Katy Kahla
James Kress
13 April 2012

Part 4 – Single Cycle Data Path

Design

This part of the project, we built a non-pipelined processor using the components that we had previously built during the course of the semester. We had to demonstrate that our design works correctly in simulation and show how fast it would execute.

We began our design by drawing a simple data path based on paths shown in class. This first data path would only implement instructions that were demonstrated in the book and was expanded in order to implement all instructions. From the basic data path, we designed our control unit and what signals would be designated to. Our finished control table can be seen below. The finished data path can be seen under the schematic section.

Operation	OpCode	RegDst	RegWrite	ALUSrc	ALUOp	MemWrite	MemToReg	PCSrc	jumpSrc	jlrSrc	retSrc	luiSrc	lliSrc	cgtSrc
NOOP	0000	0	0	0	000	0	1	0	0	0	0	0	0	0
ADD	0001	1	1	0	001	0	1	0	0	0	0	0	0	0
SUB	0010	1	1	0	010	0	1	0	0	0	0	0	0	0
NAND	0011	1	1	0	011	0	1	0	0	0	0	0	0	0
XOR	0100	1	1	0	100	0	1	0	0	0	0	0	0	0
SHL	0101	0	1	0	101	0	1	0	0	0	0	0	0	0
SHRA	0110	0	1	0	110	0	1	0	0	0	0	0	0	0
CGT	0111	1	1	0	010	0	1	0	0	0	0	0	0	1
SW	1000	x	0	1	001	1	x	0	0	0	0	0	0	0
LW	1001	0	1	1	001	0	0	0	0	0	0	0	0	0
BNQ	1010	x	0	0	010	0	x	1	0	0	0	0	0	0
LUI	1011	0	1	0	000	0	1	0	0	0	0	1	0	0
LLI	1100	0	1	0	000	0	1	0	0	0	0	0	1	0
JLR	1101	0	1	0	000	0	1	0	1	1	0	0	0	0
J	1110	x	x	x	000	x	x	0	1	0	0	0	0	0
RET	1111	x	0	x	000	x	x	0	0	0	1	0	0	0

In this report, only the instructions that caused modification to our simple data path will be discussed. Operations ADD, SUB, NAND, XOR, SHL, SHRA, SW, and LW worked on the basic data path that we first implemented. We only ran into problems with these instructions as we originally placed our PC module in the incorrect place in the data path. Originally, even if there was a reset value sent, it would increment the PC value and output 1 instead of outputting 0 as it should have. We simply had to move or PC module to the end of the PC path after the adders and muxes.

The first instructions that we implemented were LUI and LLI. We saw from the test benches that all tests needed to implement these instructions. Our original design of LUI created a new mux into the alu that would select between readRegAOut and the shifted immediate value. It would then add that value to a zeroed out register and store it to the correct register. The LLI originally relied on the idea that the last eight bits of the register would be all 0's and then it would add the unsigned immediate value to those 0's. This approach however, would not work properly if LLI was used solely by itself. Instead, to implement this, we created two muxes along the write data

back path. The first would use the luiSrc control signal and if it was 1, would change the write back data to the correct shifted value with 0's in the last eight bits otherwise, it simply let the data continue to flow along the path. The second mux would use the lliSrc control signal and if it was 1, it would take the unsigned immediate value and readRegOutB bits 15 to 9 and mess them together then write that data back to the write back path. Otherwise, it would simply let the write data to continue to along the path.

The next instruction that we worked on was the BNQ instruction. Part of how to implement this instruction was shown in the book. However, that implementation was for a BEQ instruction. To properly implement this instruction, we found that we had to invert the isEqual value from the alu and AND it with our control signal PCSrc. We then used this as a control signal to the mux after the second PC adder. This second adder would add the imputed branch address to the current PC and if we were supposed to branch, would allow that address to be assigned to the PC. After figuring out the logic of the inverted isEqual, this instruction worked as expected.

The next instructions that we implemented were J, JLR, and RET. For J, we created a jumpCalculate module that would take the PC [15:12] and the given jump address and mesh them together. This was then inputted into another mux placed after the branch much. To control this mux, was created another control signal called jumpSrc and was 1 for both J and JLR. JLR was slightly more complicated to implement. We ended up adding another mux along the write data path that would use a new control signal, jlrSrc, and if it was 1, set the write data to the calculated jump address. This address was then written to register 7. For ret, we found that we had to add another mux to the PC path after the jump mux. This mux would take the current address as one input and then the value from readRegAOut, which was the value in register 7, and use the signal retSrc to select between the two.

