Design Process Journal

Team 1E
Russell Johnson, Griffin Steffy, Charley Beeman, Stella Park

Milestone 1: Due October 3, 2017

Week Assignments:
We decided to all work collectively on this milestone in order to efficiently develop a design that we all liked and agreed upon. We met Sunday October 1st from 4:30 PM to 7:30 PM and Monday October 2nd from 5:30 PM to 8:30 PM

We decided to design a modified accumulator processor. This design would allow us to minimize the information needed for each instruction which maximizes the number of bits available for immediates in I-type instructions. Our instructions have an op-code and some have a function code. The opcode is the main label for each instruction and the function code primarily labels arithmetic operations. Because we only use additional registers beyond the accumulator in R-type instructions, we have an excess of bits available to address these registers beyond what we may need. In R-type instructions we also have the least significant bit set to be a flag to determine whether to store the result in the accumulator or the other register used. We also created an M-type instruction type used exclusively for memory instructions. This functions similarly to I-types except it only has an 8-bit immediate used to offset an address stored in the available register.

Our design has 16 dedicated registers with some of our choices resembling MIPS. Similar MIPS we have a zero, stack-pointer, return-address, (2) argument, (1) return, (2) saves, (3) temporaries, (2) in, and (2) out registers. The 16th register in our design is called $iszero. It constantly updates the register to whether the last ALU operation ended with zero as the result. This is used for branch control as we can use this register to determine if two values are equal. Primarily branches are controlled by checking the most significant bit of acc to determine if &acc is less than zero. We chose to control branches this way to be able to simplify the hardware implementation and addressing as this allows us to use the full 12-bits as an immediate to extend our branch range.

Milestone 2: Due October 10, 2017

- We met on Thursday Oct. 5 for 45 minutes to distribute the initial tasking for the weekend before our final meeting as a group on Sunday. Assignments are shown below.

Week Assignments:
Russell Johnson: Continue work on Assembler, update machine code in Design Doc, move RTL
Griffin Steffy: I-type RTL
Charley Beeman: M-Type RTL, Reformat Design Doc
Stella Park: R-Type RTL (not move)

- We also met on Sunday Oct. 8 for 3 hours as a group to combine and simplify the RTL, generate a list of component specifications, test our RTL, and edit this journal.

Based on feedback from Dr. Taylor, we made a handful of minor modifications to our design document. In the instructions section, we specified what $acc, $reg, and imm meant as well as whether or not the imm was sign extended or zero extended. We decided to get rid of the jacc instruction and add a new instruction called jr to the M-types because it gives the user the ability to jump to values in other registers and acc is now addressable. Making acc addressable also simplifies the RTL. In the registers section, we got rid of the a and v registers and used those spots for additional temporary registers in order to make the programming simpler. Also, we got rid of the second $in and $out registers so that we only have one of each that we have to use. In the addressing modes section, we specified that the immediate for Register+Offset addressing is sign extended. Also, we stated which instructions use Pseudo-Direct addressing. Reorganized the entire document by adding a hyperlinked table of contents and using tables for the Euclidean assembly and machine code section as well as the machine code fragments section.

For our RTL we decided to use a multi-cycle style of processor because of the speed benefits it provides over single cycle, and the hardware simplification that pipeline doesn't have. Our goal was to simplify the hardware as much as possible. After we created our multi-cycle RTL we were able to start a list of components for our data path and began thinking about what control signals will be needed for our hardware implementation.

Milestone 3: Due October 18, 2017

Week Assignments:

- We met on tuesday Oct. 17 in class time for 30 minutes to discuss about meeting time and got started on the milestone 3.
- We also met on wednesday Oct. 18 from 9:00am-2:00pm to design a datapath, list control signals, modify and specify component lists, and test the datapath design.
    - This milestone was all don collectively as a group

Looking at the RTL description from Milestone 2, we started by drawing block diagram of datapath. The datapath design was modified while we were going through the testing. For the testing of the RTL, load, jr, stor, jump, ori, addi, jump0, jump1, and lui were chosen. Through the testing of each instruction, we observed that RTL was working as expected.

Furthermore, we changed the format of the general components specification from chart to listing. This reformating allowed us to add on more information about building and testing each components and clarification of each components. We also added specifications for building and testing each component. Additionally, we made a detailed plan for implementing our datapath.

We ended up slightly modifying the RTL after building the datapath. We changed the relative-to-pc addressing mode so that it is PC+1 and there is no left shift because this simplifies the memory design so that everything is in blocks of 16 bits. We also modified the pseudo-direct addressing mode so that it takes the upper 4 bits of PC and 12 bits of the immediate for simplification. So, the RTL for jump0, jump1, jump, and jal all changed in order to incorporate these two modified addressing modes. Lastly, we modified the store RTL by loading acc into A at the end of cycle 3 in order to make the store faster in cycle 4.

