Lab 4 Report

Sabrina Herman (sh997), Brian Szczesniak (bs593), Raghav Kumar (rk524)
Monday 1:25 - Group 4

Introduction (Sabrina)

The purpose of this lab was to design a processor from scratch. The first part of the lab was completing the ALU. The main components of an arithmetic logic unit are three submodules: an adder, a shifter, and a logical unit. The OP code to the ALU determines which of these components is being utilized and what operations are being performed. The ALU can add, subtract, and, or, shift left logical, shift right logical, and shift right arithmetic. We were able to successfully create and test our ALU.

The second part of the lab was to create the processor. A single-cycle processor performs operations based on the instructions it receives in a single clock cycle. The processor has many components including an instruction RAM, instruction decoder, register file, ALU, data RAM, a PC, and other logic components for branching and halting. The processor starts with a 16-bit instruction from the instruction RAM. This instruction could be in two different formats: register-to-register or immediate format. The instruction decoder will convert that instruction into control words that go to other components of the processor. Those control words include the destination and source registers, immediate value, ALU function, and other values that select the operation such as write, load, and memory selects. Following the decoder is the register file, which selects values from the source register(s) to operate on in the ALU. The ALU can operate on two values from the register file, or on one value and an immediate value. The value that the ALU outputs can be written into DRAM, a value from DRAM can be retrieved, or the DRAM can be ignored. After that stage, the chosen value, if the instruction was a load instruction, will be loaded into the register file at the destination register address. This processor can be expanded to include branch instructions, which will jump to different instructions based on selected ALU outputs such as Z (1 if output equal to zero), N (1 if output negative), C (carry out value or shifted out value depending on operation), and V (1 if overflow). Implementing branch instructions is what we were successfully able to do for the third part of the lab.

Design (Brian)

We designed a single cycle microprocessor with branch instructions. This microprocessor looked at an instruction ram and executed the program contained within it. These instructions included many different operations. Programs were able to load and store values from/to memory as well as manipulate these values while they were stored in registers within the processor. The inputs to this program would be the instruction ram and the data ram. The outputs of the processor are the values present in the data ram, since these can be altered by the processor.

We designed our microprocessor in three steps.

ALU:

First we designed an arithmetic logic unit. This unit could perform all the basic logical operations we wanted for out processor: “add”, “subtract”, “shift left logical”, “shift right logical”, “shift right arithmetic”, “and”, and “or”. The ALU takes two eight bit values and a three-bit operation code as inputs. It outputs a eight-bit output value as well as four informational values:

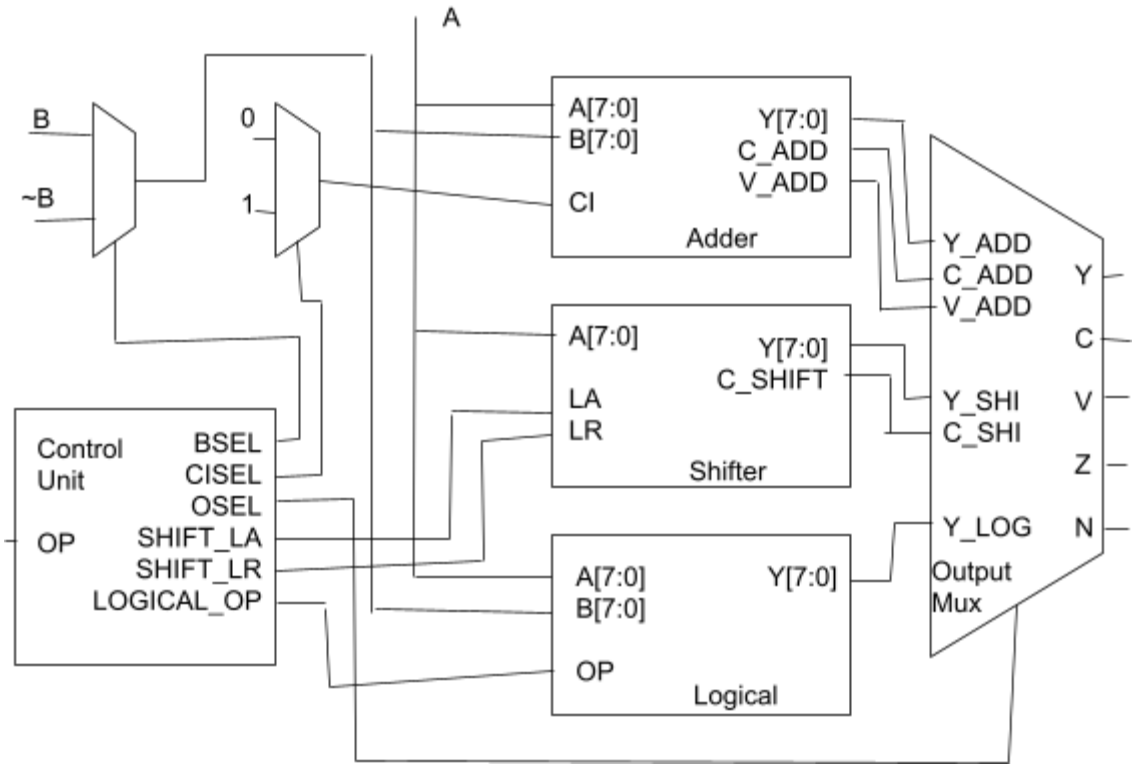
Z: One if output is zero, zero otherwise

