Design and Implementation of a Simple MIPS Processor

TAVISHI SRIVASTAVA
Roll no.: B23CS1101

Contents

1 Assumptions

To implement the required MIPS processor, the following assumptions were made:

- **Instruction Source:** The 32-bit instruction is provided from an external 32-bit input pin named `$im0`. This pin is the single source of instructions.
- **Branch Target Output:** The 32-bit branch target address is sent to an external 32-bit output pin named `$im1`. This pin's value is updated on every cycle, regardless of the instruction.
- **Program Counter (PC):** The PC is not implemented as a register. It is assumed to be fixed at address `0x00000000`. Therefore, `PC + 4` is a constant value of `0x00000004`.
- **Register File Mapping:** Standard MIPS register numbering is assumed. Eg:
  - `$zero` = Register 0 (hardwired to 0)
  - `$t0` = Register 8
  - `$t1` = Register 9
  - `$t2` = Register 10
- **Opcodes and Funct Codes:** Standard MIPS opcodes and function fields are used.
  - **R-type (Opcode 0x00):** `add` (0x20), `sub` (0x22), `slt` (0x2A), `and` (0x24), `or` (0x25).
  - **I-type:** `addi` (0x08), `beq` (0x04).
- **Sign Extension:** The 16-bit immediate value for I-type instructions is always sign-extended to 32 bits.

2 Understanding of the Task

The objective of this assignment was to integrate previously designed components (ALU, Register File, Control Units) into a single-cycle MIPS processor datapath. This processor must support a subset of MIPS instructions, specifically R-type arithmetic/logic operations, their I-type immediate counterparts, and the `beq` branch instruction.

Key constraints differentiate this from a standard MIPS processor:

- **No Instruction Memory:** Instructions are fed manually via an input pin `$im0`.
- **No Data Memory:** All operations are register-to-register or register-immediate. No `lw` or `sw` instructions are supported.
- **No PC Loop:** The PC is a fixed constant (`0x00000000`).
- **Branch Handling:** Instead of updating a PC, a taken `beq` instruction calculates its target address (`PC+4 + (offset « 2)`) and outputs this value to a pin `$im1`. If the branch is not taken, `$im1` outputs the `PC+4` value (`0x00000004`).

The task involved wiring these components together, creating necessary "glue" logic (MUXes, Sign Extend unit, Branch Unit), and routing data and control signals according to the MIPS single-cycle datapath architecture.

3 Processor Datapath Diagram

The complete datapath implemented in Logisim is shown in Figure 1. This diagram illustrates the flow of data from the instruction pin `$im0`, through the Register File and ALU, and back to the Register File, as well as the branch calculation logic feeding the `$im1` output.

