CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: Grant Pierce_____

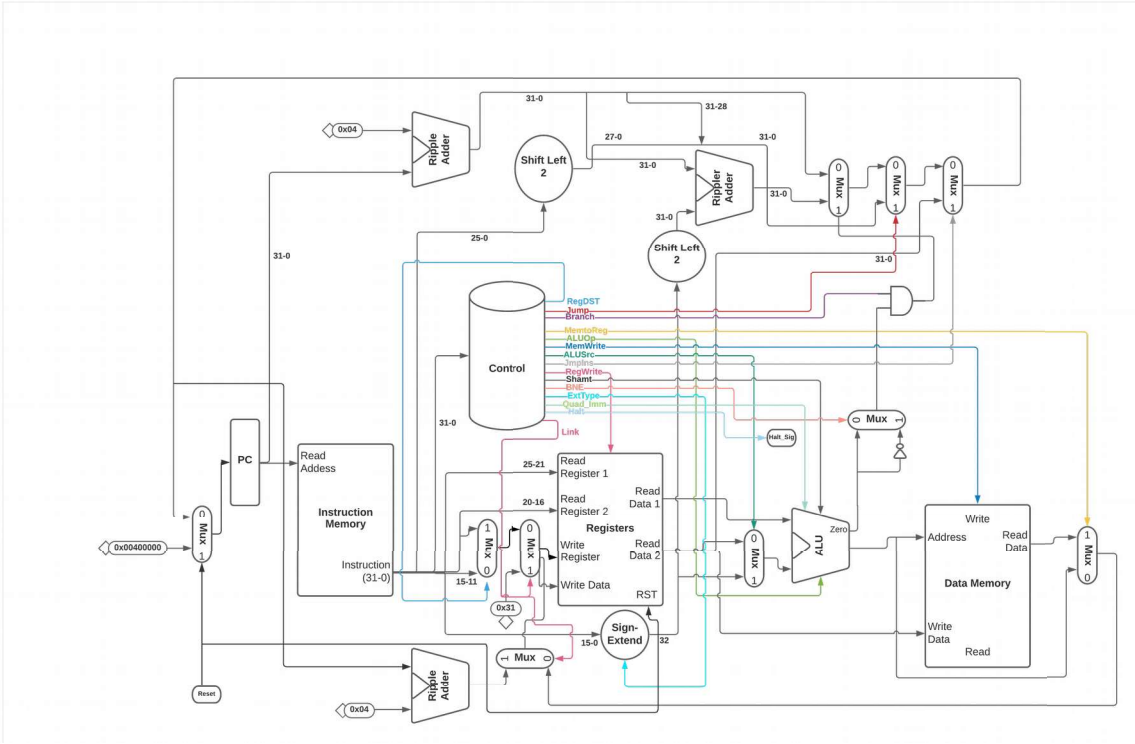
_____

_____

Project Teams Group #:_____

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.



[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.

Spreadsheet already present in the documents with names control sign.xlsx and control sign xlsm

[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).

find the code of the implementation in MIMPS directory

[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.

Branch instructions: Branch instructions are used to alter the program counter (PC) value and transfer control to a different part of the program. The instruction fetch logic must be able to handle the following control flow possibilities for branch instructions:
Unconditional branch: Transfer control to a specific target address unconditionally.
Conditional branch: Transfer control to a target address based on a specific condition (e.g., the value of a register).
Direct branch: Transfer control to a target address specified in the instruction itself.
Indirect branch: Transfer control to a target address stored in a register.
Jump instructions: Jump instructions are used to transfer control to a different part of the program without modifying the PC value. The instruction fetch logic must be able to handle the following control flow possibility for jump instructions:
Jump: Transfer control to a specific target address without modifying the PC value.
Call and return instructions: Call and return instructions are used to implement subroutines and functions in the program. The instruction fetch logic must be able to handle the following control flow possibilities for call and return instructions:
Call: Transfer control to a subroutine, saving the return address on the stack.
Return: Transfer control back to the calling subroutine, restoring the return address from the stack.

[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?

