Multi-Cycle Processor analysis

* COSE222 – 02 Assignment Report

Lee Jeong Min
2023320060
Computer Science & Engineering
Korea University
a23493307@gmail.com

Kim tae kwan
2023320135
Computer Science & Engineering
Korea University
pollarmagic@gmail.com

***Abstract*— The multi-cycle processor architecture divides instruction execution into multiple stages across several clock cycles, optimizing hardware utilization by reusing shared components. This approach enhances resource efficiency while accommodating the varying complexities of instruction types, including R-type, I-type, SB-type, and S-type. This report examines the design, operation, and performance characteristics of a multi-cycle processor, focusing on instruction execution stages and control signal coordination. By analyzing key aspects such as instruction fetch, decode, execution, memory access, and write-back, we highlight the trade-offs between hardware simplicity, cycle efficiency, and instruction latency. The findings provide valuable insights into how multi-cycle processors serve as a foundation for advanced concepts like pipelining.**

Keywords*— *Multi-cycle Processor, Instruction Execution, RISC-V Architecture, Resource Utilization, Pipeline Design, Control Signals, Instruction Stages, Hardware Optimization

I. INTRODUCTION

The multi-cycle processor architecture represents a refined approach to instruction execution in computer systems, where each instruction is divided into multiple stages—such as fetch, decode, execute, memory access, and write-back—executed across several clock cycles. This design offers improved resource utilization and flexibility by allowing shared hardware components to perform different tasks in different clock cycles. The multi-cycle processor serves as an advanced model for understanding the complexities of instruction execution, control signal coordination, data path organization, and timing optimization.

In recent years, the study of multi-cycle architectures has retained its pedagogical importance in computer architecture courses, providing students and engineers with practical insights into the trade-offs between hardware complexity and cycle efficiency. This model offers a comprehensive platform for analyzing critical design considerations, such as minimizing cycle count per instruction, optimizing data path utilization, and balancing control logic overhead with overall performance.

This report analyzes the design, operation, and performance characteristics of a multi-cycle processor, examining how its architecture handles different instruction types. It includes Action for R-type Instructions, Action for Memory Instructions and Action for Branches. By understanding its strengths and limitations, we can better appreciate the trade-offs that lead to more advanced techniques such as pipelining.

II. HOW A MULTI-CYCLE PROCESSOR OPERATES

A. What is a multi-cycle processor?

A multi-cycle processor divides the execution of instructions into multiple stages, with each stage performed in a separate clock cycle. This design reuses hardware resources, optimizing efficiency and reducing hardware requirements. The execution process is divided into five main stages: Instruction Fetch (IF), Instruction Decode and Register Fetch (ID/RF), Execute and Address Calculation (EX), Memory Access (MEM), and Register Write-back (WB). Each stage is responsible for specific tasks, enabling sequential execution of arithmetic operations, memory access, and branch instructions. The control unit manages each stage independently, generating the necessary control signals for each clock cycle. This design approach allows efficient resource utilization by sharing and reusing hardware components while dynamically allocating the required number of clock cycles for different instructions, thereby improving overall performance. However, it introduces complexities in control logic and requires the clock period to align with the slowest stage, which can pose challenges in design. Compared to a single-cycle processor, a multi-cycle processor reduces hardware cost while processing a broader range of instructions more efficiently. Detailed descriptions of each stage are provided in section B, Detailed descriptions of each stage.

B. Detailed descriptions of each stage

The operation of a multi-cycle processor consists of five stages: instruction fetch, instruction decode and register fetch, execution, memory access, and write-back. Table 1 presents these five stages along with their description.

TABLE 1. EXECUTION STAGES (MULTICYCLE DIVISIONS) OF A MULTI-CYCLE PROCESSOR

Execution Stages	Step Description	
	Stage Name	***Description***
Phase1	Instruction Fetch	Obtain instruction from instruction memory
Phase2	Instruction Decode and Register Fetch	Locate and obtain operand data
Phase3	Execute Operation	Compute result value or status
Phase4	Memory Access	Load and/or store values from/to memory
Phase5	Writeback	Deposit results in storage for later use

The following provides additional explanations for each stage. These additional explanations include simple example codes and examples that will be covered in more detail later for each stage.

1) Instruction Fetch (IF)

In the IF stage, the instruction is fetched from memory. The Program Counter (PC) provides the address of the instruction to be executed, which is sent to the memory address register (MAR). The instruction is then retrieved from memory and stored in the Instruction Register (IR). Simultaneously, the PC is incremented by 4 to prepare for the next instruction. For example, if the PC points to address 0x1000, the instruction at this address is fetched, and the PC is updated to 0x1004. The code below is a simple example.

```
IR = Memory[PC]
PC = PC + 4
```

2) Instruction Decode and Register Fetch (ID/RF)

In the ID stage, the processor decodes the instruction to determine its type and retrieves the values of the source registers specified in the instruction. For instance, in the instruction lw $t0, 4($t1), the base register $t1 is read, and the immediate value 4 is extracted and sign-extended to calculate the effective address. This stage also prepares control signals for subsequent stages based on the instruction's opcode. Below is a simple example code.

```
A = Reg[IR[19-15]]
B = Reg[IR[24-20]]
ALUOut = PC + Branch Label (i.e., sign-extend (imm12 << 1))
```

3) Execute and Address Calculation (EX)

The EX stage performs arithmetic or logical operations or computes the effective address for memory access. For arithmetic instructions such as add $t0, $t1, $t2, the ALU adds the values of $t1 and $t2, and then save this result in $t0. For memory instructions like lw $t0, 4($t1), the effective memory address is calculated as the sum of $t1 and the immediate value 4. For branch instructions such as beq $t1, $t2, label, the ALU evaluates the condition ($t1 == $t2) and calculates the target address if the branch is taken. Below is a simple example code.

