Cpre281 Final Project

Conner Ohnesorge

2024-04-10

Abstract

This is the final project for CPRE281 taught at Iowa State University By Conner Ohnesorge. The project is a single-cycle MIPS processor written in verilog with reflections from writing a similar single-cycle processor project in vhdl that can execute a subset of the MIPS instruction set with additional features such as a clock divider and displays the current instruction on the seven segment displays present on the FPGA board. The processor is implemented in Verilog and tested using a test-bench to verify the functionality of the processor. The processor is tested on an FPGA board to verify the functionality of the processor. The processor is implemented in Verilog and tested using a test-bench to verify the functionality of the processor. The processor is tested on an FPGA board to verify the functionality of the processor.

Table of Contents

Proposal

This section contains the proposal that was submitted for the project.

A MIPS processor that can execute a subset of the MIPS instruction set with additional features such as a clock divider, displaying the current instruction on the seven segment displays present on the FPGA board, and the ability to change the frequency of execution of the processor.

More specifically, the processor will be able to execute the following instructions: **LW SW J ADD ADDI BEQ ADDU SUBU AND ANDI OR ORI SUB NOR BNE SLT**

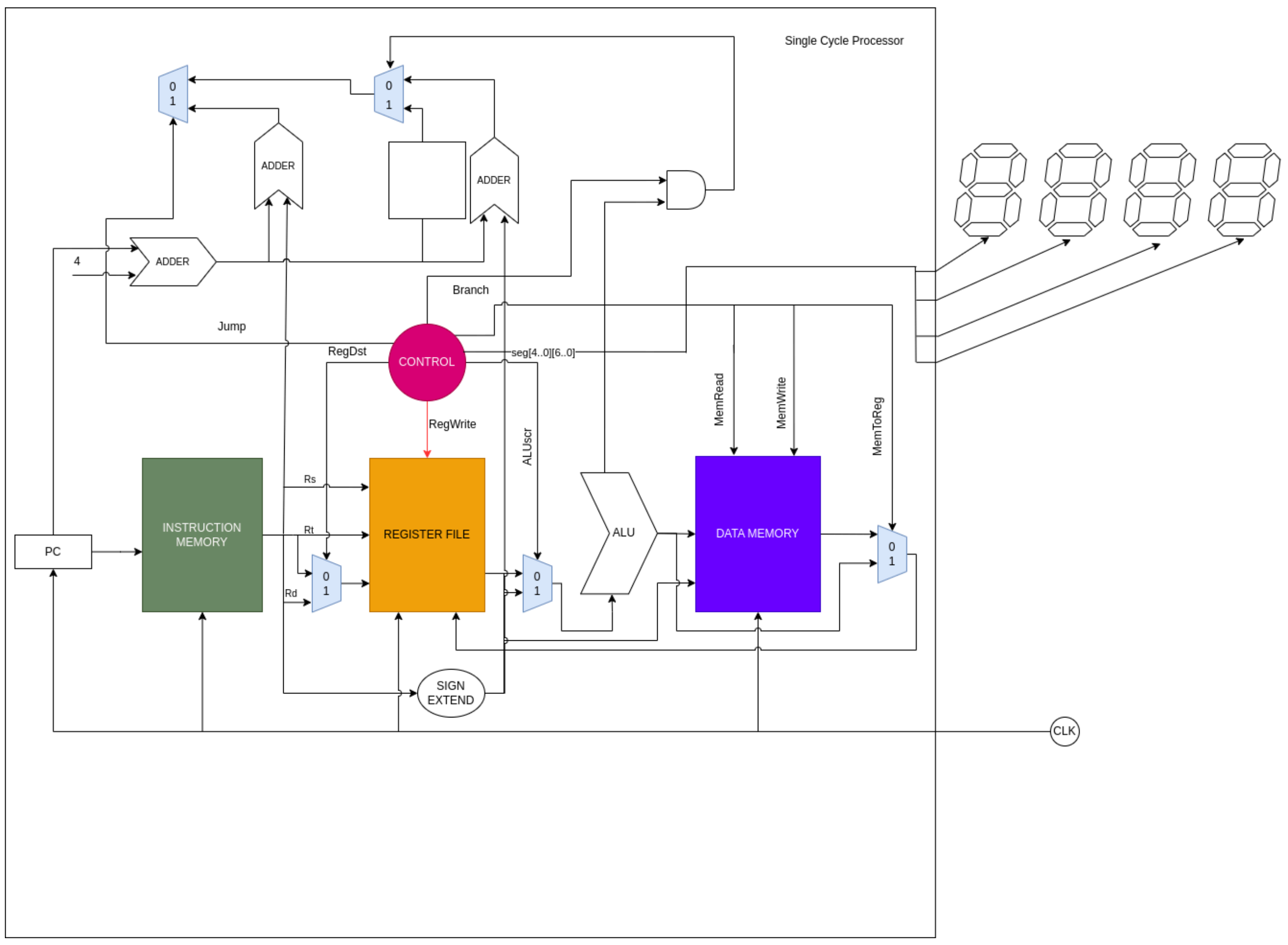
The frequency of execution for the processor will be controlled by a clock divider. The current instruction being executed will be displayed on the seven segment displays present on the FPGA board. The processor will be implemented in Verilog and tested using a test-bench.

After the processor has been verified using the test-bench, the processor will be tested on an FPGA board to verify the functionality of the processor.

The comparison and contrast of the experience writing the same processor in both Verilog and VHDL will be also included in the final report.

Introduction

The project is a single-cycle MIPS processor that can at a variable speed execute a subset of the MIPS instruction set displaying the current instruction on the seven segment displays present on **EP4CE115F29C7** FPGA board.



Circuit Diagram.png

As a result of this desired function (and it's actualation into reality), the processor can technically be used to do **all** the other projects from other students in the class. As I took this class whilst also taking CPRE381, I additionally decided to compare and contrast the experience writing the same processor in both Verilog and VHDL.

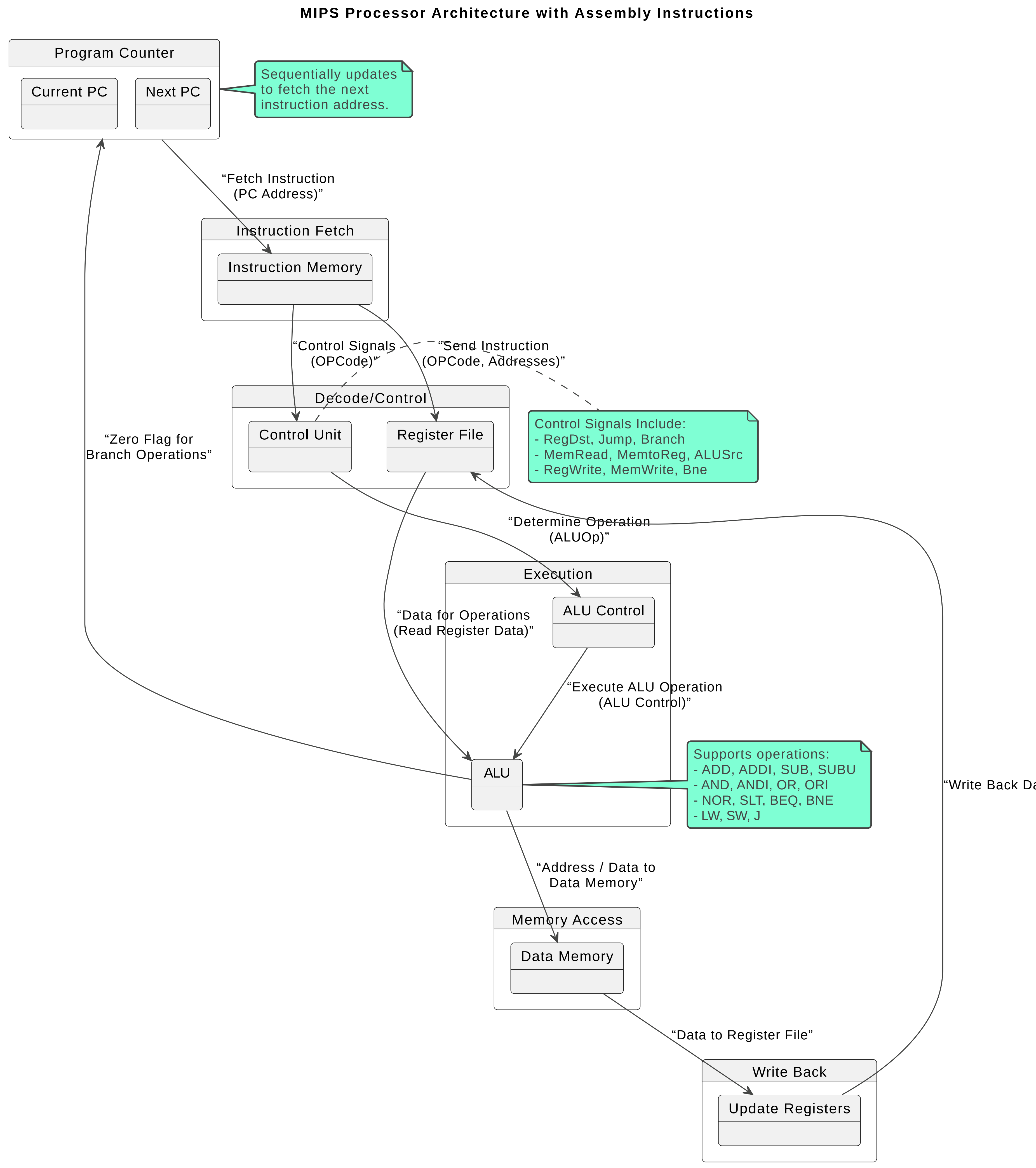
While the overall state machine will be broken down below, the main processor state machine has **five** states:

Fetch: In this state, the processor fetches the next instruction from memory. **Decode**: In this state, the processor decodes the instruction to determine what operation to perform. **Execute**: In this state, the processor executes the instruction. **Memory**: In this state, the processor accesses memory to read or write data. **Write-back**: In this state, the processor writes the results of the instruction to a register.

Supported Instructions: LW SW J ADD ADDI BEQ ADDU SUBU AND ANDI OR ORI SUB NOR BNE

State Machines

Single Cycle MIPS Processor Staging

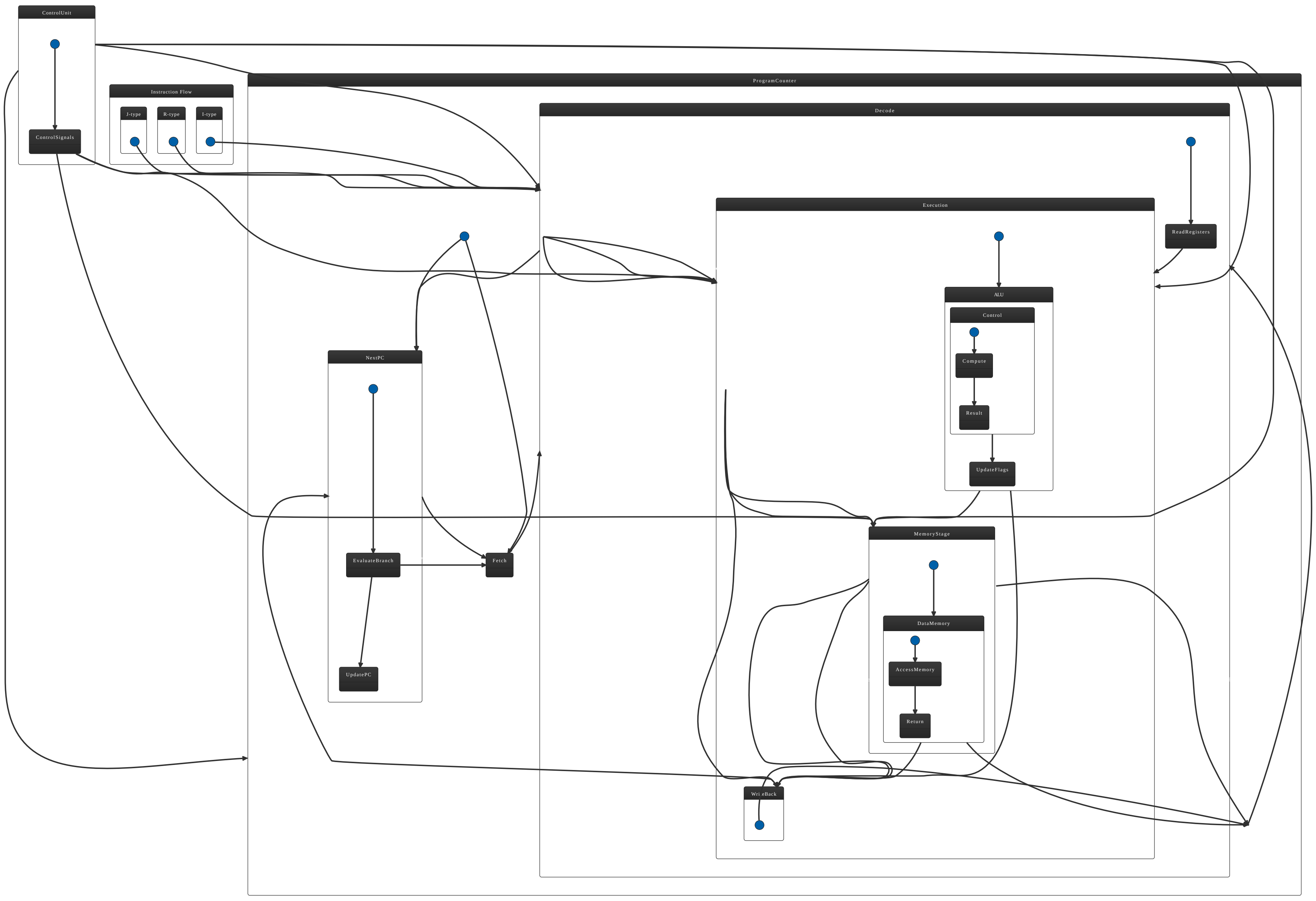

MIPS Processor Architecture with Assembly Instructions
Program Counter
Current PC
Next PC
Sequentially updates to fetch the next instruction address.
"Fetch Instruction (PC Address)"
Instruction Fetch
Instruction Memory
"Control Signals (OPCode)"
"Send Instruction (OPCode, Addresses)"
Decode/Control
Control Unit
Register File
Control Signals Include:
- RegDst, Jump, Branch
- MemRead, MemtoReg, ALUSrc
- RegWrite, MemWrite, Bne
"Zero Flag for Branch Operations"
"Determine Operation (ALUOp)"
Execution
ALU Control
"Data for Operations (Read Register Data)"
"Execute ALU Operation (ALU Control)"
ALU
Supports operations:
- ADD, ADDI, SUB, SUBU
- AND, ANDI, OR, ORI
- NOR, SLT, BEQ, BNE
- LW, SW, J
"Write Back Da
"Address / Data to Data Memory"
Memory Access
Data Memory
"Data to Register File"
Write Back
Update Registers

Execution diagram for each instruction:

Each instruction type (R-type, I-type, J-type) generally follows a similar flow with variations primarily in the Execute and Memory Access stages depending on whether the instruction involves arithmetic, memory access, or control flow.

This model provides a consistent framework for understanding how different instructions are processed in the single-cycle MIPS architecture.

Single-Cycle MIPS Execution Diagram



ADD (R-type instruction)

Name (format, op, function): `add (R,0,32)`

Syntax: `add rd,rs,rt`

Operation: `reg(rd) := reg(rs) + reg(rt);`

Instruction Overview:

The following is an overview of the operation of the `add` instruction in a MIPS processor, focusing on the key stages of the processor pipeline.

- **IF:** The instruction is fetched from memory using the program counter (PC).
- **ID:** The instruction bits are decoded to determine it is an ADD operation. Registers specified by the source register fields (`rs` and `rt`) are read.
- **EX:** The ALU performs the addition of the two register values.
- **MEM:** No action (not used by ADD).
- **WB:** The result from the ALU is written back to the destination register (`rd`).

Operation Breakdown:

The following provides a detailed breakdown of the operation of the `add` instruction in a MIPS processor, focusing on the key stages of the processor pipeline.

Stages of the `add` Instruction:

1. **Instruction Fetch (IF):**

   - The instruction is fetched from memory using the Program Counter (PC).

   - This stage corresponds to reading the instruction code from the instruction memory. The address comes from the PC which points to the location of the next instruction to execute.

     ```
     i_Instruction = Imem[i_Addr>>2];
     ```

2. **Instruction Decode (ID):**

   - The fetched instruction is decoded to determine it is a `add` operation.

   - The opcode part of the instruction (which is `000000` for R-type instructions) is identified, and the source register identifiers (`rs` and `rt`) are used to read the respective registers.

     ```
     rs = i_instruction[25:21];
     rt = i_instruction[20:16];
     ```
3. **Execute (EX):**
   - The ALU (Arithmetic Logic Unit) performs the addition of the values in the source registers (`rs` and `rt`).
   - The values from these registers are fed into the ALU where the addition is performed based on the control signal (`ALUOp`) from the control unit.

     ```
     o_ALUresult = i_data1 + i_data2;
     ```
4. **Memory Access (MEM):**
   - For the `add` instruction, this stage is not utilized as no memory access is required (i.e., no data is read from or written to the memory).

     ```
     // No memory operation required for 'add'
     ```
5. **Write Back (WB):**
   - The result from the ALU is written back into the destination register (`rd`).
   - This is where the output of the ALU operation is stored back into the register file, specifically into the register indicated by the `rd` field of the instruction.

     ```
     Reg[rd] = o_ALUresult;
     ```

Example Code Snippet:

Here is a simplified Verilog snippet that captures the essence of the `add` instruction's operation in a MIPS processor, focusing on the key stages of the mips architecture (IF, ID, EX, MEM, WB).

```
module MIPS_Processor(input clk, input reset, ...);
    // Registers and other declarations here
    reg [31:0] PC, ALUResult, Reg[31:0];
    reg [31:0] InstructionRegister, ReadData1, ReadData2;
    integer rd, rs, rt;
    always @(posedge clk) begin
        if (reset) begin
            PC <= 0;  // Reset PC
        end else begin
            // Fetch Instruction
            InstructionRegister <= Imem[PC>>2];
            PC <= PC + 4;
            // Decode Instruction
            rs = InstructionRegister[25:21]; // Extract source register indices
            rt = InstructionRegister[20:16]; // Extract target register indices
            rd = InstructionRegister[15:11]; // Extract destination register index
            ReadData1 <= Reg[rs];
            ReadData2 <= Reg[rt];
            // Execute
            ALUResult <= ReadData1 + ReadData2;
            // Write Back
            Reg[rd] <= ALUResult;
        end
    end
endmodule
```

The `add` instruction demonstrates the typical use of the R-type format in MIPS instruction set architecture, involving fetching the instruction, decoding it, executing the operation in the ALU, skipping memory access, and finally writing back the result to the register file.