N: One if the output is negative, zero if non-negative

C: If an add/sub operation is performed this is the carry out value, if a shift operation is performed this is the shifted out value, otherwise it is a zero

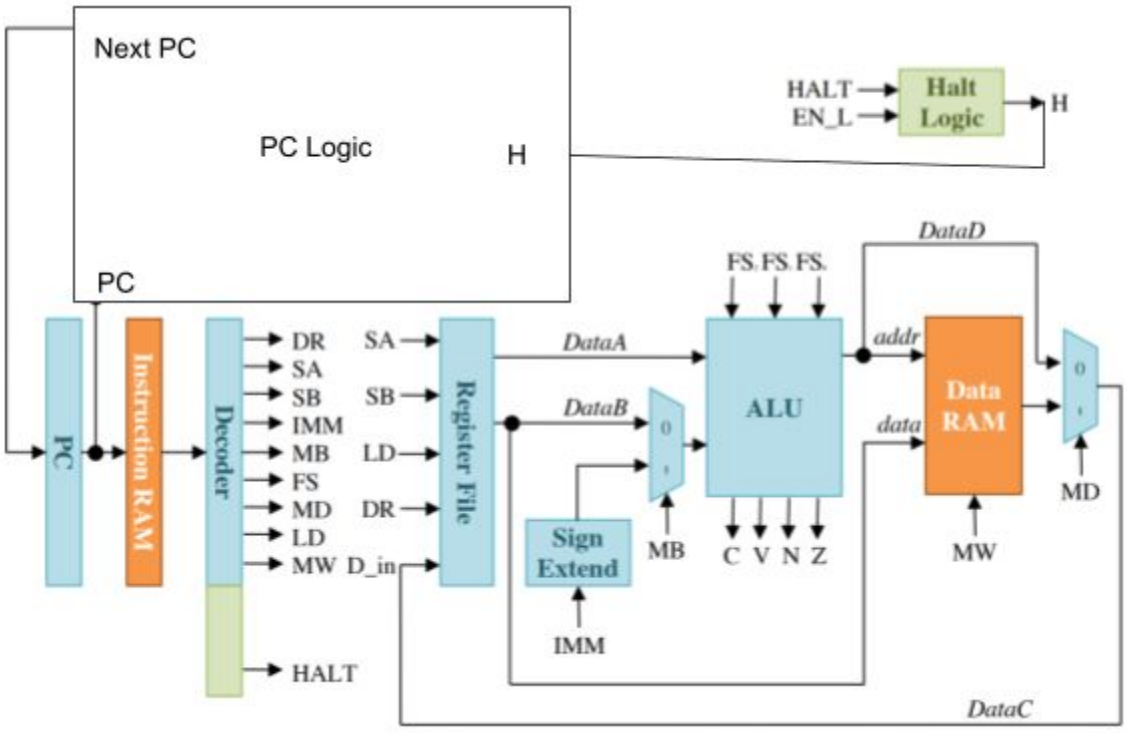
V: One if an add/sub operation had overflow, zero otherwise

The body of the ALU is three operational units, an adder, a shifter, and a logical unit. The inputs to these units are controlled based on the op code. Each unit produces outputs and a output mux selects which of the units’ outputs should be used. A diagram for the ALU can be seen below. The output mux determines N and Z internally based on the chosen value of Y.