Action for R-type Instructions	Action for Memory Instructions	Action for Branches
ALUOut = A op B	*ALUOut = A + sign-extend(imm12)*	*If (A==B) then PC = ALUOut*

4) Memory Access (MEM)

The MEM stage involves reading from or writing to the memory. For a load instruction like lw $t0, 4($t1), Read the value added by 4 to the $t1 register and store it in $t0. For a store instruction such as sw $t0, 8($t1), load the value of $t0 and store it at the memory address of $t1 plus 8. Below is a simple example code.

Action for R-type Instructions	Action for Memory Instructions
Reg[IR[11-7]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B

5) Register Write-back (WB)

In the WB stage, the results of the operation are written back to the appropriate register. For arithmetic instructions, the ALU result is stored in the destination register. For memory instructions, the data fetched during the MEM stage is written to the specified register. For example, in lw $t0, 4($t1), the data fetched from memory is stored in $t0. Below is a simple example code.

Action for Memory Instructions
Load: Reg[IR[11-7]] = MDR

C. Additional Considerations

In fact, Phase 2(Instruction Decode and Register Fetch) may not be necessary for R-type or memory data transfer instructions (load/store). This raises the question of why Phase 2 is executed uniformly across all instructions. In Phase 2, instruction decoding is not yet complete because this task requires more time than one might initially expect depending on the complexity of the instruction. In other words, the processor does not yet recognize the specific instruction currently being executed. Therefore, up to this phase, all instructions must perform the same operations. At this point, the processor designer faces a critical decision: which steps should be designated as common phases to maximize the processor's efficiency? After thorough consideration, the current structure was developed, reflecting the optimal balance between commonality and efficiency. This also explains why the example instruction table in the earlier Section B varied slightly depending on the specific phase for each instruction type.

III. ANALYZE IN DETAIL HOW THE INSTRUCTION IS EXECUTED

Below is a detailed breakdown of how each instruction type (R-type, I-type, B-type, and S-type) is executed in a multi-cycle processor, with an emphasis on RTL viewer-based module interactions, inputs/outputs, and comparisons to single-cycle execution. Each instruction is analyzed in **separate execution stages** as observed in a multi-cycle processor design. Before starting the code analysis, a table presenting the colors used in the RTL view diagram along with additional explanations is provided.

Color	***Stage Division***
Blue Line	Instruction Fetch (IF)
Light Pink Line	Instruction Decode and Register Fetch (ID/RF)
Yellow Line	Execute and Address Calculation (EX)
Pink Line	Memory Access (MEM)
Light Blue Line	Register Write-back (WB)

A. *R-type: add, sub, and, or*

R-Type instructions in the RISC-V architecture perform arithmetic or logical operations on register values. A prime example is the ADD instruction, which adds the values of two source registers and stores the result in a destination register. This report outlines the execution of the ADD instruction, detailing its stages in a pipeline and the differences between single-cycle and multi-cycle implementations. The examples are as follows:

add x1, x2, x3

The example code performs the operation: Adds the values in registers x2 and x3 and stores the result in x1. The following is an analysis of the R-Type instruction process broken down by each stage.

1) *Instruction Fetch (IF)*

In the Instruction Fetch (IF) stage, the instruction is retrieved from memory using the Program Counter (PC), which is then incremented by 4 to point to the next instruction, producing the fetched instruction and the updated PC value as outputs. The process can be represented in a table as follows.

Inputs	Program Counter (PC) Memory
Operations	The instruction is fetched from memory using the address stored in the PC. The PC is then incremented to point to the next instruction (PC + 4 in RISC-V).
Outputs	The fetched instructions. Updated PC value (PC + 4).

Below is the code from the RV321CPU.v file that corresponds to the role of the respective stage.

```
`define OP_R      7'b0110011

IF: begin
   controls <= 11'b00_10_11_00_000;
   ALUOp <= 1;
end
```

First, define the opcode for the R-type instructions. Second, Set the 11-bit control signal to determine the flow of instruction processing. The meaning of each part of the control signal is as follows:

- **00**: Initialization signal
- **10_11**: ALU operation selection
- **00_000**: Memory read/write and other control signals

Then, set ALUOp to 1 to perform the PC + 4 operation, ensuring sequential execution of instructions.

2) *Instruction Decode and Register Fetch (ID/RF)*

In the Decode and Register Fetch (ID) stage, the fetched instruction is decoded to determine the source (rs1, rs2) and destination (rd) registers, load values from x2 and x3 into rs1_data and rs2_data, and generate control signals for subsequent stages. The process can be represented in a table as follows.

