MIPS Processor design and implementation

Computer Architecture

Name: Shreekar Mane

Abstract

This report details the design, implementation, and verification of a single-cycle MIPS processor capable of executing a subset of R-type, I-type, and J-type instructions. The processor datapath and control unit were constructed and simulated using Logisim. Key components, including the ALU, Register File, Control Unit, and Memory modules, were designed and integrated to create a functional processor. The design was verified by executing a test program and observing the state of the register file and data memory, confirming the correct execution of instructions.
Following instructions are implemented in design :
R-Type : `add, sub, and, or, slt`
I-Type : `lw, sw, beq, addi`
J-Type : `j`
Pseudo : `subi, move`
Currently Memory module only supports to Read and Write 1 word (4bytes) per instruction.
I will also use assembler which I made for last "graded assignment" (which converts assembly instructions in mahcine code). Using that assembler, it will be much easy to east our MIPS processor.

Contents

1 Introduction 3

1.1 Understanding of the Task . . . 3

1.2 Assumptions and Design Choices . . . 3

2 System Architecture and Component Design 4

2.1 Overall Datapath . . . 4

2.2 Component Details . . . 4

2.2.1 Control Unit . . . 4

2.2.2 ALU and ALU Control . . . 4

2.2.3 Register File . . . 5

2.2.4 Memory Units . . . 5

3 Simulation and Verification 7

3.1 Test Program . . . 7

3.2 About Machine code (generated by assembler) . . . 8

3.3 Simulation Results . . . 8

3.3.1 Execution of : `0x14 : add $s2, $s0, $s1` . . . 9

3.3.2 Execution of : `0x28 : sw $s2, result` . . . 9

3.3.3 Execution of : `0x30 : j end_program` . . . 10

4 Challenges, Learnings, Conclusion, Extra (GitHub) 11

4.1 Challenges Faced . . . 11

4.2 Key Takeaways . . . 11

4.3 Conclusion . . . 11

4.4 Extra (GitHub Repo for this project) . . . 11

1 Introduction

1.1 Understanding of the Task

The objective of this assignment was to design and simulate a simplified MIPS processor. The core task involved integrating previously designed components—the Arithmetic Logic Unit (ALU), Register File, and Control Units—into a cohesive single-cycle datapath. The processor was required to handle a specified set of R-type and I-type instructions, demonstrating the fundamental instruction execution cycle: fetch, decode, execute, memory access, and write-back.

1.2 Assumptions and Design Choices

To create a more robust and realistic processor model, several design choices were made that extend the original assignment requirements.

1. **Implementation of Assembler:** As I made assember in c++ that takes mips assembly file as input and it produces 'data memory' and 'instruction memory' images corresponding to given assembly program. It also generates 'about progra' file, such that user will able to understand how memory images are created. I will use my assembler to create machine instructions then I will load those in curresponding memories, then I will start simulation. After simulation ends or program ends we will get desire output reflected in data memory and register file. Following instructions are supported by my assembler and processor :
   R-Type : `add, sub, and, or, slt`
   I-Type : `lw, sw, beq, addi`
   J-Type : `j`
   Pseudo : `subi, move`

2. **Implementation of Memory:** This design incorporates dedicated Instruction Memory and Data Memory modules. This approach provides a more scalable and conventional processor architecture, allowing for the execution of programs stored in memory rather than a single, hardwired instruction. In this model, memories are byte addressable but design can only read word (4bytes) or write word (4bytes).

3. **Program Counter (PC):** A Program Counter (PC) was implemented to handle instruction fetching sequentially. This is a necessary component for a memory-based design and allows for proper execution flow, including branching and jumping.

4. **Debug overloading:** We can overload value of PC register by HEX_custom_PC if we set load_custom_PC=1. Similarly we can overload value of fetched instruction with HEX_custom_instruction if we set load_custom_instruction=1.

