry file so that r0 holds the value 15. We did this because zero is the worst number to test mathematical functions with. In this stage, the registry file finds the registers needed for the current instruction, if it is needed or not, so that it can provide them for stage 2. The registry knows which register to provide because we input the values coming from the instruction register into the two inputs that choose the correct register for regA and regB. Since our memory is not working yet, we do not have the instruction register set up. This means we have to provide the instruction codes manually through an input pin that is accessible for manipulation. We labeled this pin instruction[23..0]. This pin connects to an intermediate register which outputs to the bus IR[23..0]. We connected the two input pins on the registry labeled regA and regB with IR[15..12] and IR[11..8]. Stage 5 works to feed input into Stage 1 through the regYOut[15..0] bus and cycles into it.

Stage 2: Stage 2 takes the two correct registers from the registry file (or the immediate value which we do not use in this part of the project) and saves them to what is called a source register. Our source registers are called regA and regB. These registers are needed as an input for the ALU. Once our project accepts D-Type instructions, we will need to have a working immediate value generator that feeds into the ALU.

Stage 3: Stage 3 is where the ALU is located. This is where every mathematical operation occurs. For our project, we have four possible operations: and, or, xor, add. The ALU is fed the two inputs from stage two and outputs to a single bus labeled ALUOut[15..0]. The ALU was completed in Lab 8.

Stage 4: This stage takes the results from the ALU and stores it into memory. Since our memory interface does not work yet, we can only save this value into an intermediate register. Eventually, this stage will be where memory reading and writing takes place. There is a multiplexer that decides what the output of this stage should be. The three choices for the mux are return address, memory data out, and the value stored in the intermediate register. Regardless of the choice, the data chosen is stored into another intermediate register which we called Register Y. This register is what we use to determine if our processor outputs correct data. Register Y feeds its data to stage 5 where it is in turn fed to the register file in stage 1.

Components added to our block diagram file which are not used in this part of the project:

1. Added the instruction Address generator and MuxMA
    a. Connected the clock to the clock input
    b. Connected intermediate regA to RA input
    c. Connected the PC_select input to the pc_select output from the control unit
    d. Connected the PC_enable input to the pc_enable output from the control unit
    e. Connected the immediate input to the output of the immediate value generator
    f. Connected the INC_select input to the inc_select output from the control unit
2. Added the PC Temp to the diagram but did not connect it to other components.
3. Added the Instruction address generator
    a. Connected most of the inter connection except for the connection to memory
4. Added the memory interface component
    a. Connected the output MFC to the input of the control unit
5. Added the Memory component but made not connections

Testing

The testing of the process as built was difficult. Testing is difficult due to hand-writing bit patterns for instruction, validating that it is correct, and then tracing the outputs of the results of the numerous stages. Hand-generated instructions were required because there was no memory interface in this part of the design. Also we had to change the zero constant in the registry files to a constant integer (we used 15) that was used to test the Add operation because otherwise data was not available.

1. We added many intermediated outputs so we could trace the output of each stage of execution
    a. Added an enable input and connected to the intermediate register file so the intermediate register file will accept data
    b. We have two testing outputs connected register file 1 and register file 2 so we can test the successful storage of computed data to those to registers

c. We also found that there is a maximum number outputs supported by the Cyclone II board. The design as we have it is at the maximum number of outputs. We found the build fails if more testing outputs are added past what we currently have.
d. Pins used for testing: regYOut, alu_op, RegDest, regA, regB, RF2 (register 2), ALUOut.
   i. After the 5th cycle, we use regYout to determine if our instruction outputs correctly.