Inputs	Fetched instruction
Operations	The instruction is decoded to identify the source (rs1, rs2) and destination (rd) registers. Values from the registers specified by x2 and x3 are read into rs1_data and rs2_data. Control signals for subsequent stages are generated based on the decoded instruction.
Outputs	rs1_data: Value in x2 rs2_data: Value in x3 Control signals

Below is the code from the RV321CPU.v file that corresponds to the role of the respective stage.

```
IF:
   begin
      n_state <= Decode;
end

Decode:
   if (opcode == `OP_R) n_state <= ExR;
```

The begin block signifies that after fetching an instruction from memory, the state transitions to decoding it. When in the Decode state, the opcode is checked, and if it corresponds to an R-type instruction, the state transitions to execution for R-type instructions.

3) *Execute and Address Calculation (EX)*

In the Execute (EX) stage, the ALU, guided by the control signal, performs an addition operation (ALUOp = add) on rs1_data and rs2_data, producing their sum as the ALU result. The process can be represented in a table as follows.

Inputs	rs1_data, rs2_data ALU control signal
Operations	The ALU performs the addition operation (ALUOp = add) on the values in rs1_data and rs2_data.
Outputs	ALU result: Sum of rs1_data and rs2_data

Below is the code from the RV321CPU.v file that corresponds to the role of the respective stage.

```
localparam ExR = 4'b0110;
```

Define ExR as a localparam for R-type execution. This represents the state in the instruction pipeline where R-type instructions are processed.

```
ExR: begin // Line 70
   controls <= 11'b10_00_00_00_000;

   ALUOp <= 0;
```

Set control signals to provide the necessary inputs for the ALU to execute R-type instructions. Use ALUOp to configure the ALU's operational mode.

```
case(opcode)
   `OP_R:

   begin

   case({funct7,funct3})

      10'b0000000_000: alucontrol <= 5'b00000; // addition (add)

      10'b0100000_000: alucontrol <= 5'b10000; // subtraction (sub)

      10'b0000000_111: alucontrol <= 5'b00001; // AND (and)

      10'b0000000_110: alucontrol <= 5'b00010; // OR (or)

      10'b0000000_100: alucontrol <= 5'b00011; // XOR (xor)

   default:       alucontrol <= 5'bxxxxx;

   endcase
```

Based on funct7 and funct3, determine whether the operation is add, sub, and, or, or xor, and configure the ALU control signal accordingly.

```
alu i_alu(
.a          (alusrc1), // ALU input 1

.b          (alusrc2), // ALU input 2

.alucont    (alucontrol), // ALU control signal

.result     (aluout), // ALU result

.N          (Nflag), // Negative, Zero, Carry, Overflow flags

.Z          (Zflag),

.C          (Cflag),

.V          (Vflag));
```

4) ***R-Type Completion (WB)***

In the Write Back (WB) stage, the ALU result is written to the destination register x1 in the register file, updating it with the computed value. The process can be represented in a table as follows.

Inputs	ALU result
Operations	The ALU result is written back to the destination register x1 in the register file.
Outputs	Updated register file with x1 containing the result.

Below is the code from the RV321CPU.v file that corresponds to the role of the respective stage.

```
ExR: begin

   n_state <= ALUWB;

end
```

For R-type instructions, skip the memory (MEM) stage and transition directly to the write-back (WB) stage.

```
ALUWB: begin

   controls <= 11'b00_00_00_10_100;

end
```

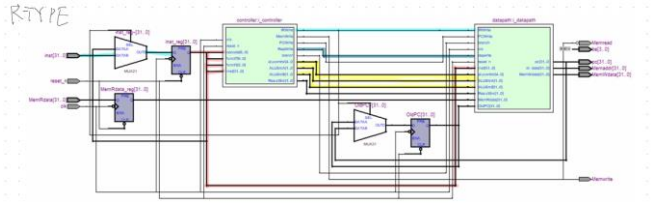
Enable RegWrite (10) to store the R-type instruction result into a register and select the destination register to store the data.

```
always @(*) begin

   if (ResultSrc == 2'b00) rd_data = aluout;

end
```

In the WB stage, decide the data to be written to the register based on the ALU output.

```
regfile i_regfile(

.clk      (clk),

.we       (regwrite),    // RegWrite signal

.rs1      (rs1),

.rs2      (rs2),

.rd       (rd),          // Destination register

.rd_data  (rd_data),     // Data to be written

.rs1_data (rs1_data),

.rs2_data (rs2_data)

);
```

This code completes the WB stage, ensuring the result is stored in the memory/register. The RTL diagram based on the code analysis is as follows.



For the remaining part, we have analyzed the differences between single-cycle and multi-cycle processors for R-type instructions.

a) ***Single-Cycle vs. Multi-Cycle Implementations***

In a single-cycle implementation, all execution steps are completed within a single clock cycle, which simplifies the control logic but requires high-performance hardware to ensure that all operations finish within one cycle, potentially leading to inefficiencies for complex instructions. On the other hand, a multi-cycle implementation divides execution across multiple clock cycles, with each step (IF, ID, EX, WB) occupying a separate cycle, allowing for resource sharing (e.g., reusing the ALU) and improved efficiency for varying instruction complexities, but at the cost of increased control complexity and longer instruction latency.

B. ***L-type: load***

L-Type instructions in the RISC-V architecture are used for operations involving immediate values or memory access. The LW (Load Word) instruction is a key example, where a 32-bit word is loaded from memory into a register. This report describes the execution stages of the LW instruction and compares its implementation in single-cycle and multi-cycle designs. The examples are as follows:

lw x1, 0(x2)

The example code performs the operation: Loads the 32-bit word from the memory address calculated as the sum of the value in x2 and the immediate offset 0, storing the result in x1. The following is an analysis of the L-Type instruction process broken down by each stage.

