Erich Viebrock

ECPE 173

Lab #5

Problem Statement

In this lab, implementations are added in order to create a more functional Beta. Newly added functionality includes branches, jumps, exceptions, and interrupts. This will allow the Beta processor to handle situations with jumps, the premise behind each of the added functions, an integral function for processors.

Design Approach

In order to implement these features, I will be moving through each step in alphabetical order, as depicted in Figure 2. First, I will implement the enhanced multiplexer feeding the PC. Then, I will implement the multiplexer with WASEL as a select, feeding the register file. After that, I will implement the functionality for sign-extension. Then, I will create the multiplexer choosing from radata or the sign-extended number, which is feeding input A of the ALU. Then, I will create the enhanced multiplexer feeding WD of the register file. Finally, I will add the inputs into the control logic, IRQ and Z.

Implementation

The first feature I decided to implement was section A, the enhanced portion of the program counter. Initially, I noticed the multiplexer used for the program counter now had 5 inputs, instead of the previous inputs. I also noticed RESET was no longer a select,

and was now PCSEL. However, the program counter still needs to identify RESET as a select. Therefore, I recognized two multiplexers would need to be used. When making a truth table for every possible outcome of PCSEL and RESET select scenarios, it became clear 2 four-bit multiplexers would need to be used. By looking at Figure 2, the inputs of the first multiplexer were PC+4, PC+4+C, JT, and ILL_OP, corresponding to input 00, 01, 10, and 11 respectively. PCSEL [1] and PCSEL[0] were used for selects, being PCSEL[2] remained constant for these 4 inputs. For XAdr, being this input is the only input used when PCSEL[2] is high, can be used in a separate multiplexer with PCSEL[2] being a select. However, in order to include our RESET select and functionality from the previous design, a second 4-bit multiplexer is used with PCSEL[2] and RESET as selects. I tied the output of the first multiplexer into the first input of the cascaded multiplexer, being a PCSEL input of 0XX would output the function from the first multiplexer. Then, I set input 1 and 3 of the cascaded multiplexer to RESET, as this would RESET the program counter whenever RESET is high. Finally, I mapped XAdr to input 2. However, given the problem statement, XAdr and ILL_OP are the constant values 8 and 4 respectively. Therefore, I created two 32-bit buffers of values 8 and 4 for XAdr and ILL_OP. The debugging for this section was not entirely difficult. However, remembering what output I created in a previous lab that I made for PC+4, was difficult. Once I found the correct output, my design for part A was functional.

Although I originally planned to implement my design in alphabetical order, I was intrigued to go after part C being it appeared simple. I recognized this section as PC+4+C, therefore I sign extended PC+4 to the 15^{th} bit of C. In order to implement this, I created a 32-bit buffer. I declared bit 16 to 31 as whatever the 15^{th} bit of C was, and bit 0

to 15 would be mapped to bit 0 to 15 from PC+4. When debugging, I noticed sign-extension is needed in the control subcircuit, as well as the program counter subcircuit. Therefore, I declared the same line of code in each subcircuit, because I was running into problems of the buffer falling out of scope. This is a lazy approach to solve this problem, but given the JSIM knowledge I have, was the only solution I could think of.

The next logic block I implemented was section B. This section seemed simple as well, as I was only adding 5 two-bit multiplexers. The reason I created 5 multiplexers is because I needed to use 5 inputs of RC with XP. Therefore, each multiplexer would use a bit from RC and XP, until the 5 bits from RC were used. WASEL was the used as a select for the multiplexers, with the multiplexers feeding in to WA. Debugging this section was very simple, given it was only a single line of code.

Then, I implemented section D. This section needed the addition of a multiplexer, a buffer, and Boolean logic gates. First, I designed the logic for Z. In order to get Z, I noticed the input was fed through a NOR gate, which has the truth table of only having output high when all inputs are low. Therefore, because JSIM only allows for 4 input NOR gates, and we are importing 32-bit value, 8 NOR gates would need to be used. Once all the bits are fed through the NOR gates, we can feed those outputs through AND gates, until we finally have a single output Z. To create JT, the supervisor bit, I sent the 31st bit of radata and the 31st bit of ia through an AND gate, to declare the 31st bit of JT. I also directly mapped bits 0 to 30 of ia to bits 0 to 30 of JT by means of a buffer. To implement the multiplexer used with ASEL as a select, I created 2-bit multiplexers inputting all 32-bits of radata and all 32-bits of SEXT, with ASEL copied 32 times as a select. These outputs were mapped to overwrite all 32-bits of radata. Debugging this section was not

difficult, but I think I may have been able to write fewer lines of code when I was creating my design for Z. JSIM may have a more optimized method than what I used, but regardless, I was able to implement what I intended.