5. **Debug Readings:** There are several components in design which are only meant to 'examine' and 'debug' purpose. e.g. HEX_PC: to examine current PC value in Hex, B_instruction: to examine current fetched instruction in Binary, etc.

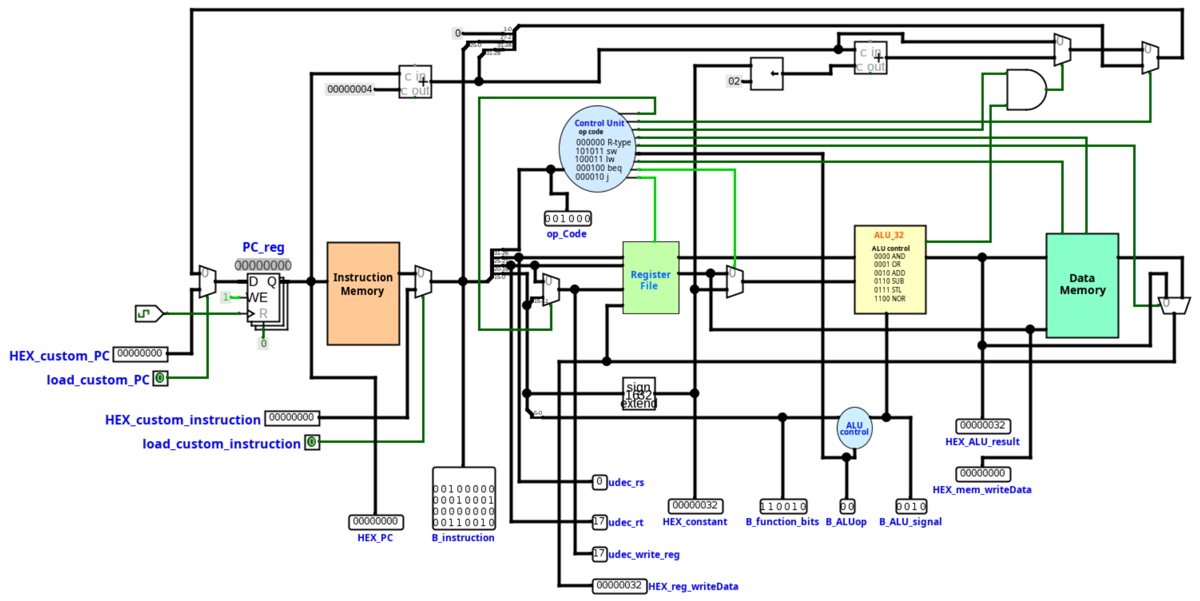


Figure 1: MIPS Processor

2 System Architecture and Component Design

2.1 Overall Datapath

The processor (Figure 1) implements a single-cycle datapath architecture. An instruction is fetched from memory, and in the same clock cycle, it travels through the entire datapath to completion. The main data flow path involves the PC, Instruction Memory, Register File, ALU, Data Memory, and various multiplexers that select the correct data for each stage based on control signals.

2.2 Component Details

The processor is built from several modular components as listed below.

2.2.1 Control Unit

The main Control Unit is a combinational logic circuit (Figure 2a) that takes the 6-bit opcode of the instruction as input and generates the primary control signals for the datapath. These signals dictate the operation of multiplexers, the Register File, and the Memory units.