ADDI (I-type instruction)

Name (format, op, function): `add immediate (I,8,na)`

Syntax: `addi rt,rs,imm`

Operation: `reg(rt) := reg(rs) + signext(imm);`

Instruction Overview:

- **IF:** Fetch the instruction from memory.
- **ID:** Decode the instruction; read the source register (`rs`).
- **EX:** ALU adds the value in the source register to the immediate value (which is sign-extended).
- **MEM:** No action.
- **WB:** The result is written back to the target register (`rt`).

Operation Breakdown:

The following further breaks down the operation of the `ADDI` instruction in a MIPS processor across the various stages of the processor pipeline.

Stages of the `ADDI` Instruction

1. **Instruction Fetch (IF):**
   - The processor retrieves the `ADDI` instruction from memory based on the current Program Counter (PC) value.
   - The instruction is then forwarded to the next stage for decoding.
2. **Instruction Decode (ID):**
   - The instruction is decoded to identify that it is a `ADDI` operation.
   - The source register (`rs`) is read to obtain its value. The immediate value (`imm`) is also extracted from the instruction during this phase.
3. **Execute (EX):**
   - The Arithmetic Logic Unit (ALU) performs the addition operation. It adds the value retrieved from the source register (`reg(rs)`) to the sign-extended immediate value (`signext(imm)`).
   - This computation involves extending the immediate value to match the register size (typically 32 bits in MIPS), preserving its sign to handle negative numbers correctly.
4. **Memory Access (MEM):**
   - The `ADDI` instruction does not involve any memory access, so this stage is effectively a no-op (no operation) for this instruction.
5. **Write Back (WB):**
   - The result of the addition from the ALU is written back to the destination register (`reg(rt)`).
   - This step updates the target register with the computed value, completing the execution of the instruction.

Explanation of the Code Implementation

The operation of `ADDI` can be modeled in a simulated or actual MIPS processor using the following Verilog-like pseudocode:

```
module addi_instruction(rs, rt, imm, output rt_value);
    input [4:0] rs, rt;       // Source and target register indices (5 bits each)
    input [15:0] imm;         // 16-bit immediate value
    output [31:0] rt_value;   // Output to target register
    wire [31:0] rs_value;     // Value from source register
    wire [31:0] extended_imm; // Sign-extended immediate value
    assign extended_imm = {
      {16{imm[15]}}, imm      // Sign-extend the immediate value
    };
    assign rt_value = rs_value + extended_imm;
endmodule
```

- `rs` and `rt` are inputs representing the source and destination register indices.
- `imm` is the 16-bit immediate value input.
- The immediate value is sign-extended to 32 bits using Verilog's bit replication and concatenation (`{{16{imm[15]}}, imm}`), where `imm[15]` is the most significant bit (MSB) of the immediate value, replicated 16 times to fill the upper half of a 32-bit word.
- The sum of the sign-extended immediate and the source register value is computed and assigned to `rt_value`, which would be written back to the register file in the actual processor hardware.

LW (Load Word)

Name (format, op, function): `load word (I,35,na)`

Syntax: `lw rt,imm(rs)`

Operation: `reg(rt) := mem[reg(rs) + signext(imm)];`

Instruction Overview:

- **IF:** Fetch the instruction.
- **ID:** Decode the instruction; read the base address register (`rs`).
- **EX:** Calculate the memory address by adding the immediate value (offset) to the base register.
- **MEM:** Access the memory at the computed address and read the word.
- **WB:** Write the loaded word into the target register (`rt`).

Operation Breakdown:

The following provides a detailed breakdown of the operation of the `lw` instruction in a MIPS processor, focusing on the key stages of the processor pipeline.

Breakdown of `lw` Instruction Execution

1. **Instruction Fetch (IF)**:
   - The processor fetches the `lw` instruction from instruction memory using the program counter (PC).
   - Code snippet showing fetching the instruction from memory:

   ```
   i_Instruction = Imem[i_Addr>>2];
   ```

2. **Instruction Decode (ID)**:
   - The fetched instruction is decoded to extract the opcode, source register (`rs`), target register (`rt`), and the immediate value.
   - The base address (content of `rs`) is read from the register file during this phase.
   - Code snippet showing the decoding and reading of the base address:

   ```
   read_data1 = RegData[i_rs];  // Assume i_rs is the source register index
   ```

3. **Execute (EX)**:
   - The effective memory address is calculated by adding the sign-extended immediate value to the base address read from `rs`.
   - This calculation typically happens in the ALU.
   - Code snippet that could represent the address calculation in ALU (not specifically shown in your snippets):

   ```
   address = read_data1 + sign_extend(imm);  // Conceptual code
   ```

4. **Memory Access (MEM)**:

- The processor accesses the memory location computed in the Execute stage.
- The word at this memory address is read.
- Code snippet showing memory access to read data:

```
if (i_MemRead == 1) {
  o_rData = Dmem[i_addr];  // Read memory at calculated address
}
```

5. **Write Back (WB)**:
   - The data retrieved from memory is written into the target register (`rt`).
   - Code snippet showing the write-back to the register:

```
RegData[i_rt] <= i_wData;  // Assume i_rt is the target register index and i_wData is data
      read from memory
```

SW (Store Word)

Name (format, op, function): store word (I,43,na)

Syntax: `sw rt,imm(rs)`

Operation: `mem[reg(rs) + signext(imm)] := reg(rt);`

Instruction Overview:

- **IF:** Fetch the instruction.
- **ID:** Decode the instruction; read the base address register (`rs`) and the register to be stored (`rt`).
- **EX:** Calculate the memory address by adding the immediate value (offset) to the base register.
- **MEM:** Write the value from `rt` into the calculated memory address.
- **WB:** No write-back step for store instructions.

Operation Breakdown:

The `SW` instruction in the MIPS architecture is used to store a 32-bit word from a register into memory. Here's an in-depth breakdown of how the `SW` instruction is executed across the various stages in a MIPS processor.

Instruction Stages:

1. **IF (Instruction Fetch):**
   - The instruction is fetched from the instruction memory using the current Program Counter (PC).
2. **ID (Instruction Decode):**
   - The instruction is decoded to identify it as a `SW` instruction.
   - The base address register (`rs`) and the register containing data to be stored (`rt`) are identified and read.
3. **EX (Execute):**
   - The effective memory address is calculated by adding the sign-extended immediate (offset) to the value in the base register (`rs`).
4. **MEM (Memory Access):**
   - The data in register `rt` is written to the calculated memory address.
5. **WB (Write Back):**
   - No write-back is performed for the `SW` instruction, as this instruction does not modify any register contents.

Verilog Implementation:

Data Memory Module (`DataMemory.v`):

Below is a simplified Verilog module for a data memory component that can be used to store and retrieve data in a MIPS processor. This module includes logic for both read and write operations.

```
module DataMemory (
    input clk,
    input memWrite,
    input [31:0] address,
    input [31:0] writeData,
    output reg [31:0] readData
);
    reg [31:0] memory [0:1023];
    always @(posedge clk) begin
        if (memWrite) begin
            memory[address >> 2] <= writeData;  // Write operation
        end else begin
            readData <= memory[address >> 2];   // Read operation
        end
    end
endmodule
```

Processor Control Logic (`ProcessorControl.v`):

Below is an extract from the ProcessorControl module that controls the behavior of the processor based on the opcode of the instruction being executed. This snippet shows how the control signals are set for the `SW` instruction.

```
module ProcessorControl (
    input [5:0] opcode,
    output reg memWrite,
    output reg aluSrc,
    output reg regDst,
    output reg memToReg,
    output reg regWrite
);
    always @(*) begin
```

```
        case (opcode)
            6'b101011: begin  // Opcode for SW
                memWrite = 1'b1;
                aluSrc = 1'b1;
                regDst = 1'b0;
                memToReg = 1'b0;
                regWrite = 1'b0;
            end
            default: begin
                memWrite = 1'b0;
                aluSrc = 1'b0;
                regDst = 1'b0;
                memToReg = 1'b0;
                regWrite = 1'b0;
            end
        endcase
    end
endmodule
```

Simplified Top-Level MIPS Module:

```
module MIPSProcessor (
    input clk,
    input reset,
    output [31:0] pc,
    input [31:0] instruction,
    output [31:0] aluResult,
    output [31:0] writeData,
    output [31:0] readData
);
    wire [5:0] opcode = instruction[31:26];
    wire [4:0] rs = instruction[25:21];
    wire [4:0] rt = instruction[20:16];
    wire [15:0] imm = instruction[15:0];
    wire [31:0] signExtImm = {{16{imm[15]}}, imm};
    wire [31:0] regDataRs, regDataRt;
    wire memWrite, aluSrc, regDst, memToReg, regWrite;
    // Instantiate control logic
    ProcessorControl control(opcode, memWrite, aluSrc, regDst, memToReg, regWrite);
    // ALU operation (assuming already instantiated and connected)
    // Data memory operation
    DataMemory dataMem(clk, memWrite, aluResult, regDataRt, readData);
    // Register file operations and other connections would be defined here
endmodule
```

BEQ (Branch if Equal)

Name (format, op, function): `branch on equal (I,4,na)`

Syntax: `beq rs,rt,label`

Operation: if reg(rs) == reg(rt) then PC = BTA else NOP;

Instruction Overview:

- **IF:** Fetch the instruction.
- **ID:** Decode the instruction; read the two registers (`rs` and `rt`) and compare them.
- **EX:** Calculate the branch target address if the comparison is equal (by adding the sign-extended, shifted immediate to the PC).
- **MEM:** No memory access.
- **WB:** No write-back; update the PC to the branch address if the condition is met, otherwise increment the PC as usual.

Operation Breakdown:

The following provides a detailed breakdown of the operation of the `BEQ` instruction in a MIPS processor, focusing on the key stages of the processor pipeline.

The `BEQ` (Branch if Equal) instruction in the MIPS architecture follows a specific flow through the processor stages. Here’s a step-by-step walkthrough of each stage using Verilog code examples to illustrate how each part of the instruction’s lifecycle is handled in hardware.

Instruction Stages for `BEQ`

1. **IF (Instruction Fetch) Stage**:
    - The instruction is fetched from the instruction memory using the current Program Counter (PC).
    - The PC is incremented to point to the next instruction (PC = PC + 4).

```
module InstructionFetch(
    input clk,
    input reset,
    input [31:0] next_pc,
    output reg [31:0] instr,
    output reg [31:0] pc
);
```

```
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            pc <= 32'h00000000; // Reset PC to start
        end else begin
            pc <= next_pc; // Update PC to next PC
            instr <= instruction_memory[pc >> 2]; // Fetch instruction from memory
        end
    end
endmodule
```

2. **ID (Instruction Decode) Stage**:
   - Decode the fetched instruction to identify it as `BEQ`.
   - Read the two source registers (`rs` and `rt`) based on the instruction fields.
   - Set up the control signals for the ALU to perform a subtraction (`rs - rt`).

```
module InstructionDecode(
    input [31:0] instr,
    output reg [4:0] rs,
    output reg [4:0] rt,
    output reg [15:0] immediate
);
    always @(*) begin
        rs = instr[25:21];
        rt = instr[20:16];
        immediate = instr[15:0]; // For branch offset
    end
endmodule
```

3. **EX (Execute) Stage**:
   - Compute the target address for branching by sign-extending the immediate field and shifting left by 2 bits (since it's word-aligned), then adding this to the PC + 4 (already incremented PC from IF stage).
   - ALU checks if `rs` and `rt` are equal by subtracting and checking if the result is zero.

```
module ALU(
    input [31:0] rs_val,
    input [31:0] rt_val,
    input [31:0] sign_ext_imm,
    input [2:0] alu_control,
    output reg zero,
    output reg [31:0] alu_result
);
    wire [31:0] branch_target = (sign_ext_imm << 2) + pc_plus_4;
    always @(*) begin
        case(alu_control)
            3'b010: begin // Subtract for BEQ
                alu_result = rs_val - rt_val;
                zero = (alu_result == 0) ? 1'b1 : 1'b0;
            end
        endcase
    end
endmodule
```

4. **MEM (Memory Access) Stage**:
   - For `BEQ`, there is no memory access or data memory operation.
5. **WB (Write-Back) Stage**:
   - Update the PC to the branch target if `rs` == `rt` (if zero flag from ALU is true).
   - If `rs` != `rt`, increment the PC to the next instruction (already done in IF).

```
always @(posedge clk) begin
    if (branch_taken) begin
        pc <= branch_target; // Update PC if branch is taken
    end
endmodule
```

Example Verilog for Complete BEQ Control

Here's how a simpler control unit might orchestrate these stages just for the `BEQ` instruction in a MIPS processor:

```
module ControlUnit(
    input [5:0] opcode,
    output reg branch,
    output reg alu_src,
    output reg [2:0] alu_control
);
    always @(*) begin
        case(opcode)
            6'b000100: begin // BEQ
                branch = 1'b1;
                alu_src = 1'b0; // Use rs, rt directly
                alu_control = 3'b010; // Set ALU to subtract
            end
            default: begin
                branch = 1'b0;
                alu_src = 1'b0;
                alu_control = 3'b000;
            end
        endcase
    end
endmodule
```

J (Jump)

Name (format, op, function): `jump (J,2,na)`

Syntax: `j`

Operation: `PC := JTA;`

Instruction Overview:

- **IF:** Fetch the instruction.
- **ID:** Decode the instruction.
- **EX:** Calculate the jump target address from the address field of the instruction.
- **MEM:** No memory access.
- **WB:** Update the PC to the jump address.

Operation Breakdown:

The following provides a detailed breakdown of the operation of the `J` instruction in a MIPS processor, focusing on the key stages of the processor pipeline. The J instruction in MIPS is a jump instruction that allows the program to continue execution from a specified address. It is used to alter the flow of control unconditionally.

Instruction Format and Operation:

- **Name (format, op, function):** `jump (J,2,na)`
- **Syntax:** `j target`
- **Operation:** `PC := JTA;` where JTA (Jump Target Address) is calculated from the instruction itself.

Stages of J Instruction Execution

Here's a breakdown of how the J instruction progresses through each stage of the MIPS pipeline:

1. **IF (Instruction Fetch):**
   - The instruction is fetched from the instruction memory at the current program counter (PC) address.
   - The PC is then incremented by 4 to point to the next sequential instruction (though this increment will be overridden by the jump).
2. **ID (Instruction Decode):**
   - The opcode of the instruction is decoded to identify it as a jump instruction.
   - No registers are read in this stage because the jump instruction does not involve any registers.
3. **EX (Execute):**
   - The jump target address (JTA) is calculated from the address field of the instruction.
   - JTA is formed by taking the upper 4 bits of the PC (from the incremented value that points to the next instruction) and concatenating them with the 26-bit address field from the instruction, shifted left by 2 bits (to word-align the address).
4. **MEM (Memory Access):**
   - There is no memory access for the jump instruction.
5. **WB (Write Back):**
   - The PC is updated to the new address calculated in the Execute stage. This is the jump target address where the program will continue executing.

Verilog Module for Program Counter with just Jump:

```
module ProgramCounter(
    input clk,
    input reset,
    input [31:0] jump_address,  // Jump target address input
    input jump,                // Control signal to indicate a jump
    output reg [31:0] pc       // Program counter output
);
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            pc <= 32'b0;  // Reset the PC to 0 on reset
        end else if (jump) begin
            pc <= jump_address;  // Update PC to the jump address if jump is asserted
        end else begin
            pc <= pc + 4;  // Increment PC by 4 on each clock cycle otherwise
        end
    end
endmodule
```

Verilog Module for Jump Address Calculation:

While in this project, this function is done by `NextProgramCounter` module, here is a simplified version of a module that calculates the jump address in a MIPS processor to further illustrate the concept of jump address calculation:

```
module JumpAddressCalculator(
    input [25:0] address_field,  // Address field from the jump instruction
    input [31:0] pc_plus_4,      // PC + 4 (the incremented PC pointing to the next instruction)
    output [31:0] jump_address   // Calculated jump target address
);
```

```
    assign jump_address = {pc_plus_4[31:28], address_field << 2};
endmodule
```

- The `ProgramCounter` module handles updating the PC based on whether a jump is taken. If a jump is taken, it sets the PC to the jump address; otherwise, it simply increments the PC.
- The `JumpAddressCalculator` module calculates the full 32-bit jump address by concatenating the upper 4 bits of the incremented PC (PC+4) with the left-shifted 26-bit address from the jump instruction.

These modules collectively illustrate how the J instruction's effect on the program counter can be implemented in hardware using Verilog.

ADDU (Add Unsigned)

name (format, op, function): `add unsigned (R,0,33)`

Syntax: `addu rd,rs,rt`

Operation: `reg(rd) := reg(rs) + reg(rt);`

Instruction Overview:

- **IF (Instruction Fetch):** The instruction is fetched from memory using the program counter (PC).
- **ID (Instruction Decode):** The opcode is decoded; registers rs and rt are read.
- **EX (Execute):** The arithmetic logic unit (ALU) adds the values from registers rs and rt.
- **MEM (Memory Access):** No action needed (pass-through).
- **WB (Write Back):** The result from the ALU is written back to the destination register rd.

Operation Breakdown:

The following provides a detailed breakdown of the operation of the `ADDU` instruction in a MIPS processor, focusing on the key stages of the processor pipeline.

The `ADDU` instruction in MIPS is an unsigned addition operation that does not raise exceptions on overflow. Here's a detailed breakdown of how the `ADDU` instruction is executed across the various stages of the processor pipeline.