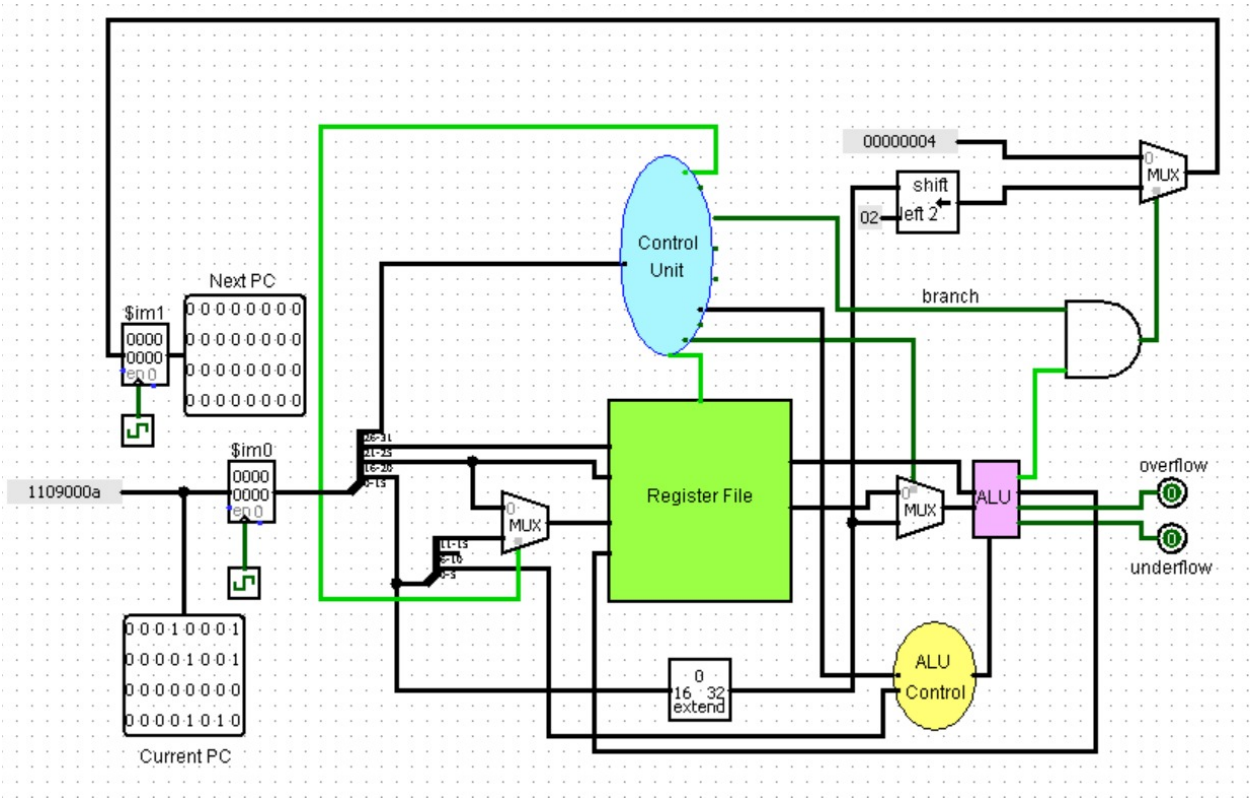


Figure 1: The main processor datapath in Logisim.

4 Sub-Components/Building Blocks

The project is comprised of several component files, which are integrated into a main circuit. Let us look at each of these components one-by-one:

- **main.circ:** This is the top-level file for the processor. It contains the complete datapath, integrating all the sub-circuits listed below.
- **register_file:** The 32x32-bit register file. It has the 5-bit ports Read Register 1 and Read Register 2. The register numbers referred to by these ports are looked up in the register file (with the help of Multiplexer selection lines) and the concerned register's data is read. It has the port regDest, which takes which register is to be considered as the destination register (i.e., where the result has to be stored). It has a writeData port, where the data to be written to the destination register is received. The outputs of the register file are two values read from the appropriate registers. It also has a Control Port, which receives the signal for the multiplexer selection of the destination register.
- **ALU_32:** The 32-bit ALU takes two 32-bit inputs and a 4-bit ALU control signal, to perform one of the operations, add, sub, and, or, slt. It outputs a 32-bit result and a 1-bit `Zero` flag.
- **control_unit:** The main control unit takes the 6-bit opcode from the instruction (`$im0[31:26]`) and generates all primary control signals: `regDest`, `jump`, `branch`, `memRead`, `memToReg`, `2-bit ALU Op`, `memWrite`, `ALUSrc` and `regWrite`.
- **aluop_funct_alucontrol:** The ALU Control Unit takes the 2-bit `ALUOp` from the main Control Unit and the 6-bit `funct` field from the instruction (`$im0[5:0]`) to generate the 4-bit signal for the ALU.
- **overflow and underflow:** These two units simply take two 32-bit inputs and signal if there is an overflow (in case of addition) or underflow (in case of subtraction) using a simple logical circuit.

5 Non-trivial Logic of the Design

The most complex parts of the design are the control units and the branch logic.

5.1 Main Control Unit Logic

The main Control Unit is a combinational circuit that maps the 6-bit opcode to the necessary control signals. Its logic is summarized in Table 1.

Table 1: Main Control Unit Signal Logic

Instruction	RegDst	Jump	Branch	MemRead	MemToReg	ALUOp	MemWrite	ALUSrc	RegWrite
R-type (0x00)	1	0	0	0	0	10	0	0	1
I-type (addi, etc.)	0	0	0	0	0	00	0	1	1
`beq` (0x04)	x	0	1	0	x	01	0	0	0

(**x** = Don't Care)

5.2 ALU Control Unit Logic

The ALU Control Unit further decodes the operation. For R-type instructions (`ALUOp`=10), it uses the `funct` field. For I-type (`ALUOp`=00), it performs an `add` (for `addi`). For `beq` (`ALUOp`=01), it performs a `sub` to check for equality.

