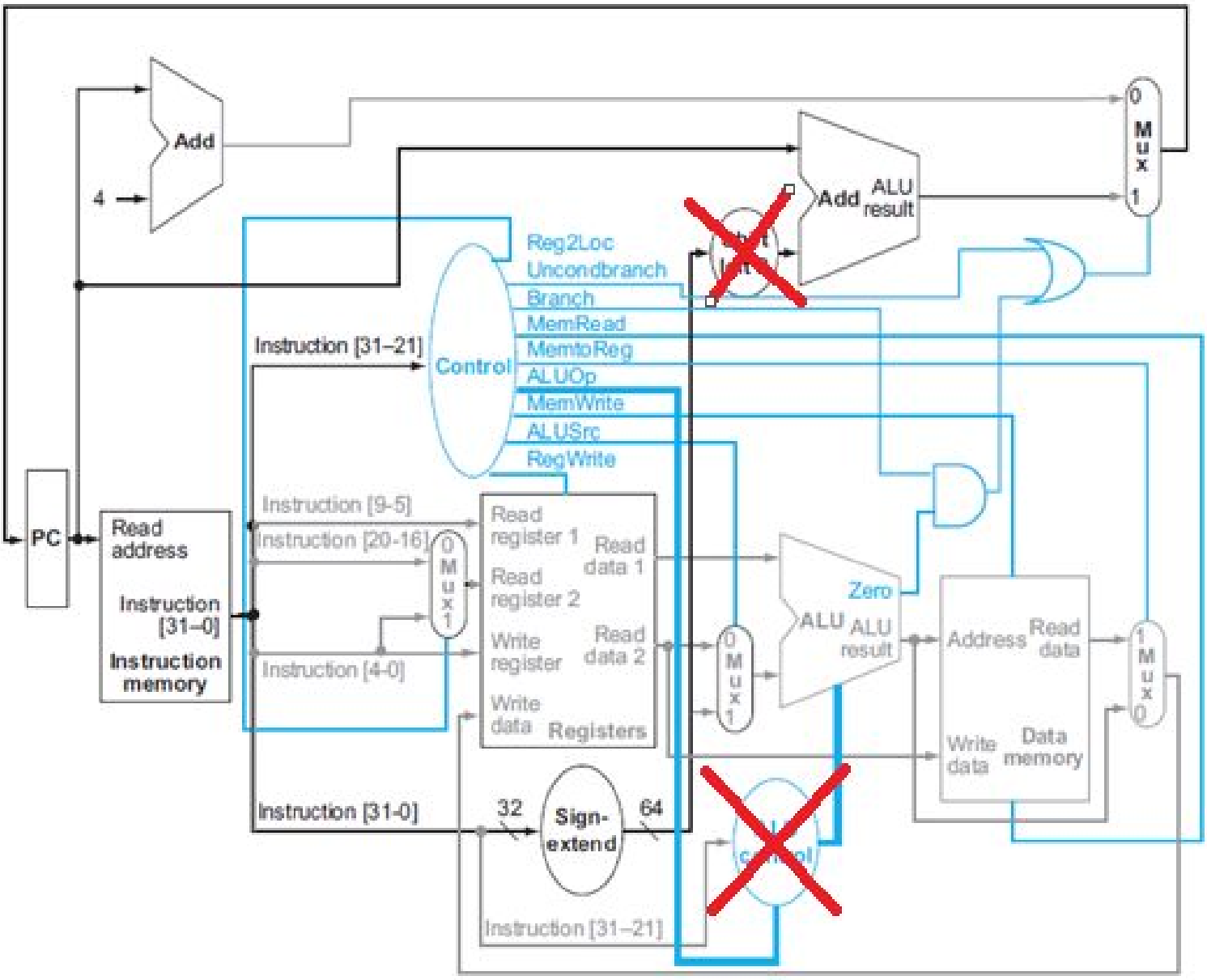
ECE 475 Processor Project

Justin Ross, Thomas Jackson, Dambaru Kafley, Thakur Bhattarai

Problem Specifications: The final project for our ECE 475 class was to program a 64-bit LEGv8 processor using a control unit and Data Path design, written in VHDL. The project was conducted on the Nexys 4 FPGA development board, and programmed using Vivado 2015. The programming was based off the figure given to us below.