Find the drawing in the file called mips.pdf

[Part 2 (b.iii)] Implement your new instruction fetch logic using VHDL. Use QuestaSim 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 QuestaSim waveforms in your writeup.
To implement the new instruction fetch logic using VHDL, the code written handles the different control flow-related instructions, such as branch, jump, and return instructions. The implementation should include the necessary control signals to allow the processor to execute these instructions correctly.

To test the new instruction fetch logic thoroughly, various input instructions can were applied to the processor and the resulting control flow can be observed using QuestaSim. The correctness of the results can be verified by comparing them to the expected control flow based on the input instructions and the control flow-related instructions being tested.

The QuestaSim waveforms can show the execution of the different control flow possibilities by displaying the values of the control signals and other relevant signals in the circuit over time. For example, when testing a branch instruction, the waveform should show the instruction being fetched, the decision to take the branch, and the new instruction address being loaded. Similarly, when testing a jump instruction, the waveform should show the instruction being fetched, the new instruction address being loaded, and the jump to the new instruction address.

By carefully designing and testing the instruction fetch logic using VHDL and QuestaSim, the control flow possibilities can be executed correctly and the resulting waveforms can show the expected behavior of the processor during these operation

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

In a logical shift (srl), the bits are shifted to the right by a specified number of positions, and the vacated bits are filled with zeros. In other words, the bits are shifted to the right, and the sign bit is not propagated. This operation is often used when dividing by a power of two.

In an arithmetic shift (sra), the bits are shifted to the right by a specified number of positions, and the vacated bits are filled with the sign bit. In other words, the bits are shifted to the right, and the sign bit is propagated. This operation is often used when performing signed arithmetic operations, such as dividing a signed number by a power of two.

MIPS (Microprocessor without Interlocked Pipeline Stages) is a popular Reduced Instruction Set Computer (RISC) architecture that is widely used in embedded systems, scientific computing, and other applications. MIPS supports logical shifts (srl and sll) and arithmetic shifts (sra), but it does not have a shift left arithmetic (sla) instruction.

The reason why MIPS does not have an sla instruction is that it can be easily implemented using the sll instruction. Since shifting a number to the left by n positions is equivalent to multiplying the number by 2^n, the sla instruction can be emulated by sll instruction with the desired shift amount, followed by any necessary sign extension operation. This simplifies the instruction set and reduces the hardware complexity of the processor.

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

In VHDL, shifting operations were implemented using the shift_left and shift_right functions, which are built-in to the language. The shift_left function performs a logical shift to the left, filling the vacated bits with zeros, while the shift_right function can perform either a logical shift to the right (filling vacated bits with zeros) or an arithmetic shift to the right (filling vacated bits with the sign bit).

To perform an arithmetic shift to the right, the shift_right function can be called with a third argument, which specifies the shift type. If the shift type is "arithmetic", then the shift_right function will perform an arithmetic shift to the right, filling the vacated bits with the sign bit. If the shift type is "logical", then the shift_right function will perform a logical shift to the right, filling the vacated bits with zeros.

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

right barrel shifter can be enhanced to also support left shifting operations by adding a multiplexer at the input of the shifter. This multiplexer selects either the original input value or the inverted value of the input, depending on the direction of the shift.

To support left shifting, the shifter's control input would need to specify the direction of the shift (left or right). When a left shift is required, the multiplexer would select the inverted value of the input as the input to the shifter, causing the bits to be shifted to the left instead of the right

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

When an arithmetic shift operation is performed, the sign bit of the input value is preserved and shifted into the vacated bits. This means that if the input value is negative, the vacated bits will be filled with ones to maintain the negative value. Conversely, if the

input value is positive, the vacated bits will be filled with zeros to maintain the positive value.

In the QuestaSim waveforms, the execution of an arithmetic shift operation is as follows:

The input value is loaded into the shifter.
The shift control signal is activated, indicating that an arithmetic shift should be performed.
The shifter performs the arithmetic shift, shifting the bits to the left or right as specified by the shift amount and filling the vacated bits with the sign bit.
The output value of the shifter is presented on the output bus.
When a logical shift operation is performed, the vacated bits are always filled with zeros, regardless of the sign of the input value.