Table 2: ALU Control Unit Signal Logic

ALUOp	Funct Field / Opcode	ALU Signal
10 (R-type)	...100000 (add)	0010 (add)
10 (R-type)	...100010 (sub)	0110 (sub)
10 (R-type)	...100100 (and)	0000 (and)
10 (R-type)	...100101 (or)	0001 (or)
10 (R-type)	...101010 (slt)	0111 (slt)
00 (I-type)	`addi`, `andi`, `ori`, `slti`	0010 (add) / 0000 (and) / 0001 (or) / 0111 (slt)
01 (`beq`)	(Don't Care)	0110 (sub)

5.3 Branch Unit Logic

The branch unit logic is non-standard.

1. The 16-bit immediate from `$im0[15:0]` is passed to the **Sign Extend** unit.
2. The 32-bit sign-extended result is **shifted left by 2**.
3. An **adder** calculates `(SignExtend(Imm) « 2) + 0x00000004`. This is the `Branch_Target_Address`.
4. A 2-to-1 **MUX** selects the value for the `$im1` register.
   - Input 0: `0x00000004` (the `PC+4` value)
   - Input 1: The `Branch_Target_Address`
5. The MUX's select bit is the result of `Branch AND Zero`.
   - If the instruction is not `beq`, `Branch`=0, MUX selects Input 0.
   - If it is `beq` but `$rs != $rt`, `Zero`=0, MUX selects Input 0.
   - If it is `beq` and `$rs == $rt`, `Branch`=1 and `Zero`=1, MUX selects Input 1.

Thus, `$im1` correctly outputs the target address only if the branch is taken, and `PC+4` otherwise.

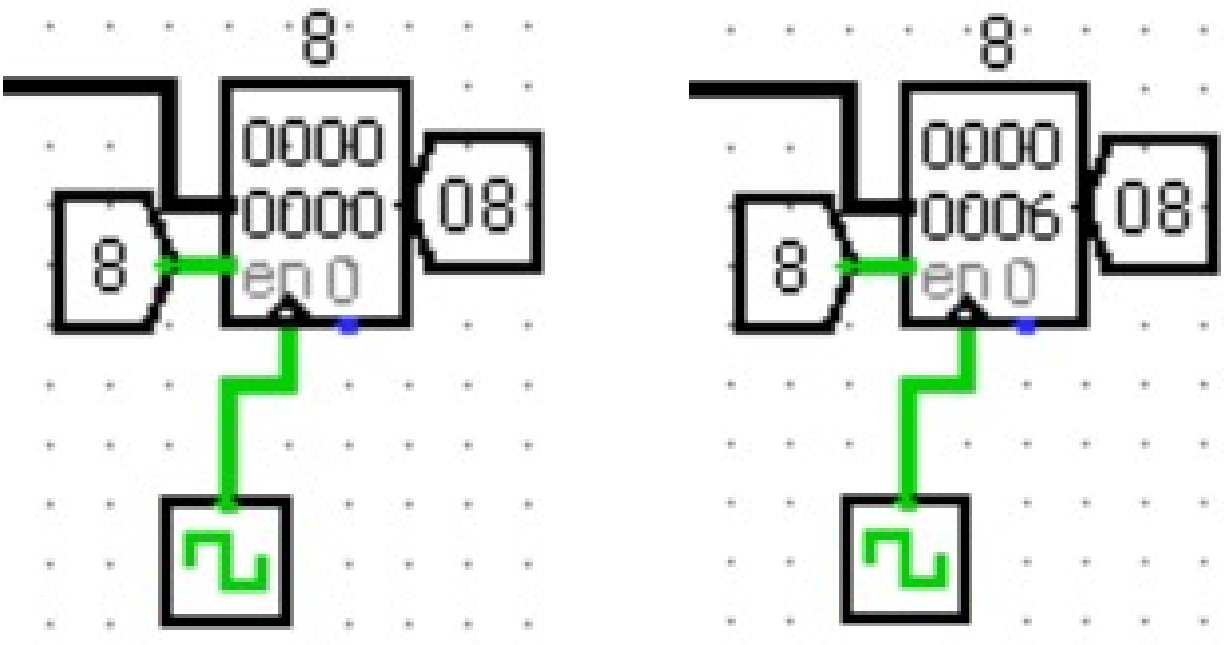
6 Modifications to Previous Components

To integrate the stand-alone components from previous assignments, several modifications and additions were necessary:

- **Integration Wiring:** The primary "modification" was wiring the components together. This involved connecting the `$im0` instruction splitter to the `Opcode` input of the Control Unit, the `Read Reg` ports of the Register File, the `Funct` input of the ALU Control, and the `Immediate` field to the Sign Extend unit.
- **Datapath MUXes:** New components were added to facilitate correct data routing:
  - **RegDst MUX:** A 2-to-1 MUX was added to select the `Write Register` address for the Register File. It chooses between `rt` (`$im0[20:16]`) for I-types and `rd` (`$im0[15:11]`) for R-types, controlled by the `RegDst` signal.
  - **ALUSrc MUX:** A 2-to-1 MUX was added to select the second operand for the ALU. It chooses between `Read Data 2` from the Register File (for R-type and `beq`) and the 32-bit sign-extended immediate (for I-type arithmetic/logic), controlled by the `ALUSrc` signal.
- **Branch Unit:** The entire Branch Unit (shifter, adder, MUX) was a new component added to fulfill the assignment requirements for `beq`.
- **Aesthetic/Wiring:** As noted, splitters and tunnels were used to bundle wires (like the `ALUOp` bits) to make the main schematic cleaner. The visual appearance (shape, color) of the custom components was also adjusted for better readability.

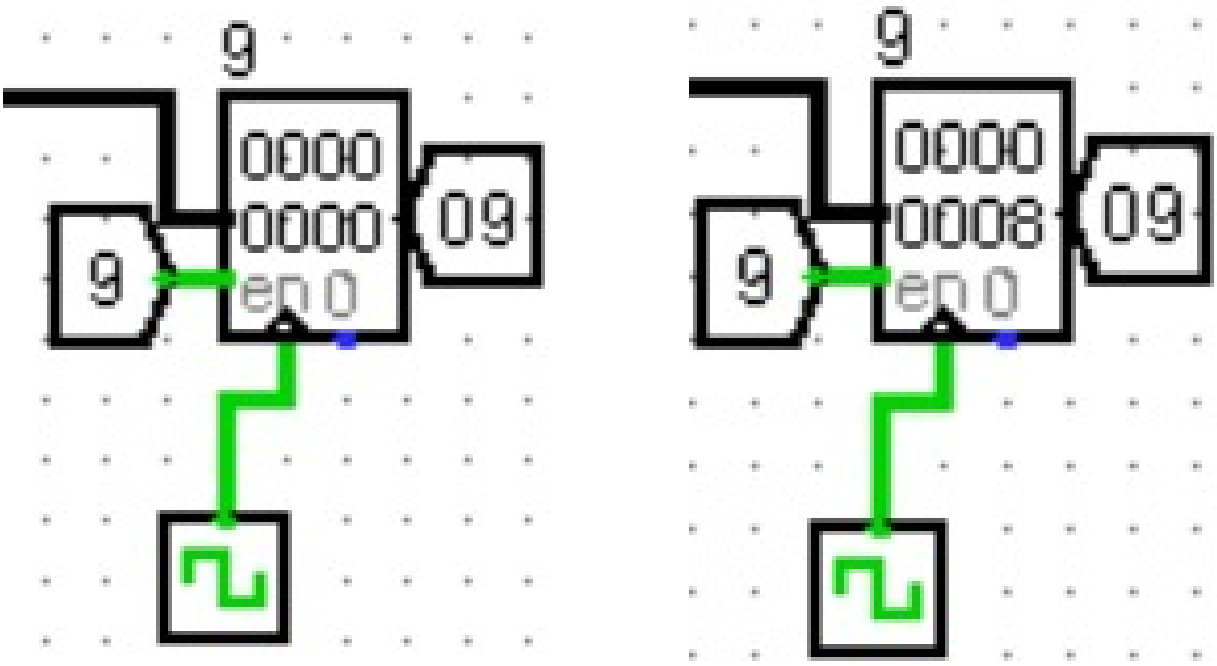
7 Test Instructions and Simulation Results