1) ***Instruction Fetch (IF)***

In the Instruction Fetch (IF) stage, the instruction is retrieved from memory using the Program Counter (PC), which is then incremented by 4 to point to the next instruction, resulting in the fetched instruction and the updated PC value (PC + 4). This stage is identical to that of R-Type instructions. The process can be represented in a table as follows.

Inputs	Program Counter (PC) Memory
Operations	The instruction is fetched from memory using the PC. The PC is incremented by 4 to point to the next instruction.
Outputs	The fetched instructions. Updated PC value (PC + 4).

Below is the code from the RV321CPU.v file that corresponds to the role of the respective stage.

```
`define OP_I_Load (7'b0000011)

localparam IF = 4'b0000;

IF: begin
  controls <= 11'b00_10_11_00_000; // PC + 4 operation
  ALUOp <= 1;                      // Perform PC increment operation
end
```

This defines the OP_I_Load opcode for Load instructions in the RISC-V architecture. It specifies that the first 7 bits of an instruction with the value 0000011 corresponds to Load-type instructions. And set the 11-bit control signal to configure the instruction fetch process. The meaning of each part of the control signal is as follows:

- **00**: Select PC as the first ALU input.
- **10_11**: Configure the second ALU input as 4 and enable instruction register write.
- **00_000**: Set default values for memory and register control signals.

Then, set **ALUOp** to 1 to perform the PC + 4 operation, ensuring the program counter increments correctly for sequential instruction execution.

2) ***Instruction Decode and Register Fetch (ID/RF)***

In the Decode and Register Fetch (ID) stage, the fetched instruction is decoded to identify the source register (rs1), destination register (rd), and immediate offset, with the offset being sign-extended to 32 bits, while the value from x2 is loaded into rs1_data, and control signals are generated for subsequent stage. The process can be represented in a table as follows.

Inputs	Fetched instruction
Operations	The instruction is decoded to determine the source register (rs1), destination register (rd), and the immediate offset. The offset is sign-extended to a 32-bit value. Additionally, the value from x2 is read into rs1_data.
Outputs	rs1_data: Value in x2 Sign-extended immediate value Control signals

Below is the code from the RV321CPU.v file that corresponds to the role of the respective stage.

```
localparam Decode = 4'b0001;

Decode: begin
  controls <= 11'b01_01_00_00_000; // Set ALU inputs to calculate address
  ALUOp <= 1;                      // Perform addition for address calculation
  if (opcode == `OP_I_Load)
    n_state <= MemAdr;             // Transition to the memory address calculation state
end
```

First, set the 11-bit control signal to configure the instruction decode process. The meaning of each part of the control signal is as follows:

- **01**: Select OldPC as the first ALU input.
- **01**: Select the immediate (imm) as the second ALU input.
- **00_00_000**: Set default values for memory and register control signals.
-

Then, set **ALUOp** to 1 to perform the addition operation for address calculation (OldPC + imm). If the opcode matches OP_I_Load, transition to the **MemAdr** state for memory address computation.

*3) **Execute and Address Calculation (EX)***

In the Address Calculation (EX) stage, the ALU computes the effective memory address by adding the base address (rs1_data from x2) to the sign-extended immediate value, producing the calculated memory address as output. The process can be represented in a table as follows.

Inputs	rs1_data (Base address from x2) Sign-extended immediate
Operations	The ALU calculates the effective memory address by adding the base address (rs1_data) to the immediate offset.
Outputs	Effective memory address

Below is the code from the RV321CPU.v file that corresponds to the role of the respective stage.

```
localparam MemAdr = 4'b0010;

MemAdr: begin
  controls <= 11'b10_01_00_00_000; // ALU calculates rs1 + imm
  ALUOp <= 1;                      // Perform addition operation for address calculation
end
```

First, set the 11-bit control signal to configure the memory address calculation process. The meaning of each part of the control signal is as follows:

- **10**: Select rs1 as the first ALU input.
- **01**: Select the immediate (imm) as the second ALU input.
- **00_00_000**: Set default values for memory and register control signals.

Then, set **ALUOp** to 1 to perform the addition operation (rs1 + imm), calculating the effective memory address for Load/Store instructions. The remaining code below is related to ALU control and execution.

ALU Configuration:

For Load instructions, the ALU is set to perform an addition (ADD) to compute the address.

```
case(opcode)
  `OP_I_Load: // Load instruction
  begin
    alucontrol <= 5'b00000; // Set ALU control for addition (ADD)
  end
endcase
```

The ALU performs the calculation rs1 + imm to generate the memory address. The result is stored in aluout.

```
alu i_alu(
  .a          (alusrc1),    // First ALU input (rs1)
  .b          (alusrc2),    // Second ALU input (imm)
  .alucont    (alucontrol), // ALU operation control
  .result     (aluout),     // ALU output (calculated address)
  .N          (Nflag),
  .Z          (Zflag),
  .C          (Cflag),
  .V          (Vflag));
```

*4) **Memory Access (MEM)***

In the Memory Access (MEM) stage, the memory is accessed at the calculated address, and the 32-bit word retrieved from that location is output as data. The process can be represented in a table as follows.

