Project 2  
- Single Cycle MIPS Simulator -

## Prepared by

**Lee Chang Yoon**

**32183641@dankook.ac.kr**

Left Days: 5

April 11, 2022

# **Table of Contents**

1. **Introduction ………………………………………………………………………………………………………………………………….…… 1**
2. **Requirement ………………………………………………………………………………………………………………………………………. 1**
3. **Concepts ………………………………………………………………………………………………………………………………….………... 2**

3-1. Von Neumann Computer ………….….……………………………………………………………………………………………………………………….……… 2

* 1. ISA (Instruction Set Architecture) ….……………………………………………………………………………………………………………………………… 2
  2. MIPS (Microprocessor without Interlocked Pipeline Stages) …………………………………………………………………………...…………… 3
  3. Data paths of MIPS ISA ………………………………………………………………………………………………………………………………………………... 3
  4. L1 Cache ………………………………………………………………………………………………………………………………………………………………….…… 5
  5. Byte Order ……………………………………………………………………………………………………………………………………………………………………. 5

1. **Single-Cycle Microarchitecture …………………………………………………………………………………………………………… 6**

4-1. Implemented Datapath ……………………………………………...………………………………………………………………………………………….……. 6

* 1. Control Signal Table …………………………………………………………………………………………...……………………………………………………...... 7
  2. Program Definition …….………………………………………………………………………………………...………………………………………………………. 8

1. **Implementation …………………………………………………………………………………………...……………………………….……. 9**

5-1.Control Unit (CU) ………….………………………………………………………………………………………………………………………...……...…………... 9

* 1. Instruction Memory (IM) ………………………………………………………………………………………………………………………...……...………….. 10
  2. Registers (RF) ……………………………………………...…………………………………………………………………………………………………………...…. 10
  3. Arithmetic Logic Unit (ALU) ………………………………………………………………………...……………………………..........................………… 11
  4. Data Memory (DM) ……………………………………………………………………………...……………………………………….……………………………...12
  5. ADR ……………………………………………………………………………………………………………………………………………………………………………… 12
  6. MUX ……………………………………………………………………………………………………………………………………...………………………...…………. 12
  7. Main ………………………………………………………………………………………………………………………………………...…………………………………. 13
     1. *Load Program …………………………………………………………………………………….……………………………………….…………………………...…. 13*
     2. *Initialize ………………………………………………………………………………………………………………………………...……………………...………….… 13*
     3. *Instruction Fetch (IF) ……………………………………………………………………………………………………………….……………………………..……. 14*
     4. *Instruction Decode / Register Operand Fetch (ID / RF) ……………………………………………………….………………………………...……… 14*
     5. *Execute / Evaluate Memory Address (EX / AG) ………………………………………………………………………………………………………….…. 14*
     6. *Store / Write Result (WB) …………………………………………………………………………………………...………………………………………….……. 15*
     7. *PC Update ……………………………………………………………………………………………………………………………………….………………………...… 15*

1. **Build Environment ……………………………………………………………………………………………….…………………….……… 16**
2. **Result ……………………………………………………………………………………………….…………………………….…………………. 16**
3. **Evaluation the Single-Cycle Microarchitectures ………………………………………………………………………….……… 22**

8-1. Analysis ……………………………………………………………………………………………………………………………………………………………...…….... 22

8-2. Single-Cycle Datapath Analysis ………………………………………………………………………………………………………...…………………….…... 22

* + 1. *R-Type and I-Type ………………………………………………………………………………………………………………………………………………………… 22*
    2. *Load Word (LW) ………………………………………………………………………………………………………………………...………………………………… 23*
    3. *Store Word (SW) ……………………………………………………………………………………………………………………………...………………………….. 23*
    4. *Brach Taken ………………………………………………………………………………………………………………………………...........................…….… 23*
    5. *Jump ……………………………………………………………………………………………………………………………...…………….……………………………… 24*

*8-3. Conclusion ………………………………………………………………………………………………………………………………………………….………………… 24*

1. **Conclusion ……………………………………………………………………………………………………….………………………………… 25**
2. **Citation …………………………………………………………………………………………….……………………………………….……… 26**

# **Introduction**

Microarchitecture is the abstracted information that includes how hardware, such as CPU and GPU, is operated. Microarchitecture and ISA (Instruction Set Architecture) constitute the computer architecture field. Therefore, understanding the microarchitecture and ISA is important in understanding computer architecture.