2.2.2 ALU and ALU Control

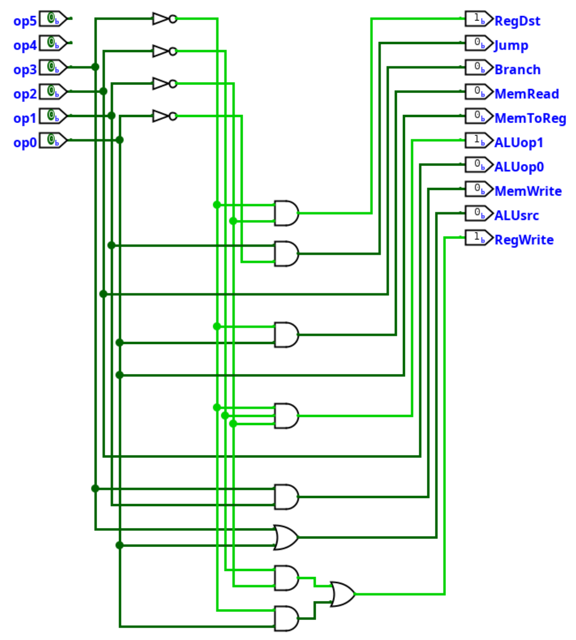
The 32-bit ALU performs arithmetic and logical operations. It takes two 32-bit operands and a 4-bit control signal from the ALU Control Unit to determine the operation. It outputs a 32-bit result and a single-bit `Zero` flag, which is asserted if the result is zero (used for the `beq` instruction). The ALU Control Unit (Figure 2b) takes the `funct` field from R-type instructions and a 2-bit `ALUOp` signal from the main Control Unit to generate the final 4-bit signal for the ALU.

2.2.3 Register File

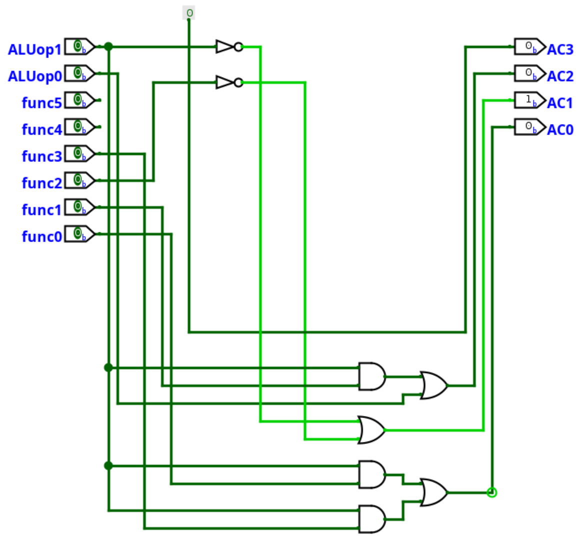
The Register File (Figure 2c) consists of 32 general-purpose 32-bit registers. It has two read ports (`Read Register 1`, `Read Register 2`) and one write port (`Write Register`). The `RegWrite` control signal must be asserted for a write operation to occur on the rising edge of the clock.

2.2.4 Memory Units

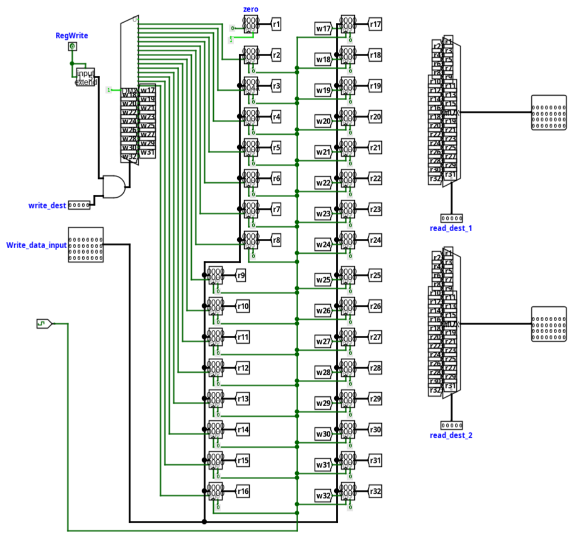
Separate Instruction Memory (Figure 2d) and Data Memory (Figure 2e) modules were used. The Instruction Memory is a read-only component that takes a 32-bit address from the PC and outputs the corresponding 32-bit instruction. The Data Memory is a read/write component used by `lw` and `sw` instructions, controlled by the `MemRead` and `MemWrite` signals.