Inputs	Effective memory address
Operations	The memory is accessed at the calculated address, and the 32-bit word at that location is read.
Outputs	Data retrieved from memory

Below is the code from the RV321CPU.v file that corresponds to the role of the respective stage.

```
localparam MemRead = 4'b0011;

MemRead: begin
  controls <= 11'b00_00_00_10_000; // Enable memory read
  n_state <= MemWB;                // Transition to Write Back state
end
```

Set the 11-bit control signal to configure the memory read operation. The meaning of each part of the control signal is as follows:

- **00_00_00**: Set ALU and other inputs to default values as they are not needed for memory read.
- **10**: Enable memory read to fetch data from the calculated address.
- **000**: Set default values for branch, register, and memory write controls.

After completing the memory read operation, transition to the **MemWB** state to prepare for writing the fetched data to the register.

5) ***Register Write-back (WB)***

In the Write Back (WB) stage, the data retrieved from memory is written to the destination register x1, updating the register file with the loaded value. The process can be represented in a table as follows.

Inputs	Data retrieved from memory
Operations	The data from memory is written to the destination register x1.
Outputs	Updated register file with x1 containing the loaded value

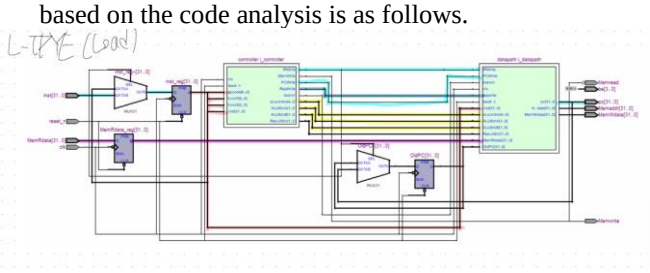
Below is the code from the RV321CPU.v file that corresponds to the role of the respective stage.

```
localparam MemWB = 4'b0100;

MemWB: begin
  controls <= 11'b00_00_00_01_100; // Enable register write
end

...

always @(*)
begin
  if (ResultSrc == 2'b01)  // Select memory read data
    rd_data = MemRdata;
end
```

Set the 11-bit control signal to configure the Write Back operation. The meaning of each part of the control signal is as follows:

- **00_00_00**: Set ALU and memory controls to default values as no computation or memory operation is needed.
- **01**: Enable register write to store the fetched data into the destination register.
- **100**: Select memory read data (MemRdata) as the source for writing to the register.

And then, when ResultSrc is 2'b01, the memory read data (MemRdata) is written into the destination register (rd) in the register file, ensuring the value fetched during the MEM stage is correctly stored. The RTL diagram based on the code analysis is as follows.



For the remaining part, we have analyzed the differences between single-cycle and multi-cycle processors for L-type instructions.

a) ***Single-Cycle vs. Multi-Cycle Implementations***

In a single-cycle implementation, all stages (IF, ID, EX, MEM, WB) are completed within one clock cycle, offering simplified design and control logic but risking inefficiencies by combining memory access and register write-back in the same cycle, whereas in a multi-cycle implementation, stages are spread across multiple cycles, reducing critical path delay and enabling resource sharing, albeit with increased control complexity and longer instruction latency.

C. ***SB-type: BEQ***

SB-Type instructions in the RISC-V architecture are used for conditional branching, enabling program control to jump to a specific instruction address based on a condition. The BEQ (Branch if Equal) instruction is a fundamental example that compares two register values and updates the Program Counter (PC) to the branch target address if they are equal. This report details the execution stages of the BEQ instruction and compares its single-cycle and multi-cycle implementations. The examples are as follows:

beq x1, x2, offset

The example code performs the operation: If the values in x1 and x2 are equal, the PC is updated to the branch target address calculated as PC + offset; otherwise, the PC is incremented by 4. The following is an analysis of the B-Type instruction process broken down by each stage.

1) ***Instruction Fetch (IF)***

In the Instruction Fetch (IF) stage, the instruction is retrieved from memory using the Program Counter (PC), which is then incremented by 4 to point to the next instruction, resulting in the fetched instruction and the updated PC value, identical to the process in R-Type instructions. The process can be represented in a table as follows.

Inputs	Program Counter (PC) Memory
Operations	The instruction is fetched from memory using the PC, which is then incremented by 4 to point to the next instruction.
Outputs	The fetched instructions. Updated PC value (PC + 4).

Below is the code from the RV321CPU.v file that corresponds to the role of the respective stage.

```
`define OP_B        7'b1100011

IF: begin
    controls <= 11'b00_10_11_00_000;
    ALUOp <= 1; // add PC + 4
end
```

This defines the opcode for **B-type instructions** in the RISC-V ISA. Set the 11-bit control signal to configure the Instruction Fetch (IF) operation. The meaning of each part of the control signal is as follows:

- **00_10_11**: Configure ALU sources and enable instruction register write (**IRWrite**) for fetching the instruction.
- **00**: Disable register write and memory write as no memory or register operation is required in this stage.
- **000**: Default branch and control settings for sequential instruction execution.

When **ALUSrcA** is 00 and **ALUSrcB** is 10, the ALU computes PC + 4, preparing the address of the next sequential instruction, ensuring the pipeline continues smoothly.