In this report, we will describe how single-cycle microarchitecture works, its implementations, and the evaluation of the project. The work presented in this report is part of a large project which is designed to implement the pipelined and optimized multi-cycle microarchitecture which uses the instruction set of the MIPS. Before we get into the large scale of the project, we must build the microarchitecture that executes the instructions in a single cycle.

The first step in this project is to specify the requirements for the single-cycle microarchitecture. Second, we move on to concepts that are critically related to the implementation of the microarchitecture: Von Neumann Computer, ISA, MIPS, Data paths. Third, we will state the total data path that we have implemented, including the control signal table and program definitions. Fourth, we will describe how we implemented the data paths and single-cycle MIPS simulator according to the control signals and program definitions. Then, there will be some results by executing the binary programs using an implemented simulator. In the end, we will evaluate the single-cycle simulator with some assumptions and the flow of the data paths.

# **Requirements**

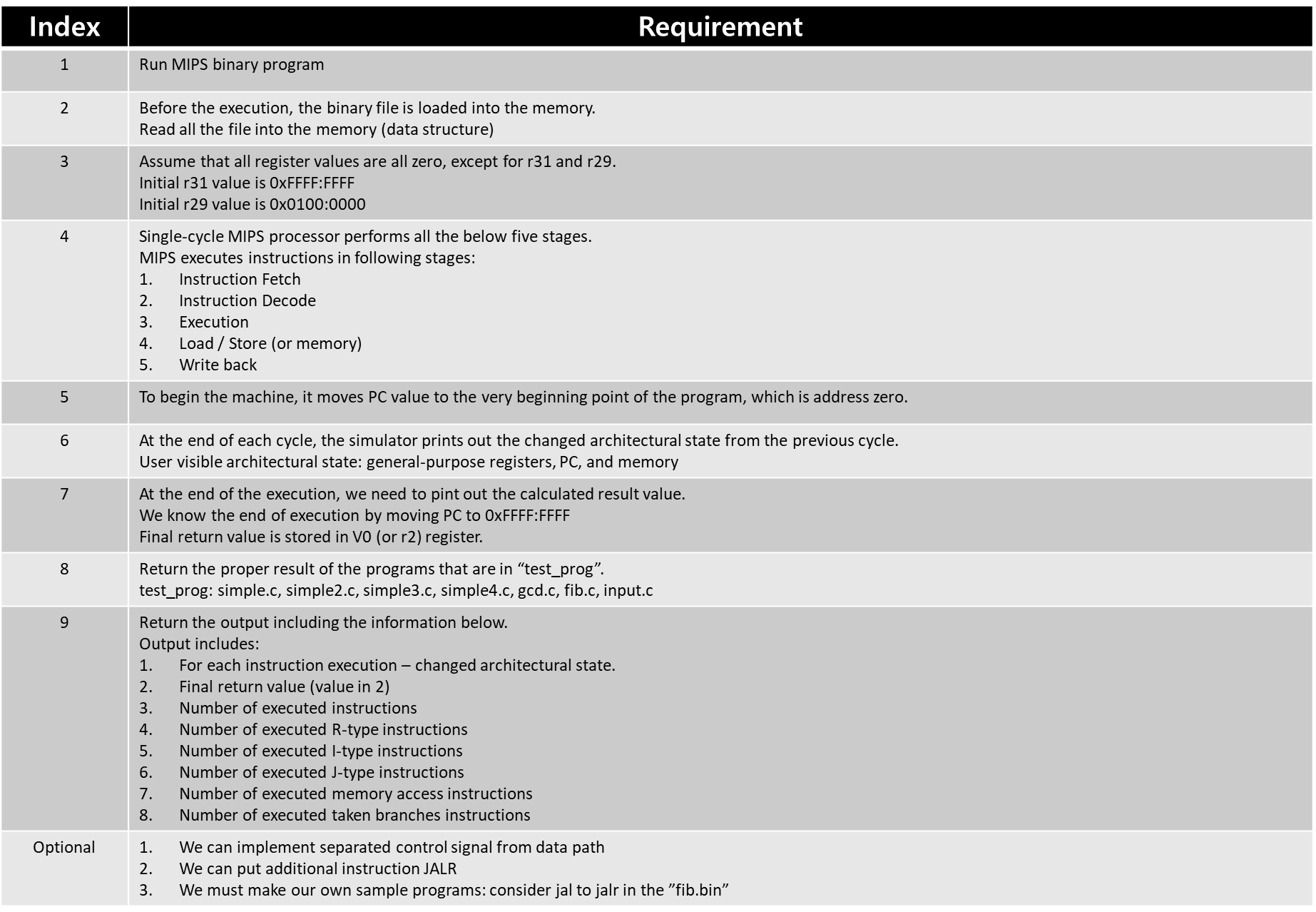


Figure 1 - Requirement Specification

Figure 1 shows the requirements for a single-cycle MIPS simulator. The implementations for these requirements will be described in detail afterward.

# **Concepts**

Before we get into the description of the single-cycle MIPS simulator, we will briefly discuss the concepts that are mainly used in the implementation: ISA, MIPS, and Data Paths.

* 1. **Von Neumann Computer**

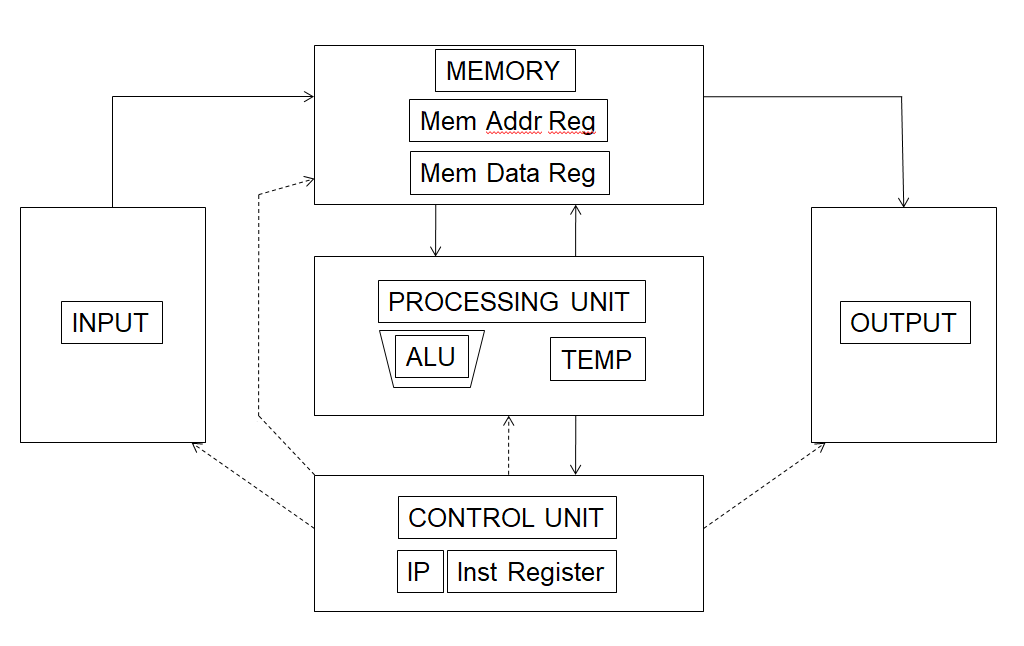


Figure 2 - Von Neumann Model (*Project1, CHANGYOON*)

The Von Neumann architecture is a computer architecture based on a description by John Von Neumann in 1945. Figure 2 shows the model of the Von Neumann architecture. Von Neumann's computer consists of a processing unit containing an arithmetic logic unit (ALU) and processor register, a controlling unit (CU) containing an instruction register (IR) and instruction pointer (IP), a memory to store both data and instructions, and input and output mechanisms. The meaning has evolved to be any stored-program computer in which instruction fetch and a data operation cannot occur at the same time because of the share of the common bus (*Project1, CHANGYOON*).