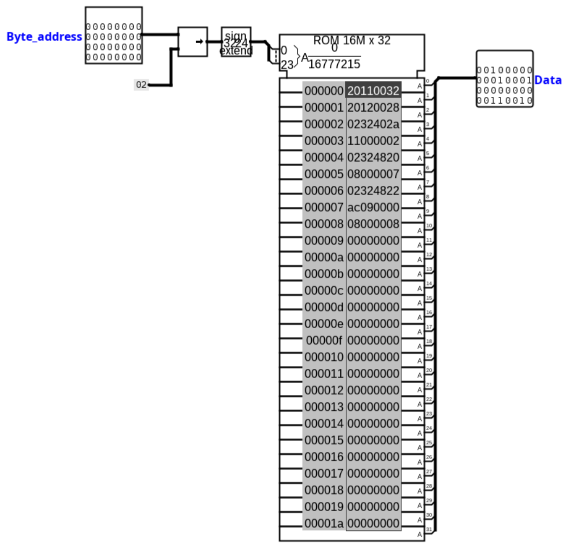
(a) Control Unit Circuit



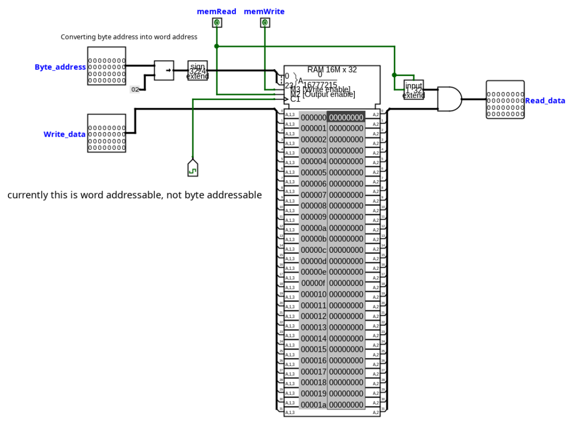
(b) ALU Control Circuit



(c) Register File



(d) Instruction Memory


memRead
memWrite
Converting byte address into word address
Byte_address
RAM 16M x 32
Read_data
Write_data
currently this is word addressable, not byte addressable

(e) Data Memory

Figure 2: Key hardware components of the MIPS processor.