In the QuestaSim waveforms, the execution of a logical shift operation would be seen as follows:

The input value is loaded into the shifter.
The shift control signal is activated, indicating that a logical shift should be performed.
The shifter performs the logical shift, shifting the bits to the left or right as specified by the shift amount and filling the vacated bits with zeros.
The output value of the shifter is presented on the output bus.

[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.

My design approach for the implementation of the MIPS processor in VHDL was to first identify the functional units required to execute the MIPS instructions and their interconnections, as well as the necessary control signals for each unit. I used the MIPS instruction set architecture specification as a reference to identify the required functional units and control signals.

Once I had identified the required functional units and control signals, I proceeded to implement each unit in VHDL and simulate their behavior using a testbench. I used the QuestaSim simulation tool to simulate the VHDL design and verify its correctness.

One design decision I had to make was how to implement the instruction fetch unit. I decided to use a separate instruction memory component to store the program instructions and a program counter register to keep track of the address of the next instruction to be fetched. I also implemented a control unit that generates the necessary control signals for the other functional units based on the opcode of the fetched instruction.

To implement the instruction fetch unit, I had to decide on the size and format of the instruction memory and how to interface it with the other functional units. I chose to use a synchronous read-only memory (ROM) component with a 32-bit address bus and a 32-

bit data bus to store the program instructions. I also decided to use a multiplexer to select the appropriate instruction based on the value of the program counter register.

Overall, my design approach was to implement each functional unit separately and then integrate them together to form the complete MIPS processor. I used the MIPS instruction set architecture specification and the QuestaSim simulation tool to guide my design decisions and verify the correctness of the implementation.

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

Instruction Fetch: The instruction fetch stage retrieves the instruction from memory and increments the program counter to point to the next instruction. In the waveforms, this stage is usually represented by a rising edge on the clock signal and a sequence of signals that indicate the address of the instruction being fetched and the opcode of the instruction.

Instruction Decode: The instruction decode stage decodes the opcode of the instruction and generates the necessary control signals for the other functional units to execute the instruction. In the waveforms, this stage is usually represented by a sequence of signals that indicate the opcode of the instruction and the control signals generated by the control unit.

Register Fetch: The register fetch stage retrieves the operands specified by the instruction from the register file. In the waveforms, this stage is usually represented by a sequence of signals that indicate the register numbers of the operands and the data values read from the register file.

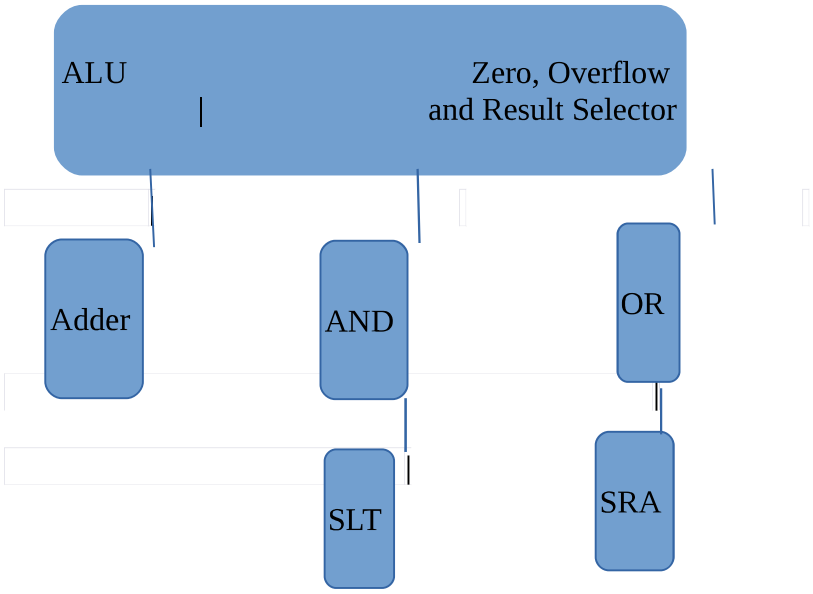
ALU Operation: The ALU operation stage performs the operation specified by the instruction using the operands retrieved from the register file. In the waveforms, this stage is usually represented by a sequence of signals that indicate the operation being performed, the values of the operands, and the result of the operation.