IF (Instruction Fetch)

In this stage, the instruction is fetched from the instruction memory based on the current value of the Program Counter (PC). Here's how you might see this operation in Verilog:

```
// Instruction Fetch module
module InstructionFetch(
    input [31:0] i_PC,            // Program Counter
    output [31:0] o_Instruction   // Fetched instruction
);
    reg [31:0] instruction_memory[255:0]; // Memory array

    // Fetch the instruction
    assign o_Instruction = instruction_memory[i_PC >> 2]; // Word aligned access
endmodule
```

ID (Instruction Decode)

During this stage, the opcode of the fetched instruction is decoded, and the register file is accessed to read the contents of registers `rs` and `rt`.

```
// Instruction Decode module
module InstructionDecode(
    input [31:0] i_Instruction,   // Input from IF stage
    output [4:0] o_rs, o_rt, o_rd // Register specifiers
);
    // Decode the instruction
    assign o_rs = i_Instruction[25:21];
    assign o_rt = i_Instruction[20:16];
    assign o_rd = i_Instruction[15:11];
endmodule
```

EX (Execute)

The ALU adds the values from registers `rs` and `rt`. Here's a snippet of the ALU performing this addition:

```
// Arithmetic Logic Unit (ALU) module
module ALU(
    input [31:0] i_data1, i_data2,    // Data from registers rs and rt
    input [3:0] i_ALUControl,         // Control signals
    output reg [31:0] o_result        // Result of the ALU operation
);
    always @(i_data1, i_data2, i_ALUControl) begin
        case (i_ALUControl)
            4'b0010: o_result = i_data1 + i_data2; // ADDU operation
            // Other ALU operations...
        endcase
    end
endmodule
```

MEM (Memory Access)

This stage is a pass-through for the ADDU instruction since it does not involve memory access.

```
// Memory Access Stage - No action needed for ADDU
module MemoryAccess(
    input [31:0] i_ALUResult,
    output [31:0] o_MemOut
);
    assign o_MemOut = i_ALUResult;  // Direct pass-through
endmodule
```

WB (Write Back)

The result from the ALU is written back to the destination register `rd`.

```
// Write Back stage
module WriteBack(
    input [31:0] i_ALUResult,       // Result from ALU
    input [4:0] i_rd,               // Destination register
    output reg [31:0] o_WriteData   // Data to write back
);
    // Write the data back to the register file
    always @(i_ALUResult) begin
        o_WriteData = i_ALUResult;
    end
endmodule
```

SUB (Subtract)

Name (format, op, function): `subtract (R,0,34)`

Syntax: `sub rd,rs,rt`

Operation: `reg(rd) := reg(rs) [ reg(rt);`

Instruction Overview:

- **IF:** Fetch the instruction using the PC.
- **ID:** Decode the instruction; read registers rs and rt.
- **EX:** The ALU subtracts the value in rt from rs.
- **MEM:** No action needed (pass-through).
- **WB:** The ALU result is written back to register rd.

Operation Breakdown:

The following provides a detailed breakdown of the operation of the `SUB` instruction in a MIPS processor, focusing on the key stages of the processor pipeline.

1. **Instruction Fetch (IF) Stage:** In this stage, the processor fetches the instruction from instruction memory using the Program Counter (PC).

```
// Instruction Fetch (IF) stage
module InstructionFetch(
    input [31:0] i_pc,
    output reg [31:0] o_instruction
);
    // Assume IMem is an array storing instructions
    reg [31:0] IMem[0:1023];

    // Fetch instruction
    always @(i_pc) begin
        o_instruction = IMem[i_pc >> 2]; // Word aligned fetch
    end
endmodule
```

2. **Instruction Decode (ID) Stage:** Here, the instruction is decoded, and the relevant registers are read. The operation is identified, and signals are prepared for the execution stage.

```
// Instruction Decode (ID) stage
module InstructionDecode(
    input [31:0] i_instruction,
    output reg [4:0] o_rs, o_rt, o_rd,
    output reg [5:0] o_opcode, o_funct
);
    always @(i_instruction) begin
        o_opcode = i_instruction[31:26];
        o_rs = i_instruction[25:21];
        o_rt = i_instruction[20:16];
        o_rd = i_instruction[15:11];
        o_funct = i_instruction[5:0];
    end
endmodule
```

3. **Execution (EX) Stage:** The ALU performs the subtraction based on the decoded instruction. The operands are taken from the registers identified in the ID stage.

```verilog
// Execution (EX) stage - ALU for SUB operation
module ALU(
    input [31:0] i_data1, i_data2,
    input [3:0] i_ALUcontrol,
    output reg [31:0] o_result
);
    always @(*) begin
        case(i_ALUcontrol)
            4'b0110: o_result = i_data1 - i_data2; // SUB operation
            // Additional cases for other ALU operations
        endcase
    end
endmodule
```

4. **Memory (MEM) Stage:** For the `SUB` instruction, there is no memory operation needed. This stage can be passed through or handled with a control signal that disables memory operations.

```verilog
// Memory (MEM) stage pass-through for SUB
module MemoryStage(
    input i_MemRead, i_MemWrite,
    input [31:0] i_address, i_writeData,
    output reg [31:0] o_readData
);
    // Memory array
    reg [31:0] DMem[0:1023];

    always @(*) begin
        if (i_MemWrite) DMem[i_address >> 2] = i_writeData;
        if (i_MemRead) o_readData = DMem[i_address >> 2];
    end
endmodule
```

5. **Write Back (WB) Stage:** The result of the ALU operation is written back to the register file, particularly in the register specified by `rd`.

```verilog
// Write Back (WB) Stage
module WriteBack(
    input [31:0] i_ALUresult,
    input [4:0] i_rd,
    input i_RegWrite,
    output reg [31:0] o_writeData
);
    always @(i_ALUresult) begin
        if (i_RegWrite) begin
            o_writeData = i_ALUresult;
        end
    end
endmodule
```

SUBU (Subtract Unsigned)

Name (format, op, function): `subtract unsigned (R,0,35)`

Syntax: `subu rd,rs,rt`

Operation: `reg(rd) := reg(rs) [ reg(rt);`

Instruction Overview:

- **IF:** Fetch the instruction using the PC.
- **ID:** Decode the instruction; read registers rs and rt.
- **EX:** The ALU subtracts the value in rt from rs.
- **MEM:** No action needed (pass-through).
- **WB:** The ALU result is written back to register rd.

Operation Breakdown:

1. **Instruction Fetch (IF)**
   - The instruction is fetched from the instruction memory using the Program Counter (PC).
   - Verilog snippet:

   ```verilog
   // IF Stage
   always @(posedge clk) begin
       if (reset)
           pc <= 0;
       else if (pc_src)
           pc <= pc_next;
       else
           pc <= pc + 4;
   end
   ```

2. **Instruction Decode (ID)**

- The fetched instruction is decoded to identify the operation as `SUBU` and the source (`rs`, `rt`) and destination (`rd`) registers are identified.
- Verilog snippet:

```
// ID Stage
reg [31:0] instruction;
wire [4:0] rs, rt, rd;
assign rs = instruction[25:21];
assign rt = instruction[20:16];
assign rd = instruction[15:11];
```

3. **Execution (EX)**
   - The actual subtraction of the contents of the registers `rs` and `rt` is performed. The result does not account for overflow because it is unsigned.
   - Verilog snippet:

```
// EX Stage
reg [31:0] reg_data[31:0];  // Register file
wire [31:0] rs_value, rt_value, result;
assign rs_value = reg_data[rs];
assign rt_value = reg_data[rt];
assign result = rs_value - rt_value;
```

4. **Memory Access (MEM)**
   - `SUBU` does not require a memory operation, so this stage can be considered a pass-through.
   - No action required for SUBU
5. **Write Back (WB)**
   - The result of the subtraction is written back to the destination register `rd`.
   - Verilog snippet:

```
// WB Stage
always @(posedge clk) begin
    if (reg_write)
        reg_data[rd] <= result;
end
```

```
module MIPS_Processor(input clk, input reset, output [31:0] pc);
    reg [31:0] pc, next_pc;
    reg [31:0] reg_file[31:0];  // Register file

    // Instruction Fetch
    always @(posedge clk) begin
        if (reset)
            pc <= 0;
        else
            pc <= next_pc;
    end

    // Instruction Decode
    reg [31:0] instruction;
    wire [4:0] rs, rt, rd;
    wire [5:0] opcode;
    assign opcode = instruction[31:26];
    assign rs = instruction[25:21];
    assign rt = instruction[20:16];
    assign rd = instruction[15:11];
    // Execute
    wire [31:0] rs_value, rt_value, alu_result;
    assign rs_value = reg_file[rs];
    assign rt_value = reg_file[rt];
    assign alu_result = (opcode == 6'b000000) ? (rs_value - rt_value) : 32'b0;  // SUBU Opcode
        assumed
    // Memory Access
    // No memory access for SUBU
    // Write Back
    always @(posedge clk) begin
        if (opcode == 6'b100011)  // SUBU Opcode
            reg_file[rd] <= alu_result;
    end
    // Program Counter Update
    always @(*) begin
        next_pc = pc + 4;  // Simple sequential execution
    end
endmodule
```

In this example, `opcode == 6'b100011'` is the actual opcode for `SUBU`.

AND (Bitwise AND)

- **IF:** Instruction is fetched.
- **ID:** Instruction is decoded. For AND, rs and rt are read;
- **EX:** The ALU performs an AND operation between operands.
- **MEM:** No action needed.
- **WB:** Result is written back to rd (AND)

Instruction Breakdown

The AND instruction performs a bitwise AND operation between two operands and stores the result in a destination register.

1. **Instruction Fetch (IF) stage:**
   - The Program Counter (PC) contains the address of the AND instruction.
   - The Instruction Memory module (`InstructionMemory.v`) fetches the instruction from the memory location pointed to by the PC.
   - The fetched instruction is passed to the next stage.
2. **Instruction Decode (ID) stage:**
   - The Control Unit module (`ControlUnit.v`) decodes the opcode of the AND instruction.
   - Based on the opcode, the Control Unit generates the appropriate control signals for the data-path components.
   - The Register File module reads the values of the source registers specified in the AND instruction.
3. **Execution (EX) stage:**
   - The ALU module (`ALU.v`) performs the bitwise AND operation between the values of the source registers.
   - The ALU control signal generated by the Control Unit determines the specific operation to be performed (AND in this case).
4. **Memory Access (MEM) stage:**
   - The AND instruction does not require any memory access, so this stage is a pass-through.
5. **Write Back (WB) stage:**
   - The result of the AND operation from the ALU is written back to the destination register specified in the AND instruction.
   - The RegWrite control signal generated by the Control Unit enables the writing of the result to the Register File.

Within the control unit, the AND instruction is identified by its opcode, and the appropriate control signals are set to execute the AND operation.

The ALU module performs the bitwise AND operation between the source register values, and the result is written back to the destination register.

Here’s an example of how the AND instruction flows through the different stages of the single-cycle MIPS processor starting within the control unit.

```
// Control Unit
always @(i_instruction) begin
  case (i_instruction[31:26])
    // ...
    6'b001100: begin   // andi
      o_RegDst = 0;     // Destination register is rt
      o_ALUSrc = 1;     // Second operand is immediate value
      o_MemtoReg = 0;  // ALU result is written to register
      o_RegWrite = 1;  // Write to register file
      o_MemRead = 0;   // No memory read
      o_MemWrite = 0;  // No memory write
      o_Branch = 0;     // No branch
      o_Bne = 0;        // No branch if not equal
      o_ALUOp = 2'b11; // ALU operation is AND
      o_Jump = 0;       // No jump
      // ...
    end
    // ...
  endcase
end

// ALU
always @(i_data1, data2, i_ALUcontrol) begin
  case (i_ALUcontrol)
    // ...
    4'b0000:  // AND
      o_ALUresult = i_data1 & data2;
    // ...
  endcase
  // ...
end
```

In the Control Unit module, when the opcode of the instruction matches the AND opcode (`6'b001100` in this case), the appropriate control signals are set.

The `ALUSrc` signal is set to 1 to select the immediate value as the second operand, and the `ALUOp` signal is set to indicate an AND operation.

In the ALU module, when the `ALUcontrol` signal matches the AND operation (4’b0000), the bitwise AND operation is performed between the two input operands (i_data1 and data2), and the result is assigned to `o_ALUresult`.

ANDI (AND Immediate)

`and immediate (I,12,na)`

`andi rt,rs,imm`

`reg(rt) := reg(rs) & zeroext(imm);`

Instruction Overview:

- **IF:** Instruction is fetched.
- **ID:** Opcode decoded. Registers rs and immediate for ANDI are read.
- **EX:** ALU performs an AND operation.
- **MEM:** No memory access.
- **WB:** Result written to or rt (ANDI).

Instruction Breakdown

The ANDI (AND Immediate) instruction performs a bitwise AND operation between a register value and an immediate value.

Here’s an explanation of how the ANDI instruction goes through each stage of the MIPS processor pipeline:

1. **Instruction Fetch (IF):**
   - The Program Counter (PC) contains the address of the ANDI instruction in the Instruction Memory.
   - The instruction is fetched from the Instruction Memory using the PC value.
   - Example code in the Instruction Memory module (`InstructionMemory.v`):

   ```
   always @(i_Addr) begin
     if (i_Addr == -4) begin         // init
       i_Instruction = 32'b11111100000000000000000000000000;
     end else begin
       i_Instruction = Imem[i_Addr>>2];
     end
     i_Ctr = i_Instruction[31:26];
     i_Funcode = i_Instruction[5:0];
   end
   ```

2. **Instruction Decode (ID):**
   - The fetched instruction is decoded to determine the operation to be performed.
   - The Control Unit generates the necessary control signals based on the opcode and function code of the instruction.
   - The register to be read (rs) is determined from the instruction, and the immediate value is sign-extended.
   - Example code in the Control Unit module (`ControlUnit.v`):

   ```
   6'b001100: begin  // andi
     o_RegDst = 0;
     o_ALUSrc = 1;
     o_MemtoReg = 0;
     o_RegWrite = 1;
     o_MemRead = 0;
     o_MemWrite = 0;
     o_Branch = 0;
     o_Bne = 0;
     o_ALUOp = 2'b11;
     o_Jump = 0;
     // ...
   end
   ```

3. **Execute (EX):**
   - The ALU performs the bitwise AND operation between the value of register rs and the sign-extended immediate value.
   - The result of the AND operation is stored in a temporary register.
   - Example code in the ALU module (`ALU.v`):

   ```
   always @(i_data1, data2, i_ALUcontrol) begin
     case (i_ALUcontrol)
       // ...
       4'b0000:  // AND
         o_ALUresult = i_data1 & data2; // bitwise AND
       // ...
     endcase
     // ...
   end
   ```

4. **Memory Access (MEM):**
   - The ANDI instruction does not involve memory access, so no action is needed in this stage.
   - The result from the Execute stage is simply passed through to the next stage.
5. **Write Back (WB):**
   - The result of the AND operation, stored in the temporary register, is written back to the destination register (rt) in the Register File.
   - Example code in the Register File module (`RegisterFile.v`):

   ```
   always @(posedge i_Clk or posedge i_Rst) begin
     if (i_Rst) begin
       // Reset all registers to zero
       for (j = 0; j < 32; j = j + 1) begin
         RegData[j] = 32'b0;
       end
     end else if (i_RegWrite) begin
       // Write data to the specified register
       RegData[i_wReg] = i_wData;
     end
   end
   ```

Here’s an example of how the ANDI instruction would look in machine code:

```
001100 01000 01010 0000000000001111
```

In this example: - `001100` is the opcode for the ANDI instruction. - `01000` represents the source register (rs), which is $8 in this case. - `01010` represents the destination register (rt), which is $10 in this case. - `0000000000001111` is the immediate value, which is 15 in decimal.

OR

- **IF:** Instruction is fetched.
- **ID:** Opcode decoded. Registers rs and rt are read for OR; rs and immediate for ORI.
- **EX:** ALU performs an OR operation.
- **MEM:** No action needed.
- **WB:** Result written to rd (OR) or rt (ORI).

Instruction Breakdown

The `OR` instruction in the MIPS architecture performs a bitwise OR operation on two register values and stores the result in a destination register.

1. **Instruction Fetch (IF) Stage:**

   - The Program Counter (PC) holds the address of the `OR` instruction to be fetched.
   - The Instruction Memory module (`InstructionMemory.v`) retrieves the instruction from the memory based on the PC value.
   - The fetched instruction is passed to the next stage.

   Example code from `InstructionMemory.v`: `verilog always @(i_Addr) begin   if (i_Addr == -4) begin i_Instruction = 32'b11111100000000000000000000000000;   end else begin     i_Instruction = Imem[i_Addr>>2];   end   i_Ctr = i_Instruction[31:26];   i_Funcode = i_Instruction[5:0]; end`

2. **Instruction Decode (ID) Stage:**

- The fetched instruction is decoded to identify the opcode and register operands.
- The Control Unit module (`ControlUnit.v`) sets the appropriate control signals based on the opcode.
- For the `OR` instruction, the `ALUOp` control signal is set to indicate an OR operation.
- The register operands (`rs` and `rt`) are read from the Register File.

Example code from `ControlUnit.v`:

```verilog
always @(i_instruction) begin
  case (i_instruction[31:26])
    // ...
    6'b000000: begin  // ARITHMETIC
      o_RegDst = 1;
      o_ALUSrc = 0;
      o_MemtoReg = 0;
      o_RegWrite = 1;
      o_MemRead = 0;
      o_MemWrite = 0;
      o_Branch = 0;
      o_Bne = 0;
      o_ALUOp = 2'b10;
      o_Jump = 0;
      // ...
    end
    // ...
  endcase
end
```

3. **Execute (EX) Stage:**

- The ALU module (`ALU.v`) performs the bitwise OR operation on the values of `rs` and `rt` based on the `ALUOp` control signal.
- The result of the OR operation is stored in a temporary register.

Example code from `ALU.v`:

```verilog
always @(i_data1, data2, i_ALUcontrol) begin
  case (i_ALUcontrol)
    // ...
    4'b0001:  // OR
      o_ALUresult = i_data1 | data2; // bitwise OR
    // ...
  endcase
  // ...
end
```

4. **Memory Access (MEM) Stage:**
   - For the `OR` instruction, no memory access is needed, so this stage is a pass-through.
5. **Write Back (WB) Stage:**

- The ALU result, which is the result of the OR operation, is written back to the destination register (`rd`) in the Register File.

Example code for writing back to the Register File:

```verilog
always @(posedge i_clk) begin
  if (i_RegWrite) begin
    RegData[i_wReg] = i_wData;
  end
end
```

Throughout the execution of the `OR` instruction, the control signals generated by the Control Unit module (`ControlUnit.v`) orchestrate the flow of data and the operations performed in each stage.

The ALU Control module (`ALUControl.v`) decodes the `ALUOp` signal and the function code of the instruction to generate the appropriate `ALUControl` signal for the ALU module (`ALU.v`) to perform the OR operation.

ORI (OR Immediate)

- **IF:** Instruction is fetched.
- **ID:** Opcode is decoded. Registers rs and rt are read for OR; rs and immediate for ORI.
- **EX:** ALU performs an OR operation.
- **MEM:** No action needed.
- **WB:** Result is written to rd (OR) or rt (ORI).

Instruction Breakdown

The `ORI` (OR Immediate) instruction in MIPS is an I-type instruction that performs a bitwise OR operation between a register and a zero-extended immediate value.

Instruction Format:

```
ORI rt, rs, immediate
```

- `rt`: The destination register where the result will be stored.
- `rs`: The source register containing one of the operands.
- `immediate`: A 16-bit immediate value that will be zero-extended to 32 bits.

Instruction Encoding:

```
| opcode (6 bits) | rs (5 bits) | rt (5 bits) | immediate (16 bits) |
```

Processor Stages: 1. Instruction Fetch (IF): - The instruction is fetched from the instruction memory using the PC (Program Counter). - The PC is incremented by 4 to point to the next instruction.