3 Simulation and Verification

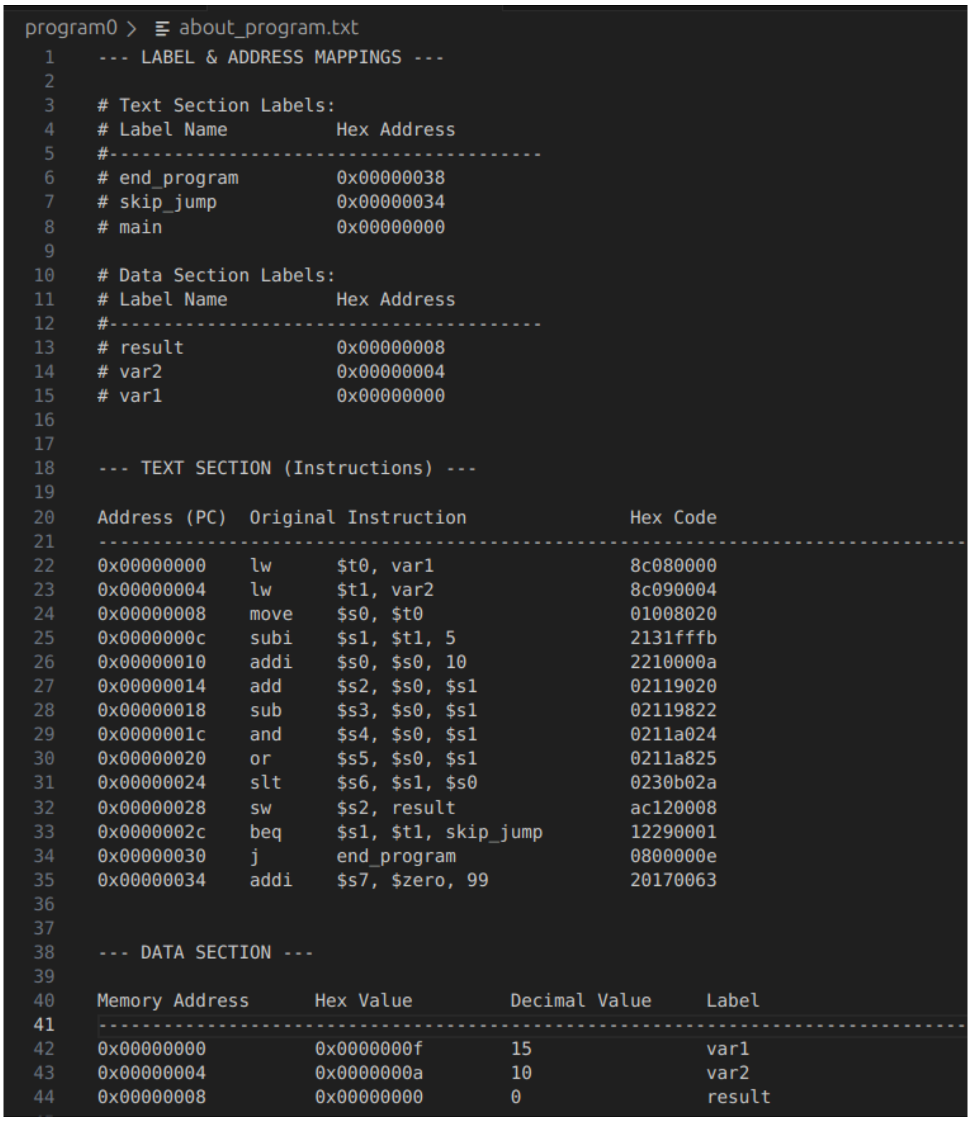
3.1 Test Program

To verify the correctness of the processor, the following MIPS assembly program was created and converted into a machine code image for the Instruction Memory. I tried to add various instructions in the following program.

```
.data
var1:   .word 15    # var1 = 15
var2:   .word 10    # var2 = 10
result: .word 0

.text

main:
    lw      $t0, var1       # $t0 = 15
    lw      $t1, var2       # $t1 = 10

    move    $s0, $t0        # $s0 = 15
    subi    $s1, $t1, 5     # $s1 = 5

    addi    $s0, $s0, 10    # $s0 = 25

    add     $s2, $s0, $s1   # $s2 = 30
    sub     $s3, $s0, $s1   # $s3 = 20
    and     $s4, $s0, $s1   # $s4 = 1
    or      $s5, $s0, $s1   # $s5 = 29
    slt     $s6, $s1, $s0   # $s6 = 1

    sw      $s2, result     # result = 30

    beq     $s1, $t1, skip_jump # Branch not taken (5 != 10)

    j       end_program

skip_jump:
    addi    $s7, $zero, 99  # $s7 = 99

end_program:
    # End of program
```

Listing 1: MIPS Assembly Test Code

3.2 About Machine code (generated by assembler)



```
program0 > about_program.txt
--- LABEL & ADDRESS MAPPINGS ---

# Text Section Labels:
# Label Name          Hex Address
#-------------------------------------
# end_program         0x00000038
# skip_jump           0x00000034
# main                0x00000000

# Data Section Labels:
# Label Name          Hex Address
#-------------------------------------
# result              0x00000008
# var2                0x00000004
# var1                0x00000000


--- TEXT SECTION (Instructions) ---

Address (PC)  Original Instruction              Hex Code
--------------------------------------------------------------------------------------
0x00000000    lw      $t0, var1                 8c080000
0x00000004    lw      $t1, var2                 8c090004
0x00000008    move    $s0, $t0                  01008020
0x0000000c    subi    $s1, $t1, 5               2131fffb
0x00000010    addi    $s0, $s0, 10              2210000a
0x00000014    add     $s2, $s0, $s1             02119020
0x00000018    sub     $s3, $s0, $s1             02119822
0x0000001c    and     $s4, $s0, $s1             0211a024
0x00000020    or      $s5, $s0, $s1             0211a825
0x00000024    slt     $s6, $s1, $s0             0230b02a
0x00000028    sw      $s2, result               ac120008
0x0000002c    beq     $s1, $t1, skip_jump       12290001
0x00000030    j       end_program               0800000e
0x00000034    addi    $s7, $zero, 99            20170063


--- DATA SECTION ---

Memory Address      Hex Value         Decimal Value     Label
--------------------------------------------------------------------------------------
0x00000000          0x0000000f        15                var1
0x00000004          0x0000000a        10                var2
0x00000008          0x00000000        0                 result
```