7.1 Instruction 1: `addi $t0, $zero, 6`



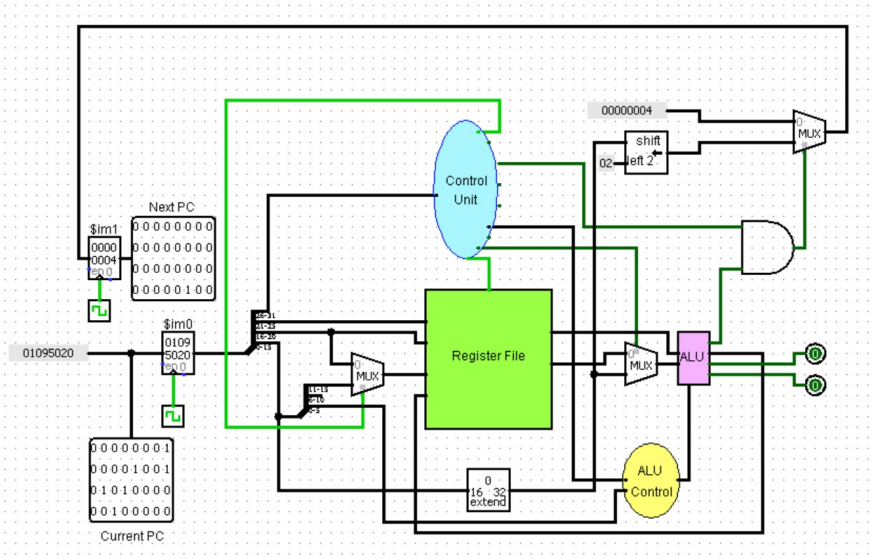
Result: Register $t0 loaded with immediate value 6.

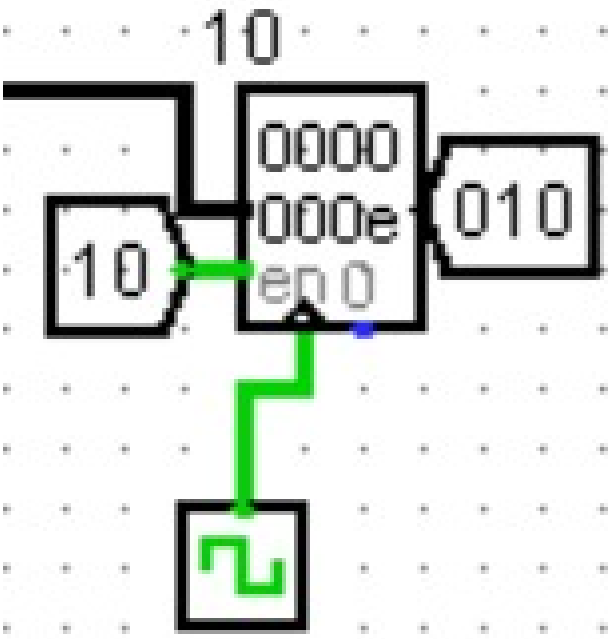
7.2 Instruction 2: `addi $t1, $zero, 8`



Result: Register $t1 loaded with immediate value 8.

