Project Part 1 Report (CPRE 381)

CprE 381: Computer Organization and Assembly-Level Programming

Team Members:

- Conner Ohnesorge
- Levi Wenck

Project Teams Group #:TermProj1_2_02

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)]

Task

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.

Instruction	Opcode (Binary)	Funct (Binary)	jr [jr mux]	jal [jal mux]	MemtoReg	MemWrite (we_mem)	RegWrite (we reg)	Reg
I-TYPE								
`addi`	"001000"	"------"	0	0	0 [addi does NOT read from memory]	O [addi does NOT write to memory]	1 [addi writes back to a register]	0 [a uses destir regi

Instruction	Opcode (Binary)	Funct (Binary)	jr [jr mux]	jal [jal mux]	MemtoReg	MemWrite (we_mem)	RegWrite (we reg)	Reg
								rather r
`addiu`	"001001"	"------"	0	0	0 [addiu does NOT read from memory]	0 [addiu does NOT write to memory]	1 [addiu writes back to a register]	0 [a uses destir regi rather r
`andi`	"001100"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [us a destir regi rather r
`xori`	"001110"	"-----"	0	0	0 [does NOT read from memory]	O [does NOT write to memory]	1 [writes back to a register]	0 [us a destir regi rather r
`ori`	"001101"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [us a destir regi rather r
`slti`	"001010"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [us a destir regi rather r
`lui`	"001111"	"-----"	0	0	O[does NOT read from memory]	O [does NOT write to memory]	1 [writes back to a register]	0 [us a destir regi

Instruction	Opcode (Binary)	Funct (Binary)	jr [jr mux]	jal [jal mux]	MemtoReg	MemWrite (we_mem)	RegWrite (we reg)	Reg
								rather r
`beq`	"000100"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	0 [does NOT write to a register]	0 [d not m
`bne`	"000101"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	0 [does NOT write to a register]	0 [d not m
`Iw`	"100011"	"------"	0	0	1 [reads from memory]	0 [does NOT write to memory]	1 [writes back to a register]	a destir regis
`sw`	"101011"	"-----"	0	0	O [does not matter]	1 [writes to memory]	0 [does NOT write to a register]	0 [d not m
R-TYPE								
`add`	000000	100000	0	0	0 [add does NOT read from memory]	O [add does NOT write to memory]	1 [add writes back to a register]	1 [a uses destir
`addu`	000000	100001	0	0	0 [addu does NOT read from memory]	O [addu does NOT write to memory]	1 [addu writes back to a register]	1 [a uses destir

Instruction	Opcode (Binary)	Funct (Binary)	jr [jr mux]	jal [jal mux]	MemtoReg	MemWrite (we_mem)	RegWrite (we reg)	Reg
`and`	000000	100100	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [us a destir
`nor`	000000	100111	0	0	o[does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [us a destir
`xor`	000000	100110	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [us a destir
`or`	000000	100101	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [us a destir
`sit`	000000	101010	0	0	O [does NOT read from memory]	o [does NOT write to memory]	1 [writes back to a register]	1 [us a destir
`sll`	000000	000000	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 (us a destir

Instruction	Opcode (Binary)	Funct (Binary)	jr [jr mux]	jal [jal mux]	MemtoReg	MemWrite (we_mem)	RegWrite (we reg)	Reg
`srl`	000000	000010	0	0	O [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 (us a destir
`sra`	000000	000011	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [us a destir
`sub`	000000	100010	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 (us a destir
`subu`	000000	100011	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [us a destir
`jr`	"000000"	"001000"	1	0	O [does not matter]	0 [does NOT write to memory]	0 [does not matter]	0 [d not m
J-TYPE								
`j`	"000010"	"-----"	0	0	0 [does not matter]	0 [does not matter]	0 [does not matter]	0 [d not m

Instruction	Opcode (Binary)	Funct (Binary)	jr [jr mux]	jal [jal mux]	MemtoReg	MemWrite (we_mem)	RegWrite (we reg)	Reg
`jal`	"000011"	"-----"	0	1	O[does NOT read from memory]	O[does NOT write to memory]	1	0 [d not m

[Part 2 (a.ii)]

Task

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)]

Task

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.

Answer

i. Control Flow Possibilities

The instruction fetch logic must accommodate several control flow possibilities, including:

1. Sequential Execution: The default case where the Program Counter (PC) is simply incremented to point to the next instruction in memory.
2. Branches (Conditional and Unconditional):

- Conditional Branches: The PC is updated based on a condition. If the condition is true, the PC is set to a specific address; otherwise, it proceeds to the next sequential instruction.
- Unconditional Jumps: The PC is directly set to a specific address, regardless of any conditions.