Like most computers, single-cycle MIPS microarchitecture follows the concept of Von Neumann architecture. It follows the execution sequence, fetch – decode – execute – store. First, the control unit fetches the next instruction from the memory. The instruction pointer (IP) or program counter (PC) in the control unit contains the address of the next instruction to fetch. The memory unit then reads the bytes stored at the specified address and sends them to the control unit on the data bus. The instruction register (IR) stores the bytes of the instructions received from the memory unit. The control unit also increments the IP or PC value to store the address of the new next instruction to fetch. Next, the control unit decodes the instructions stored in IR. Third, the processing unit executes the instruction. Lastly, the control unit stores the results in memory (*Project1, CHANGYOON*).

* 1. **ISA (Instruction Set Architecture)**

An Instruction Set Architecture (ISA) is part of the abstract model of a computer that defines how the CPU is controlled by the software. The ISA acts as an interface between the hardware and the software, specifying both what the processor can do as well as how it gets done. The ISA provides the only way through which a user can interact with the hardware. It can be viewed as a programmer’s manual because it is the portion of the machine that is visible to the assembly language programmer, the compiler writer, and the application programmer. The ISA defines the supported data types, the registers, how the hardware manages main memory, key features (such as virtual memory), which instructions a microprocessor to execute, and the input/output model of multiple ISA implementations. The ISA can be extended by adding instructions or other capabilities, or by adding support for larger addresses and data values. Each CPU has its own ISA. For example, AMD has AMD64 and ARM, and Intel has x86. In this project, we will use MIPS ISA, which is then built by MIPS Technologies. The concepts and details of MIPS ISA will be described in section 3-3.

* 1. **MIPS (Microprocessor without Interlocked Pipeline Stages)**

MIPS (Microprocessor without Interlocked Pipeline Stages) is the RISC (Reduced Instruction Set Computer) ISA which has been built by MIPS Technology. The reason is that MIPS ISA is RISC type ISA and all the lengths of instructions are set to be 32bit. According to the different information for CPU needed to process the different jobs, there must be several formats to distinguish the necessary information for types of jobs. Therefore, in MIPS-32 ISA, instructions are distinguished into three types: R-Type, I-Type, and J-Type.

* R-Type:



**opcode** (6bit): opcode is always zero in R-type instructions. MIPS distinguishes the operation type due to the funct field.

**rs** (5bit): Fist source register index.

**rt** (5bit): Second source register index.

**rd** (5bit): Write register index.

**shamt** (5bit): Shift amount. This field is only used for shift operations.

**funct** (6bit): Distinguishes the operation type.

* I-Type



**opcode** (6bit): Distinguishes the operation type.

**rs** (5bit): Fist source register index.

**rt** (5bit): Write register index.

**immediate** (16bit): number that is used for the second source.

* J-Type



**opcode** (6bit): Distinguishes the operation type.

**address** (5bit): Target address to jump. It represents (target instruction memory address / 4).

* 1. **Data Paths of MIPS ISA**

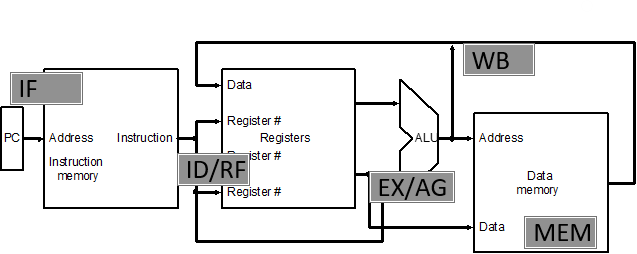
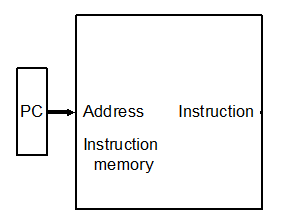