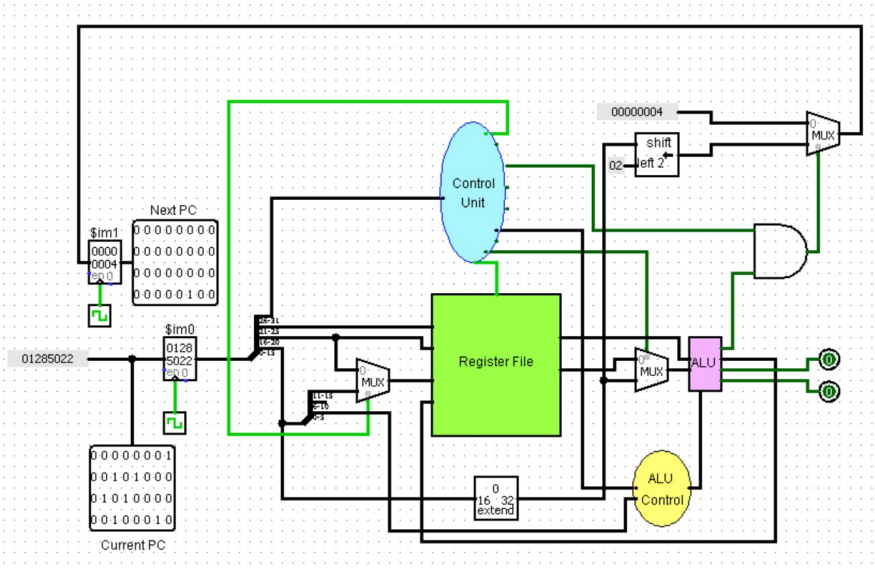
7.3 Instruction 3: `add $t2, $t0, $t1`

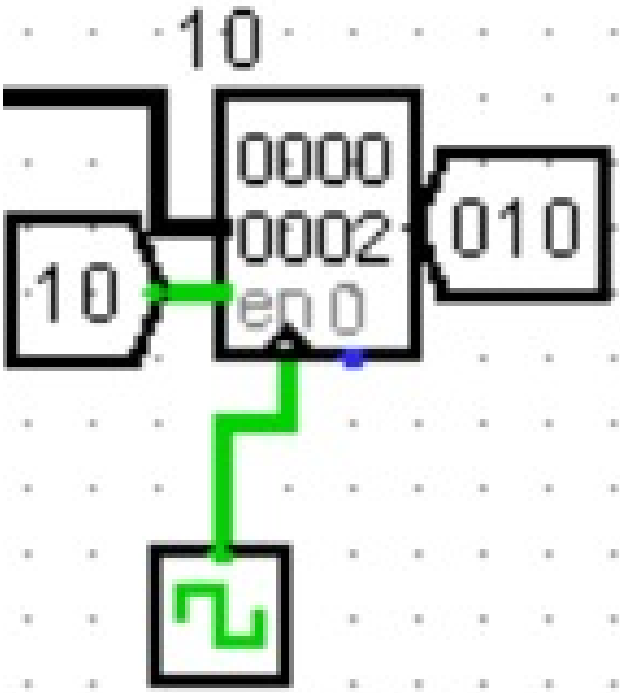




Result: $t2 = $t0 + $t1 = 6 + 8 = 14.