2. Instruction Decode (ID):
   - The instruction is decoded by the control unit.
   - The register file is accessed to read the values of the source register `rs` and the destination register `rt`.
   - The 16-bit immediate value is zero-extended to 32 bits.

Verilog code for the instruction decode stage:

```
// Control Unit
always @(i_instruction) begin
  case (i_instruction[31:26])
    // ...
    6'b001101: begin  // ORI
      o_RegDst = 0;
      o_ALUSrc = 1;
      o_MemtoReg = 0;
      o_RegWrite = 1;
      o_MemRead = 0;
      o_MemWrite = 0;
      o_Branch = 0;
      o_ALUOp = 2'b11;  // ALU control for OR operation
      o_Jump = 0;
    end
    // ...
  endcase
end
```

3. Execute (EX):
   - The ALU performs the bitwise OR operation between the value in the source register `rs` and the zero-extended immediate value.
   - The ALU control unit generates the appropriate control signal for the OR operation based on the `ALUOp` signal from the control unit.

Verilog code for the ALU control unit:

```
// ALU Control
always @(i_ALUOp or i_Funcode) begin
  case (i_ALUOp)
    // ...
    2'b11: begin  // ORI
      o_ALUcontrol = 4'b0001;  // ALU control for OR operation
    end
    // ...
  endcase
end
```

4. Memory (MEM):
   - No memory access is needed for the `ORI` instruction, so this stage is a pass-through.
5. Write Back (WB):
   - The ALU result is written back to the destination register `rt` in the register file.

Verilog code for the write-back stage:

```
// Register File
always @(posedge i_clk) begin
  if (i_RegWrite) begin
    RegData[i_writeReg] <= i_writeData;
  end
end
```

Here’s an example of how the `ORI` instruction would be processed in the single-cycle MIPS processor:

```
ORI $t0, $s0, 0xFFFF
```

This instruction performs a bitwise OR operation between the value in register `$s0` and the immediate value `0xFFFF` (16 bits), and stores the result in register `$t0`.

In the IF stage, the instruction is fetched from the instruction memory using the PC.

In the ID stage, the instruction is decoded, and the values of `$s0` and `$t0` are read from the register file. The immediate value `0xFFFF` is zero-extended to 32 bits.

In the EX stage, the ALU performs the bitwise OR operation between the value in `$s0` and the zero-extended immediate value. The ALU control unit generates the appropriate control signal for the OR operation based on the `ALUOp` signal from the control unit.

NOR

Intruction Overview

- **IF:** Fetch instruction.
- **ID:** Decode opcode; read rs and rt.
- **EX:** ALU performs NOR operation on rs and rt.
- **MEM:** No action needed.
- **WB:** Result is written back to rd.

Instruction Breakdown

The NOR instruction in the MIPS architecture performs a bitwise NOR operation on the values of two registers and stores the result in a destination register.

1. **Instruction Fetch (IF) Stage:**
   - The instruction is fetched from the instruction memory using the program counter (PC).
   - The instruction memory module (`InstructionMemory.v`) retrieves the instruction based on the address provided by the PC.

```
// Inside the InstructionMemory module
always @(i_Addr) begin
  if (i_Addr == -4) begin         // init
    i_Instruction = 32'b11111100000000000000000000000000;
  end else begin
    i_Instruction = Imem[i_Addr>>2];
  end
  i_Ctr = i_Instruction[31:26];
  i_Funcode = i_Instruction[5:0];
end
```

2. **Instruction Decode (ID) Stage:**
   - The fetched instruction is decoded by the control unit (`ControlUnit.v`).
   - The opcode of the instruction (bits [31:26]) is used to determine the type of instruction.
   - For the NOR instruction, the opcode is 6’b100111 (binary representation of 39).

```
// Inside the ControlUnit module
always @(i_instruction) begin
  case (i_instruction[31:26])
    // ...
    6'b100111: begin  // NOR
      o_RegDst = 1;
      o_ALUSrc = 0;
      o_MemtoReg = 0;
      o_RegWrite = 1;
      o_MemRead = 0;
      o_MemWrite = 0;
      o_Branch = 0;
      o_ALUOp = 2'b11;
      o_Jump = 0;
      // ...
    end
    // ...
  endcase
end
```

3. **Execute (EX) Stage:**
   - The ALU performs the bitwise NOR operation on the values of the source registers (rs and rt).
   - The ALU control unit generates the appropriate control signal (4’b1100) for the NOR operation based on the ALUOp bits from the control unit.

```
// Inside the ALU module
always @(i_data1, data2, i_ALUcontrol) begin
  case (i_ALUcontrol)
    // ...
    4'b1100:  // NOR
      o_ALUresult = i_data1 | ~data2;
    // ...
  endcase
  // ...
end
```

4. **Memory (MEM) Stage:**
    - For the NOR instruction, no memory access is required, so this stage is a pass-through.
5. **Write Back (WB) Stage:**
    - The result of the NOR operation from the ALU is written back to the destination register (rd) in the register file.

```
// Inside the RegisterFile module
always @(posedge i_clk) begin
  if (i_wEn == 1) begin
    RegData[i_wDst] <= i_wData;
  end
end
```

Here's an example of how the NOR instruction would be encoded in MIPS assembly and its corresponding machine code:

```
# MIPS Assembly
nor $t0, $s1, $s2   # Perform bitwise NOR of $s1 and $s2 and store the result in $t0

# Machine Code (in binary)
000000 10001 10010 01000 00000 100111
```

In the machine code, the bits are organized as follows: - Bits [31:26]: Opcode (000000 for R-type instructions) - Bits [25:21]: Source register 1 (rs) - Bits [20:16]: Source register 2 (rt) - Bits [15:11]: Destination register (rd) BNE (Branch Not Equal)

- **IF:** Fetch instruction.
- **ID:** Decode instruction; read registers rs and rt.
- **EX:** Compare values of rs and rt.
- **MEM:** No action needed.
- **WB:** If rs != rt, PC is updated to branch address (PC + offset); otherwise, move to next sequential instruction.

Instruction Breakdown

The BNE (Branch Not Equal) instruction is a conditional branch instruction in the MIPS architecture.

It compares the values of two registers and transfers control to a target address if the values are not equal.

1. **Instruction Fetch (IF) Stage:** In the IF stage, the instruction is fetched from the Instruction Memory using the Program Counter (PC).

```
// Fetch the instruction from the Instruction Memory
i_Instruction = Imem[i_Addr>>2];
```

2. **Instruction Decode (ID) Stage:** In the ID stage, the instruction is decoded, and the registers rs and rt are read from the Register File.

```
// Decode the instruction
i_Ctr = i_Instruction[31:26];
i_Funcode = i_Instruction[5:0];

// Read registers rs and rt from the Register File
wire [4:0] rs = i_Instruction[25:21];
wire [4:0] rt = i_Instruction[20:16];
wire [31:0] rs_data = RegData[rs];
wire [31:0] rt_data = RegData[rt];
```

3. **Execution (EX) Stage:** In the EX stage, the values of rs and rt are compared using the ALU.

```
// Compare the values of rs and rt using the ALU
wire ALU_zero;
ALU alu (
    .i_data1(rs_data),
    .i_read2(rt_data),
    .i_ALUcontrol(ALU_control),
    .o_Zero(ALU_zero),
    .o_ALUresult(ALU_result)
);
```

4. **Memory Access (MEM) Stage:**

The BNE instruction does not require any memory access, so this stage is a pass-through.

5. **Write Back (WB) Stage:**

In the WB stage, if the values of rs and rt are not equal (ALU_zero is 0), the Program Counter (PC) is updated to the branch target address (PC + offset). Otherwise, the PC moves to the next sequential instruction.

```
// Update the PC based on the branch condition
wire [31:0] branch_target = PC + {{14{i_Instruction[15]}}, i_Instruction[15:0], 2'b00};
wire branch_taken = i_Branch & ~ALU_zero;
assign PC_next = branch_taken ? branch_target : PC + 4;
```

Here's an example of how the BNE instruction can be represented in Verilog:

```
module BNE_example (
    input [31:0] rs_data,
    input [31:0] rt_data,
    input [15:0] offset,
    input [31:0] PC,
    output reg [31:0] PC_next
);
    wire ALU_zero;
    wire [31:0] branch_target = PC + {{14{offset[15]}}, offset, 2'b00};
    // Compare rs and rt using ALU
    assign ALU_zero = (rs_data == rt_data) ? 1 : 0;
    // Update PC based on branch condition
    always @(*) begin
        if (~ALU_zero)
            PC_next = branch_target;
        else
            PC_next = PC + 4;
    end
endmodule
```

In this example, the `BNE_example` module takes the values of rs and rt (`rs_data` and `rt_data`), the branch offset (`offset`), and the current Program Counter (`PC`) as inputs.

It compares rs and rt using the ALU and updates the `PC_next` output based on the branch condition.

If rs and rt are not equal (`ALU_zero` is 0), `PC_next` is set to the branch target address (`branch_target`).

Otherwise, `PC_next` is set to the next sequential instruction address (`PC + 4`).

Comparing Verilog vs VHDL

The following section will compare the experience of developing a single-cycle processor in Verilog vs VHDL.

Interesting notes about verilog

The loose typing of Verilog can lead to some useful modules that can be defined in small amounts of code.

Additionally, in verilog, you do not have to declare your components in the component that uses them which also decreases the amount of code that is needed to be written.

For example, the following module is the mux module that is used in the processor to select between two inputs based on a control signal.

```
module mux #(parameter size = 1) (
  input select,
  input [size - 1:0] in_0,
  input [size - 1:0] in_1,
  output [size - 1:0] out
);
  assign out = (select) ? in_1 : in_0;
endmodule
```

Another example of this is the sign extender module that is used to extend the sign of a 16-bit number to a 32-bit number.

```
module signextender (
  input [15:0] in,
  output [31:0] out
);
  assign out = {{16{in[15]}}, {in}};
endmodule
```

Additionally, the fact that in Verilog you do not need to preemptively define components before using them allows for a more flexible design and faster development.

Another nice feature of Verilog (specifically VerilogHDL) is the ability to print out the values of signals without having to worry about their types in the waveform viewer inside modelsim/questasim.

This is a feature that is not present in VHDL and is very useful for debugging and understanding the behavior of the processor.

Interesting notes about VHDL

While you can do this “print debugging” in VHDL, it is not as easy as it is in Verilog because in VHDL you must deal with the typings of the signals and the fact that you must declare the signals before you can use them.

As an example, the following is the code in VHDL that prints out the values of the signals in the waveform viewer for an n-bit register which was used in a project for CPRE381.

The following test-bench shows the code that can be executed inside modelsim/questasim to print out the values of the signals in the waveform viewer.

It displays the additional hurdles that must be overcome in VHDL to print out the values of the signals in the waveform viewer because of the strict typing of the language.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY tb_nbitregister IS
  GENERIC (
    gCLK_HPER : TIME := 50 ns;
    N : INTEGER := 32);
END tb_nbitregister;
```

```vhdl
ARCHITECTURE behavior OF tb_nbitregister IS

  -- Calculate the clock period as twice the half-period
  CONSTANT cCLK_PER : TIME := gCLK_HPER * 2;
  COMPONENT nbitregister
    PORT (
      i_CLK : IN STD_LOGIC; -- Clock input
      i_RST : IN STD_LOGIC; -- Reset input
      i_WE : IN STD_LOGIC; -- Write enable input
      i_D : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0); -- Data value input
      o_Q : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0) -- Data value output
    );
  END COMPONENT;
  -- Temporary Signals to connect to the nbitregister component.
  SIGNAL s_CLK, s_RST, s_WE : STD_LOGIC;
  SIGNAL s_D, s_Q : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
BEGIN
  DUT : nbitregister
  PORT MAP(
    i_CLK => s_CLK,
    i_RST => s_RST,
    i_WE => s_WE,
    i_D => s_D,
    o_Q => s_Q
  );
  P_CLK : PROCESS
  BEGIN
    s_CLK <= '0';
    WAIT FOR gCLK_HPER;
    s_CLK <= '1';
    WAIT FOR gCLK_HPER;
  END PROCESS;
  P_TB : PROCESS
  BEGIN
    s_RST <= '1';
    s_WE <= '0';
    s_D <= "00000000000000000000000000000000";
    WAIT FOR cCLK_PER;
    -- TEST CASE 1 - STORE '1'
    -- DESCRIPTION: The register should store the new data value
    -- EXPECTED RESULT: The new data value should be stored in the register
    s_RST <= '0';
    s_WE <= '1';
    s_D <= "11111111111111111111111111111111";
    WAIT FOR cCLK_PER;
    IF (s_Q /= "11111111111111111111111111111111") THEN
      REPORT "Test 1 failed";
      REPORT "Expected: 11111111111111111111111111111111";
      REPORT "Actual:   " & STD_LOGIC_VECTOR'image(s_Q);
    ELSE
      REPORT "TEST 1 PASSED - STORE '1'";
    END IF;
      // ...
END behavior;
```

Conclusion

I think that VHDL actually provides more flexibility within the development of the processor.

While the language is more verbose because you must reinstantiate a component within another component to use it, it is more type-safe, and allows for more control over the design of the processor.

Verilog is more concise and easier to read, but I think that VHDL is more powerful and allows for more control over the design of the processor because of it's type-safety.

To support this, I present the result of the actual lines of code that were written for the processor in VHDL and Verilog.

VHDL Single Cycle MIPS Processor Code Statistics:

```
Language                         files          blank        comment           code
-------------------------------------------------------------------------------------
VHDL                                65            694           1085           4677
```

Verilog Single Cycle MIPS Processor Code Statistics:

```
Language                         files          blank        comment           code
-------------------------------------------------------------------------------------
Verilog-SystemVerilog               10              0              9            555
```

As you can see, the VHDL processor has almost **10** times the amount of code as the Verilog processor.

Furthermore, I think that the fact that the name of a file in verilog must match the module name is a limitation that VHDL does not have (atleast in our Quartus simulator).

To summarize, I think that VHDL is more suited for larger projects and more complex designs where the type-safety and , while Verilog is more suited for smaller projects and simpler designs.

Breaking down decoding a signal to 7-segment displays

As the signal representing the instruction is 5 bits long inside the `controller.v` file, we need to decode this signal to display the current instruction on the 7-segment displays.

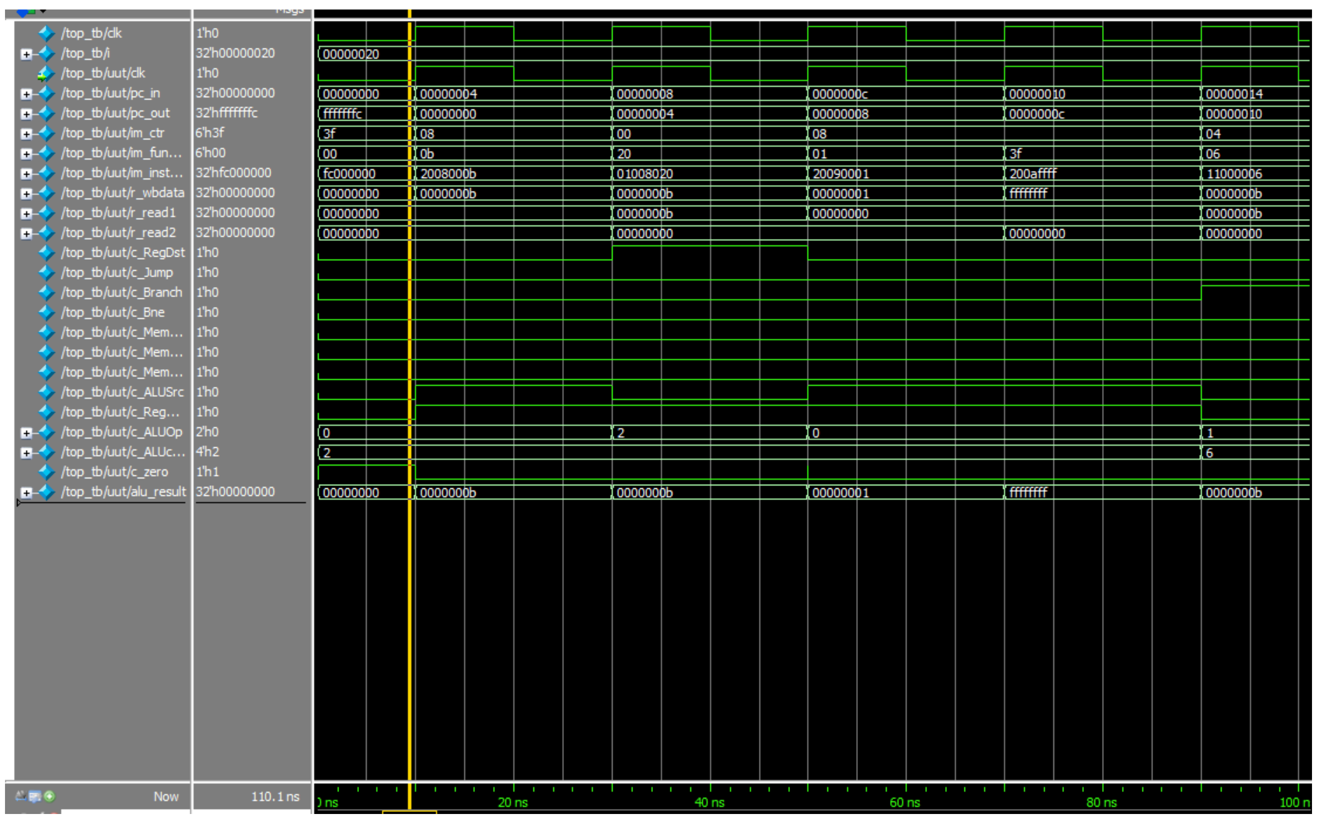
This means that we need to decode a 5-bit signal to a 35-bit signal that will be used to display the current instruction on the 7-segment displays.

If 5-bits are used to represent the instruction, and 7-bits are needed to represent a character on a 7-segment display, then 35-bits are needed to represent the current instruction on 5 7-segment displays as 7 * 5 = 35.

Furthermore, the longest word that can be displayed on the 7-segment displays is 5 characters long, so 5 * 7 = 35 bits are needed to represent the current instruction on the 7-segment displays.

Func_in	O_out	Operation	Description
1000	ox	$(A+B)$	ADD
1000	1X	(A−B)	SuB
1001	00	$(A\&B)$	AND
1001	01	$(A \mid B)$	OR
1001	π	~ (A|B)	NOR
101	xx0	signed (A) < signed (B)	Set-Less-Than signed
101	$xx1$	$A < B$	Set-Less-Than unsigned
111	000	A	BLTZ (Branch if Less Than Zero)
111	001	A	BGEZ (Branch if Greater or Equal to Zero)
111	010	A	J/AL (Jump and Link)
111	011	A	JR/AL (Jump Register and Link)
111	100	A	BEQ((Branch if Equal)
111	101	A	BNE (Branch if Not Equal)

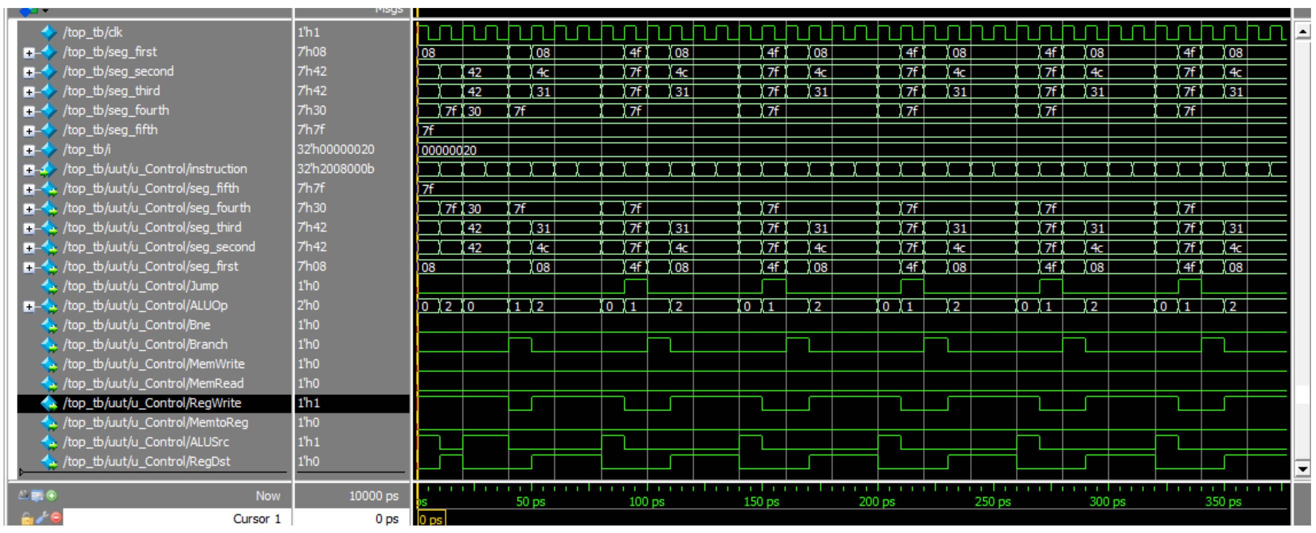
The following is the wave-diagram from modelsim/questasim for my test-bench of my processor without the added seven segment displays.



/top_tb/clk 1'h0
/top_tb/i 32'h00000020
/top_tb/uut/clk 1'h0
/top_tb/uut/pc_in 32'h00000000
/top_tb/uut/pc_out 32'hfffffffc
/top_tb/uut/im_ctr 6'h3f
/top_tb/uut/im_fun... 6'h00
/top_tb/uut/im_inst... 32'hfc000000
/top_tb/uut/r_wbdata 32'h00000000
/top_tb/uut/r_read1 32'h00000000
/top_tb/uut/r_read2 32'h00000000
/top_tb/uut/c_RegDst 1'h0
/top_tb/uut/c_Jump 1'h0
/top_tb/uut/c_Branch 1'h0
/top_tb/uut/c_Bne 1'h0
/top_tb/uut/c_Mem... 1'h0
/top_tb/uut/c_Mem... 1'h0
/top_tb/uut/c_Mem... 1'h0
/top_tb/uut/c_ALUSrc 1'h0
/top_tb/uut/c_Reg... 1'h0
/top_tb/uut/c_ALUOp 2'h0
/top_tb/uut/c_ALUc... 4'h2
/top_tb/uut/c_zero 1'h1
/top_tb/uut/alu_result 32'h00000000
Now 110.1 ns
20 ns
40 ns
60 ns
80 ns
100 ns

WaveDiagramWithoutSevenSegment.png

Below is the captured wave-diagram from modelsim/questasim with the seven segment ports included:



/top_tb/clk 1'h1
/top_tb/seg_first 7'h08
/top_tb/seg_second 7'h42
/top_tb/seg_third 7'h42
/top_tb/seg_fourth 7'h30
/top_tb/seg_fifth 7'h7f
/top_tb/i 32'h00000020
/top_tb/uut/u_Control/instruction 32'h2008000b
/top_tb/uut/u_Control/seg_fifth 7'h7f
/top_tb/uut/u_Control/seg_fourth 7'h30
/top_tb/uut/u_Control/seg_third 7'h42
/top_tb/uut/u_Control/seg_second 7'h42
/top_tb/uut/u_Control/seg_first 7'h08
/top_tb/uut/u_Control/Jump 1'h0
/top_tb/uut/u_Control/ALUOp 2'h0
/top_tb/uut/u_Control/Bne 1'h0
/top_tb/uut/u_Control/Branch 1'h0
/top_tb/uut/u_Control/MemWrite 1'h0
/top_tb/uut/u_Control/MemRead 1'h0
/top_tb/uut/u_Control/RegWrite 1'h1
/top_tb/uut/u_Control/MemtoReg 1'h0
/top_tb/uut/u_Control/ALUSrc 1'h1
/top_tb/uut/u_Control/RegDst 1'h0
Now 10000 ps
Cursor 1 0 ps
50 ps
100 ps
150 ps
200 ps
250 ps
300 ps
350 ps

SevenSegmentWaveDiagram.png

The wave-forms show the output of the following test-bench, `mips_tb.v`, which is used to test the single-cycle MIPS processor, `mips.v`.

```verilog
`timescale 1ns / 1ps
`define CYCLE_TIME 20
module mips_tb;
  reg clk;
  reg rst;
  // segments for the 7-segment displays
  wire [6:0] seg_first, seg_second, seg_third, seg_fourth, seg_fifth;
  integer i;
  always #(`CYCLE_TIME / 2) clk = ~clk;
  mips uut (
      .i_Clk(clk),
      .i_Rst(rst),
      .o_Seg_first(seg_first),
      .o_Seg_second(seg_second),
      .o_Seg_third(seg_third),
      .o_Seg_fourth(seg_fourth),
      .o_Seg_fifth(seg_fifth)
  );
  initial begin
    // Initialize data memory
    for (i = 0; i < 32; i = i + 1) begin
      uut.inst_DataMemory.Dmem[i] = 32'b0;
    end
    // Initialize Register File
    for (i = 0; i < 32; i = i + 1) begin
      uut.inst_RegisterFile.RegData[i] = 32'b0;
    end
    clk = 0;
  end
  initial begin
    #1800 $finish;
  end
endmodule
```

The given Verilog code represents my test-bench module that was used for testing the single-cycle MIPS processor, `mips.v`.

1. The test-bench module is named `mips_tb` (titled `mips_tb.v`), and it operates on a timescale of 1ns/1ps.

2. The module declares two reg variables:

   - `clk`: Represents the clock signal for the processor.
   - `rst`: Represents the reset signal for the processor.

3. It also declares five wire variables (`seg_first`, `seg_second`, `seg_third`, `seg_fourth`, `seg_fifth`) to represent the segments for the 7-segment displays. These wires are used to display the current instruction being executed by the processor.

4. The `integer` variable `i` is declared as a loop variable for initializing memory.

5. The `always` block generates the clock signal by toggling the `clk` variable every half of the clock cycle time (`CYCLE_TIME/2`).

6. The `mips` module (the actual MIPS processor) is instantiated as `uut` (unit under test) with the following connections:

- `i_Clk` is connected to the `clk` signal.
- `i_Rst` is connected to the `rst` signal.
- The 7-segment display outputs (`o_Seg_first`, `o_Seg_second`, `o_Seg_third`, `o_Seg_fourth`, `o_Seg_fifth`) are connected to the corresponding wires in the test-bench.

7. The first `initial` block is used to initialize the data memory and the register file of the MIPS processor:
   - It uses a `for` loop to iterate over the first 32 locations of the data memory (`Dmem`) and initializes each location to zero.
   - Similarly, it initializes the first 32 registers in the register file (`RegData`) to zero.
   - Finally, it sets the `clk` variable to 0.
8. The second `initial` block is used to specify the duration of the simulation. It uses the `$finish` system task to terminate the simulation after 1800 time units.

The purpose of this test-bench is to provide a simulation environment for the MIPS processor. It initializes the necessary components (data memory and register file), generates the clock signal, and instantiates the MIPS processor module. The testbench also specifies the duration of the simulation.

The test-bench interacts with other components of the processor through the instantiated `mips` module (`uut`). It provides the clock and reset signals to the processor and observes the output signals for the 7-segment displays.

Overall, this test-bench serves as a framework to verify the functionality of the single-cycle MIPS processor by providing the necessary inputs, initializing the memory, and specifying the simulation duration.

Schematics

The following section will include the schematics for the various components of the MIPS processor as written in verilog.

Control Unit Schematic

The following is the schematic for the control unit of the MIPS processor.

[[Pasted image 20240503123506.png]]

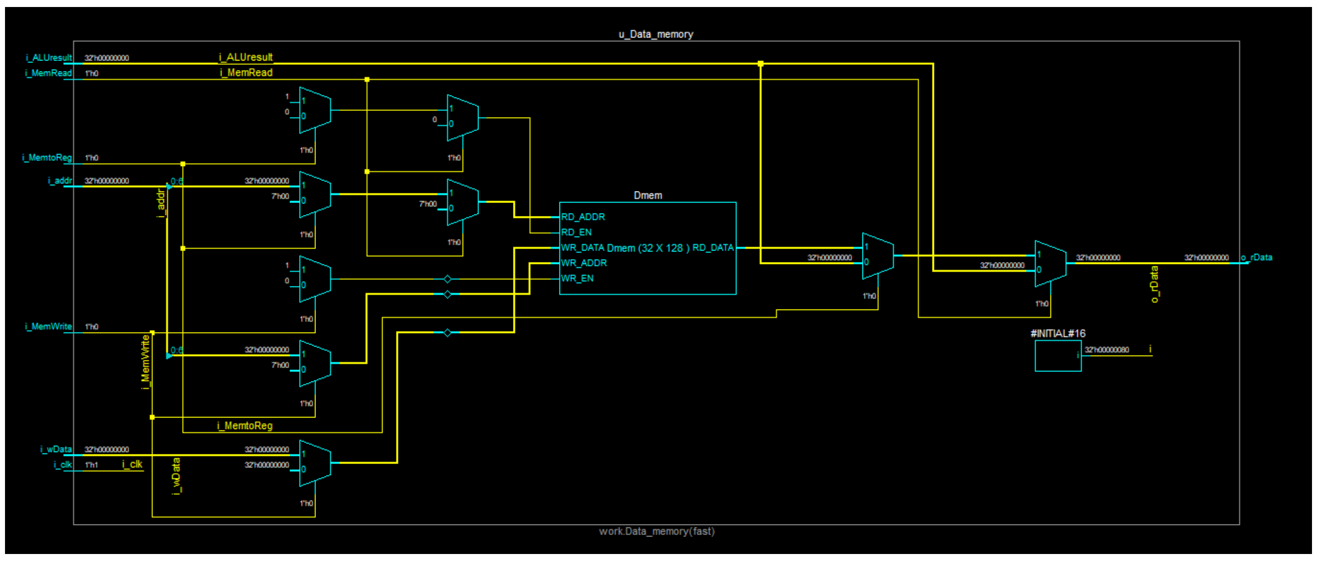
Register File

The following is the schematic for the register file of the MIPS processor.

[[RegisterFile.png]]

Data Memory

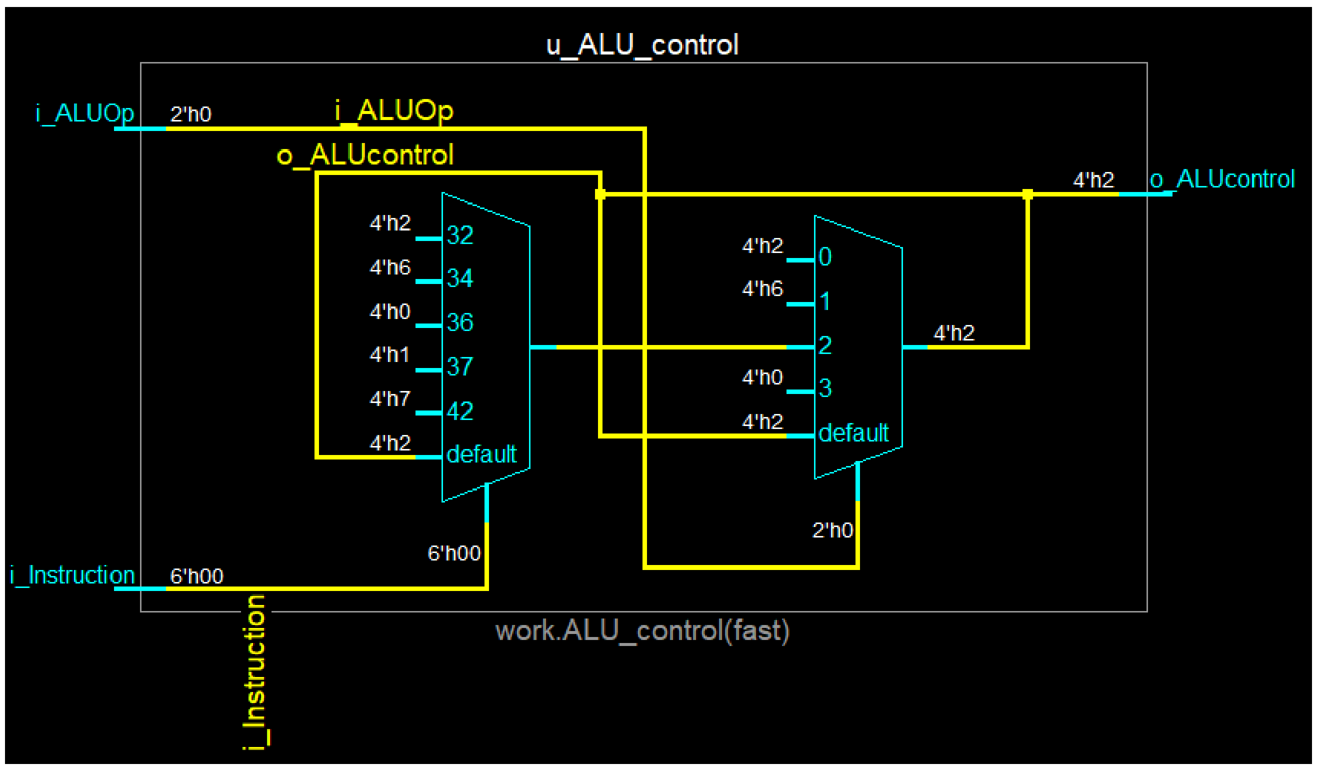
The following is the schematic for the data memory of the MIPS processor.



SchematicDataMemory.png

ALU Control

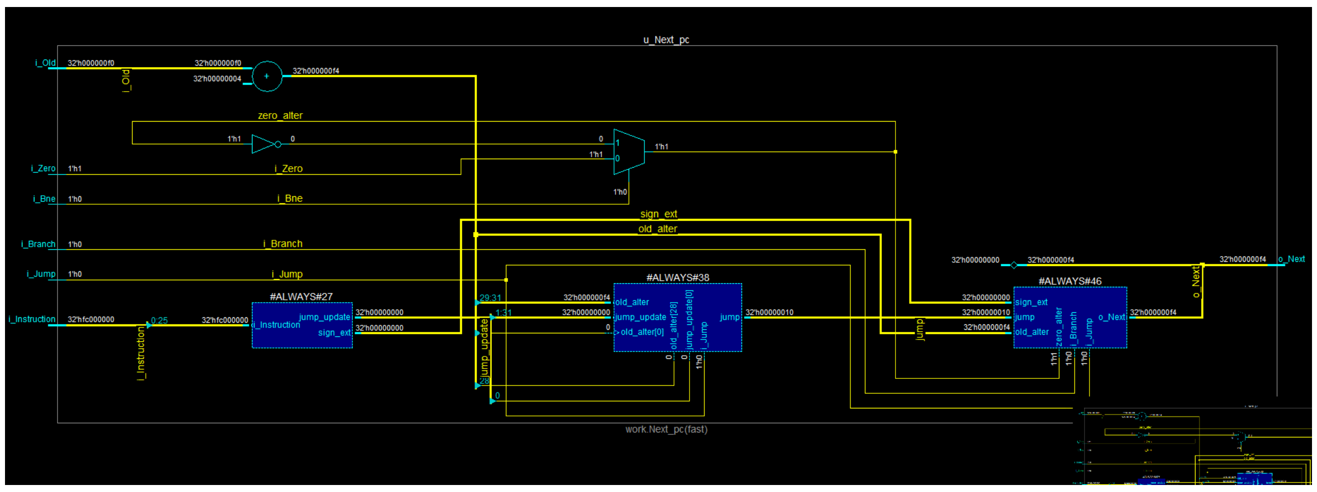
The following is the schematic for the ALU control of the MIPS processor.



SchematicALUControl.png

Program Counter Control

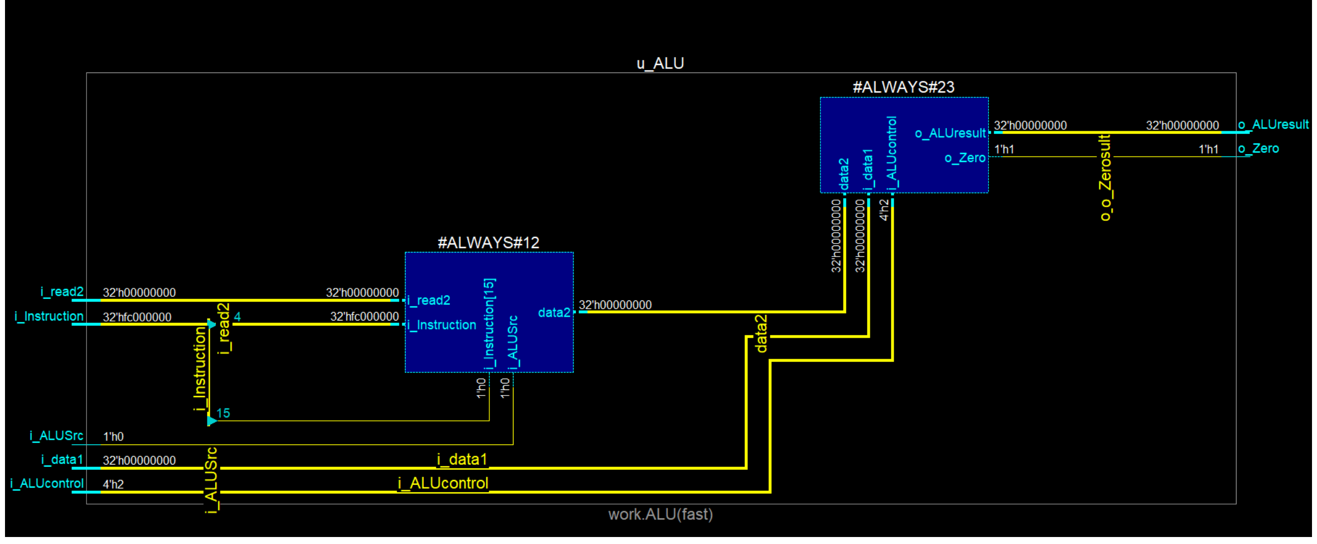
The following is the schematic for the program counter control of the MIPS processor.



SchematicNextProgramCounter.png

ALU

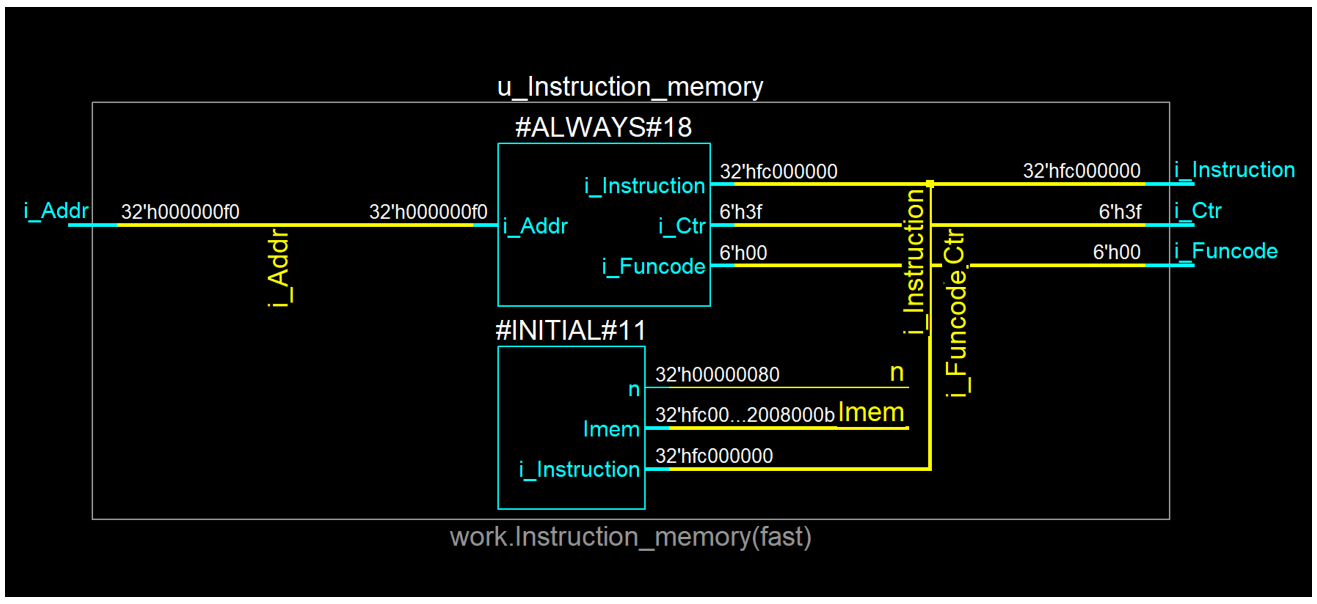
The following is the schematic for the ALU of the MIPS processor.



SchematicALU.png

Instruction Memory

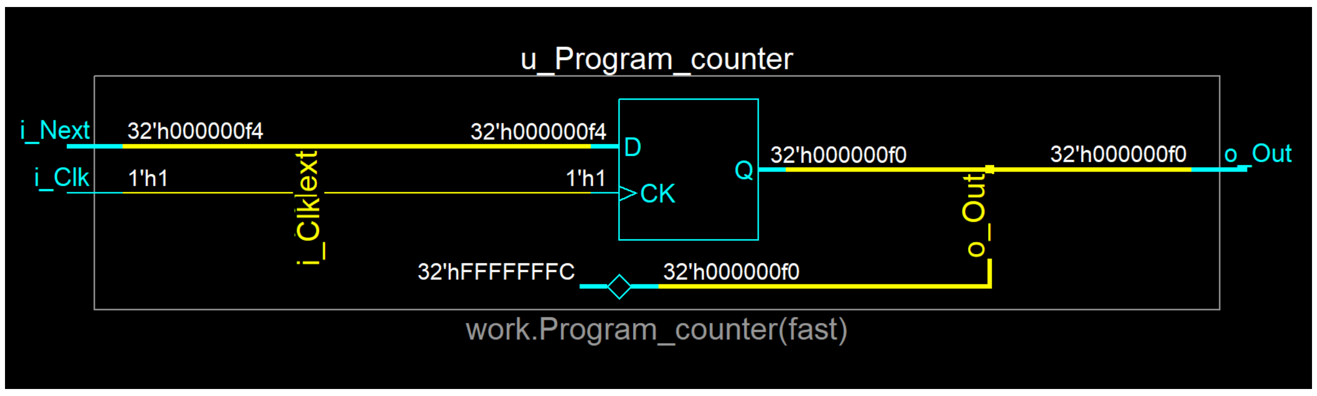
The following is the schematic for the instruction memory of the MIPS processor.



SchematicInstructionMemory.png

Program Counter

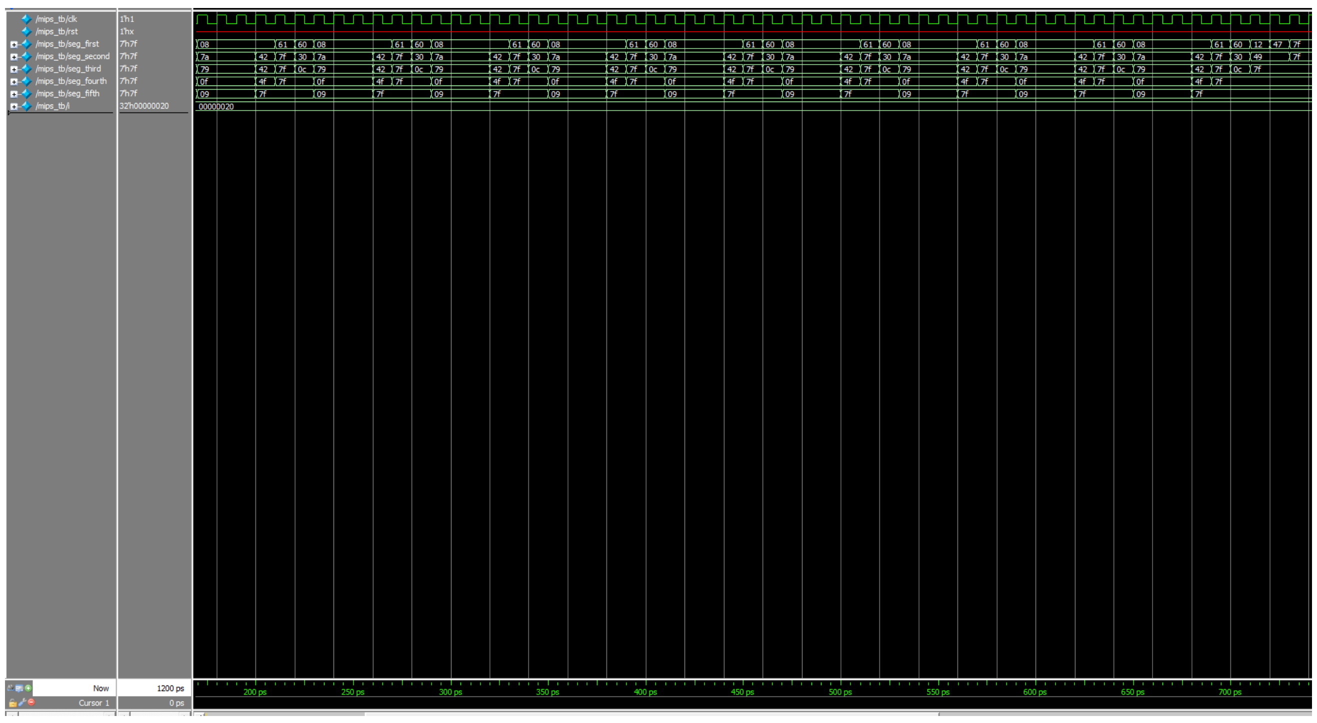
The following is the schematic for the program counter of the MIPS processor.



SchematicProgramCounter.png

Waveform

The following is the wave form of the Processor from modelsim/questasim:



TestbenchWaveform.png

Tooling

First, as learned in CPRE381, I enjoy having test-benches for my code.

Test-benches allow for faster debugging and more efficient development by allowing you to test your code without having to run it on the FPGA board, directly seeing the signals, how they interact with one another, and allows for testing to make sure that progress is being made.

I used a test-bench to test my processor and ensure that it was working correctly.

The following is the main test-bench that I used to test my processor, `mips_tb.v`.