2) ***Instruction Decode and Register Fetch (ID/RF)***

In the Decode and Register Fetch (ID) stage, the fetched instruction is decoded to determine the source registers (rs1 and rs2) and the branch offset, which is sign-extended to 32 bits, while the values from x1 and x2 are loaded into rs1_data and rs2_data, respectively. The process can be represented in a table as follows.

Inputs	Fetched instruction
Operations	The instruction is decoded to identify the source registers (rs1 and rs2) and the branch offset. The offset is sign-extended to 32 bits, and the values from registers x1 and x2 are loaded into rs1_data and rs2_data.
Outputs	rs1_data: Value in x1 rs2_data: Value in x2 Sign-extended branch offset

Below is the code from the RV321CPU.v file that corresponds to the role of the respective stage.

```
IF: begin
   n_state <= Decode;
end

Decode: begin
   if (opcode == `OP_B) n_state <= Branch;
end
```

After fetching the instruction in the IF stage, the system transitions to the Decode stage, where it checks if the opcode indicates a branch instruction (OP_B), and if so, transitions to the Branch state.

3) ***Execute and Address Calculation (EX)***

In the Branch Comparison (EX) stage, the ALU subtracts rs2_data from rs1_data to compare their values, sets the zero flag if the result is zero (indicating equality), and calculates the branch target address as PC + offset if branching occurs. The process can be represented in a table as follows.

Inputs	rs1_data (x1 value) rs2_data (x2 value) Sign-extended branch offset
Operations	The ALU compares rs1_data and rs2_data by subtracting one from the other. If the result is zero (indicating equality), the zero flag is set, and the branch target address is calculated as PC + offset.
Outputs	Zero flag (indicates if x1 equals x2) Branch target address (if branching occurs)

Below is the code from the RV321CPU.v file that corresponds to the role of the respective stage.

```
localparam Branch = 4'b1010;
```

This line defines the state for processing SB-type (branch) instructions, such as BEQ, BNE, BLT, and BGE.

```
Branch: begin
   controls <= 11'b10_00_00_10_001; // ALUSrcA: rs1, ALUSrcB: immediate
   ALUOp <= 0; // Subtraction
end
```

This part is Branch Control Signal Configuration. The control signal configures the ALU to perform subtraction between the rs1 register and the branch offset (immediate), ensuring the branch condition is evaluated based on the result of rs1 - rs2.

```
case(opcode)
   `OP_B: begin
      case(funct3)
         3'b000: alucontrol <= 5'b10000; // BEQ (branch if equal)
         3'b001: alucontrol <= 5'b10000; // BNE (branch if not equal)
         3'b100: alucontrol <= 5'b10000; // BLT (branch if less than)
         3'b101: alucontrol <= 5'b10000; // BGE (branch if greater or equal)
         default: alucontrol <= 5'bxxxxx; // Undefined
      endcase
   end
endcase
```

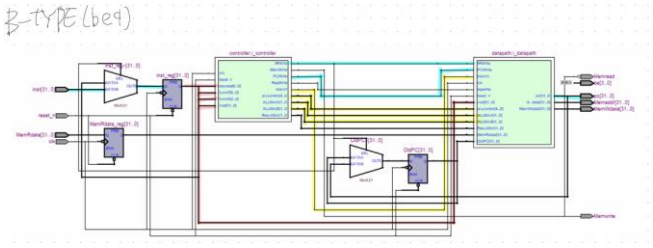
The funct3 value specifies the branch operation type (BEQ, BNE, BLT, BGE), and the ALU control signal is set to perform subtraction for condition evaluation.

```
assign beq_taken  = branch & f3beq & Zflag;  // BEQ: Zero flag set
assign bne_taken  = branch & f3bne & ~Zflag; // BNE: Zero flag not set
assign blt_taken  = branch & f3blt & (Nflag != Vflag); // BLT: Negative != Overflow
assign bgeu_taken = branch & f3bgeu & Cflag; // BGEU: Carry flag set
```

Branch conditions (BEQ, BNE, etc.) are evaluated using ALU flags (N, Z, C, V), and the branch signal activates if the condition is satisfied.

```
always @(negedge clk, negedge reset_n) begin
   if (!reset_n)
      pc <= 0;
   else if (PCWrite | beq_taken | bne_taken | blt_taken
| bgeu_taken)
      pc <= rd_data;
end
```

If the branch condition is satisfied, the program counter (PC) is updated with the branch target address; otherwise, it proceeds to the next sequential instruction (PC + 4). The RTL diagram based on the code analysis is as follows.



For the remaining part, we have analyzed the differences between single-cycle and multi-cycle processors for SB-type instructions.

a) ***Single-Cycle vs. Multi-Cycle Implementations***

In single-cycle implementations, all stages (IF, ID, EX) are completed within one clock cycle, simplifying design and control logic but risking inefficiencies and critical path delays by performing branch comparison and decision in the same cycle, whereas multi-cycle implementations distribute stages across multiple cycles, reducing pipeline stalls and critical path delays by separating branch computation but increasing control complexity and instruction latency.

D. ***S-type: Store***

S-Type instructions in the RISC-V architecture are used for storing data from a register into memory. The SW (Store Word) instruction is a fundamental example that calculates a memory address based on a base address and an immediate offset, then writes data from a register to the calculated address. This report outlines the execution steps of the SW instruction and compares single-cycle and multi-cycle implementations. The examples are as follows:

sw x1, 0(x2)

The example code performs the operation: Stores the value in x1 to the memory address calculated as the sum of the value in x2 and the immediate offset 0. The following is an analysis of the S-Type instruction process broken down by each stage.

