Single-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*— This report presents an in-depth analysis of a single-cycle processor architecture, focusing on its design, operation, and timing simulation results. The single-cycle processor serves as a fundamental model for understanding instruction execution within a single clock cycle, offering insights into control signal behavior, data path organization, and key design trade-offs. Through modifications and simulations, we demonstrate the reset functionality and the propagation of key signals, emphasizing the interplay between control logic and processor performance.**

Keywords—Single-cycle processor, R-type, S-type, R-type, SB-type

I. Introduction

The single-cycle processor architecture represents a fundamental approach to instruction execution in computer systems, where each instruction completes its execution in a single clock cycle. This design offers simplicity and serves as a foundational model for understanding more complex pipelined architectures. The single-cycle processor provides critical insights into instruction set design, control signal behavior, data path organization, and timing analysis.

In recent years, the study of single-cycle architectures has retained its pedagogical significance for computer architecture courses, allowing students and engineers to gain hands-on experience in the detailed workings of a processor. This model offers a platform for exploring key design considerations such as clock cycle time, data path efficiency, and the balance between hardware complexity and performance.

This report analyzes the design, operation, and performance characteristics of a single-cycle processor, examining how its architecture handles different instruction types, including arithmetic, memory, and control instructions. By understanding its strengths and limitations, we can better appreciate the trade-offs that lead to more advanced designs such as multi-cycle and pipelined processors.

II. How to operate single-cycle processor

A. *What is a single-cycle processor?*

A single-cycle processor is designed such that a single instruction completes all stages (instruction fetch, instruction decode and register read, execution, memory access, and write-back) within one clock cycle. In other words, each instruction is processed in a single clock cycle. This design is relatively simple and has the advantage of easy control because instructions are processed sequentially. However, since each instruction must complete all stages within the same clock period, the longest stage determines the overall clock cycle time, which can lead to inefficiencies in terms of speed.

****baseline code is used to express skeleton code.***

B. *How a Single Cycle Processor operates?*

The operation of a single-cycle processor consists of five stages: instruction fetch, instruction decode and register read, execution, memory access, and write-back. Table 1 presents these five stages along with their associated modules. Here, the related modules refer to those associated in the code. And, The detailed descriptions of each stage are provided below Table 1.

Table 1. Execution stages of a single cycle processor

Execution Stages	**Step Description**	
	Stage Name	***Related Modules (in baseline code)***
Phase1	Instruction Fetch	ram2port_inst_data, ALTPLL_clkgen
Phase2	Instruction Decode and Register Read	Addr_Decoder
Phase3	Execute Operation	RV32I
Phase4	Memory Access	ram2port_inst_data, TimerCounter, miniUART, GPIO
Phase5	Writeback	RV32I

In the first phase, Instruction Fetch, "The clock signal from the ALTPLL_clkgen module is used to provide the address stored in the PC (Program Counter) to the instruction memory, retrieving the instruction (ram2port_inst_addr module). And in the next phase, Instruction Decode and Register Read, the instruction decoder extracts necessary informationd by interpreting the fetched instruction (Addr_Decoder module). Data from source registers is read from the register file to prepare for execution. In the third phase, Execute Operation, Performs ALU operations (RV32I module) or calculates addresses for memory access. For branch instructions, it evaluates branch conditions and performs branching. In the fourth phase, Memory Access, Uses the address calculated by the ALU (ram2port_inst_data module) to read data from memory or store register data to the calculated address. It also performs read or write operations related to timers (TimerCounter), carries out serial communication with external devices (miniUART), or checks the state of switches input by the user (GPIO). In the final phase, Writeback, Writes the result of the operation or data read from memory back to the destination register (RV32I module). A more detailed process of instruction execution is described in the following section.

III. Analyze in Detail

How the Instruction Is Executed

There are seven main signals that control the Single-Cycle Control. The names and descriptions of each signal are provided in Table 2. When the value of a signal described in Table 2 is 1, it indicates True, and when it is 0, it indicates False.