We found that when testing these instructions, that the inputs to the jlrMux were incorrect. We originally were giving it the calculated jumpAddress and the write data. Instead, we needed to give it the output of the first pc adder and the write data. By fixing this, we were able to correctly implement JLR, J, and RET.

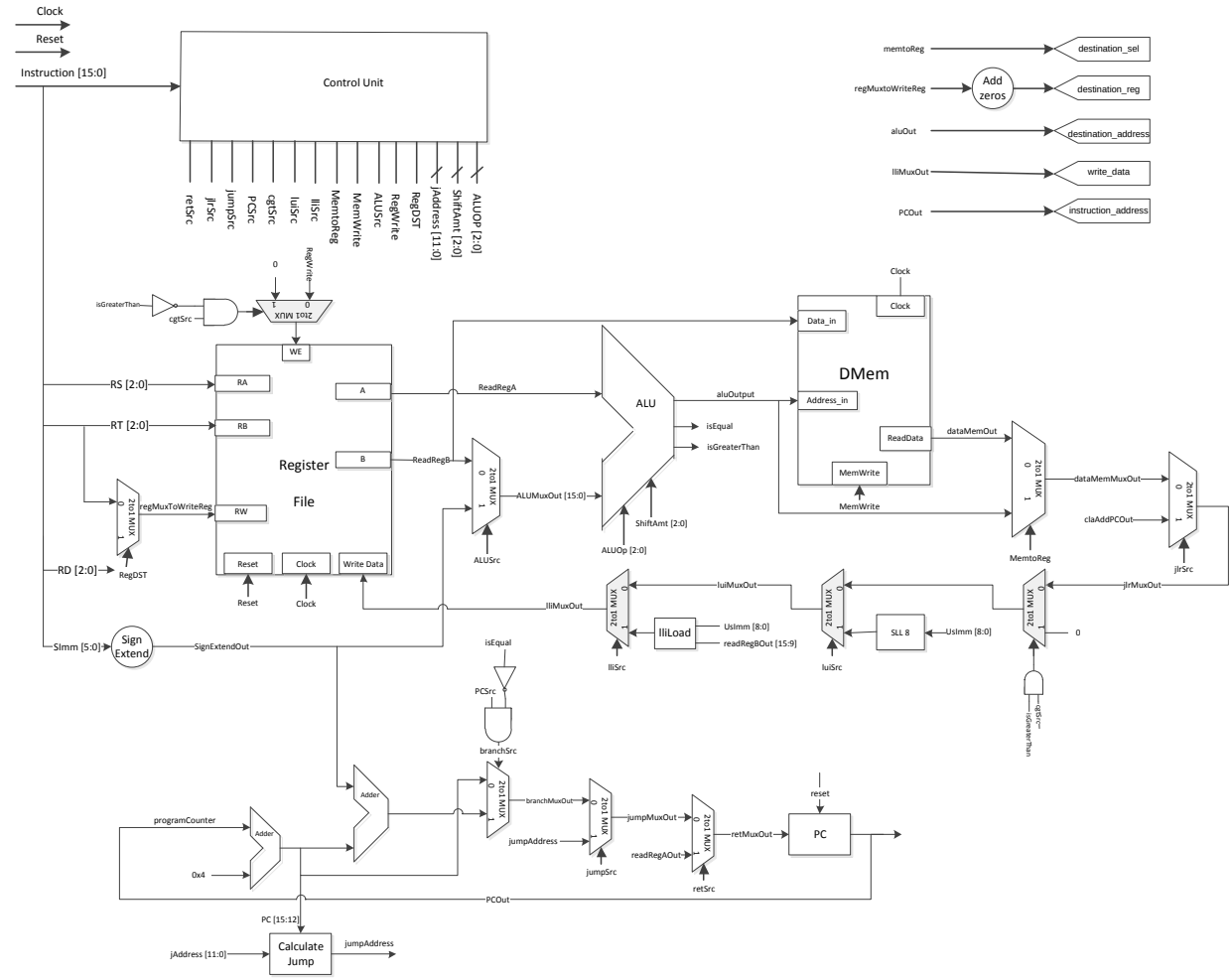
The final instruction that we worked on was CGT. For this instruction, was also added a mux to the write data path to control when we need to write 0's to the register and when we should not. However, for this we noticed that sometimes we did not want to write any data at all meaning that we had to update the RegWrite signal. To do this, we created a mux for that signal that would use an ANDED cgtSRC and inverted isGreaterThan signal as its inputs. We were not able to get this instruction to function properly however. We saw in schematic that the isGreaterThan value would change to 1 and the corresponding muxes would change to the correct values but the mux values would not latch to that value for some reason. We tried using all different sorts of always blocks and assign statements to fix this but were unable to get the mux values to latch without messing up other instructions. In the end, we left the logic in place hoping to fix it later.