Figure 3 - 5 Generic Steps of MIPS Data Path

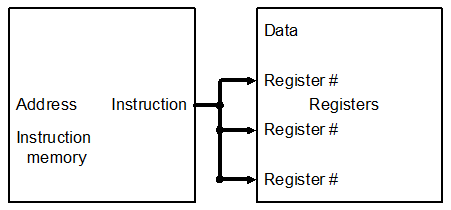
Datapath is the structure that refers to all elements that process and operate the data, address, and registers in the CPU, and we can broadly divide it into 5 phases: IF – ID/RF – EX/AG – MEM – WB. Figure 3 shows the data path for each phase.

* Instruction Fetch (IF)



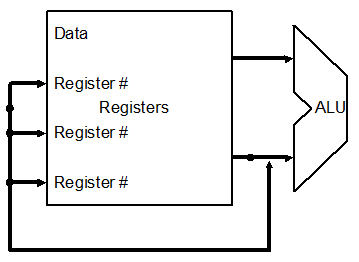
Reads the instruction memory address from the PC and update the PC.

* Instruction Decode and Register operand fetch (ID / RF)



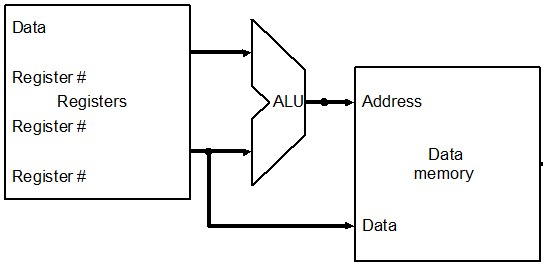
Decode the instruction and read the data that stored in following registers.

* Execute / Evaluate memory address (EX / AG)



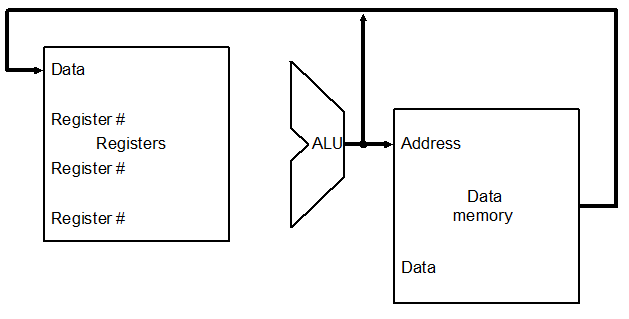
Execute the operation for the following instruction and if it needs to get the memory address, it calculates the memory address.

* Memory operand fetch (MEM)



For instructions that are related to the memory access, such as load word (LW) and store word (SW), the components above conduct the operation like reading or storing the data from the following memory address or storing the data into the following memory address.

* Store / Write Back result (WB)



Store the ALU result into the target register.

* 1. **L1 Cache**

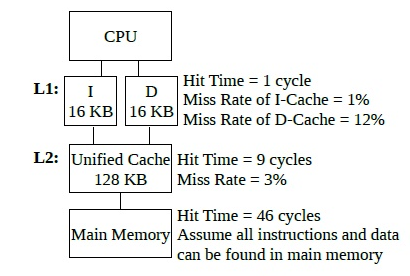


Figure 4 – Cache Hierarchy

L1 cache is the closest cache from the processor. Due to the processing speed, the L1 cache is separated into an instruction cache (I$) and a data cache (D$). Figure 4 shows the cache hierarchy of each level of the caches. Instruction cache deals with the memory text area and data cache deal with every data without the text area. In this project, we will apply this concept and therefore, we will make separate memory arrays for instruction memory and data memory.