3. Function Calls and Returns:

- Calls: Similar to unconditional jumps, but the address of the next instruction (return address) is saved on a stack or in a register.
- Returns: The PC is set to the return address, popping it from the stack or reading from a register, to continue execution from the point after the function call.

4. Interrupts and Exceptions: Special cases where the PC might be set to a handler routine's address based on external events or errors during execution.

More specifically, the control flow possibilities are as follows:

`bne`: must take a Branch not equal input to an AND gate "&'ed" with not Zero, this must activate the branch address or keep the normally incremented PCAddress.

`beq`: must take a Branch is Equal input to an AND gate with ZERO, to then choose for a multiplexer whether to branch or stay on same PC Address.

`Jal`: Jump and link must set register 31 to the PC Address incremented 4, by using a multiplexer on the write address port of the register file and porting the PC Address + 4 into a multiplexer to choose to take the output from the mem or ALU or the PCAddress and activates the PC Address when Jal is active to 1.

`j`: Jump runs to a multiplexer to choose between the given jump address extended by 2 0s and taking the first bits of the PC Address, or to keep the normal PC Address.

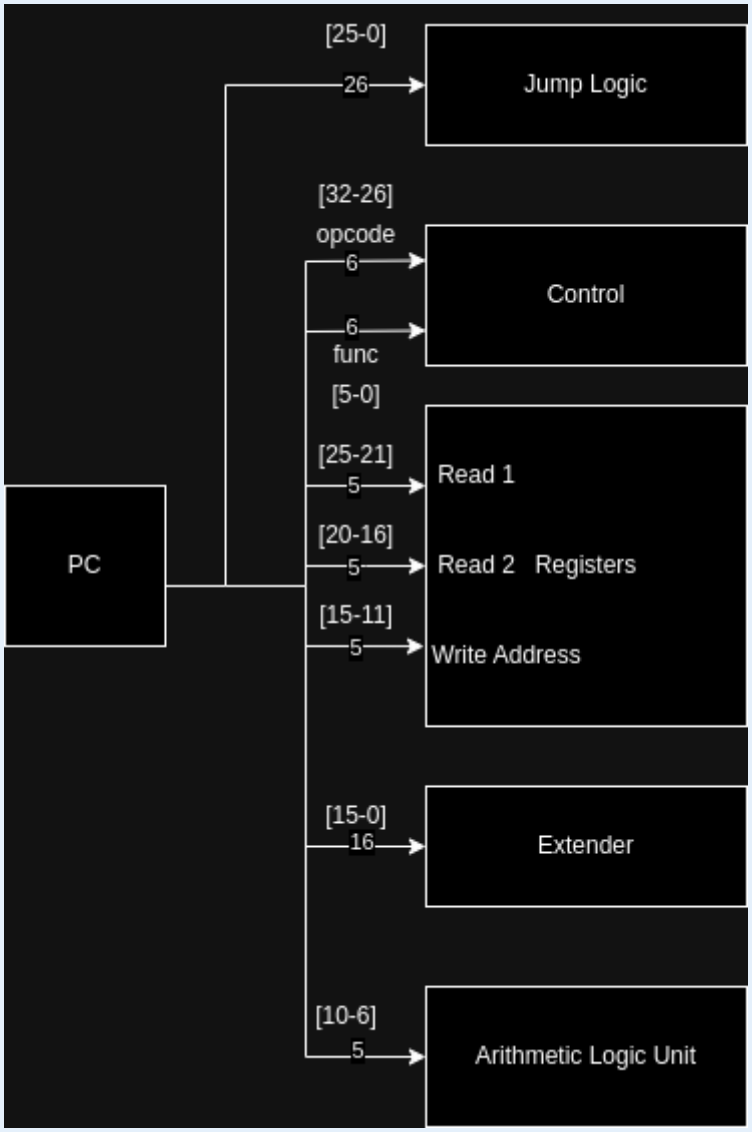
`jr`: Jump Register jumps to the saved register address in RS usually register 31, Jr signal is used to choose between the Jr Address input, or the normal PC + 4 Address.

[Part 2 (b.ii)]

Part 2 (b.ii)

Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions.

Part 2 (b.ii)



What additional control signals are needed?

The additional control signals needed are the following:

- `Branch`: This signal is used to determine whether the PC should be updated based on a branch condition.ќ

[Part 2 (b.iii)]

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 report.

[Part 2 (c.i.1)]

Part 2 (c.i.1)

Describe the difference between logical (`srl`) and arithmetic (`sra`) shifts. Why does MIPS not have a `sla` instruction?

Part 2 (c.i.1)

Logical shifts (`srl`) and arithmetic shifts (`sra`) are two types of bit shifting operations.