TABLE 2. SIGNALS OF SINGLE-CYCLE PROCESSOR

Signal	*Description*
RegWrite	RegWrite Specify if the destination register needs to be written
ALUSrc	Select whether source of ALU is register or immediate
ALUOp	Specify operation of ALU
MemWrite	Specify whether memory needs to be written
MemRead	Specify whether memory needs to read
MemtoReg	Select whether memory or ALU output is used for "Write data" of register
PCSrc	Select whether PC + 4 or branch target address is used for the next PC

In this section, we will focus on how each type of instruction operates, centering on the provided code and the signals described in Table 2. The Single Cycle Processor diagram used to explore the signals is shown in Figure 1.

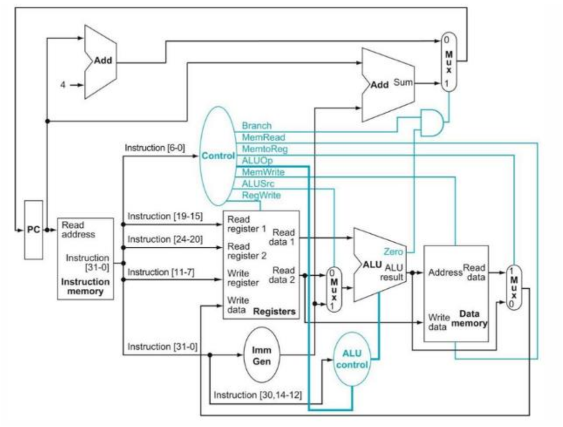


Figure 1. Diagram of Single-Cycle Processor

The signals corresponding to the colored lines used in the diagram are listed in Table 3, and detailed explanations are provided afterward.

TABLE 3. SIGNALS OF SINGLE-CYCLE PROCESSOR

Color	*Related signals*
Green Line	PCSrc
Red Line	Instruction Fields
Purple Line	MemtoReg
Blue Line	Related to Data Memory
Pink Line	Related to Control Signals

The green line indicates the path used to determine the address of the next instruction in the Program Counter (PC). By default, the PC performs PC + 4 to proceed to the next instruction. However, it can be modified to point to the branch target address during conditional branching. The red line represents the fields of the instruction fetched from instruction memory. For example, Instruction [31-0] represents the entire instruction, while Instruction [19-15], Instruction [24-20], and Instruction [11-7] each represent register address fields, which are then passed to the register file. The purple line indicates the data transfer path related to data memory. This line handles the movement of data between memory and registers in the CPU, such as writing data read from memory into a register. The purple line is particularly significant when the MemtoReg control signal is active, as it transfers data read from memory to the register, enabling the storage of this data in the register after memory access operations. The blue line represents the flow of data related to data memory. This path is used for read and write operations, handling the retrieval or storage of data during memory read and write processes. Lastly, the pink line represents control signals that determine how each component operates, including signals like RegWrite, ALUSrc, ALUOp, MemWrite, MemRead, and MemtoReg. These signals control the execution of instructions in the processor. In summary, the pink lines indicate the paths where each signal is active (value of 1) during operations.

We will now analyze how each instruction (R-type, I-type, SB-type, S-type) operates in detail. Additionally, each command will be examined from both the baseline code and the theoretical perspective, focusing on the signals involved.

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

R-type instructions are a type of register-to-register operation, with add, sub, and, and or being representative examples. Each instruction performs an operation on the values of two source registers and stores the result in a destination register.

The *add* instruction adds the values of two source registers and stores the result in the destination register. The *sub* instruction subtracts the value of the second source register from the value of the first source register and stores the result in the destination register. The *and* instruction performs a bitwise AND operation on the values of two source registers and stores the result in the destination register. The *or* instruction performs a bitwise OR operation on the values of two source registers and stores the result in the destination register.