* 1. **Byte Order**

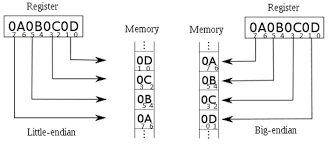


Figure 5 - Byte Order

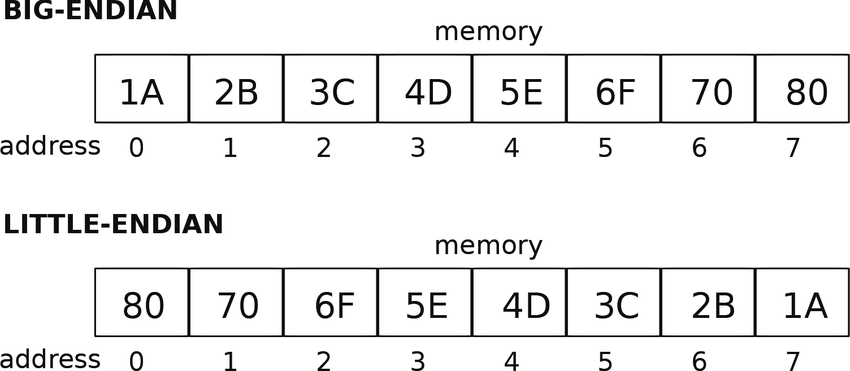


Figure 6 - Endian

Endian means the method of arranging several consecutive objects in a one-dimensional space such as a computer’s memory. Endian can be divided into Big-Endian, in which a large unit precedes, and Little-Endian, in which a small unit precedes. These days, almost all the computers that use x86 architecture use Little-Endian, and we call this Intel format. In this project, we will convert these instructions represented in Little-Endian into the Big-Endian and store them in the memory. Figures 5 and 6 show how each byte order method is represented in the memory.

# **Single-Cycle Microarchitecture**

In this section, we will mainly discuss the ideas and methods to implement our own single-cycle MIPS simulator. In single-cycle machines, each instruction takes a single clock cycle, and all state updates are made at the end of an instruction’s execution. How long each instruction takes is determined by how long the slowest instruction takes to execute, even though many instructions do not need that long to execute. Therefore, in MIPS ISA, because the instruction that takes the longest execution time is load word (LW) instruction (the reason why will be explained in detail in section 8, evaluation), one cycle of the instruction in MIPS architecture follows the flow of load word (LW) instruction execution. Our implementation of the single-cycle MIPS simulator contains this concept. Also, we tried to implement most of the MIPS instructions except the ones that deal with floating-point.