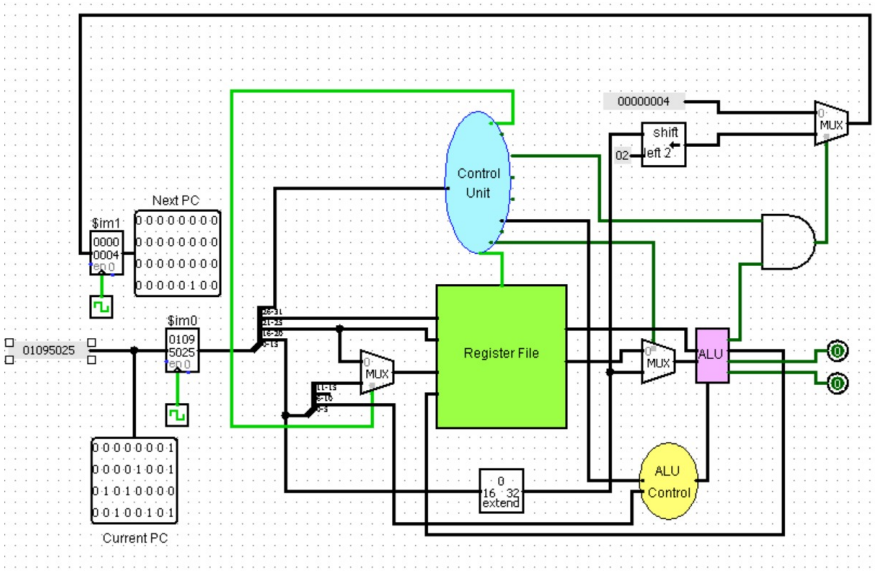
7.4 Instruction 4: `sub $t2, $t1, $t0`

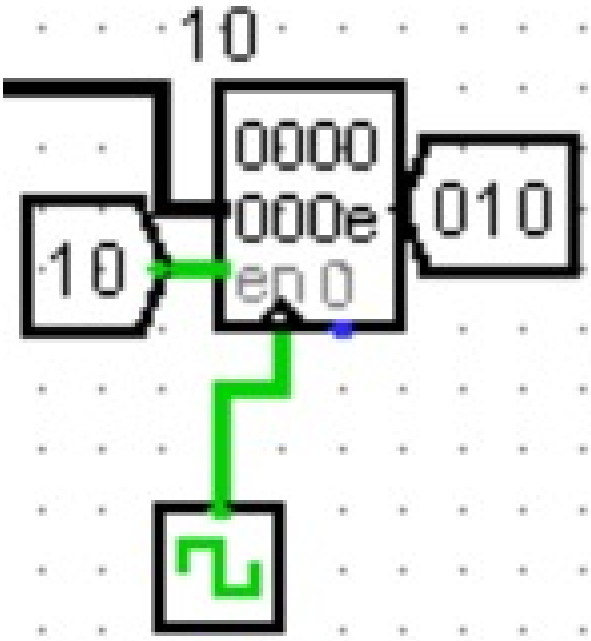




Result: $t2 = $t1 - $t0 = 8 - 6 = 2.

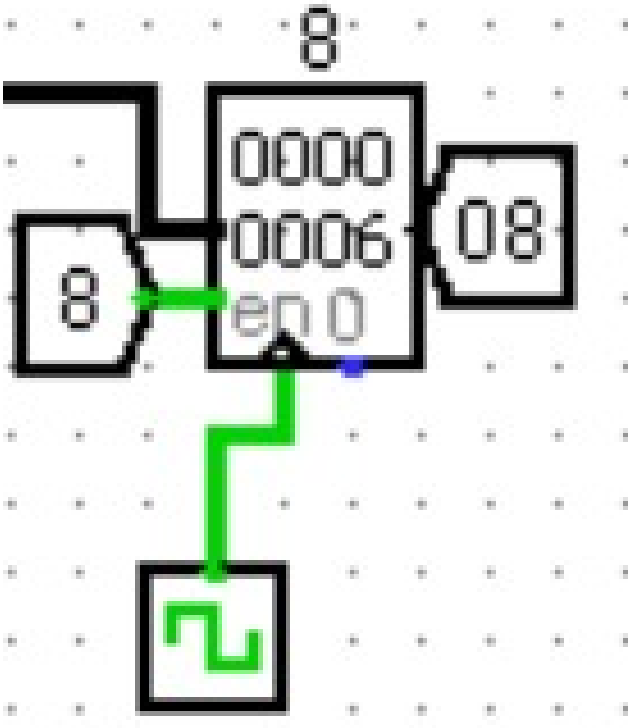
7.5 Instruction 5: `or $t2, $t0, $t1`





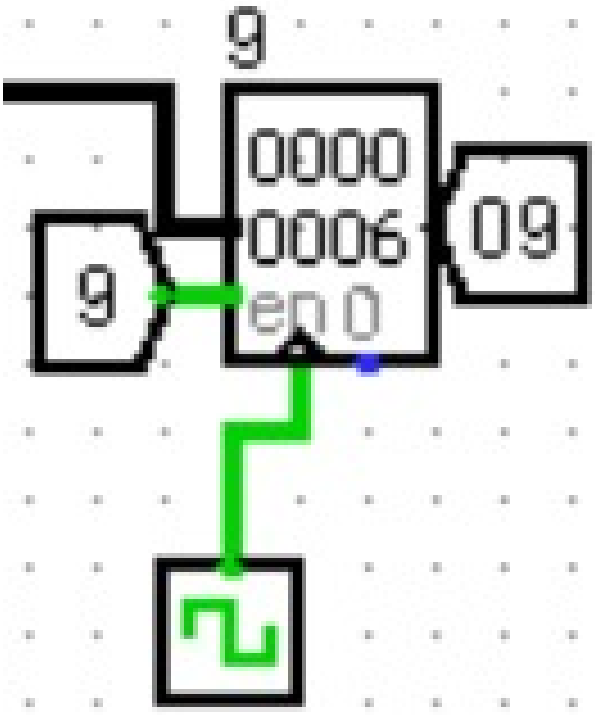
Result: Bitwise OR performed on $t0 and $t1.