These instructions are commonly used for arithmetic and logical operations and are executed by the Arithmetic Logic Unit (ALU) of the Central Processing Unit (CPU). And now, we will introduce R-type instructions in two perspectives: 1) Perspective of baseline code, 2) Perspective of theoretical (Signal on table2). As mentioned above, the rest (I-type, SB-type, S-type) will also be introduced in the same manner.

1) *Perspective of baseline code*

In baseline code, the RV32I processor interprets instructions based on the clock (clk90) signal and performs operations using the ALU (Arithmetic Logic Unit) by adding the values of two registers and storing the result.

2) *Perspective of theoretical*

The execution of R-type instructions operates by combining the values of two registers through ALU operations to generate a result, which is then written back to a register. It does not perform memory read or write operations, and after the operation, the PC (Program Counter) increments to move to the next instruction. These control signals reflect the characteristics of R-type instructions, ensuring that the given instruction operates correctly within the CPU. This is illustrated in Figure 2.

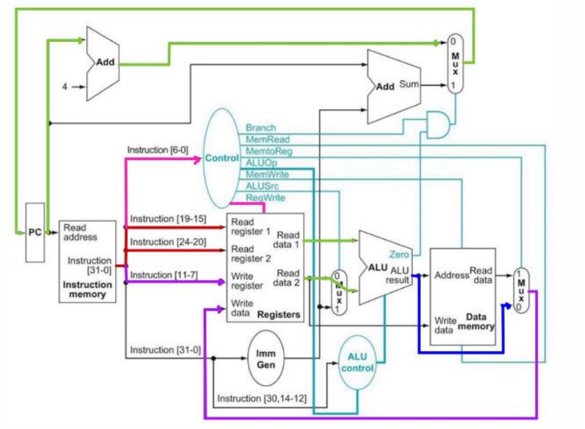


Figure 2. Diagram of Single-Cycle Processor of R-type instructions

A detailed description of each signal is provided in the table below (Table 4).

TABLE 4. SIGNALS OF SINGLE-CYCLE PROCESSOR OF R-TYPE INSTRUCTIONS

Signal	*Value*	*Description*
RegWrite	1	When this signal is set to 1, the result of the executed instruction is written on the register file. In other words, the result of the operation is stored in the destination register.
ALUSrc	0	When selecting the second input to the ALU, the value of the Read Data 2 register is chosen. This indicates that R-type instructions perform operations between register values, differentiating them from I-type instructions that use immediate (constant) values.
ALUOp	OP	This signal selects the ALU operation to be performed. Based on the OP value, the ALU performs specific operations such as addition, subtraction, or logical operations. The behavior of the ALU is determined by the function (opcode) of the R-type instruction.
MemWrite	0	Memory writes are disabled. Since R-type instructions do not directly write the result of operations to memory but instead store them in a register, this value is set to 0.
MemRead	0	Memory reads are disabled. R-type instructions do not fetch values from memory, so this signal is set to 0.
MemtoReg	0	The ALU result is selected as the data source for writing to the register. Thus, the operation result is delivered to the destination register, contrasting with cases where data is read from memory.
PCSrc	0	The program counter (PC) value is incremented by PC + 4 to proceed to the next instruction. Since R-type instructions are not related to branching, the default increment of the PC is selected.

B. *L-type: load*

L-type instructions load data from memory and store it in a register.

1) Perspective of baseline code

In baseline code, data_addr represents the memory address, and read_data refers to the data read from memory. The cs_mem_n signal controls memory access, and when data is loaded, it is transferred to the RV32I processor through the read_data signal.

2) Perspective of theoretical

L-Type instructions operate by calculating a memory address and then reading data from that address to store it in the destination register. The ALU is used to compute the memory address, and the process involves reading values from memory. After the operation, the PC (Program Counter) increments to move to the next instruction. Control signals are tailored to reflect the characteristics of L-Type instructions, ensuring that the instruction is executed correctly within the CPU. This is illustrated in Figure 3 and detailed further in Table 5.