```
`timescale 1ns / 1ps
`define CYCLE_TIME 20
module mips_tb;
  reg clk;
  reg rst;
  wire [6:0] seg_first, seg_second, seg_third, seg_fourth, seg_fifth;
  integer i;
  always #(`CYCLE_TIME / 2) clk = ~clk;
  mips uut (
      .i_Clk(clk),
      .i_Rst(rst),
      .o_Seg_first(seg_first),
      .o_Seg_second(seg_second),
      .o_Seg_third(seg_third),
      .o_Seg_fourth(seg_fourth),
      .o_Seg_fifth(seg_fifth)
  );
  initial begin
    // Initialize data memory
    for (i = 0; i < 32; i = i + 1) begin
      uut.inst_DataMemory.Dmem[i] = 32'b0;
    end
    // Initialize Register File
    for (i = 0; i < 32; i = i + 1) begin
      uut.inst_RegisterFile.RegData[i] = 32'b0;
    end
    clk = 0;
  end
  initial begin
    #1800 $finish;
  end
endmodule
```

I used do files to more easily compile and run my code.

The following is the main `.do` file, `run.do` that I used to compile and run my code.

```
set target "mips_tb"
set file "proj/${target}.v"
if { [file exists "work"] } {
    vdel -all
}
vlog *.v
vsim -voptargs=+acc -debugDB $target
force -freeze sim:/$target/clk 1 0, 0 {5 ps} -r 10
# force -freeze sim:/$target/rst 0 0, 1 {80 ps} -r 100
add wave -position insertpoint \ ../$target/*
run 1200
```

In addition to the tooling already mentioned, I used modelsim/questasim to simulate my processor and test-bench.

Furthermore, I used the Quartus Prime software to compile my code and program my FPGA board.

Even further, I used tools like git, GitHub, and markdown to manage my code, version control, and documentation.

For editor tooling, I used NeoVim with a combination of popular language servers that are used for VerilogHDL.

These language servers that I used for development include verible, Tree-Sitter, veridian, and more to provide completion, syntax highlighting, code actions, linting, and more.

I think that using these language servers and tools in combination with NeoVim (my personal open-sourced config has a startup time of <80ms) allowed me to develop my processor more efficiently and effectively.

Components and Explanations

The following section explains the components of the MIPS processor and their functionalities.

This is done by providing the Verilog code for each component and explaining its role in the processor.

ALU

The following is the code for the ALU module of the MIPS processor called `ALU.v`. (It can be found in the `./proj/` directory)

```
`timescale 1ns / 1ps
module ALU (
    input      [31:0] i_data1,        // data 1
    input      [31:0] i_read2,        // data 2 from MUX
    input      [31:0] i_Instruction,  // used for sign-extension
    input             i_ALUSrc,
    input      [ 3:0] i_ALUcontrol,
    output reg        o_Zero,
    output reg [31:0] o_ALUresult
);
  reg [31:0] data2;
  always @(i_ALUSrc, i_read2, i_Instruction) begin
    if (i_ALUSrc == 0) begin
      data2 = i_read2;
    end else begin
      if (i_Instruction[15] == 1'b0) begin
        data2 = {16'b0, i_Instruction[15:0]};
      end else begin
        data2 = {{16{1'b1}}, i_Instruction[15:0]};
      end
    end
  end
  always @(i_data1, data2, i_ALUcontrol) begin
    case (i_ALUcontrol)
      4'b0000:  // AND
      o_ALUresult = i_data1 & data2; // bitwise AND
      4'b0001:  // OR
      o_ALUresult = i_data1 | data2; // bitwise OR
      4'b0010:  // ADD
      o_ALUresult = i_data1 + data2; // addition
      4'b0110:  // SUB
      o_ALUresult = i_data1 - data2; // subtraction
      4'b0111:  // SLT
      o_ALUresult = (i_data1 < data2) ? 1 : 0; // set-on-less-than
      4'b1100:  // NOR
      o_ALUresult = i_data1 | ~data2; // bitwise NOR
      default: ;
    endcase
    if (o_ALUresult == 0) begin
      o_Zero = 1;
    end else begin
      o_Zero = 0;
    end
  end
endmodule
```

The above code represents the ALU (Arithmetic Logic Unit) module of the single-cycle MIPS processor.

The ALU is responsible for performing arithmetic and logical operations on the input data based on the ALU control signal.

Inputs: - `i_data1` (32-bit): The first input data for the ALU operation. - `i_read2` (32-bit): The second input data from the MUX. - `i_Instruction` (32-bit): The instruction used for sign-extension. - `i_ALUSrc` (1-bit): A control signal indicating whether to use the second input data from the MUX or the sign-extended immediate value. - `i_ALUcontrol` (4-bit): The ALU control signal that determines the specific operation to be performed.

Outputs: - `o_Zero` (1-bit): A flag indicating whether the ALU result is zero. - `o_ALUresult` (32-bit): The result of the ALU operation.

Functionality: 1. Data Selection: - The module first determines the second input data for the ALU operation based on the `i_ALUSrc` control signal. - If `i_ALUSrc` is 0, the second input data is taken directly from `i_read2`. - If `i_ALUSrc` is 1, the second input data is obtained by sign-extending the 16-bit immediate value from the `i_Instruction`.

2. ALU Operation:
    - Based on the `i_ALUcontrol` signal, the module performs the corresponding ALU operation on `i_data1` and the selected second input data (`data2`).
    - The supported ALU operations include AND, OR, ADD, SUB (subtract), SLT (set-on-less-than), and NOR.
    - The result of the ALU operation is stored in `o_ALUresult`.
3. Zero Flag:
    - After performing the ALU operation, the module checks if the result is zero.
    - If the result is zero, the `o_Zero` flag is set to 1; otherwise, it is set to 0.

Significance in the MIPS Processor: - The ALU is a crucial component in the MIPS processor’s data-path. - It performs arithmetic and logical operations on the input data based on the instructions being executed. - The ALU receives input data from the register file or the immediate value in the instruction, depending on the `i_ALUSrc` control signal. - The ALU control signal (`i_ALUcontrol`) determines the specific operation to be performed, which is decoded by the ALU control module based on the instruction opcode and function code. - The result of the ALU operation is used for various purposes, such as storing it back to the register file, using it as a memory address, or making branching decisions based on the zero flag.

Interaction with Other Components: - The ALU receives input data from the register file (`i_data1` and `i_read2`) and the instruction (`i_Instruction`). - The ALU control module generates the `i_ALUcontrol` signal based on the instruction opcode and function code, which determines the ALU operation to be performed. - The ALU result (`o_ALUresult`) is used by other components, such as the data memory for memory operations or the register file for storing the result. - The zero flag (`o_Zero`) is used by the control unit to make branching decisions based on the comparison result.

Overall, the ALU module performs the necessary arithmetic and logical operations in the MIPS processor based on the instruction being executed.

The ALU allows the processor to execute instructions and produce the desired results.

Control Unit

The following is the code for the control unit of the MIPS processor called `ControlUnit.v`. (It can be found in the `./proj/` directory)

As named, the `ControlUnit`, `ControlUnit.v` is responsible for decoding the instruction and generating the control signals for the various components of the processor.