The control unit for the system was implemented using a “one-hot” state-machine in which each state sent bits to various parts of the system to enable them, or to distinguish what pieces of data went where. Our project used a multiple cycle control unit, which broke up the various steps of fetch, decode, execute. The processor must be capable of executing the instruction set:

- add
- addi
- sub
- subi
- lsl
- lsr
- mul
- cbz
- cbnz
- and
- orr
- eor
- ldur
- stur
- b

Our diagram differentiated from the one given because we combined the control unit and the ALU control into one component. The goal of this project was to test the information and knowledge that we have gathered throughout the year.

Requirement Specification:

- 7 Segment Display: This component displays data that is written onto memory, specifically, to address 2048. The data is displayed in hexadecimal. The display must be multiplexed in order to show several different numbers at the same time. This consists of turning the display on/off, and enabling desired segments with the number you wish to be displayed. The frequency at which the display is alternated on and off exceeds the human eye’s ability to pick up on it, and appears to constantly be on.

- Program Counter: The PC is a register that contains the address of the location that is currently being executed .When the current instruction is executed, the PC increases its current value by 1, and points to the next instruction in the

sequence. Our PC is preloaded with a .coe file, and counts from 0-31. Once the processor gets restarted, or is reset, the PC reverts back to 0.

- Instruction Memory: This component contains the instruction data, which is 32 bits. The block of memory is 32x32 bits, which is sufficient for this assignment because we do not exceed 32 bits of instruction. In a more advanced setting, this would need to be increased to allow more instructions to be stored. Specifically, the instruction memory holds the machine code based off of the assembly that was given. In traditional RAM, the memory is made up of transistors. In our case, we generated memory using Vivado's block memory generator.

- First Adder: This adder (located in the top left) allows the PC to execute the next instruction stored in the instruction memory. Adder components can be made several ways, such as a full adder, a ripple-carry adder, and a carry-lookahead adder, which are, for the most part, made with a combination of AND and XOR gates.

- Sign Extender: The sign extender takes a 32-bit input, and outputs 64 bits. The process is completed by adding a series of 0's in front of the existing 32 bits. It is paramount that both the value and the sign of the number is preserved after sign extension. This component is necessary for several instructions like shifts, loads, branches, and I- Format instructions.

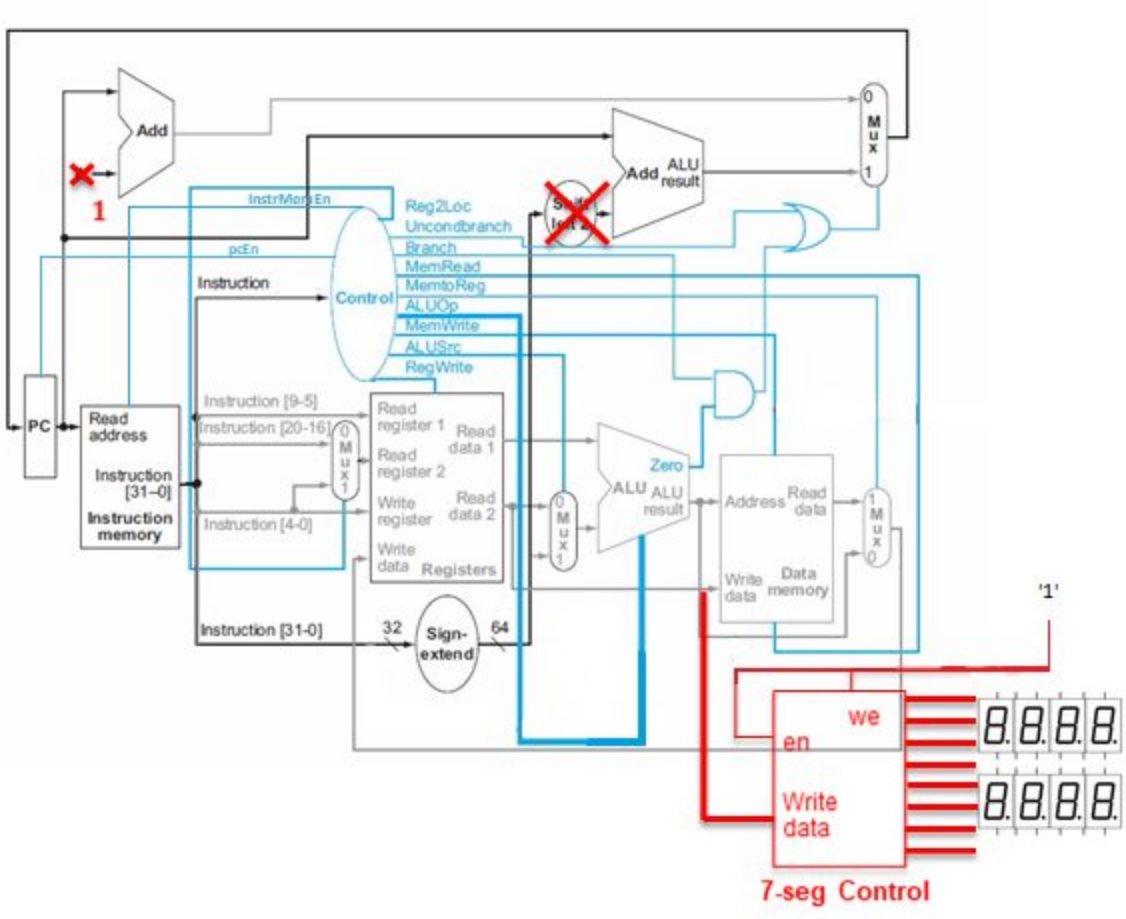
- Control Unit & ALU Control: An integral part of the processor, the control unit has the important job of controlling what actions are implemented within the system. This component tells the subunits of the computer how to respond given certain instructions. Additionally, it controls how data flows inside the processor, and provides connections to the rest of the computer to direct data and instructions.In our case, we combined the ALU control and control unit, so that the control unit directly outputs the 4-bit control signal. This effectively removes one component from our architecture, however, adds more control lines.