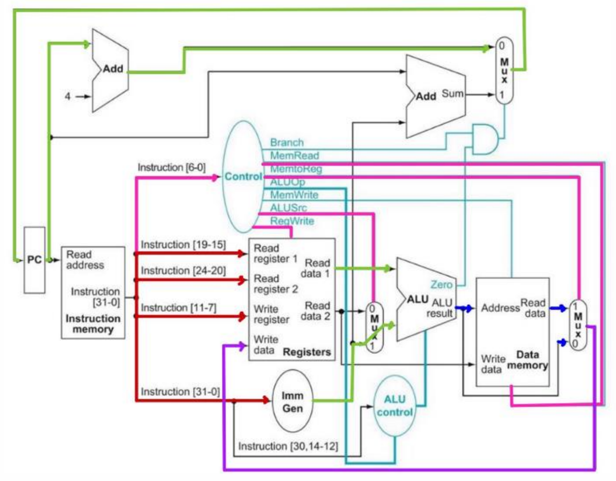


Figure 3. Diagram of Single-Cycle Processor of L-type instructions

TABLE 5. SIGNALS OF SINGLE-CYCLE PROCESSOR OF L-TYPE INSTRUCTIONS

Signal	*Value*	*Description*
RegWrite	1	When this signal is set to 1, data read from memory is written to the destination register (rd). In the case of I-Type instructions, they mainly perform operations where data is fetched from memory and stored in a register.
ALUSrc	1	The control configures the ALU's second input to select the immediate value (Immediate Offset). Since I-Type instructions often operate using both a register and an immediate value, the ALUSrc signal is activated to facilitate this.
ALUOp	OP	The ALU determines the operation to be performed. For example, it may add Read Data 1 and the immediate value (or perform subtraction or other operations if necessary). The operation of the I-Type instruction is defined by this ALU behavior.
MemWrite	0	Memory write operations are disabled. Since I-Type Load instructions only perform data reading from memory, memory writing is unnecessary.
MemRead	1	Data reading from memory is enabled. When this signal is set to 1, the CPU fetches data from memory to store it in a register.
MemtoReg	1	It selects the source for writing data to the register. Instead of storing the ALU result, the value read from memory is stored in the destination register.
PCSrc	0	The control sets the default next instruction address to PC + 4. Since I-Type instructions are not branch instructions, the PC simply proceeds to the next instruction.

C. *SB-type: BEQ*

SB-Type instructions perform a branch to a specific address if the values of two registers are equal.

1) Perspective of baseline code

In the RV32I processor, conditional branch instructions compare the values of two registers during the instruction decode stage. If the condition is met, inst_addr is updated to move to a new instruction address.

2) Perspective of theoretical

The execution of SB-Type instructions uses the ALU to compare the values of two registers, and if the condition is true, the PC (Program Counter) is changed to the branch address. No memory read or write operations are performed, and if the condition is false, the PC increments to move to the next instruction. These control signals are tailored to reflect the characteristics of SB-Type instructions, ensuring that the given instruction performs conditional branching correctly within the CPU. One distinctive feature of the SB-Type instruction diagram is the inclusion of a Shift Left 1 unit, as shown in Figure 4. Further details are provided in Table 6.