Our accumulator-based architecture influenced our design of our datapath by simplifying writing to memory since the address always come from the ALU and the data always comes from acc. Also, everything is centered around a register file which we are familiar with and is easier to use. Based on the way that our RTL was laid out, multicycle was the most logical fit.

Our process for testing was to test the basic functionality of the component with multiple cases as well as making sure that edge cases were tested if needed. As described previously, we did end up modifying our RTL and addressing modes slightly based on errors discovered while making the datapath.

Milestone 4: Due October 25, 2017

Week Assignments:

- Griffin: Xilinx implementation and testing of Muxes, Extenders, and shifter, Reg File
- Russell & Griffin: Xilinx implementation and testing of ALU and Register File, revamp integration plan, and start integration
- Charlie: Labs and FSM
- Stella: Updating Design Document and Control Unit Design

For this milestone we began by fixing the RTL for the move instruction. We recognized that at this point we can't read and write from the register file properly simultaneously, so we split the operation into two steps. Second we saw that some of the wires and muxes were unneeded/mistakenly pointing to the wrong location in the datapath so we fixed those issues as well.

We started by implementing the easy components like Muxes, Extenders, and shifters. We verified that those components were working to our standards through our test benches. We also implemented our ALU and tested to make sure that all operations worked correctly as well as the "iszero" functionality, however, we do not have overflow implemented at this time. Next we began implementing the Register File and created a test for that as well. We tested the reading and writing functionality to make sure that is was working properly as well. After the Reg File we started Implementing other components except the memory component.

Before we began integrating our design we first redesigned our plan to be more simple and easier to follow than originally. After implementing our components we knew which components would be the easiest start with so we decided to start with the register File as well as the A, B, and IR registers. This was the first step we implemented and tested by making sure that you the register file would read and write properly. We then implemented the second step which involved adding the mux for controlling the input to A, the mux for controlling the input to B, and the mux controlling whether you SE or ZE the 12 bit immediate going into B. We were able to get a successful test bench from that step as well.

The last thing we did for this operation was design the Control Unit for our datapath. We decided to use a Finite State Machine because it complied with multi-cycle the best. We also began thinking about how we were going to test our control unit. We decided that we would test it similar to our components, by looping through the possible situations and ensuring that the correct control signals were being triggered properly. We will do this testing in steps for each type of instruction. So the single test is not an information overload.

Milestone 5: Due November 1, 2017

Week Assignments:

- We met on Sunday Oct. 29 from 4:00 pm to 7:30 pm to modify RTL, datapath, and control unit and continue on implementing parts, unit tests, and processor.
- We met again on Oct. 30 in class time and from 5:00 pm to 7:00 pm to additionally work on processor and integration..
- We met once again on Oct. 31 from 10:00 am to 11:30 am to fix some previous RTL and integration steps, as well as continuing to work on the control unit. We also met from 5:15 pm to 10:00 pm to finish up the control unit and integrate the last step of integration.
- We also met on Nov. 1 from 10:00am to 11:30am as well as 1:00pm to 2:30pm in order to go over the control unit and start implementing tests for it
- Charley: Control Unit design and testing
- Griffen: help Russell with xilinx testing, Visio datapath, test Control Unit
- Russell :finish implementing and testing all parts, integrate xilinx, test Control Unit
- Stella: update design doc, assist with testing

Our first task of the week was adding a instruction that we forgot originally, and that instruction was jal. Since this is instruction is very similar to jal, we had the workings of it figured out already, it was just missing from the RTL. Second we optimized a few cycles on our original RTL, by combining cycle two of the I-Type jump0, jump1, and arithmetic instructions. We were able to combine them into a single cycle instead of splitting the jumps into two seperate cycles.

By changing our RTL and adding in an instruction we had to make changes in our datapath and FSM control unit design. We changed the toAcc control to destAddr and it became a 2 bit signal, Also we changed the 5 bit input mux for destData to become a 6 input mux. We added some branches on the FSM to accommodate for the jal instruction, as well as match our optimized RTL. The next step we took was when we were redoing our datapath we recreated it in the microsoft program visio, to make the final design much cleaner. We added the new files to the repository.

Next we finished up integration by integrating the final steps of our design. We developed a graphical representation of our steps in the plan to aid us when doing the integration. We created exhaustive tests for each integration step, which also verified the step before it was workings as well. The last stage we implemented was memory because we were waiting to see if our Lab results were optimal. We are starting with block memory because it is the most simple type, but later we are leaving the option open to use distributed memory to increase the speed and performance of our processor.