- Data Memory: Similar to the instruction memory in architecture, the data memory stores the results of the operations that we execute. The data memory is 64x2048 bits. This memory was also generated using Vivado's block memory generator.

- Multiplexor: A standard component in electronics design, the multiplexer selects data based on the 'select' bit that is passed into it. Our system contains both a

5-bit multiplexer, and two 64 bit multiplexer, with a single select bit. Implementing multiplexers is beneficial because it allows several signals to share one component, rather than having an individual component for each signal.

Design Specifications:



The above diagram is a better representation of what we ended up building. It's only slightly different from the original design, the most notable differences being the two additional control lines and the absence of the ALU Control module. Those decisions will be explained in further detail later.

The program counter keeps track of what instruction is going to be executed, the output of the PC is directly tied to the address line of the instruction memory. This will select the instruction that will be executed. The instruction memory outputs the 32 bit instruction, which goes to a lot of different places. The entire instruction is sent to the sign extender, which has some logic that allows it to select bits of interest and extends those to be 64 bits. For example, the 12 bits in the I-Format instruction that carry the immediate information will be grabbed and extended. The entire instruction also goes to

the control unit. The control unit has logic that decodes the instruction and selects the next state based on the current state and that input instruction. Both the PC and Instruction memory have enables in order to minimize errors.

The instruction also is broken up and sent into the registers. Instruction bits 9 through 5 are sent to the read register 1 input, this is because in all of the instructions that involve reading from the registers, a register number is always stored in bits 9 through 5. Instruction bits 20 through 16 are multiplexed with instruction bits 4 through 0. Depending on the instruction, one of the sets of bits will be selected to be the read register 2 input. Instruction bits 4 through 0 are also used as the write register input, this is because all of the instructions with a write back step place the destination register in the last 5 bits of the instruction. The register file has a write enable line that comes from the control unit.

The first data output from the register file goes directly to the first input of the ALU. The second data output runs to the write data input of the data memory and also the write data line of the seven segment display. The second data output from the register file is also multiplexed with the output from the sign extender. The select line comes from the control unit, and will select the correct input depending on the type of instruction. The ALU determines what operation to do with the 4 bit signal that comes from the control unit. The ALU has a zero flag that will raise whenever the result of the ALU is 0. The result of the ALU is tied to the address input of the data memory, it also is multiplexed with the result of the data memory. The select line for that multiplexor comes from the control unit. The output of the multiplexor is the write data input for the register file.

The output of the program counter also runs to an adder that will increment the current count by one. The output of that adder will go to a multiplexer in which the other input is the output of another adder that takes the output of the sign extender as the other input. This is used to determine the address if there’s a branch. The select line of the multiplexor is the the zero flag from the alu anded with the branch line from the control unit. The output of that and gate is ored with the unconditional branch line from the alu. The output from that or gate is the select line of the multiplexor. The output from the multiplexor is the input line to the program counter.