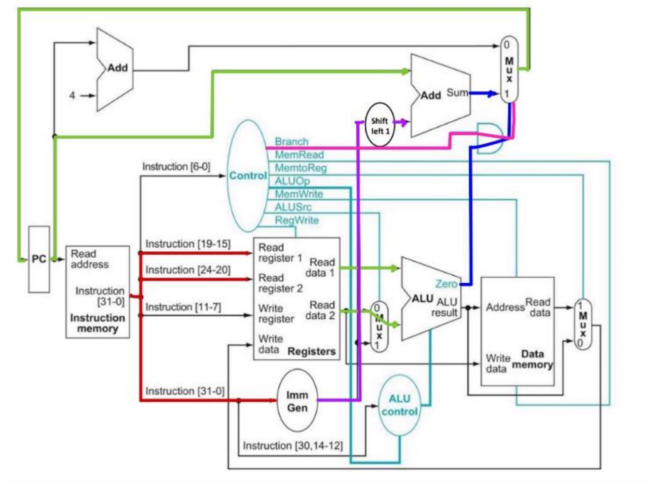


Figure 4. Diagram of Single-Cycle Processor of SB-type instructions

TABLE 6. SIGNALS OF SINGLE-CYCLE PROCESSOR OF SB-TYPE INSTRUCTIONS

Signal	*Value*	*Description*
RegWrite	0	Since SB-Type instructions do not write values to registers, the RegWrite signal is deactivated. This is because they perform conditional branching based on comparisons and do not need to store the result of operations in a register.
ALUSrc	0	The second input for the ALU is configured to select Read Data 2 from the register. SB-Type instructions use this for comparing the values of two registers.
ALUOp	OP	The ALU performs a comparison or subtraction operation on the two input values (Read Data 1 and Read Data 2) to evaluate the condition. The branching decision is based on this comparison result.
MemWrite	0	Memory write operations are disabled as SB-Type instructions do not involve writing data to memory, setting this signal to 0.
MemRead	0	Reading data from memory is also disabled. SB-Type instructions are solely concerned with evaluating a condition and branching if necessary, making memory reading unnecessary.
MemtoReg	X (Not using)	Since SB-Type instructions do not need to fetch data from memory and store it in a register, this signal is not utilized.
PCSrc	1	When the branch condition is met, the program counter (PC) is updated to move to the target branch address, altering the program flow accordingly.

D. S-type: Store

S-Type instructions store the value from a register into a specified memory address.

1) Perspective of baseline code

Using the data memory (ram2port_inst_data module), the write_data is stored at the specified data_addr location. This occurs only when the data_we signal is in the write-enabled state, allowing values to be written to memory.

2) Perspective of theoretical

The execution of S-Type instructions operates by using an ALU operation to calculate the memory address and storing the value from the source register at that address. Memory read operations are not performed; instead, S-Type instructions primarily function to write data to memory. After execution, the PC (Program Counter) increments to move to the next instruction. These control signals are tailored to reflect the characteristics of S-Type instructions, ensuring that the given instruction correctly performs memory write operations within the CPU. This process is illustrated in Figure 5, with further details provided in Table 7.

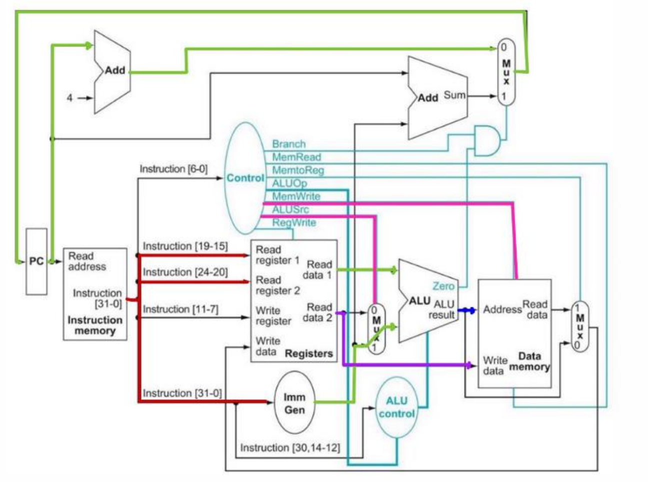


Figure 5. Diagram of Single-Cycle Processor of S-type instructions

TABLE 7. SIGNALS OF SINGLE-CYCLE PROCESSOR OF S-TYPE INSTRUCTIONS

Signal	*Value*	*Description*
RegWrite	0	When this signal is set to 0, S-Type instructions perform memory write operations and do not record data into a register.
ALUSrc	1	The control configures the ALU's second input to select the immediate value (Immediate Offset). Since S-Type instructions often involve calculating an address using a register value combined with an immediate value, the ALUSrc signal is activated to enable this.
ALUOp	OP	The ALU determines the operation to perform. For example, when calculating the memory address to store a value, it may involve adding the register value and the immediate value. This ALU operation defines how S-Type instructions behave.
MemWrite	1	Data writing to memory is enabled. When this signal is set to 1, the S-Type instruction writes data to memory at the computed address.
MemRead	0	Memory reading operations are disabled since S-Type instructions solely perform write operations, making memory reads unnecessary.
MemtoReg	X (Not using)	This signal is not used in S-Type instructions, as they only involve writing data to memory and do not require storing data into a register.

