# CprE 381: Computer Organization and Assembly-Level Programming

# Project Part 1 Report

Team Members: Joseph Barnes III

Josh Arceo

Evan Shiber

## Project Teams Group #: Sec C 03

***Refer to the highlighted language in the project 1 instruction for the context of the following questions****.*

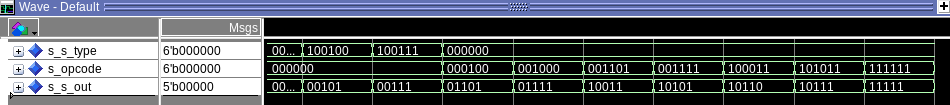
[Part 1 (d)] Include your final MIPS processor schematic in your lab report.

A diagram of a computer

Description automatically generated

[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your datapath implementation. The end result should be an *N*\**M* table where each row corresponds to the output of the control logic module for a given instruction.

[Part 2 (a.ii)] Implement the control logic module using whatever method and coding style you prefer. Create a testbench to test this module individually, and show that your output matches the expected control signals from problem 1(a).



[Part 2 (b.i)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.

The fetch logic needs to be able to manage three different control types:

* branch
* jump
* regular increment (PC+4 for most instructions).

It is necessary that it supports both the branch and jump commands.

* The regular increment is utilized for all other command sets.

Additionally, a *halt* instruction (which assists in halting program execution process when required) should have been included in the fetch logic.

[Part 2 (b.ii)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?

A diagram of a circuit

Description automatically generated

[Part 2 (b.iii)] Implement your new instruction fetch logic using VHDL. Use Modelsim to test your design thoroughly to make sure it is working as expected. Describe how the execution of the control flow possibilities corresponds to the Modelsim waveforms in your writeup.

[Part 2 (c.i.1)] Describe the difference between logical (srl) and arithmetic (sra) shifts. Why does MIPS not have a sla instruction?

Logical shift (srl):

* Shifts bits right and fills leftmost bits with zeros, regardless of original sign of the number
* Usually for unsigned integers

Arithmetic Shift (sra):

* Shifts bits right and preserves sign bit (MSB) by filling leftmost bits with the sign bit (0 for positive, 1 for negative)
* Usually for signed integers to maintain correct value

Absence of a Shift Left Arithmetic (sla) instruction:

* Shifting left by one bit (using sll) inherently preserves the sign of the number, since it doesn’t depend on the sign bit.
  + All bits shift left and zeros fill in from the right, which effectively multiplies the number by 2.
* Thus, a separate sla instruction is unnecessary
  + sll suffices for both signed and unsigned integers when shifting left.

[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

**Arithmetic Shifting**

Sign Preservation

* Arithmetic shifting preserves the sign bit during right shifts.
  + This ensures that the result retains the correct value for negative numbers.

Control Signal

* The s\_shift\_type signal is set to '1' for arithmetic shifts, allowing the shifter to recognize when to maintain the sign bit.

Implementation

* In the case of a right arithmetic shift (ex: sra), when the input is 11110000 (representing -16 in two's complement), the shifter will shift right and fill the leftmost bit with 1, resulting in 11111000 (representing -8).

**Logical Shifting**

Zero Filling

* Logical shifting fills the vacated bit positions with zeros.
  + Usually for unsigned binary values.

Control Signal

* The s\_shift\_type signal is set to '0' for logical shifts, instructing the shifter to use zero for any new bits introduced.

Implementation

* For a logical right shift (ex: srl), shifting 11110000 right will result in 01111000, where the leftmost bit is filled with 0, effectively changing the sign of the number if treated as signed.

[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

**Method 1: Input and Output Modification**

Bidirectional Inputs:

* Configure multiplexers to accept input for both left and right shifts, allowing for flexible data routing.

Flexible Outputs:

* Ensure outputs can represent results from either operation by connecting them accordingly.

**Method 2: Multiplexer Configuration**

Cascading Design: Maintain cascade of 2:1 multiplexers:

* For left shifting:
  + Each multiplexer routes bits to higher positions
    - Ex: MUX[0] receives D1 for Q0, D2 for Q1, etc.
* For right shifting:
  + Configure inputs so that MUX[0] gets D0 for Q0, D1 for Q1, filling with zeros as needed for higher bits.

**Method 3: Control Signal Integration**

Control Logic:

* Introduce a single control signal to determine shift direction
  + Ex: SHL for left, SHR for right
* This signal will direct multiplexers to select appropriate inputs based on desired operation

[Part 2 (c.i.4)] Describe how the execution of the different shifting operations corresponds to the Modelsim waveforms in your writeup.

[Part 2 (c.ii.1)] In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.

In the ALUcontrol, output different signals for every instruction and make a different calculation for every instruction in the ALU based on the inputted control signal.

[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.

This depends on the instruction. There will be different instruction control signals and machine code, and the outputs in the waveform.

[Part 2 (c.iii)] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: how is Overflow calculated? How is Zero calculated? How is slt implemented?

A diagram of a computer system

Description automatically generated

[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.

Depending on the instructions, the ALU will output different fetch signal and overflow.

[Part 2 (c.viii)] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.

[Part 3] In your writeup, show the Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

[Part 3 (a)] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially, 1 per cycle while data written into registers can be effectively retrieved and used by later instructions). Name this file Proj1\_base\_test.s.

[Part 3 (b)] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1\_cf\_test.s.

[Part 3 (c)] Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm ([link](http://en.wikipedia.org/wiki/Bubble_sort)). Name this file Proj1\_bubblesort.s.

[Part 4] report the maximum frequency your processor can run at and determine what your critical path is. Draw this critical path on top of your top-level schematics. What components would you focus on to improve the frequency?