Our final design while implement all instruction except CGT which we hope to fix in the pipelined version of this program.

Performance

We tested the performance of our data path extensively while adding module and after completing the path. We determined that fastest that our data path could go was 105 ns. We made sure that this time would work for all instructions that were given to the data path. We used not only the tests that were given but would modify those and add to them to test the functionality.

Schematic



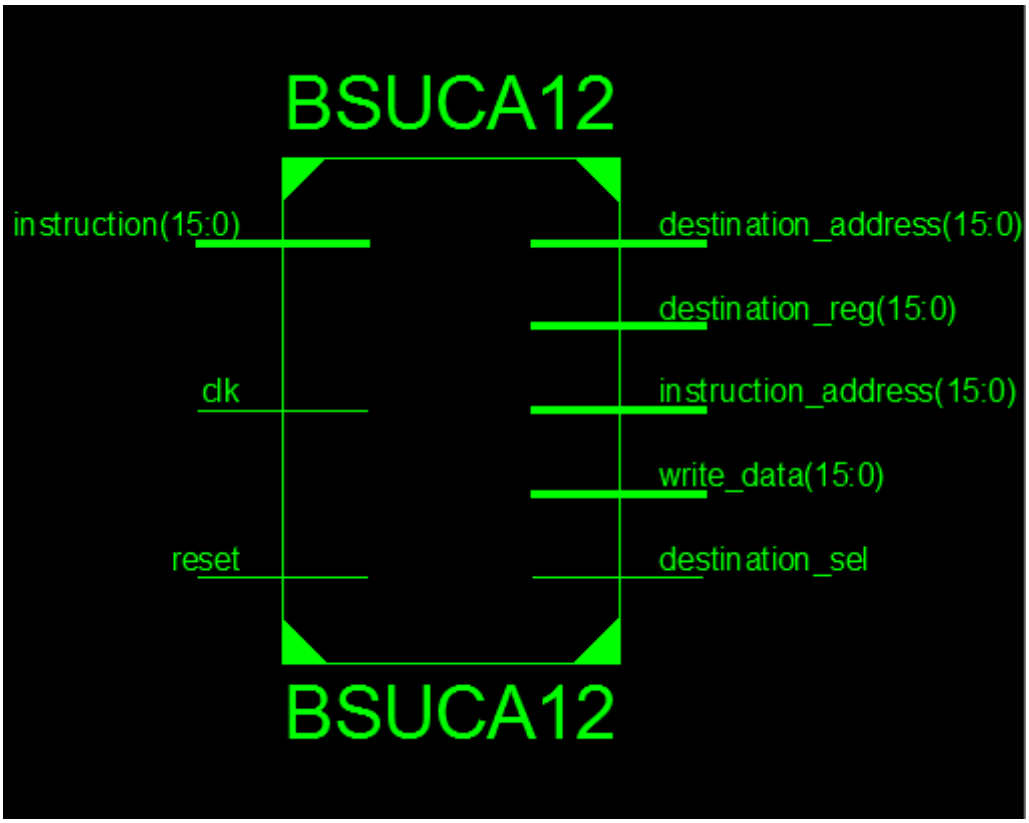


Figure 1: Capture of the Top Level Processor

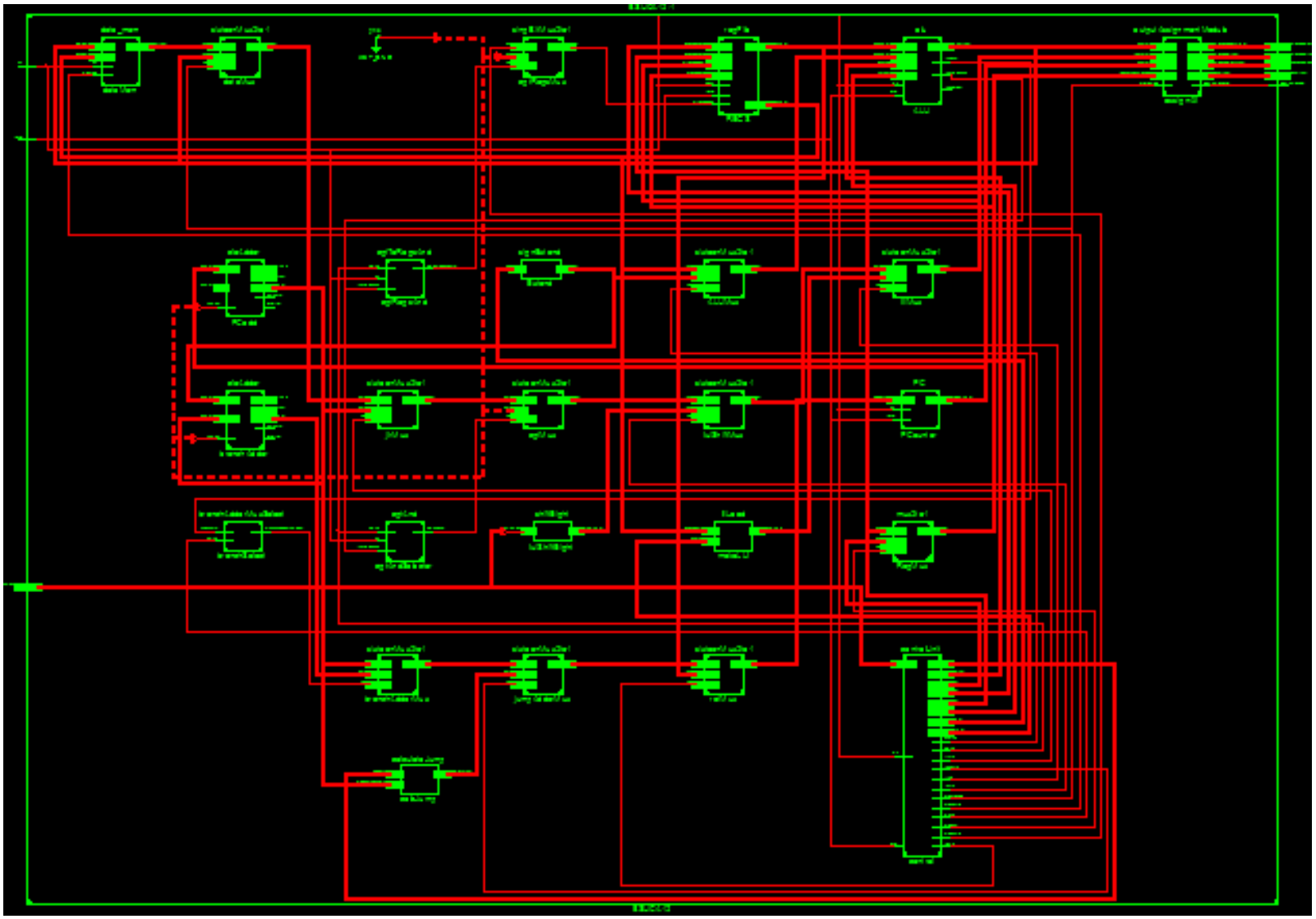


Figure 2: Capture of Processor's Internal Components

Testing

To test our processor we went through many iterations of simulation, waveform reading, and diagnosis. We started off our processor by implementing the more basic instructions such as add and subtract and then running tests on them. To do this we modified the given test benches and did cases as individual cases rather than using a large testing structure. This made the testing of edge cases a little easier to focus on.

After we had the basic instructions working, we moved on to the more difficult like jump and link, jump and cgt. We got all but cgt working. We attempted to diagnose cgt for many hours, and kept running into the problem of waveforms not holding their value when we needed them

to. We couldn’t come to a solution through testing, so it is the only one that was not fully implemented.

Overall, testing was a combination of synthesis and checking all of the individual components as we designed them, using the waveforms in modelsim. This gave us invaluable information and allowed us to modify several of our components before implementation, saving debugging time later.