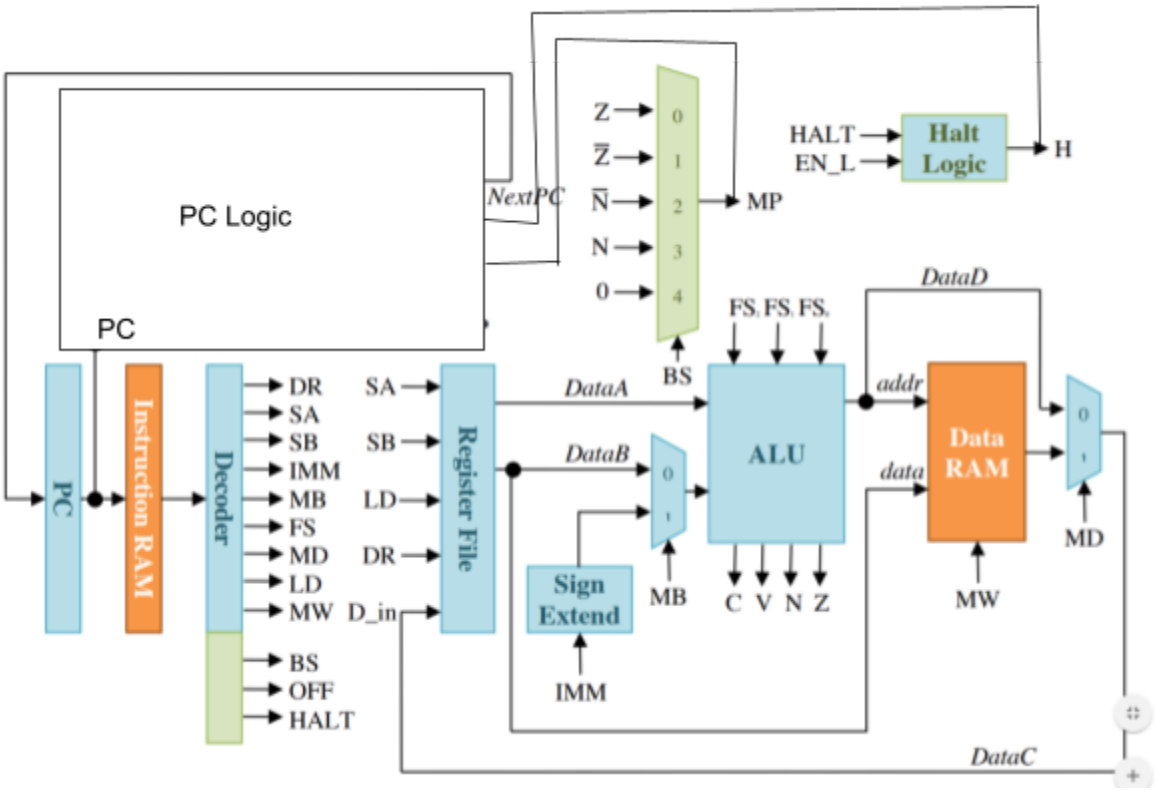
Branchless Microprocessor:

Second we used this ALU as a component in microprocessor without branch instructions. This unit read instructions from an Iram, decoded them, and carried out the instructions. This processor used a program counter to keep track of which IRAM instruction to consider. This PC acted as an IRAM address reading instructions out one at a time. These instructions could be R to R instructions or a number of IMM type instructions; specifically “ANDI”, “ORI”, “ADDI”, “LOAD”, STORE”, “NOP” and “HALT”. Below is a diagram of our data path for this microprocessor, note that it differs from the lab diagram by combining all of the PC logic into a single place. This combination will be more significant when adding branches as it helps simplify the design.



Branched Microprocessor:

The final step of our process was to add branch instructions to the microprocessor to allow more complex programs. This addition required two main changes to our processor. First, the decoder had to be expanded to interpret branch instructions. This meant adding two outputs, one for the type of branch criteria being used and one for the size of the offset. Second, the PC logic had to be expanded to decide whether or not to branch. A diagram of the final data path can be seen below.



Components in final microprocessor:

PC: This is a register holding the current PC value, this keep track of what line of the program is being executed.

Instruction RAM: A ram file containing appropriately formatted instructions to be carried out. This is one of the two inputs for the program.

Decoder: This breaks down the instructions from the instruction ram and distributes to information to various other components.

Register File: This contains internal registers used in instructions, new values can be stored in these registers using most instruction types.

Sign Extend: This sign extends the the IMM value as a two's complement number, this allows the 6 bit IMM to be used with the 8 bit data values from the register file

ALU: Described above.

Data RAM: This is a memory for storing information. This is the second input to the system as well as the only output.

PC Logic: This decides how to advance the PC, normally this is just a plus two, but can change depending on branch instructions and halt instructions.

BS Mux: This interprets the branch condition and produces a result that says whether or not to branch

Halt Logic: This takes hare of halting and unhalting based inputs from the user.

Implementation and Testing (Brian and Raghav)

Most of the systems described above are relatively simple; however, a couple deserve further explanation.

Adder: The adder unit inside the ALU is somewhat complex. Instead of simple gates as in the shift and logical units, the ALU is made of of 8 one bit adders wired up as a ripple carry out adder.

Decoder: The decoder is very complex and involves assigning values the different outputs based incoming instructions. This is done differently for different outputs. Some of the outputs involve passing signals through the decoder from parts of the input bus to the output pins. For example the last six bits of the instruction are always passed to the IMM output. Other signals, such as branch select, are determined using more complicated logic to turn the four bit op code into a branch select code.