Finally, I created the logic section of E. This section was incredibly easy to implement, being it was a slight add on from the previous design. The new addition was adding PC+4 as the 0 input, which could be directly ported from the PC+4 I created in the program counter subcircuit. Debugging this line of code was not difficult, because it was a simple add-on from the previous subcircuit.

Unfortunately, I did not get to part F, upgrading the control logic. This required the addition for IRQ and Z. I also was unable to implement my Beta to handle any instructions with jumps, including branches, jumps, exceptions, and interrupts. I plan to implement these features very soon, as the proceeding lab will surely use them.

When it came to debugging my design, finding errors was troublesome. If I compiled, I would receive a single error notification, indicating where there error was occurring. After fixing an error, a new error would arise. Although it would make sense that fixing each error one by one would eventually lead to a functional design, a fix to correct a line of code may cause a different problem elsewhere. Therefore, it is a bad idea to solely rely on the compiler to pinpoint the mistakes in the design. Therefore, I would also use my best judgment when reviewing my code for errors. If the logic made sense as to what I was trying to achieve, and how it was implemented in JSIM, then the problem most likely wouldn’t reside in where the compiler was indicating. Rather, the problem may had been caused by something that portion was using, which was creating an error.

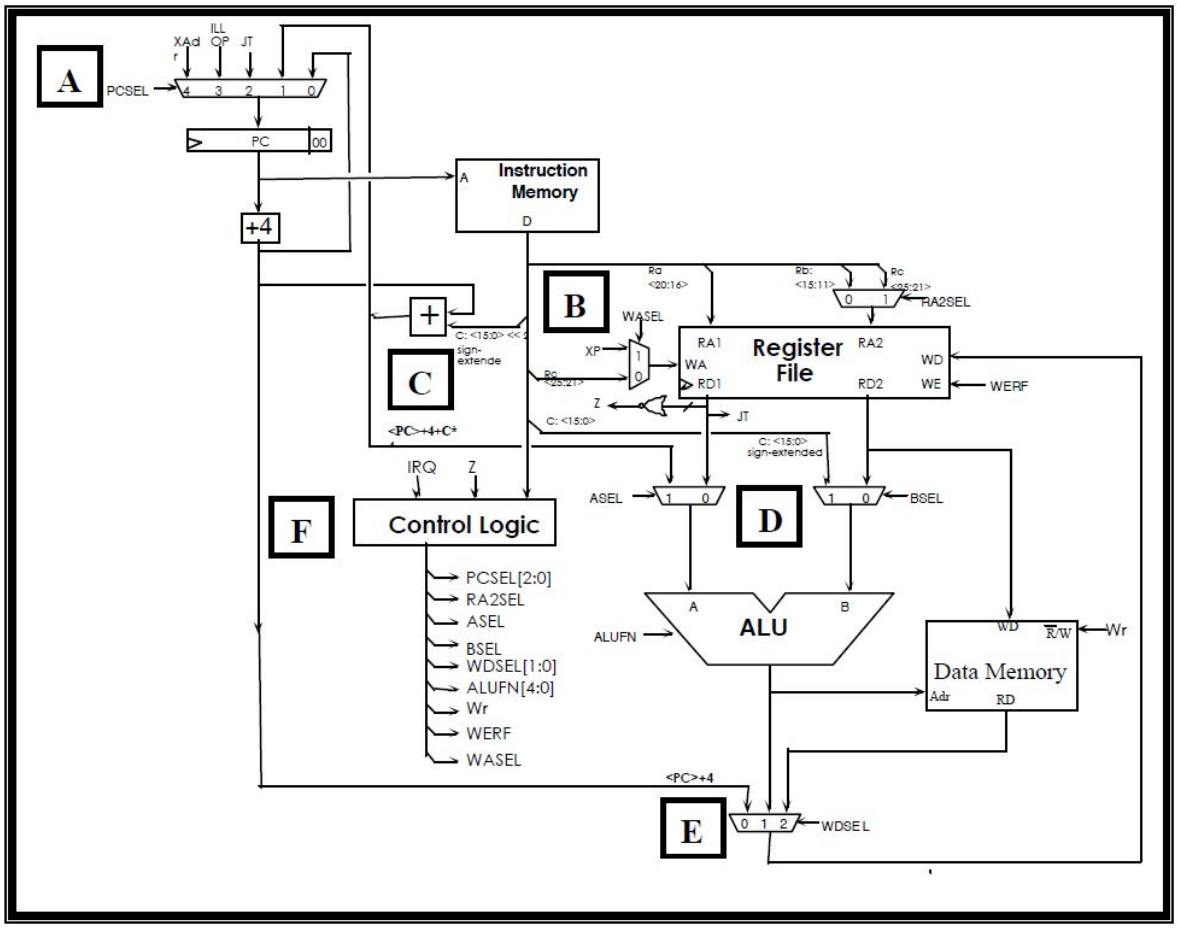

Figure 2: Expanded Beta Design