```
`timescale 1ns / 1ps
module ControlUnit (
    input [31:0] i_instruction,
    output reg o_RegDst,
    output reg o_Jump,
    output reg o_Branch,
    output reg o_Bne,
    output reg o_MemRead,
    output reg o_MemtoReg,
    output reg [1:0] o_ALUOp,
    output reg o_MemWrite,
    output reg o_ALUSrc,
    output reg o_RegWrite,
    output reg [6:0] o_seg_first,
    output reg [6:0] o_seg_second,
    output reg [6:0] o_seg_third,
    output reg [6:0] o_seg_fourth,
    output reg [6:0] o_seg_fifth
);
  initial begin
    o_RegDst = 0;
    o_Jump = 0;
    o_Branch = 0;
    o_MemRead = 0;
    o_MemtoReg = 0;
    o_ALUOp = 2'b00;
    o_MemWrite = 0;
    o_ALUSrc = 0;
    o_RegWrite = 0;
    o_seg_first = 7'b1111111;  // Blank
    o_seg_second = 7'b1111111;  // Blank
    o_seg_third = 7'b1111111;  // Blank
    o_seg_fourth = 7'b1111111;  // Blank
    o_seg_fifth = 7'b1111111;  // Blank
  end
  always @(i_instruction) begin
    case (i_instruction[31:26])
      6'b000000: begin  // ARITHMETIC
        o_RegDst = 1;
        o_ALUSrc = 0;
        o_MemtoReg = 0;
        o_RegWrite = 1;
        o_MemRead = 0;
        o_MemWrite = 0;
        o_Branch = 0;
        o_Bne = 0;
        o_ALUOp = 2'b10;
        o_Jump = 0;
        o_seg_first =  7'b0001000;  // A
        o_seg_second = 7'b1111010;  // R
        o_seg_third =  7'b1111001;  // I
        o_seg_fourth = 7'b0001111;  // T
        o_seg_fifth =  7'b0001001;  // H
      end
      6'b001000: begin  // addi
        o_RegDst = 0;
        o_ALUSrc = 1;
        o_MemtoReg = 0;
        o_RegWrite = 1;
        o_MemRead = 0;
        o_MemWrite = 0;
        o_Branch = 0;
        o_Bne = 0;
        o_ALUOp = 2'b00;
        o_Jump = 0;
        o_seg_first = 7'b0001000;  // A
        o_seg_second = 7'b1000010;  // d
        o_seg_third = 7'b1000010;  // d
        o_seg_fourth = 7'b1001111;  // i
        o_seg_fifth = 7'b1111111;  // Blank
      end
      6'b001100: begin  // andi
        o_RegDst = 0;
        o_ALUSrc = 1;
        o_MemtoReg = 0;
        o_RegWrite = 1;
        o_MemRead = 0;
        o_MemWrite = 0;
        o_Branch = 0;
        o_Bne = 0;
        o_ALUOp = 2'b11;
        o_Jump = 0;
        o_seg_first = 7'b0001000;  // A
        o_seg_second = 7'b0101011;  // n
        o_seg_third = 7'b1000010;  // d
        o_seg_fourth = 7'b1001111;  // i
        o_seg_fifth = 7'b1111111;  // Blank
      end
      6'b100011: begin  // lw
        o_RegDst = 0;
        o_ALUSrc = 1;
        o_MemtoReg = 1;
        o_RegWrite = 1;
        o_MemRead = 1;
        o_MemWrite = 0;
        o_Branch = 0;
        o_Bne = 0;
        o_ALUOp = 2'b00;
        o_Jump = 0;
        o_seg_first = 7'b1000111;  // L
        o_seg_second = 7'b1001001;  // w
        o_seg_third = 7'b1111111;  // Blank
        o_seg_fourth = 7'b1111111;  // Blank
        o_seg_fifth = 7'b1111111;  // Blank
      end
      6'b101011: begin  // sw
        o_RegDst = 0;  // X
        o_ALUSrc = 1;
        o_MemtoReg = 0;  // X
        o_RegWrite = 0;
        o_MemRead = 0;
        o_MemWrite = 1;
        o_Branch = 0;
        o_Bne = 0;
        o_ALUOp = 2'b00;
        o_Jump = 0;
        o_seg_first = 7'b0010010;  // S
        o_seg_second = 7'b1001001;  // w
        o_seg_third = 7'b1111111;  // Blank
        o_seg_fourth = 7'b1111111;  // Blank
        o_seg_fifth = 7'b1111111;  // Blank
      end
      6'b000100: begin  // beq
        o_RegDst = 0;  // X
        o_ALUSrc = 0;
        o_MemtoReg = 0;  // X
        o_RegWrite = 0;
        o_MemRead = 0;
        o_MemWrite = 0;
        o_Branch = 1;
        o_Bne = 0;
        o_ALUOp = 2'b01;
        o_Jump = 0;
        o_seg_first = 7'b1100000;  // b
        o_seg_second = 7'b0110000;  // e
        o_seg_third = 7'b0001100;  // q
        o_seg_fourth = 7'b1111111;  // Blank
        o_seg_fifth = 7'b1111111;  // Blank
      end
      6'b000101: begin  // bne
        o_RegDst = 0;  // X
        o_ALUSrc = 0;
        o_MemtoReg = 0;  // X
        o_RegWrite = 0;
        o_MemRead = 0;
        o_MemWrite = 0;
        o_Branch = 1;
        o_Bne = 1;
        o_ALUOp = 2'b01;
        o_Jump = 0;
        o_seg_first = 7'b1100000;  // b
```

```
          o_seg_second = 7'b0101011;  // n
          o_seg_third = 7'b0110000;  // e
          o_seg_fourth = 7'b1111111;  // Blank
          o_seg_fifth = 7'b1111111;  // Blank
        end
        6'b000010: begin  // j
          o_RegDst = 0;  // X
          o_ALUSrc = 0;
          o_MemtoReg = 0;  // X
          o_RegWrite = 0;
          o_MemRead = 0;
          o_MemWrite = 0;
          o_Branch = 0;
          o_Bne = 0;
          o_ALUOp = 2'b01;
          o_Jump = 1;
          o_seg_first = 7'b1100001;  // J
          o_seg_second = 7'b1111111;  // Blank
          o_seg_third = 7'b1111111;  // Blank
          o_seg_fourth = 7'b1111111;  // Blank
          o_seg_fifth = 7'b1111111;  // Blank
        end
        default: begin
          o_RegDst = 0;  // X
          o_ALUSrc = 0;
          o_MemtoReg = 0;  // X
          o_RegWrite = 0;
          o_MemRead = 0;
          o_MemWrite = 0;
          o_Branch = 0;
          o_Bne = 0;
          o_ALUOp = 2'b00;
          o_Jump = 0;
          o_seg_first = 7'b1111111;  // Blank
          o_seg_second = 7'b1111111;  // Blank
          o_seg_third = 7'b1111111;  // Blank
          o_seg_fourth = 7'b1111111;  // Blank
          o_seg_fifth = 7'b1111111;  // Blank
        end
      endcase
    end
endmodule
```

The above Verilog code represents the Control Unit component of the single-cycle MIPS processor.

The Control Unit is responsible for generating control signals based on the input instruction, which determine the behavior of various components within the processor.

IO

The Control Unit, `ControlUnit.v`, has the following inputs and outputs:

Inputs: - `i_instruction`: The 32-bit instruction fetched from the Instruction Memory.

Outputs: - Various control signals: - `o_RegDst`: Selects the destination register for the instruction (0 for rt, 1 for rd). - `o_Jump`: Indicates if the instruction is a jump instruction. - `o_Branch`: Indicates if the instruction is a branch instruction. - `o_Bne`: Indicates if the instruction is a “branch not equal” instruction. - `o_MemRead`: Enables reading from the Data Memory. - `o_MemtoReg`: Selects the source of data to be written to the register (0 for ALU result, 1 for memory data). - `o_ALUOp`: A 2-bit signal that specifies the ALU operation. - `o_MemWrite`: Enables writing to the Data Memory. - `o_ALUSrc`: Selects the second source for the ALU (0 for register, 1 for immediate). - `o_RegWrite`: Enables writing to the Register File. - 7-segment display outputs: - `o_seg_first` to `o_seg_fifth`: Control signals for displaying the instruction type on 7-segment displays.

Processor Context:

The following is the context (the purpose and functionality) of the Control Unit in the single-cycle MIPS processor:

1. The Control Unit initializes all control signals to default values in the `initial` block.

2. The `always` block is triggered whenever the `i_instruction` changes. It uses a case statement to determine the type of instruction based on the opcode (bits 31 to 26 of the instruction).

3. Depending on the instruction type, the Control Unit sets the appropriate control signals:

    - For R-type instructions (arithmetic), it sets `RegDst` to 1, enables `RegWrite`, sets `ALUOp` to 2’b10, and displays “ARITH” on the 7-segment displays.
    - For I-type instructions (addi, andi, lw, sw), it sets `ALUSrc` to 1, enables `RegWrite` (except for sw), sets `ALUOp` based on the instruction, and displays the instruction type on the 7-segment displays.
    - For branch instructions (beq, bne), it sets `Branch` to 1, sets `ALUOp` to 2’b01, and displays the instruction type on the 7-segment displays.
    - For the jump instruction, it sets `Jump` to 1, sets `ALUOp` to 2’b01, and displays “J” on the 7-segment displays.

4. If the instruction does not match any of the defined cases, the Control Unit sets all control signals to their default values and displays blank on the 7-segment displays.

The Control Unit is critial to correctly orchestrating the operation of the single-cycle MIPS processor.

It interprets the instruction and generates the necessary control signals to control the data-path components, such as the ALU, Register File, and Data Memory.

The control signals determine the flow of data and the operations performed in each stage of the processor pipeline.

- The Control Unit receives the instruction from the Instruction Memory.
- It sends control signals to various components, such as the ALU, Register File, and Data Memory, to control their behavior based on the instruction being executed.
- The control signals generated by the Control Unit are used by the data-path components to perform the required operations and route the data accordingly.

Data Memory

The following is the code for the Data Memory module in the MIPS processor called `DataMemory.v`. (It can be found in the `./proj/` directory)

```verilog
`timescale 1ns / 1ps
module DataMemory (
    input i_clk,
    input [31:0] i_addr,
    input [31:0] i_wData,
    input [31:0] i_ALUresult,
    input i_MemWrite,
    input i_MemRead,
    input i_MemtoReg,
    output reg [31:0] o_rData
);
  parameter SIZE_DM = 128;       // size of this memory, by default 128*32
  reg [31:0] Dmem[SIZE_DM-1:0];  // instruction memory
  integer i;
  initial begin
    for (i = 0; i < SIZE_DM; i = i + 1) begin
      Dmem[i] = 32'b0;
    end
  end
  always @(i_addr or i_MemRead or i_MemtoReg or i_ALUresult) begin
    if (i_MemRead == 1) begin
      if (i_MemtoReg == 1) begin
        o_rData = Dmem[i_addr];
      end else begin
        o_rData = i_ALUresult;
      end
    end else begin
      o_rData = i_ALUresult;
    end
  end
  always @(posedge i_clk) begin  // MemWrite, wData, addr
    if (i_MemWrite == 1) begin
      Dmem[i_addr] = i_wData;
    end
  end
endmodule
```

The provided code snippet is the implementation of the Data Memory module (`DataMemory.v`) in the single-cycle MIPS processor.

The Data Memory module serves as the main memory for storing and retrieving data in the processor.

IO

The following are the detailed input and output ports of the Data Memory module, `DataMemory.v`: - `i_clk`: Input clock signal. - `i_addr`: Input address for reading or writing data. - `i_wData`: Input write data to be stored in memory. - `i_ALUresult`: Input ALU result, which can be used as the address or data depending on the control signals. - `i_MemWrite`: Input control signal indicating a memory write operation. - `i_MemRead`: Input control signal indicating a memory read operation. - `i_MemtoReg`: Input control signal indicating whether to pass the memory read data or ALU result to the output. - `o_rData`: Output read data from the memory.

Functionality

Memory Initialization:

- The module defines a parameter `SIZE_DM` representing the size of the data memory (default is 128 words).
- It declares a register array `Dmem` of size `SIZE_DM` to store the memory contents.
- In the initial block, all memory locations are initialized to zero using a loop.

Memory Read Operation:

- The first always block is triggered whenever the input signals `i_addr`, `i_MemRead`, `i_MemtoReg`, or `i_ALUresult` change.
- If `i_MemRead` is asserted (equals 1), it indicates a memory read operation.
  - If `i_MemtoReg` is also asserted, the data at memory location `i_addr` is assigned to the output `o_rData`.
  - Otherwise, the ALU result `i_ALUresult` is assigned to `o_rData`.
- If `i_MemRead` is not asserted, the ALU result `i_ALUresult` is directly assigned to `o_rData`.

Memory Write Operation:

- The second always block is triggered on the positive edge of the clock signal `i_clk`.

- If `i_MemWrite` is asserted (equals 1), it indicates a memory write operation.
- The data `i_wData` is written to the memory location specified by `i_addr`.

Significance in the Processor

Interaction with Other Components: - The Data Memory module interacts with the ALU and the Control Unit in the processor. - The ALU provides the address (`i_ALUresult`) for memory read or write operations. - The Control Unit generates the control signals (`i_MemWrite`, `i_MemRead`, `i_MemtoReg`) to control the behavior of the Data Memory module. - The Register File provides the data to be written to memory (`i_wData`) during a memory write operation. - The output read data (`o_rData`) is passed back to the Register File or used as needed in subsequent stages of the processor pipeline.

Simply put, the Data Memory module allows storing and retrieving data in the MIPS processor.

It responds to memory read and write requests based on the provided address and control signals, and it interacts with other components such as the ALU, Control Unit, and Register File to facilitate data storage and retrieval operations.

Instruction Memory

The following is the code for the Instruction Memory module in the MIPS processor called `InstructionMemory.v`. (It can be found in the `./proj/` directory)

```
`timescale 1ns / 1ps
module InstructionMemory (
    input [31:0] i_Addr,
    output reg [5:0] i_Ctr,           // [31-26]
    output reg [5:0] i_Funcode,       // [5-0]
    output reg [31:0] i_Instruction  // [31-0]
);
  parameter SIZE_IM = 128;            // size of this memory, by default 128*32
  reg [31:0] Imem[SIZE_IM-1:0];       // instruction memory
  integer n;
  initial begin
    for (n = 0; n < SIZE_IM; n = n + 1) begin
      Imem[n] = 32'b11111100000000000000000000000000;
    end
    $readmemb("instructions.mem", Imem);
    i_Instruction = 32'b11111100000000000000000000000000;
  end
  always @(i_Addr) begin
    if (i_Addr == -4) begin          // init
      i_Instruction = 32'b11111100000000000000000000000000;
    end else begin
      i_Instruction = Imem[i_Addr>>2];
    end
    i_Ctr = i_Instruction[31:26];
    i_Funcode = i_Instruction[5:0];
  end
endmodule
```

The provided code represents “the Instruction Memory module `InstructionMemory.v` for the single-cycle MIPS processor.”

Purpose:

The Instruction Memory module is responsible for storing the processor’s instructions and providing them to the other components of the processor.

It acts as a read-only memory (ROM) that holds the program instructions.

IO

Inputs and Outputs:

- `i_Addr` (input, 32-bit): Represents the memory address from which the instruction should be fetched.
- `i_Ctr` (output, 6-bit): Outputs the control bits of the fetched instruction (bits [31:26]).
- `i_Funcode` (output, 6-bit): Outputs the function code of the fetched instruction (bits [5:0]).
- `i_Instruction` (output, 32-bit): Outputs the complete 32-bit fetched instruction.

Functionality

1. The module defines a parameter `SIZE_IM` which represents the size of the instruction memory.

By default, it is set to 128, meaning the memory can hold 128 32-bit instructions.

2. The module declares a register array `Imem` of size `SIZE_IM` to store the instructions.

3. In the initial block:

   - The memory is initialized with a default instruction (32’b11111100000000000000000000000000) using a loop.
   - The instructions are then loaded from a file named “instructions.mem” using the `$readmemb` system task. This file contains the binary representation of the instructions.
   - The `i_Instruction` output is initialized with the default instruction.

4. The module has an “always” block that is triggered whenever the `i_Addr` input changes:

   - If `i_Addr` is equal to -4 (used for initialization), the `i_Instruction` output is set to the default instruction.
   - Otherwise, the instruction is fetched from the `Imem` array using the address `i_Addr` shifted right by 2 bits (assuming word-aligned addresses).

  - The control bits (`i_Ctr`) and function code (`i_Funcode`) are extracted from the fetched instruction and assigned to the respective outputs.

Processor Context

The following is the context of the Instruction Memory module, `InstructionMemory.v` in the single-cycle MIPS processor, `mips.v`:

- The Program Counter (`PC`) module provides the memory address (`i_Addr`) to the Instruction Memory module to fetch the instruction at that address.
- The fetched instruction (`i_Instruction`) is then passed to other components of the processor, such as the Control Unit and the Register File, for further processing and execution.
- The control bits (`i_Ctr`) and function code (`i_Funcode`) are used by the Control Unit to generate appropriate control signals for the processor's data-path.

Significance

The Instruction Memory module is a crucial component of the MIPS processor as it holds the program instructions that the processor executes.

It provides the instructions to the processor's data-path, enabling the processor to perform the desired operations and execute the program stored in the memory.

Summary

Essentially, the Instruction Memory module in the single-cycle MIPS processor acts as a read-only memory that stores the program instructions. It fetches instructions based on the provided memory address and outputs the complete instruction along with its control bits and function code for further processing by other components of the processor.

Verbose Components Code

The following section shows detailed, commented code files for each of the components of the processor.

It includes detailed code comments to better explain the functionality and purpose of each component within the actual verilog code.

Data Memory

The following is the commented code for the Data Memory module in the MIPS processor called `DataMemory.v`:

```
// File: DataMemory.v
// Description: This file contains the data memory module for the MIPS processor.
// Purpose: The data memory stores data values and provides read and write access to the
        processor.
//          It is responsible for handling memory read and write operations based on the control
        signals
//          received from the control unit.
`timescale 1ns / 1ps
module DataMemory (
    input i_clk,                    // Clock input
    input [31:0] i_addr,            // Address input for memory access
    input [31:0] i_wData,           // Write data input
    input [31:0] i_ALUresult,       // ALU result input (used for memory address calculation)
    input i_MemWrite,               // Control signal for memory write operation
    input i_MemRead,                // Control signal for memory read operation
    input i_MemtoReg,               // Control signal for selecting memory or ALU result as the
        output
    output reg [31:0] o_rData       // Read data output
);
  parameter SIZE_DM = 128;           // Size of the data memory (default: 128 * 32 bits)
  reg [31:0] Dmem[SIZE_DM-1:0];      // Data memory array
  integer i;
  // Initialize the data memory
  initial begin
    // Fill the data memory with zeros
    for (i = 0; i < SIZE_DM; i = i + 1) begin
      Dmem[i] = 32'b0;
    end
  end
  // Memory read operation
  always @(i_addr or i_MemRead or i_MemtoReg or i_ALUresult) begin
    if (i_MemRead == 1) begin                // If memory read is enabled
      if (i_MemtoReg == 1) begin             // If MemtoReg is 1, select memory data as output
        o_rData = Dmem[i_addr];              // Read data from the memory array
      end else begin
        o_rData = i_ALUresult;               // If MemtoReg is 0, select ALU result as output
      end
    end else begin
      o_rData = i_ALUresult;                 // If memory read is not enabled, select ALU result
        as output
    end
  end
  // Memory write operation
  always @(posedge i_clk) begin              // Triggered on the positive edge of the clock
```

```
    if (i_MemWrite == 1) begin                  // If memory write is enabled
      Dmem[i_addr] = i_wData;                   // Write data to the memory array
    end
  end
endmodule
```

Interactions with other components: - The `DataMemory` module receives the address (`i_addr`), write data (`i_wData`), and control signals (`i_MemWrite`, `i_MemRead`, `i_MemtoReg`) from the `ControlUnit` and `ALU` modules. - It provides the read data (`o_rData`) to the `RegisterFile` module for store instructions or to the `ALU` for load instructions. - The `i_ALUresult` input is used as the memory address for read and write operations. - The `i_MemWrite` control signal determines whether a memory write operation should be performed. - The `i_MemRead` control signal determines whether a memory read operation should be performed. - The `i_MemtoReg` control signal selects whether the memory data or the ALU result should be output as the read data.

The `DataMemory` module is critical for providing data storage and handling memory read and write operations.

It interacts with the control unit, ALU, and register file to facilitate data movement and manipulation in the processor.

Instruction Memory

Here the `ProgramCounter.v` file with detailed code comments explaining its purpose, functionality, and interactions with other components in the MIPS processor:

```
// File: ProgramCounter.v
// Description: This file contains the program counter module for the MIPS processor.
// Purpose: The program counter keeps track of the current instruction address and updates it
//          to the next instruction address on each clock cycle. It is responsible for providing
//          the address of the instruction to be fetched from the instruction memory.
`timescale 1ns / 1ps
module ProgramCounter (
    input i_Clk,                // Input clock signal
    input [31:0] i_Next,        // Input next instruction address
    output reg [31:0] o_Out     // Output current instruction address
);
  // Initialize the program counter
  initial begin
    o_Out = -4;                 // Set the initial address to -4 (used for reset or
        initialization)
  end
  // Update the program counter on the positive edge of the clock
  always @(posedge i_Clk) begin
    o_Out = i_Next;             // Update the current address with the next address
  end
endmodule
```

Interactions with other components: - The `ProgramCounter` module receives the next instruction address (`i_Next`) from the `NextProgramCounter` module. - It provides the current instruction address (`o_Out`) to the `InstructionMemory` module to fetch the corresponding instruction. - The `ProgramCounter` is updated on the positive edge of the clock signal (`i_Clk`), which is typically connected to the global clock signal of the processor.

The `ProgramCounter` module is a critical component in the MIPS processor pipeline. It keeps track of the current instruction address and updates it on each clock cycle to fetch the next instruction. The program counter ensures the sequential execution of instructions and enables the processor to navigate through the program.

Program Counter

Here the `ProgramCounter.v` file with detailed code comments explaining its purpose, functionality, and interactions with other components in the MIPS processor:

```
// File: ProgramCounter.v
// Description: This file contains the program counter module for the MIPS processor.
// Purpose: The program counter keeps track of the current instruction address and updates it to
        the next address.
//          It is responsible for providing the current instruction address to the instruction
        memory and updating
//          the address based on the next address input.
`timescale 1ns / 1ps
module ProgramCounter (
    input i_Clk,                    // Input clock signal
    input [31:0] i_Next,            // Input next instruction address
    output reg [31:0] o_Out         // Output current instruction address
);
  // Initialize the program counter
  initial begin
    o_Out = -4;                     // Set the initial instruction address to -4
  end
  // Update the program counter on the positive edge of the clock
  always @(posedge i_Clk) begin
    o_Out = i_Next;                 // Update the current instruction address with the next address
  end
endmodule
```

Purpose and Functionality: - The `ProgramCounter` module keeps track of the current instruction address in the MIPS processor. - It is responsible for providing the current instruction address to the instruction memory (`InstructionMemory`) for fetching the corresponding instruction. - The program counter is updated on the positive edge of the clock signal (`i_Clk`). - The next instruction address (`i_Next`) is provided as an input to the module, which is used to update the current instruction address (`o_Out`) on each clock cycle. - The initial value of the program counter is set to -4, which represents the initial state before the first instruction is fetched.

Interactions with other components: - The `ProgramCounter` module receives the next instruction address (`i_Next`) from the `NextProgramCounter` module, which calculates the next address based on the current instruction and control signals. - It provides the current instruction address (`o_Out`) to the `InstructionMemory` module to fetch the corresponding instruction. - The `ProgramCounter` is updated on the positive edge of the clock signal (`i_Clk`), which is typically connected to the main processor clock.

The `ProgramCounter` module is to manage the flow of execution by keeping track of the current instruction address.

It ensures that instructions are fetched and executed in the correct order by updating the address on each clock cycle based on the next address input provided by the `NextProgramCounter` module.

ALU

Here the `ALU.v` file with detailed code comments explaining its purpose, functionality, and interactions with other components in the MIPS processor:

```
// File: ALU.v
// Description: This file contains the Arithmetic Logic Unit (ALU) module for the MIPS processor.
// Purpose: The ALU performs arithmetic and logic operations based on the ALU control signals.
//          It takes two input operands (i_data1 and i_read2/immediate value) and performs the
        specified operation.
//          The ALU also generates a zero flag (o_Zero) to indicate if the result is zero.

`timescale 1ns / 1ps

module ALU (
    input      [31:0] i_data1,        // Input operand 1 (from RegisterFile)
    input      [31:0] i_read2,        // Input operand 2 (from RegisterFile or immediate value)
    input      [31:0] i_Instruction,  // Input instruction (used for sign-extension of immediate
        value)
    input             i_ALUSrc,       // Control signal to select between i_read2 or immediate
        value
    input      [ 3:0] i_ALUcontrol,   // Control signal to specify the ALU operation
    output reg        o_Zero,         // Output zero flag (1 if the ALU result is zero, 0
        otherwise)
    output reg [31:0] o_ALUresult     // Output ALU result
);
  reg [31:0] data2;
  // Determine the second operand based on the ALUSrc control signal
  always @(i_ALUSrc, i_read2, i_Instruction) begin
    if (i_ALUSrc == 0) begin
      data2 = i_read2;                   // Use i_read2 as the second operand
    end else begin
      // Sign-extend the immediate value
      if (i_Instruction[15] == 1'b0) begin
        data2 = {16'b0, i_Instruction[15:0]};  // Zero-extend if the immediate value is positive
      end else begin
        data2 = {{16{1'b1}}, i_Instruction[15:0]};  // Sign-extend if the immediate value is
        negative
      end
    end
  end
  // Perform the ALU operation based on the ALUcontrol signal
  always @(i_data1, data2, i_ALUcontrol) begin
    case (i_ALUcontrol)
      4'b0000:  // AND
        o_ALUresult = i_data1 & data2;
      4'b0001:  // OR
        o_ALUresult = i_data1 | data2;
      4'b0010:  // ADD
        o_ALUresult = i_data1 + data2;
      4'b0110:  // SUB
        o_ALUresult = i_data1 - data2;
      4'b0111:  // SLT (Set Less Than)
        o_ALUresult = (i_data1 < data2) ? 1 : 0;
      4'b1100:  // NOR
        o_ALUresult = ~(i_data1 | data2);
      default: ;
    endcase
    // Set the zero flag if the ALU result is zero
    o_Zero = (o_ALUresult == 0) ? 1 : 0;
  end

endmodule
```

Interactions with other components: - The `ALU` module receives input operands (`i_data1` and `i_read2`) from the `RegisterFile` module. - The `i_ALUSrc` control signal from the `ControlUnit` determines whether the second operand is `i_read2` or an immediate value from the instruction (`i_Instruction`). - The `i_ALUcontrol` signal from the `ALUControl` module specifies the ALU operation to be performed. - The `ALU` module outputs the result (`o_ALUresult`) to the `DataMemory` and `RegisterFile` modules for memory access and register writeback. - The zero flag (`o_Zero`) is used by

Control Unit

Certainly! Here’s a detailed explanation of the Verilog module provided for a Control Unit in a single cycle MIPS processor. Each line of the module is annotated to explain its function and relevance.

```
`timescale 1ns / 1ps
// Defines the time unit as 1 nanosecond and the simulation time precision as 1 picosecond.
module ControlUnit (
    input [31:0] i_instruction,     // 32-bit input for the instruction.
    output reg o_RegDst,            // Determines if rd (1) or rt (0) should be the destination
        register.
    output reg o_Jump,              // Control signal for jumping to an instruction address.
    output reg o_Branch,            // Control signal for branching (beq).
    output reg o_Bne,               // Control signal for branching not equal (bne).
    output reg o_MemRead,           // Enables reading from memory (used by lw).
    output reg o_MemtoReg,          // Determines if the value should come from memory (1) or ALU
        (0).
    output reg [1:0] o_ALUOp,       // Control signal for ALU operation type.
    output reg o_MemWrite,          // Enables writing to memory (used by sw).
    output reg o_ALUSrc,            // Determines if the second ALU operand is an immediate (1) or
        register (0).
    output reg o_RegWrite,          // Enables writing to the register file.
    output reg [6:0] o_seg_first,   // Segment display outputs to visually represent instruction
        types or states.
    output reg [6:0] o_seg_second,  // Each segment holds a 7-segment representation.
    output reg [6:0] o_seg_third,
    output reg [6:0] o_seg_fourth,
    output reg [6:0] o_seg_fifth
);
initial begin
    // Initialize all control signals and display outputs to their default (usually disabled)
        states.
    o_RegDst = 0;
    o_Jump = 0;
    o_Branch = 0;
    o_MemRead = 0;
    o_MemtoReg = 0;
    o_ALUOp = 2'b00; // Default ALU operation, no operation specified.
    o_MemWrite = 0;
    o_ALUSrc = 0;
    o_RegWrite = 0;
    o_seg_first = 7'b1111111;  // All segments off (blank).
    o_seg_second = 7'b1111111;
    o_seg_third = 7'b1111111;
    o_seg_fourth = 7'b1111111;
    o_seg_fifth = 7'b1111111;
end
always @(i_instruction) begin
    // Control logic triggered by any change in the instruction input.
    case (i_instruction[31:26]) // Decode the opcode part of the instruction.
      6'b000000: begin  // ARITHMETIC (R-type instructions)
        o_RegDst = 1;
        o_ALUSrc = 0;
        o_MemtoReg = 0;
        o_RegWrite = 1;
        o_MemRead = 0;
        o_MemWrite = 0;
        o_Branch = 0;
        o_Bne = 0;
        o_ALUOp = 2'b10; // Specific ALU operation for arithmetic.
        o_Jump = 0;
        // Display setup for ARITHMETIC.
        o_seg_first =  7'b0001000;  // A
        o_seg_second = 7'b1111010;  // R
        o_seg_third =  7'b1111001;  // I
        o_seg_fourth = 7'b0001111;  // T
        o_seg_fifth =  7'b0001001;  // H
      end
      6'b001000: begin  // addi
        o_RegDst = 0;
        o_ALUSrc = 1;
        o_MemtoReg = 0;
        o_RegWrite = 1;
        o_MemRead = 0;
        o_MemWrite = 0;
        o_Branch = 0;
        o_Bne = 0;
        o_ALUOp = 2'b00;
        o_Jump = 0;
        o_seg_first = 7'b0001000;  // A
        o_seg_second = 7'b1000010; // d
        o_seg_third = 7'b1000010;  // d
        o_seg_fourth = 7'b1001111; // i
        o_seg_fifth = 7'b1111111;  // Blank
      end
      6'b001100: begin  // andi
        o_RegDst = 0;
        o_ALUSrc = 1;
        o_MemtoReg = 0;
        o_RegWrite = 1;
        o_MemRead = 0;
        o_MemWrite = 0;
        o_Branch = 0;
        o_Bne = 0;
        o_ALUOp = 2'b11;
        o_Jump = 0;
        o_seg_first = 7'b0001000;  // A
        o_seg_second = 7'b0101011;  // n
        o_seg_third = 7'b1000010;  // d
        o_seg_fourth = 7'b1001111;  // i
        o_seg_fifth = 7'b1111111;  // Blank
      end
      6'b100011: begin  // lw
        o_RegDst = 0;
        o_ALUSrc = 1;
        o_MemtoReg = 1;
        o_RegWrite = 1;
        o_MemRead = 1;
        o_MemWrite = 0;
        o_Branch = 0;
        o_Bne = 0;
        o_ALUOp = 2'b00;
        o_Jump = 0;
        o_seg_first = 7'b1000111;  // L
        o_seg_second = 7'b1001001;  // w
        o_seg_third = 7'b1111111;  // Blank
        o_seg_fourth = 7'b1111111;  // Blank
        o_seg_fifth = 7'b1111111;  // Blank
      end
      6'b101011: begin  // sw
        o_RegDst = 0;  // X
        o_ALUSrc = 1;
        o_MemtoReg = 0;  // X
        o_RegWrite = 0;
        o_MemRead = 0;
        o_MemWrite = 1;
        o_Branch = 0;
        o_Bne = 0;
        o_ALUOp = 2'b00;
        o_Jump = 0;
        o_seg_first = 7'b0010010;  // S
        o_seg_second = 7'b1001001;  // w
        o_seg_third = 7'b1111111;  // Blank
        o_seg_fourth = 7'b1111111;  // Blank
        o_seg_fifth = 7'b1111111;  // Blank
      end
      6'b000100: begin  // beq
        o_RegDst = 0;  // X
        o_ALUSrc = 0;
        o_MemtoReg = 0;  // X
        o_RegWrite = 0;
        o_MemRead = 0;
        o_MemWrite = 0;
        o_Branch = 1;
        o_Bne = 0;
        o_ALUOp = 2'b01;
        o_Jump = 0;
        o_seg_first = 7'b1100000;  // b
        o_seg_second = 7'b0110000;  // e
        o_seg_third = 7'b0001100;  // q
        o_seg_fourth = 7'b1111111;  // Blank
        o_seg_fifth = 7'b1111111;  // Blank
      end
      6'b000101: begin  // bne
        o_RegDst = 0;  // X
        o_ALUSrc = 0;
        o_MemtoReg = 0;  // X
        o_RegWrite = 0;
        o_MemRead = 0;
        o_MemWrite = 0;
        o_Branch = 1;
```

```
        o_Bne = 1;
        o_ALUOp = 2'b01;
        o_Jump = 0;
        o_seg_first = 7'b1100000;  // b
        o_seg_second = 7'b0101011; // n
        o_seg_third = 7'b0110000;  // e
        o_seg_fourth = 7'b1111111; // Blank
        o_seg_fifth = 7'b1111111;  // Blank
      end
      6'b000010: begin  // j
        o_RegDst = 0;  // X
        o_ALUSrc = 0;
        o_MemtoReg = 0;  // X
        o_RegWrite = 0;
        o_MemRead = 0;
        o_MemWrite = 0;
        o_Branch = 0;
        o_Bne = 0;
        o_ALUOp = 2'b01;
        o_Jump = 1;
        o_seg_first = 7'b1100001;  // J
        o_seg_second = 7'b1111111; // Blank
        o_seg_third = 7'b1111111;  // Blank
        o_seg_fourth = 7'b1111111; // Blank
        o_seg_fifth = 7'b1111111;  // Blank
      end
      default: begin
        // Default case sets all outputs to zero or disables them, providing a safe default state.
        o_RegDst = 0;
        o_ALUSrc = 0;
        o_MemtoReg = 0;
        o_RegWrite = 0;
        o_MemRead = 0;
        o_MemWrite = 0;
        o_Branch = 0;
        o_Bne = 0;
        o_ALUOp = 2'b00;
        o_Jump = 0;
        // Display all segments off for undefined instructions.
        o_seg_first = 7'b1111111;  // Blank
        o_seg_second = 7'b1111111; // Blank
        o_seg_third = 7'b1111111;  // Blank
        o_seg_fourth = 7'b1111111; // Blank
        o_seg_fifth = 7'b1111111;  // Blank
      end
    endcase
end

endmodule
```

This code serves as the control logic for a single cycle MIPS processor, managing the routing and operations of data based on the instruction being executed. It adjusts the path and operation of the data in various parts of the processor according to the opcode of the instruction, with added visual output for debugging or educational purposes through a 7-segment display configuration.

Testbench

Here is the `mips_tb.v` file with detailed code comments explaining its purpose, functionality, and interactions with other components in the MIPS processor:

```
// File: mips_tb.v
// Description: This file contains the testbench for the MIPS processor.
// Purpose: The testbench is used to simulate and verify the functionality of the MIPS processor.
//          It instantiates the MIPS processor module, provides clock and reset signals, and
//          initializes the data memory and register file. It also displays the output on 7-
        segment displays.
`timescale 1ns / 1ps
`define CYCLE_TIME 20
module mips_tb;
  reg clk;                 // Clock signal
  reg rst;                 // Reset signal
  // Segments for the 7-segment displays
  wire [6:0] seg_first, seg_second, seg_third, seg_fourth, seg_fifth;
  integer i;               // Loop variable
  // Generate clock signal
  always #(`CYCLE_TIME / 2) clk = ~clk;
  // Instantiate the MIPS processor module
  mips uut (
      .i_Clk(clk),
      .i_Rst(rst),
      .o_Seg_first(seg_first),
      .o_Seg_second(seg_second),
      .o_Seg_third(seg_third),
      .o_Seg_fourth(seg_fourth),
      .o_Seg_fifth(seg_fifth)
  );
  // Initialize data memory and register file
  initial begin
    // Initialize data memory
    for (i = 0; i < 32; i = i + 1) begin
      uut.inst_DataMemory.Dmem[i] = 32'b0;
    end
    // Initialize register file
    for (i = 0; i < 32; i = i + 1) begin
      uut.inst_RegisterFile.RegData[i] = 32'b0;
    end
    clk = 0;                // Initialize clock signal
  end
  initial begin
    #1800 $finish;
  end
endmodule
```

Interactions with other components:

- The `mips_tb` module instantiates the `mips` module, which represents the MIPS processor.
- It provides the clock signal (`clk`) to the `mips` module for synchronization.
- The reset signal (`rst`) is not used in this testbench but can be used to reset the processor if needed.
- The testbench initializes the data memory (`inst_DataMemory.Dmem`) and register file (`inst_RegisterFile.RegData`) of the `mips` module to zero.
- The 7-segment display outputs (`seg_first`, `seg_second`, `seg_third`, `seg_fourth`, `seg_fifth`) from the `mips` module are connected to the testbench for monitoring purposes.

The `mips_tb` module serves as a testbench to simulate and verify the functionality of the MIPS processor.

It provides the necessary inputs (clock and reset) and initializes the memory and registers. The testbench can be modified to apply different test cases through loading different binary converted assembly files and allows one to monitor the processor's behavior through the 7-segment display outputs.

Conclusion

In this final project for CPRE281, I successfully designed and implemented a single-cycle MIPS processor in Verilog. The processor is capable of executing a subset of the MIPS instruction set, including arithmetic, logical, memory, and control flow instructions. The processor features additional components such as a clock divider and displaying the current instruction on the seven-segment displays present on the FPGA board.

Throughout the project, I gained valuable experience in designing and implementing a processor using Verilog. I learned about the different stages of the processor pipeline, including instruction fetch, decode, execute, memory access, and write-back. I also gained a deeper understanding of the MIPS instruction set architecture and how instructions are encoded and executed.

Implementing the processor in Verilog allowed me to apply my knowledge of digital design and hardware description languages. I utilized various Verilog constructs, such as modules, always blocks, and case statements, to model the behavior of the processor components. I also learned about the importance of proper synchronization and timing in hardware design.

To ensure the correctness of the processor implementation, I developed test-benches in Verilog to verify the functionality of individual components as well as the overall processor. The test-benches allowed me to simulate the processor's behavior and debug any issues that arose during the development process.

In addition to the Verilog implementation, I also compared and contrasted my experience with writing a similar single-cycle processor in VHDL as part of CPRE381. This comparison provided insights into the differences and similarities between the two hardware description languages. While Verilog offers a more concise and flexible syntax, VHDL provides stronger typing and more explicit component instantiation. Both languages have their strengths and weaknesses, and the choice between them often depends on the specific project requirements and personal preference.

Throughout the project, I utilized various tools and technologies to aid in the development process. I used Quartus Prime for synthesizing and programming the FPGA board, ModelSim for simulation and debugging, and version control systems like Git for managing the project files. I also leveraged language servers and integrated development environments to enhance my coding experience and productivity.

In conclusion, this final project for CPRE281 has been a valuable learning experience. It has deepened my understanding of processor design, hardware description languages, and the MIPS instruction set architecture. The project has also provided hands-on experience in implementing a functional processor using Verilog and comparing it with a VHDL implementation. The skills and knowledge gained from this project will be beneficial for future endeavors in the field of computer engineering and hardware design.