Signal	*Value*	*Description*
PCSrc	0	The control sets the default next instruction address to PC + 4. Since S-Type instructions are not branch instructions, the PC simply moves to the next instruction.

IV. Timing Simulation Results

To perform timing simulation in Quartus, we created a Vector Waveform File (VWF) and set the necessary input signals and clock. We confirmed that the provided ARM_System.vwf file exists and used it to conduct the timing simulation.

First, we ran the timing simulation without any additional processing and observed that the TimerCounter value increased. Next, we applied a reset to the TimerCounter and reran the timing simulation to verify the results. The method used for applying the reset is described below.

A. *timing simulation without any additional processing*

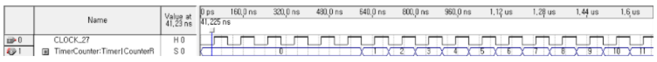


Figure 6. timing simulation without any additional processing

Figure 6 shows the results of the timing simulation executed without any additional processing, where the TimerCounter value increases periodically over a set time interval. This demonstrates the TimerCounter's behavior under initial conditions and indicates that the basic timing operation is functioning correctly.

B. *timing simulation with reset operation*

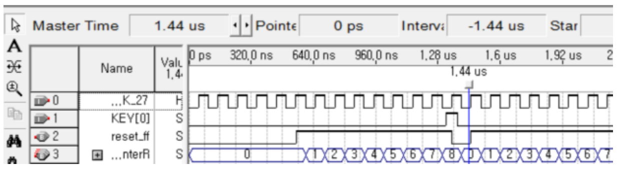


Figure 7. timing simulation with reset operation

Figure 7 displays the results of the timing simulation after applying a reset to the TimerCounter. The TimerCounter value increases up to 8 and then resets to 0, demonstrating that the reset was successfully applied. The baseline code before and after the modification, as well as the reset process, is described below.

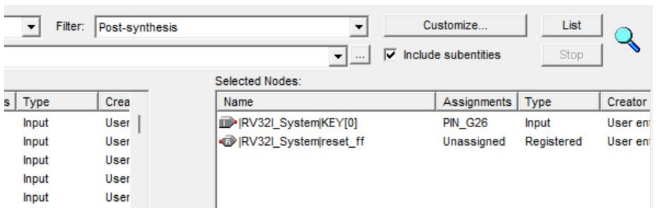
1) *BaseLine Code before the modification*

```
// reset = KEY[0]
// if KEY[0] is pressed, the reset goes down to "0"
// reset is a low-active signal
assign reset_poweron = KEY[0];
assign reset = reset_poweron;
```

2) *Revised Code*

```
// reset = KEY[0]
// if KEY[0] is pressed, the reset goes down to "0"
// reset is a low-active signal
assign reset_poweron = ~KEY[0];
assign reset = reset_poweron;
```

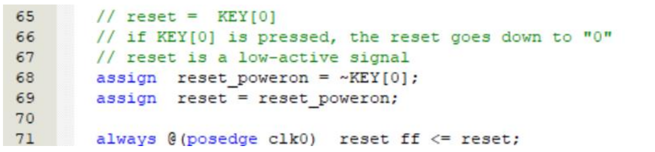
The reset handling was adjusted to ensure that the reset signal operates as a low-active signal, consistent with the provided guidance (comments in baseline code). Specifically, the modification involved assigning the inverse of *KEY[0]* to *reset_poweron*, as shown in the revised code segment. This approach guarantees that when *KEY[0]* is pressed, the reset signal transitions to a low state, thereby meeting the intended design behavior. Following this modification, a compilation was performed, and additional verification was carried out using the vwf (waveform) view, where *KEY[0]* and *reset_ff* nodes were observed to validate the correct operation of the reset logic. Figure 8 displays the simulation setup screen reflecting the modified code.