Memory Access: The memory access stage retrieves or stores data in memory, depending on the instruction being executed. In the waveforms, this stage is usually represented by a sequence of signals that indicate the address of the memory location being accessed, the data being read or written, and the type of memory access being performed.

Write Back: The write back stage writes the result of the operation to the destination register specified by the instruction. In the waveforms, this stage is usually represented by a sequence of signals that indicate the register number of the destination operand and the data being written to the register file.

Overall, the QuestaSim waveforms should show the execution of each instruction in the processor, including the instruction fetch, instruction decode, register fetch, ALU operation, memory access, and write back stages

[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?



The ALU consists of three main parts: an adder, a bitwise AND operator, and a bitwise OR operator. The adder is used to perform addition and subtraction operations, the bitwise AND operator is used to perform logical AND operations, and the bitwise OR operator is used to perform logical OR operations.

The ALU also includes a Zero/Overflow/Result Selector block, which is responsible for calculating the zero and overflow flags, as well as selecting the result of the ALU operation. The zero flag is set to 1 if the result of the operation is zero, and 0 otherwise. The overflow flag is set to 1 if the result of a signed operation exceeds the range of a signed 32-bit integer, and 0 otherwise.

The slt (set less than) operation is implemented using a comparator that compares the two input values and sets the output to 1 if the first input is less than the second input, and 0 otherwise. In addition, the slt operation also uses the overflow flag to handle signed comparisons.

Overall, this simplified high-level schematic shows the main components of the 32-bit ALU, including how the Zero and Overflow flags are calculated and how the slt operation is implemented

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

waveform diagrams show the behavior of different signals in a digital circuit over time. They can be used to visualize how the execution of different operations affects the inputs and outputs of a circuit.

For example, consider the execution of an add operation in the ALU. During the add operation, the two input values (A and B) are added together by the adder component, and the result is passed to the Zero/Overflow/Result Selector block. The Zero/Overflow/Result Selector block calculates the zero and overflow flags based on the result of the add operation, and selects the result as the output of the ALU. The waveform diagram for an add operation might show the input signals (A and B) and the output signals (Result, Zero, and Overflow) over time, with spikes in the signals corresponding to changes in the input or output values.

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

A comprehensive test plan should aim to test all aspects of the digital circuit being designed, including functional and non-functional requirements. This may involve testing the circuit under different input conditions, testing different combinations of inputs, and testing the circuit under different operating conditions (e.g. different clock speeds or power levels). The test plan should also aim to test for edge cases and potential errors in the circuit design.

To demonstrate the functioning of test programs, waveforms can be used to show the values of signals in the circuit over time. A waveform should show the expected behavior of the circuit under test, as well as any anomalies or unexpected behavior that may occur during testing. The waveform should clearly show the input signals, the output signals, and any intermediate signals in the circuit. It should also clearly show the timing of the signals relative to the clock cycle.

In addition to waveforms, the test plan should also include documentation that explains the purpose of each test case, the expected results, and any deviations from the expected results that were observed during testing. This documentation should be clear and concise, so that anyone reviewing the test plan can easily understand the testing process and results.

Overall, a comprehensive test plan should aim to test all aspects of the digital circuit being designed and include documentation that clearly explains the testing process and results. Waveforms can be used to demonstrate the functioning of test programs by showing the values of signals in the circuit over time.

[Part 3] In your writeup, show the QuestaSim 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.

check the test application in the files called test files inside MIMPS directory

[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.

check the file in the directory called test files and find the 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). Name this file Proj1_bubblesort.s.

check the test application in the directory called test files

[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 schematic from part 1. What components would you focus on to improve the frequency?

The critical path can be identified by analyzing the timing paths in the circuit and determining which path has the longest delay. This delay is determined by the propagation delay of the gates, the capacitance of the wires, and other factors that affect signal propagation.

Once the critical path has been identified, the components that affect its delay can be optimized to improve the maximum operating frequency. This can include reducing the gate delay by using faster or smaller gates, reducing the capacitance of the wires by using shorter or wider wires, and optimizing the placement and routing of the components to minimize signal propagation delay.

Drawing the critical path on top of the top-level schematic can help to visualize the path and identify the components that need to be optimized. By focusing on these components and making appropriate optimizations, the maximum operating frequency of the processor can be improved.