Figure 3: About Program (generated by assembler)

3.3 Simulation Results

The execution of each instruction was simulated, for some instructions i will show the Register File and Data Memory was captured before and after execution to demonstrate correctness.

3.3.1 Execution of : `0x14 : add $s2, $s0, $s1`

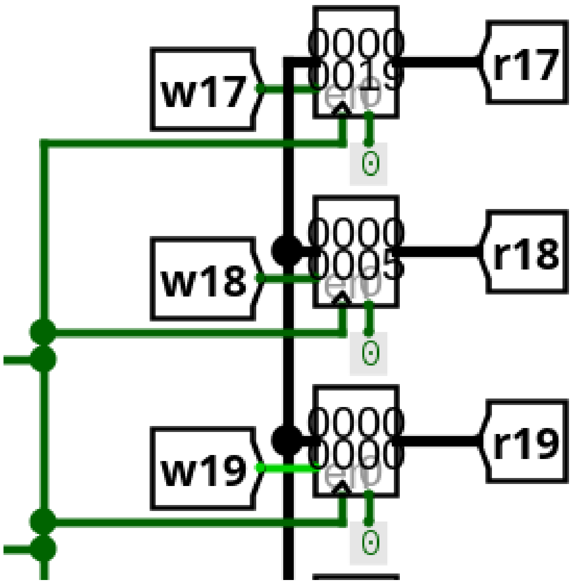


Figure 4: Registers just before execution : s0=0x19 s1=0x5 s2=0x0

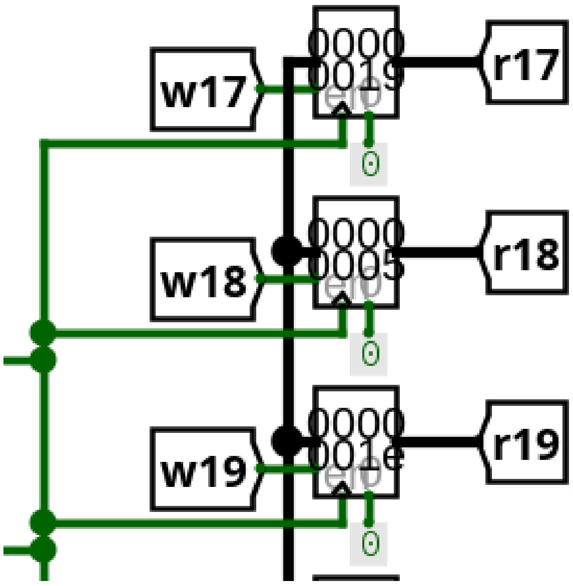


Figure 5: Registers just after execution : s0=0x19 s1=0x5 s2=0x1e

3.3.2 Execution of : `0x28 : sw $s2, result`

This instruction stores value of $s2 (which is 0x1e) into result (whose byte address is 0x00000008 in data memory (= 0x2 word address)