2. These outputs will be removed once no longer needed in order to make room for more testing pins.

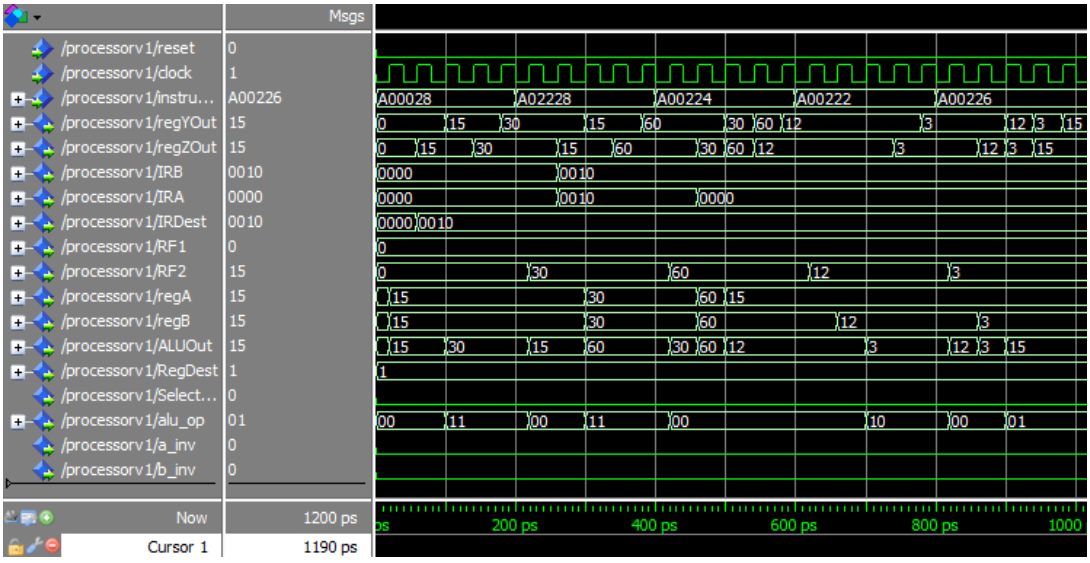
Testing with VSIM

Setup: To facilitate testing we forced the output pins regYout, regZout, RF1, RF2, regA, and regB to output their states in decimal form. The instruction pin is displayed in hexadecimal form. The clock has a period of 40ms. Its beginning state is OFF and it transitions to ON after 20 picoseconds. The reset pin is always OFF and the enable pin is always ON. With this clock configuration, the processor completes one instruction every 200 picoseconds. Every instruction begins with the code "force instruction *large_binary_number time*" where *large_binary_number* is the instruction code and *time* is equal to $200+time_{previous}$. We save the result of every instruction in register 2. The following are our tests.

1. Adding r0 with itself: To begin we decided to add r0 (which holds the value 15) with itself. This was the first thing we tried since it is the most rudimentary. The result obviously needs to be 30. In assembly, the code would be "add r2, r0, r0". Translating this to instruction bit-code results in "`10100000000000000010100`0". The uncolored bits are instruction type (4 bits), condition (4 bits), and set bit (1 bit) respectively. The full code we used to add 15+15 was "`force instruction 10100000000000000101000 0`"
2. Adding using register 2: This test was necessary because we wanted to make sure the result of the last instruction saved correctly into the register file. We also needed to test to make sure our ALU would allow using the same register twice as well as saving the result in the same register. Using the result from our previous computation, our new result when we add r2 with itself should be 60. In assembly the code is "add r2, r2, r2" and the bit-code is "`10100000001000100010100`0". Our do file code was "`force instruction 10100000001000100010100 200`". Notice that the time is now 200 since we need to do this test after we assign the value 30 to r2.
3. And using r0 with r2: This test was necessary to confirm that registers can be used interchangeably. Since r2 is now 60 (111100) and r0 is 15 (001111), the result of this operation should be 12 (001100). In assembly the code is "and r2, r0, r2" and the bit-code is "`10100000000001000100100`". We still need to increment the time by 200 so the do file code is "`force instruction 10100000000001000100100 400`"
4. XOR using r0 with r2: We tested this instruction next because it mad the most sense sequentially. The result of XOR with the values 15 (1111) and 12 (1100) equals 3 (0011). In assembly the code is "xor r2, r0, r2" and the bit-code is "`10100000000001000100010`". Our do file code was "`force instruction 10100000000001000100010 600`".

5. Or using r0 with r2: This was our last test since using r0 with any register value under 4-bits in total length is 15 (1111) and r2 holds the value 3. In assembly the code is "or r2, r0, r2" and the bit-code is "10100000000000100010010110". The code needed for VSIM is "force instruction 10100000000000100010010110 800".

Results of our testing:



The numbers in the yellow boxes are the results of our instructions listed in part 3 of this document.