Name	Assignments	Type	Creator
|RV32I_System|KEY[0]	PIN_G26	Input	User en
|RV32I_System|reset_ff	Unassigned	Registered	User en

Figure 8. the simulation setup screen reflecting the modified code

The subsequent content describes the variables used in the modified code and explains the corresponding logic. Figure 9 is an image of a portion of the code that will be used for the explanation.



```
// reset =  KEY[0]
// if KEY[0] is pressed, the reset goes down to "0"
// reset is a low-active signal
assign  reset_poweron = ~KEY[0];
assign  reset = reset_poweron;

always @(posedge clk0)  reset_ff <= reset;
```

Figure 9. Modified Code Segment for Explanation

C. *Detaied Explanation of Variables*

1) KEY[0]: This input signal indicates the state when a user presses (=1) or releases (=0) the button. KEY[0] serves as the primary control input for the system reset process.

2) reset_poweron: This variable is assigned the negated (~) value of KEY[0]. Consequently, when the user presses KEY[0] (i.e., KEY[0] = 1), reset_poweron becomes 0, effectively triggering a system reset.

3) reset: Acting as a low-active signal, reset controls the system's reset state. When reset is 0, the system enters the reset state. The signal is managed by reset_poweron to ensure proper reset logic.

4) reset_ff: This register synchronizes the reset signal at the rising edge of the clock clk0. It ensures that the reset signal propagates accurately and uniformly across all system modules for consistent initialization.

D. *Flow of Operations*

In the section "C. Detailed Explanation of Variables," the variables described have been utilized to elaborate on the logic within the box below.

Flow of Operations

1) When KEY[0] is pressed (KEY[0] = 1), the reset signal is inverted and set to 0 (reset = ~KEY[0]), initiating the reset state for the system.

2) The reset signal is synchronized to reset_ff on the rising edge of clk0. This synchronization guarantees precise timing for reset activation.

3) The reset_ff signal is then propagated to all relevant modules, performing a full system initialization. For instance, the TimeCounter resets from its maximum count of 8 back to 0, demonstrating the effective application of the reset.

4) When KEY[0] is released (KEY[0] = 0), the reset signal becomes inactive, allowing the system to return to normal operational state.

This structured approach ensures robust system initialization, proper reset propagation, and a smooth transition to normal operation once the reset is deactivated. By adhering to this logic, the system achieves reliable and predictable behavior under reset conditions.

Conclusion

In this report, we explored the design and operational characteristics of a single-cycle processor, emphasizing its simplicity and foundational role in computer architecture. We discussed key control signals and data paths that define instruction execution, focusing on different instruction types, including R-type, I-type, SB-type, and S-type operations. Through timing simulations, we verified the proper operation of a modified reset mechanism, demonstrating robust system behavior and consistent initialization. By analyzing the strengths and limitations of the single-cycle processor, we gain valuable insights into the design considerations that influence more advanced architectures, such as multi-cycle and pipelined designs.

References

[1] CA-5-Processor-1-ygkim. Young Geun Kim

[2] verilog_summary. hyuk ju Na

[3] tutorial. hyuk ju Na

[4] Project_material_. hyuk ju Na

Describes in detail how a single-cycle processor operates. (2 points)

Analyze in detail how the following instruction is executed.

Instruction (1 point each, total 4 points) (Use any register or value)

- R-type add
- I-type lw (load word)
- B-type beq
- S-type sw (store word)

Refer to the picture on page 4 and describe the operation of each unit in detail.

Run the timer simulation, capture the results (the numbers going up) and attach them to your report. (1 point)

Please explain how to reset and attach the results (capture image). (1 point)

Please describe the reset process in detail. (2 points)