* 1. **Implemented Datapath**

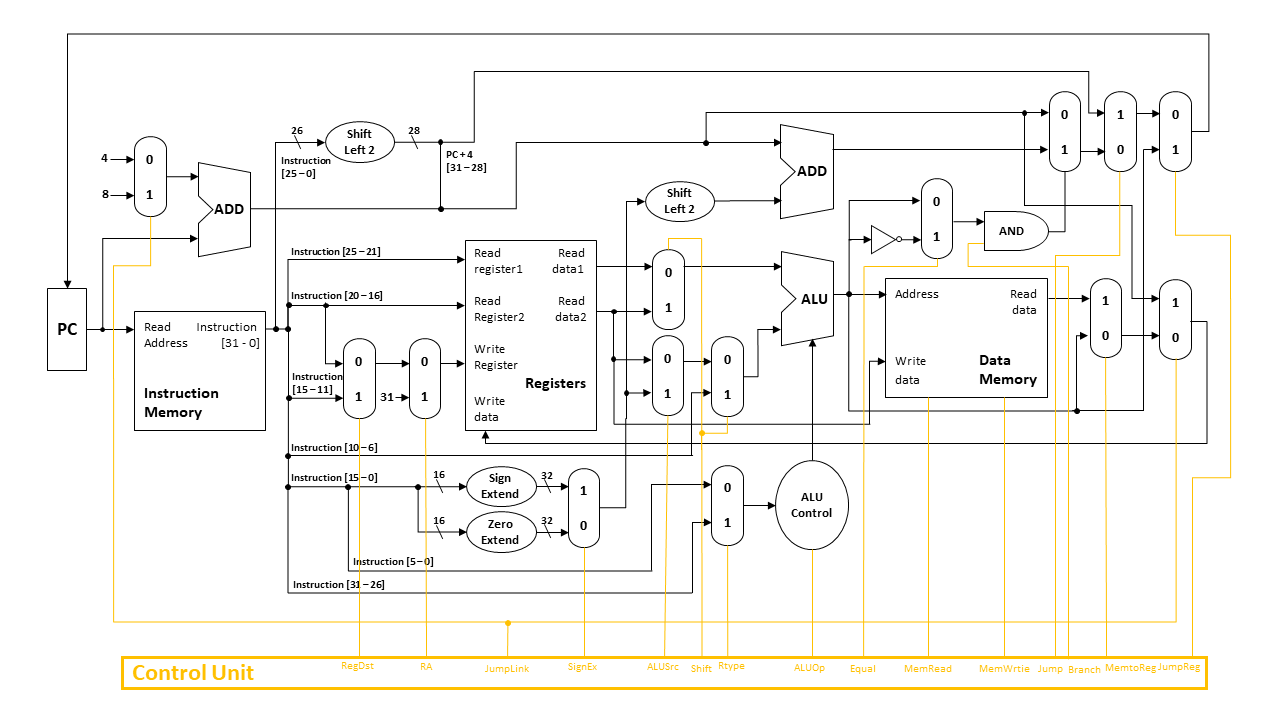


Figure 7 - Single-Cycle MIPS Architecture

Figure 7 shows the total data paths of our own single-cycle MIPS simulator implementation. This structure has three main concepts: Combinational reading, Synchronous writing, and Synchronous memory. In Combinational reading, the output of the read data port is a combinational function of the register file contents and the corresponding read select port. In synchronous writing, the selected register is updated in the positive edge clock transition when the write enablement is asserted, and this cannot affect read output in between clock edges. In synchronous memory, contrast this with memory that tells when data is ready. For example, the ready bit indicates that the read or write is done.

* By including these concepts into our implementation, we can build the simulator that allows following instructions:

1. R-Type and I-Type arithmetic instructions: add, and, nor, or, slt, sub, lui, sll, srl
2. Load Word instructions: lw
3. Store Word instructions: sw
4. Branch Taken and Branch Not Taken instructions: beq, bne
5. Jump instructions: j, jal, jr, jalr
   1. **Control Signal Table**

테이블이(가) 표시된 사진

자동 생성된 설명

Figure 8 - Control Signal Table

In our simulator, every component in the MIPS microprocessor is controlled by these control signals. Data just flows into the components through the data paths as we implement. However, the operations of processing the data in each component are determined by the control signals that are generated by the control unit. That means, these control signals are the main component in the microarchitecture, and it guarantees that all instructions have the same execution cycle in this implementation. Figure 8 shows the generated control signals due to the type of instruction.

* Role of each signal is like below:

1. RegDst: GPR write select according to rt or rd
2. SignEx: Set immediate is whether sign-extended or zero-extended
3. Shift: First ALU input from first GPR read port or shamt filed
4. ALUSrc: second ALU input from second GPR read port or sign-extended 16-bit immediate
5. MemtoReg: Steer ALU result or memory load to GPR write port.
6. RegWrite: GPR write disabled or abled
7. MemRead: Memory read disabled or read port and return the load value
8. MemWrite: Memory writes disabled or abled
9. Branch: Next PC is PC + 4 or based on 16-bit immediate branch target
10. Jump: Next PC is PC + 4 or based on 26-bit immediate jump target
11. JumpReg: Next PC is PC + 4 or based on the 32-bit address in rs register.
12. JumpLink: Next PC is PC + 4 or based on immediate jump target.
13. Rtype: ALU Control component read funct field or opcode
14. Equal: Defines whether the branch instruction is branch equal or branch not equal
15. RA: This bit is only for jump and link instruction. It sets the write register to r31.
16. ALUOp: Defines the type of instruction so that the ALU control unit can determine which instruction type it is
    1. **Program Definition**

Before implementing the single-cycle MIPS simulator in the physical schema, we will state the program definitions that will be used in the real implementations.