For the decoder we were only provided with a basic test case and thus had to make a more comprehensive test suit that tested out all the instructions and the decoder's functionality. We first started with the normal register to register instructions with the opcode 1111. We used random registers for the Source A, B and Destinations registers. Then we moved on to the immediate instructions and tested those out. To ensure maximum code coverage, we made sure that we had at least one test for each instruction so that all conditionals in our decoder module got accessed at least once by the test bench.

We had a similar strategy for the ALU but for the ALU ADD and SUB instructions we had to think of several combinations in order to comprehensively test our code. We checked for the conditions that varied the C, V, N, and Z outputs . These combinations helped increase our confidence in our design and increased code coverage. Below is a table for the test cases for the decoder and the ALU--

Tests for Decoder	Tests for ALU
Adding R2 and R6 and storing in R0	SRL - 2
SRA on R0 and storing in R2	AND - 2
SRL on R0 and storing in R2	OR - 2
SLL on R1 and storing in R3	SLL - 2
ORing R2 and R0 and storing in R0	SRA - 2
Subtracting R7 and R6 and storing in R0	ADD (Large and Small Positive, Zeros, Negative, Negative and Positive + some other combinations like overflow)
ANDing R1 and R2 and storing in R2	SUB (2 positives, 2 negatives with underflow, Positive and Negative, 2 negatives normal and some other combinations)
Immediate ADD of R3 and 5; Storing in R2	

Immediate AND of R0 and 1; Storing in R1	
Immediate OR of R1 and 2; Storing in R0	
Getting data from memory destination stored in R5 (IMM=0) and storing in R2	
Storing value of R3 in memory location given by R0 + 1	

The test bench for part C runs our microprocessor in parallel with a working microprocessor provided for us. It compares certain results such as PC values and a couple of the values in DRAM to see if our code was generating the same results. Although the values stored in the registers was hidden from the testbench, if all the other factors agree throughout, it is likely that the register values were changing as expected. Further investigation of register values could also be checked by looking at waveforms. Finally, the test benches did not check halt instructions, these needed to be checked by looking at the waveform.

Modifications (Sabrina)

One change to reduce average number of instructions to write a program:

To reduce the average number of instructions, our design would need to change the instruction format such that it could incorporate multiple instructions into a single instruction. For example, if you wanted to add two values together that are stored in memory and store the sum elsewhere in memory, that requires two load instructions that get the values being added out of memory, followed by the add instruction which would produce their sum, and lastly a store instruction that would store the sum in memory. The new instruction being described would look like “ADD 2(R0), 0(R0), 1(R0)” rather than “LB R1, 0(R0) / LB R2, 1(R0) / ADD R3, R1, R2 / SB R3, 2(R0)”. Note however, that while this would decrease the average number of instructions needed, it would not improve the overall functionality of the processor otherwise. The processor would still require four clock cycles to execute the example instruction, operating as if it had received the four separate instructions.

One change to design to increase the processor’s maximum clock frequency:

To increase the maximum clock frequency, we could design a pipelined processor instead of a single cycle processor. In a pipeline processor, instructions are executed in stages, such that the processor can be processing multiple instructions at once. The time it takes an instruction to go through one stage is less time than it takes the instruction to go through the overall processor. The clock cycle is now the amount of time to go through one stage. Though it now takes five clock cycles for an instruction to go through the pipelined microprocessor, the maximum clock frequency has increased.

Conclusion (Raghav)

Overall, this lab incorporated some important topics from our lectures including Processors and the underlying hardware behind basic computers and microprocessors. The lab also helped us accentuate our understanding of the fundamental components of hardware design such as the data and instruction RAM, the register file, the ALU etc. The incremental approach in design; starting with the ALU and finishing with a branch equipped processor was interesting because it helped appreciate the complexity behind such designs in a methodical manner. This was the only lab that required us to do our own testing. It taught us the techniques to make our code more robust and reliable in addition to giving us an exposure of building comprehensive testbenches. Seeing two different programs run on our architecture was not only fun but also motivated us to write new programs which we went on to do in the 5th lab. Finally, our report talks about the modifications that we could make to our processor to make it faster or have a higher throughput. We could build off our current design and try making a pipelined processor which has a higher throughout with a lower cycle time. We could also try reducing the total number of instructions if we changed our ISA to one which combines ALU operations with the Load and Stores.

Overall, our design worked and executed properly in the lab sessions. The lab was very helpful for our general understanding of computer hardware and implementing the processor in Verilog added a new dimension to our understanding of the class material.

Work Distribution

See subheadings for a general idea of who contributed to which sections of the report.