1) ***Instruction Fetch (IF)***

In the Instruction Fetch (IF) stage, the instruction is retrieved from memory using the Program Counter (PC), which is incremented by 4 to point to the next instruction, resulting in the fetched instruction and the updated PC value (PC + 4). This stage is identical to that of R-Type instructions. The process can be represented in a table as follows.

Inputs	Program Counter (PC) Memory
Operations	The instruction is fetched from memory using the PC, which is then incremented by 4 to point to the next instruction.
Outputs	The fetched instructions. Updated PC value (PC + 4).

Below is the code from the RV321CPU.v file that corresponds to the role of the respective stage.

```
`define OP_S        7'b0100011

IF: begin
  controls <= 11'b00_10_11_00_000;
  ALUOp <= 1; // PC + 4
end
```

The opcode for S-type instructions is defined, which is used to handle instructions that store data in memory (e.g., SW, SH, SB). Subsequently, like all instructions, S-type instructions also fetch the instruction from memory during the IF stage. The ALU performs the PC + 4 operation to fetch the next instruction. Consequently, ALUOp is set to 1 to execute the PC increment operation.

2) ***Instruction Decode and Register Fetch (ID/RF)***

In the Decode and Register Fetch (ID) stage, the instruction is decoded to identify the source registers (rs1 and rs2) and the immediate offset, which is sign-extended to 32 bits, while the values from x2 (base address) and x1 (data to be stored) are loaded into rs1_data and rs2_data, respectively. The process can be represented in a table as follows.

Inputs	Fetched instruction
Operations	The instruction is decoded to identify the source registers (rs1 and rs2) and the immediate offset. The offset is sign-extended to 32 bits. The values from registers x2 (base address) and x1 (data to be stored) are loaded into rs1_data and rs2_data, respectively.
Outputs	rs1_data: Value in x2 rs2_data: Value in x1 Sign-extended immediate value

Below is the code from the RV321CPU.v file that corresponds to the role of the respective stage.

```
Decode: begin
  if (opcode == `OP_S)
    n_state <= MemAdr;
end
```

If the instruction is identified as S-type (OP_S), the next state transitions to the MemAdr stage, where address calculation is performed to prepare for storing data in memory.

3) ***Execute and Address Calculation (EX)***

In the Address Calculation (EX) stage, the ALU computes the effective memory address by adding the base address (rs1_data from x2) to the sign-extended immediate value, producing the calculated memory address as output. The process can be represented in a table as follows.

Inputs	rs1_data (x2 value) Sign-extended immediate
Operations	The ALU calculates the effective memory address by adding the base address (rs1_data) to the immediate offset.
Outputs	Effective memory address

Below is the code from the RV321CPU.v file that corresponds to the role of the respective stage.

```
MemAdr: begin
  controls <= 11'b10_01_00_00_000;
  ALUOp <= 1; // rs1 + offset
end
```

In the MemAdr stage, the controls signal is set to 11'b10_01_00_00_000, which configures the ALU to calculate the memory address by using rs1 as the first input (ALUSrcA = 10) and the immediate offset as the second input (ALUSrcB = 01), while ALUOp is set to 1 to perform the addition operation rs1 + offset.

4) ***Memory Access (MEM)***

In the Memory Access (MEM) stage, the value in rs2_data (x1) is written to the memory location specified by the effective memory address, with no additional outputs produced. The process can be represented in a table as follows.

Inputs	Effective memory address rs2_data (x1 value)
Operations	The value in rs2_data is written to the memory location specified by the effective address.
Outputs	None

Below is the code from the RV321CPU.v file that corresponds to the role of the respective stage.

```
MemW: begin
  controls <= 11'b10_01_00_10_010; // activate memory write
end
```

In the MemW stage, the controls signal is set to 11'b10_01_00_10_010, which activates the memory write operation (MemWrite = 1), while ALUSrcA and ALUSrcB retain the previously calculated memory address (rs1 + offset) as the input for the write operation.

5) Supplementary explanation

1. ALU Control for Address Calculation

```
`OP_S: begin
   alucontrol <= 5'b00000; // add
end
```

In this stage, the ALU control signal is set to 5'b00000, instructing the ALU to perform an addition operation. This addition is necessary to calculate the memory address by adding the base address (rs1) and the offset

2. ALU Input Configuration

```
always @(*) begin
   if (ALUSrcB == 2'b01)
      alusrc2 <= se_imm_stype; // Offset (S-type Immediate)
end
```

The S-type instruction requires an offset to calculate the target memory address. Here, the ALUSrcB signal ensures that the offset is provided as the second input to the ALU, allowing the address computation to incorporate the immediate value.

3. Memory Write Data Configuration

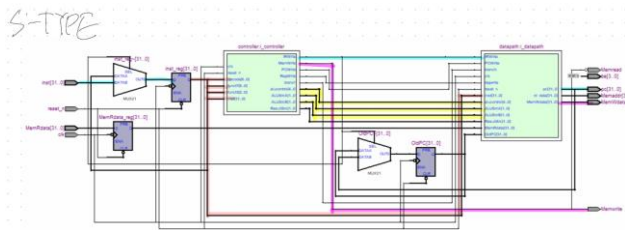
```
assign MemWdata = rs2_data_reg;
```

The value to be written into memory is fetched from the rs2 register, with the data being passed through the MemWdata signal. This configuration ensures that the correct data is stored at the computed memory address.