Next based off of our FSM we began implementing our control unit into xilinx. We used a case switch architecture to switch the unit between states based off of the previous state. We will test this unit by inputting different opcodes and ensuring that it follows the correct state path. We will begin by testing one instruction at a time to verify visually with the waveform, but once we begin adding more and more instructions to test we will begin using pass fail statements to verify our results, after we understand how to test the control unit properly.

We modified our control unit testing in the design doc to be based off of our FSM in order to make testing more practical. We also wrote a general system test plan in the design doc. Lastly, we touched up our FSM and Datapath to make sure we had consistent naming structure for all the control signals.

Milestone 6: Due November 8, 2017

Week Assignments:

- We met as a group on Sunday, Nov. 5 from 2:00pm to 11:00pm in order to work on integration of the control unit into our complete processor schematic. We also made modifications to the datapath and control unit as specified below.
- We met on Monday, Nov. 6 from 5:30pm to 7:30pm in order to look at our potions for using different types of memory: single port block, dual port block, and distributed.
- Also, we met from Tuesday, Nov. 7 at 7:30pm to Wednesday, Nov. 8 at 3:30am in order to finalize the full implementation of our processor, maximize speed and efficiency, and calculate performance of our processor
- **Griffin**: Integration testing of overall processor as well as the control unit, update datapath & FSM, update design doc, assist Russell with using LCD on FPGA
- **Charlie** :Integration testing of overall processor as well as the control unit, work on setting up input for processor on FPGA
- **Russell**: Integration testing of overall processor as well as the control unit, finalize the processor schematic and testing in Xilinx, finalize the assembler, update design doc, set up output to LCD on FPGA
- **Stella**: Integration testing of overall processor as well as the control unit, update design doc

We integrated the control unit into the final processor schematic in Xilinx. We ran a rigorous test procedure including the testing of a multitude of instructions that tested every possible state in the control unit and made sure that the correct outputs were achieved. While testing the addition of the control unit, we ended up having to make a handful of hardware modifications to our processor. We found out that we needed to use a buffer in order to append the upper 4 bits of PC to the 12 bit immediate for jumps. Xilinx had a 1 bit buffer symbol, so we made our own append symbol that took in the two signals, used 16 of the 1 bit buffers, and output the appended 16 bit value. We also decided to switch from Single Port Block Memory to Simple Dual Port Block Memory so that we could have a separate read and write address which saved us having to add in additional delays and simplified our design. We also needed to have our instructions available a state earlier so we added a mux that switches between the output from the IR register and the output from the memory unit; so we can pass the instruction to the desired places in the datapath a full cycle earlier whenever IRwrite is enabled (Bypass the waiting for the instruction to be written to the IR register and then read), once IRwrite is set back to 0, then the output from IR is passed through. This solution solved many of our initial issues with programs not running properly. We decided to make the processor faster by eliminating the double clock for the memory unit and instead put in a couple delay cycles into our control unit: 1 that all the jumps go to in order to give enough time for the PC to be updated before reading the next instruction and 1 that all instructions go to before returning to the first state in order to give the memory unit time to read. This increased the processor from around 50MHz to around 90MHz.

After making the changes to our design, we ran through the control unit to finalize any changes made, and note them. After that we updated the FSM to then show the changes we made. This included adding write signals, adding in wait states, as well as reset states.

Once we finalized our changes in our design, we update our datapath as well. This included adding muxes, changing a few connections. We updated our design using visio.

We then ran the relPrime program with an initial “n” value of 0x13B0 that is placed in the $in register, and we got the correct answer of 0x000B in the $out register.

After our processor was complete, we began doing performance analytics on our design. We started of by finding the number of bytes used by our program. Using the memory file after running a simulation we were able to use that to find the total number of bytes between program and data memory to be 134 bytes. We then found the number of instructions and cycles when running our relPrime program with the input 5,040. We used our control unit to do this by displaying a counter in two different locations. After we obtained those values, we used the Synthesis report to find our cycle time of 10.669ns (93.733 MHz). We used these two sets of numbers to calculate our programs execution time. We calculated our execution time to be 4.358 ms. After getting the execution time, we found the Utilization summary from the synthesis report.

We then started to work on putting our processor on an FPGA board as well as setting up inputs and outputs. Eventually we ended up setting it up so that the user must first move the rotary button until the desired input is displayed on the LCD and then hit the reset button in order to run the new input through the processor and have the correct corresponding output displayed on the LCD.