7.6 Instruction 6: `addi $t0, $zero, 6`



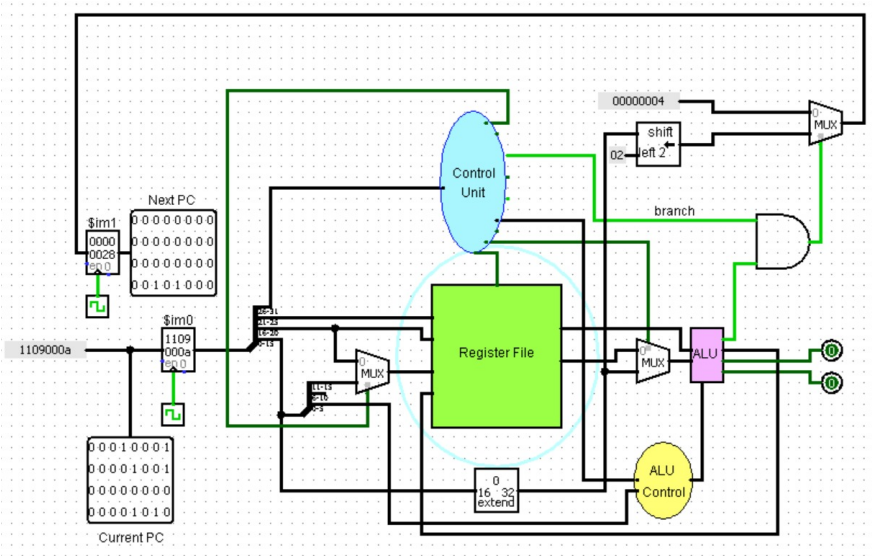
Result: Register $t0 loaded with immediate value 6.

7.7 Instruction 7: `addi $t1, $zero, 6`



Result: Register $t1 loaded with immediate value 6.

7.8 Instruction 8: `beq $t0, $t1, 10`



Result: Since $t0 = $t1, branch taken to PC + 4 + (10 × 4).

8 Challenges Faced and Mitigation

- **Challenge:** Understanding and implementing the non-standard I/O. Using `$im0` as an instruction "register" (pin) and `$im1` as a branch output pin was confusing, as it breaks the conventional PC loop.
- **Mitigation:** I treated `$im0` as a simple 32-bit input splitter, wiring its fields directly to the components that needed them (Control, RegFile, SignExtend). For `$im1`, I built the branch logic (shifter, adder, MUX) as a separate unit and wired its final output directly to the `$im1` pin, effectively isolating it from the main register-write datapath.
- **Challenge:** Debugging wiring errors in the main datapath. With so many components and bundled wires, it was easy to cross signals or miss a connection.
- **Mitigation:** I used Logisim's "Poke" tool extensively to manually set instruction bits in `$im0` and observe the control signals and data values at each stage. Testing one instruction class at a time (e.g., test all R-type, then all I-type) helped isolate problems to specific MUXes or control signals (like `ALUSrc`).
- **Challenge:** Implementing the `RegDst` MUX logic correctly, which selects between `rt` and `rd` for the write address.
- **Mitigation:** This was solved by carefully splitting the `$im0` instruction bus and feeding both the `rt` field (`[20:16]`) and `rd` field (`[15:11]`) into a 2-to-1 MUX, with the `RegDst` signal from the Control Unit as the selector.

9 Take-aways from the Assignment

This assignment was extremely valuable in bridging the gap between theoretical processor components and a functional datapath.

- **Datapath Integration:** I gained a concrete understanding of how the Register File, ALU, and Control Units are "glued" together with MUXes to create a working processor.
- **Control is Key:** I now have a much clearer appreciation for the Main Control Unit. It truly acts as the "brain," directing the flow of data by telling the MUXes which data to select, telling the ALU what operation to perform, and telling the Register File when to write.
- **Instruction Formats Matter:** I saw firsthand *why* R-type and I-type instructions have different formats. The fixed fields for `opcode`, `rs`, `rt`, `rd`, `funct`, and `immediate` are not arbitrary; they are designed to be easily fed into the datapath's components.
- **Single-Cycle View:** I learned how all operations for a single instruction (fetch, decode, execute, write-back) can be represented as a single, large combinational logic circuit (with the register file as a sequential element) that stabilizes to a final value within one clock cycle.
- **Problem-Solving:** The non-standard constraints (`$im0`, `$im1`) forced me to think about what the datapath *does* rather than just copying a standard diagram. I had to adapt the design to meet specific, unusual requirements, which is a key engineering skill.

10 Resources referred

- Class Notes (Simple MIPS Processor)
- MIPS register numbers