4. State Transition Logic

```
MemAdr: begin
   if (opcode == `OP_S)
      n_state <= MemW;
end

MemW: begin
   n_state <= IF; // Return to IF after completion
end
```

In the MemAdr state, the S-type instruction transitions to the MemW state to execute the memory write operation. Once the write operation is completed, the system returns to the IF state to fetch the next instruction, maintaining the instruction pipeline. The RTL diagram based on the code analysis is as follows.



For the remaining part, we have analyzed the differences between single-cycle and multi-cycle processors for S-type instructions.

*a) **Single-Cycle vs. Multi-Cycle Implementations***

In single-cycle implementations, all stages (IF, ID, EX, MEM) are completed in one clock cycle, simplifying design but increasing complexity and critical path length by performing address calculation and memory write simultaneously, whereas multi-cycle implementations separate these stages across cycles, reducing critical path delays but increasing control complexity and instruction latency.

E. *Differences Between Single-Cycle and Multi-Cycle Processors*

Multi-cycle execution uses fewer resources per cycle with shared hardware and relaxed timing constraints, prioritizing area efficiency, while single-cycle execution completes all stages in one cycle with higher resource demands, stricter timing, and a trade-off between performance and increased area and power consumption.

IV. Skeleton Code Application: Code Modification

Based on the previously explained concepts, the blank sections of the skeleton code provided in the assignment have been completed. The completed code, along with detailed explanations of its functionality, is described below.

1) Skeleton Code before the modification

```
// inst, OldPC register
  always @(negedge clk, negedge reset_n)
  begin
    if (!reset_n)
    begin
      inst_reg <= 0;
      OldPC <= 0;
    end
    else if (IRWrite)
    begin
     /********** TODO **********/
    end
  end
```

2) Revised Code

```
// inst, OldPC register
  always @(negedge clk, negedge reset_n)
  begin
    if (!reset_n)
    begin
      inst_reg <= 0;
      OldPC <= 0;
    end
    else if (IRWrite)
    begin
     inst_reg <= inst;
     OldPC <= pc;
    end
  end
```

The following describes the functionality of the updated code, including the handling of the TODO sections where the inst_reg and OldPC need to be updated. This involves storing the current instruction in the instruction register (inst_reg) and updating the previous PC value (OldPC). The variables are explained as follows:

- **inst_reg**: A register that stores the current instruction for decoding and execution.
- **OldPC**: A register that holds the previous value of the program counter (PC), needed for jump calculations in instructions like JALR.

The updated code, along with an explanation of its functionality, is as follows:

3) Reset Signal (reset_n) and Initialization Behavior

When the reset signal (reset_n) is set to 0, the processor resets to its initial state. In this case:

- inst_reg is set to 0, indicating that the instruction register does not hold a valid instruction.
- OldPC is also initialized to 0, signifying the absence of a previous PC value.
- The CPU begins normal operation only after the initialization is complete.

4) Updating the Instruction Register (inst_reg)

When the IRWrite signal is activated (set to 1), the instruction (inst) read in the current cycle is stored in inst_reg.

- The inst_reg will be used in subsequent instruction decoding and execution stages.
- Example:
  - If an instruction is fetched from memory, it is passed to inst and then stored in inst_reg.
  - The instruction stored in inst_reg is used in the next state (Decode) to extract register or operation-related information.

5) Storing the Previous PC (OldPC)

During the same cycle when IRWrite is activated, the current value of the program counter (pc) is stored in OldPC.

- OldPC is primarily used in instructions like JALR, which calculate a new jump address by adding an offset to the current PC value.
- Example:
  - OldPC can be represented as pc - 4 and is used as a return address after a jump instruction.

Based on this, the results of the time simulation were executed and are attached below.

Through the figure, it can be observed that each instruction type has a different number of cycles. Instructions such as 00021463, FEDFF06F, 20042303, and 00100F93 have 3, 4, 5, and 4 cycles, respectively. From this, it can be inferred that these instructions correspond to **Action for Branches**, **Action for R-type Instructions**, **Action for Memory Instructions**, and **Action for R-type Instructions**, respectively.

CONCLUSION

The multi-cycle processor architecture represents an efficient model for executing instructions by leveraging hardware reuse and separating execution into distinct stages. Through detailed analysis of instruction types, including R-type, I-type, B-type, and S-type, this report underscores the advantages of reduced hardware complexity and enhanced flexibility in handling diverse instruction sets. However, the design also introduces challenges in control signal coordination and increased instruction latency. Comparisons between single-cycle and multi-cycle implementations highlight the trade-offs in performance, resource allocation, and design complexity. Ultimately, multi-cycle processors provide an essential foundation for understanding advanced architectural techniques, enabling informed decisions in processor design for both academic and practical applications.

REFERENCES

[1] CA-5-Processor-1-ygkim. Young Geun Kim
[2] CA-5-Processor-2-ygkim. Young Geun Kim
[3] verilog_summary. hyuk ju Na
[4] tutorial. hyuk ju Na
[5] Project_material_. hyuk ju Na