Implementing and Testing:

```
int main () {

   int a = 10, b = 6;
   int i,f,g;
   int vals[40];

   f = a * b;
   g = a + b;


   /* a in $s0 , b in $s1, f in $s2, address of vals[0] in $s3
     i in $t0 , g in $t1 */
   for( i = 0; i < 40; i = i + 1 ){
    vals[i] = f + i;
   }

   if (vals[g] > (g * 4)){
    f = a - b;
     printf("value of f is : %d\n", f); // Display on 7-segment displays
    }
   else{
    f = a + b;
     printf("value of f is : %d\n", f); // Display on 7-segment displays
    }

   return 0;
}
```

This project was built to run above code that was assigned to us, this code was written in C, so part of the assignment was to translate that C code to Assembly, and that Assembly to machine code. The Assembly code was written by hand, and the following code was found.

```
ADDI X19,X31,#10        //a=10
ADDI X20,X31,#6         //b=6
MUL X21,X19,X20         //f=a*b
ADD X10,X19,X20         //g=a+b
ADDI X11,X31,#40        //X11 = 40
ADDI X9,X31,#0          //i=0
//for loop
loop:ADD X12,X21,X9     //X12 = f+i
ADD X13,X22,X9          //X13=&vals[i]
STUR X12,[X13,#0]       //val[i] = f+i
ADDI X9,X9,#1           //i=i+1
SUB X14,X11,X9          //40-i
CBNZ loop               //Branch if not 0

//If Statement
ADD X13,X22,X10         //X13=&vals[g]
LSL X12,X10,#2          //X12 = g*4
LDUR X14,[X13, #0]      //X14 = vals[g]
SUB X15,X14,X13         //X15 = vals[g]-g**4
CBNZ X15,check          //check if 0
less:ADD X21,X19,X20    //f = a+b
ADDI X11,X31,#2047      //X11 = 2048
STUR X21,[X11 #1]       //Print f to Seven Segment
B return                //branch to return

check:LSR X15,X15,#64   //get top bit
CBNZ X15,less           //if it's not 0, then we know it's negative
SUB X21,X19,X20         //f = a-b
ADDI X11,X31,#2047      //X11 = 2048
STUR X21,[X11,#1]       //print f on Seven Segment
return:LDUR X23,[X31,#0]
```

This code was slightly modified, removing unnecessary spaces and all of the comments, and fed into a program that parses the strings and converts them into the proper binary codes. Those binary codes are then split up into groups of 32 and put into a .coe file with a radix of 2. When that file is written, it can be stored into the Instruction memory and the processor can be tested. The assembler program was tested with all of the different types of commands individually and they all matched what the codes should've been.

To test the designed CPU, first,each of the individual components were tested individually. Most of the items inside the data path had testbenches written to verify their functionality. So, it was assumed that the data path, if wired correctly, would work out of the box. Which could simply be the bounce-back of the button.

The next component that needed testing was the Control Unit, and that was tested on the board. Each of the different instructions were sent in and the state changes were noted. Each state was able to be reached, but there was some seemingly random state jumping.

Once the data path and control unit were connected together, the system was tested on the board, first with a few individual instructions, ending with a store command that should display on the seven segment display. This is the stage where debugging was crucial. When something was not working, the slideshow that depicted the fetch, decode, execute, memory access, and write-back steps was consulted. The connections and control unit outputs were put under a microscope and errors were corrected. The datapath and control unit were put on a 180 degrees out of phase clock as well to help minimize timing issues.

Discussion

Issues with the Design

Although we got all of the components built and connected together, the CPU did not end up working. As stated earlier, the Control Unit does seemingly skip states from time to time, but that could be due to the bounceback of the button. The bigger issue we believe is the connections in the data path. This is thought to be the issue because whenever we tried to store something to the seven segment display, it just displayed 0. This could be due to how the register file selects data 2, the multiplexor could be selecting the wrong thing, in which case the state machine is not providing the correct outputs. But without the stur command functioning, there was no good way to determine if the system was functioning correctly.

Soft Skills

Basically, we learn to build different components and connect all of them together to work as a processor. Ultimately, we got a chance to learn how the processor works. It was a long, difficult project that tested students knowledge which we learnt during a

semester. We needed to think logical, and think like a processor to succeed. We believe we have an issue on the control unit that's why our outcome doesn't works as expected. While the project was not on the level of a large design in the engineering field, it still became a great way to teach and share our ideas with team members. The biggest things beside this project is working with the team and communicating with each other. It was a great experience and opportunity to work in team, and sharing ideas between the team, to practice our engineering ethics. Computer architecture is the only course that we all take together. Since, we have different class schedule, it is hard to manage time to work together. We split the work and combine the components but it did not work as expected. Furthermore, we started working together during weekends to improve the design. Finally, our project work up to some point but not as course required.