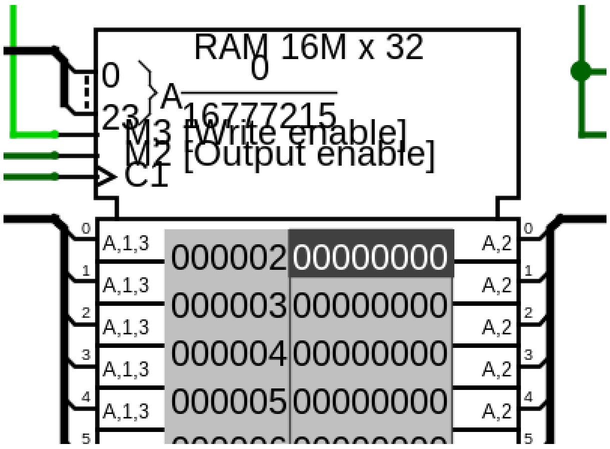


Figure 6: Data memory before sw : value is 0 at 0x8(byte address) which is 0x2 if word addressed memory

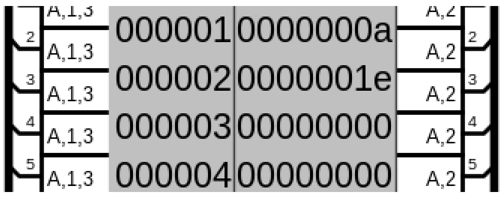


Figure 7: Data memory after sw : value is 0x1e at 0x8(byte address) which is 0x2 if word addressed memory

3.3.3 Execution of : `0x30 : j end_program`

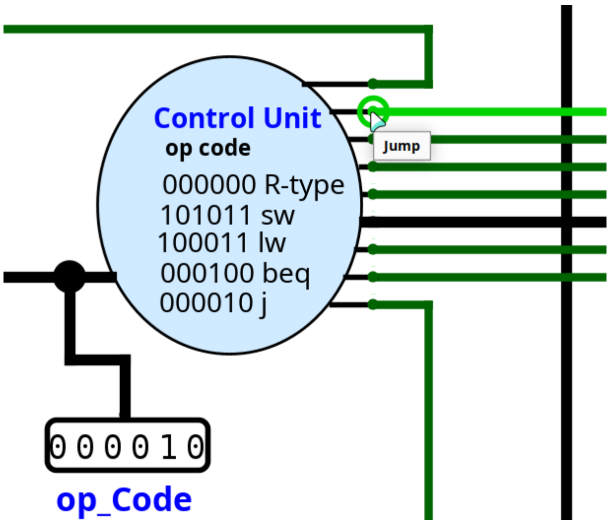


Figure 8: Control Unit state on jump instruction

4 Challenges, Learnings, Conclusion, Extra (GitHub)

4.1 Challenges Faced

One of the primary challenges was the physical wiring of the datapath in Logisim. With numerous 32-bit buses and control signals, keeping the design clean and avoiding incorrect connections required careful use of tunnels and labels. Debugging the control logic for the `beq` instruction was also non-trivial, as it involved ensuring the PC was correctly updated with the calculated branch address only when the `Zero` flag from the ALU was asserted.

4.2 Key Takeaways

This assignment provided invaluable hands-on experience in computer architecture. The key takeaway was a deep understanding of the intricate relationship between the datapath and the control unit. Designing each component modularly and then integrating them highlighted the importance of abstraction in complex digital systems. Furthermore, the process of testing and debugging solidified my understanding of how a processor executes different instruction formats at a fundamental level.

4.3 Conclusion

A single-cycle MIPS processor was successfully designed, simulated, and verified. The final implementation supports a functional set of R-type, I-type, and J-type instructions, including arithmetic, logical, memory access, and control flow operations. The project meets all the core objectives of the assignment and demonstrates a practical application of theoretical concepts in computer organization and architecture.

4.4 Extra (GitHub Repo for this project)

I have maintained github repository for this project, as this is large project so if something terrible happenes then it will be easy to recover.
This is github repo : `https://github.com/shreekar2005/MIPS_processor`

References

1. Class Notes
2. Some short videos to learn about logisim