The primary difference lies in how they handle the introduction of new bits during the shift. In a logical shift, zeroes are inserted into the vacated bit positions. On the other hand, an arithmetic shift copies the sign bit (the most significant bit for signed numbers) into the vacated positions, preserving the number's sign.

MIPS architecture does not include a specific "shift left arithmetic" (`sla`) instruction. This is because the sign bit's significance is primarily at the most significant end of the number. Shifting left would involve filling in the least significant bit, but for arithmetic purposes, the action of simply adding zeros on the right (which happens in both logical and arithmetic left shifts) is sufficient.

Incorporating a `sla` operation that specifically modifies the sign in a manner different from normal left shifts would not align with how numbers are typically managed in binary arithmetic, potentially leading to unintended alterations in the value of the data being shifted.

[Part 2 (c.i.2)]

In your report, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

[Part 2 (c.i.3)]

In your report, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

The right barrel shifter can be enhanced to support left shifting operations by simply reversing the bits before the shift operation. This can be achieved by changing the direction of the shift operation, effectively shifting the bits in the opposite direction. By modifying the control signals to indicate a left shift instead of a right shift, the barrel shifter can be repurposed to perform left shifts.

[Part 2 (c.i.4)]

Describe how the execution of the different shifting operations corresponds to the QuestaSim waveforms in your report.

[Part 2 (c.ii.1)]

In your report, 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.

[Part 2 (c.ii.2)]

Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your report.

[Part 2 (c.iii)]

Task

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?

[Part 2 (c.v)]

Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your report.

[Part 2 (c.viii)]

Task

Justify why your test plan is comprehensive.
Include waveforms that demonstrate your test programs functioning.

[Part 3] In your report,

Task

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

Task

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

Task

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`.

```
.data
msg: .asciiz "Final result: "

.text
.globl main
```

```
# Function declarations for clarity
# Each function calls the next, incrementing a value passed through
registers
func1:
    addi $a0, $a0, 1      # Increment the argument
    jal func2             # Call func2
    jr $ra                # Return to caller

func2:
    addi $a0, $a0, 1      # Increment the argument
    jal func3             # Call func3
    jr $ra                # Return to caller

func3:
    addi $a0, $a0, 1      # Increment the argument
    jal func4             # Call func4
    jr $ra                # Return to caller

func4:
    addi $a0, $a0, 1      # Increment the argument
    jal func5             # Call func5
    jr $ra                # Return to caller

func5:
    addi $a0, $a0, 1      # Increment the argument
    jr $ra                # Return to caller with final value in $a0

main:
    # Initial setup
    li $a0, 0             # Initialize argument to 0

    # Call the first function in the chain
    jal func1

    # After returning from the call chain, print the result
    move $a0, $v0         # Move the result to $a0 for printing
    li $v0, 4             # System call for print_string
    la $a0, msg           # Load address of msg into $a0
    syscall

    li $v0, 1             # System call for print_int
    move $a0, $v0         # Move final result into $a0
    syscall

    # Exit program
```

```
    li $v0, 10            # System call for exit
    syscall
```

[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`.

```
.data
array: .word 5, 3, 8, 4, 2  # Example array to sort
n: .word 5                   # Number of elements in the array

.text
.globl main

main:
    la $t0, array           # Load the address of the array into $t0
    lw $t1, n               # Load the size of the array into $t1

    addi $t1, $t1, -1       # Decrease n by 1 because we need (n-1) passes
    move $t2, $t1           # Copy $t1 to $t2 for the outer loop counter

outer_loop:
    blez $t2, end_sort      # If $t2 is 0 or less, we're done sorting
    move $t3, $t0           # Copy base address of array to $t3 for inner
loop
    li $t4, 0               # Inner loop index i = 0

inner_loop:
    slt $t5, $t4, $t2       # Check if i < outer loop counter $t2
    beq $t5, $zero, update_outer # If not, update outer loop counter
    lw $t6, 0($t3)          # Load A[i]
    lw $t7, 4($t3)          # Load A[i+1]

    # Compare and swap if necessary
    ble $t6, $t7, no_swap
    sw $t6, 4($t3)          # Swap A[i] and A[i+1]
    sw $t7, 0($t3)

no_swap:
    addi $t3, $t3, 4        # Move to the next element in the array
    addi $t4, $t4, 1        # Increment inner loop index
    j inner_loop            # Jump back to the start of the inner loop
```

```
update_outer:
    addi $t2, $t2, -1       # Decrement outer loop counter
    j outer_loop            # Jump back to the start of the outer loop

end_sort:
    # Sorting is done, implement any post-sorting logic here
    # For now, just exit
    li $v0, 10              # Exit syscall
    syscall
```

[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?