* Global Variables

테이블이(가) 표시된 사진

자동 생성된 설명

The interesting thing is that we separated instruction memory and data memory. We can check that we applied the concept of the L1 cache in the implementation.

* Modules and Functions

테이블이(가) 표시된 사진

자동 생성된 설명

# **Implementation**

* 1. **Control Unit (CU)**

텍스트이(가) 표시된 사진

자동 생성된 설명

Figure 9 - CU\_Init

텍스트이(가) 표시된 사진

자동 생성된 설명

Figure 10 - CUOperation

Figure 9 shows the CU\_Init function, which initializes all the control bits to zero. Figure 10 shows the CUOperation function, which controls the control bits due to the given opcode or funct field. According to the control signal table in sections 4-2, we can implement the action of setting the bits like c codes in Figure 10. In the case of ALUOP, which is the ALU operation bit, because they need two bits to determine the ALU control bits, we used conditional statements to distinguish the needed ALU operation bit. By implementing the control unit components in this way, we can generate the control bits and set them in a proper situation.

* 1. **Instruction Memory (IM)**

텍스트이(가) 표시된 사진

자동 생성된 설명

Figure 11 - IM\_Init

텍스트이(가) 표시된 사진

자동 생성된 설명

Figure 12 - IM\_ReadMemory

Figure 9 shows IM\_Init. This function reads the following binary program for every 4 bytes until it reaches the end of the file. Since the data in the binary file is stored in Little-Endian, we converted the byte order into Big-Endian form and stored them in the instruction memory per 1 byte. Figure 10 shows the IM\_ReadMemory function that returns the instruction stored in the following Read Address.

* 1. **Registers (RF)**

텍스트이(가) 표시된 사진

자동 생성된 설명

Figure 13 - RF Functions

Figure 11 shows the functions implemented in the RF module. RF\_Init function initializes the register r29 and r31 in the data represented in the requirement specification, which is 0x0100:0000 and 0xffff:ffff. RF\_Read returns the read data from the given registers and returns them through the first and second GPR read data. In RF\_Write, if the RegWrite control bit is 1, it writes the given data into the following register.

* 1. **Arithmetic Logic Unit (ALU)**

텍스트이(가) 표시된 사진

자동 생성된 설명 테이블이(가) 표시된 사진

자동 생성된 설명

Figure 12 - ALU Control Figure 13 - ALU Control Table

텍스트이(가) 표시된 사진

자동 생성된 설명

Figure 14 - ALU Operation

Figure 12 shows the ALU Control unit. This function returns the ALU control bit (4 bit) according to the opcode (of funct) field. It generates the ALU control bit according to the rule represented in the table in Figure 13. After that, this ALU control flows into the ALU and is processed in the ALUOperation function in Figure 14. It returns the result of the calculation of operands and operators.

* 1. **Data Memory (DM)**

텍스트이(가) 표시된 사진

자동 생성된 설명

Figure 15 - Data Memory

Figure 15 shows the DM\_MemoryAccess function. If the MemRead bit is 1, then it returns the read data from the given address to the GPR read data. If MemWrite bit is 1, then it writes the data into the following address in the data memory per byte.

* 1. **ADDR**

텍스트이(가) 표시된 사진

자동 생성된 설명

Figure 16 - ADDR

Figure 16 shows the adders that are used in the MIPS architecture. Sign extend adder is used for I-type instructions except andi and ori instruction. Zero extend adder is only used for andi and ori. Branch adder is used for branch taken instructions such as branch equal and branch not equal instruction. Jump adder is used for jump instructions and PC adder is used for PC update

* 1. **MUX**

텍스트이(가) 표시된 사진

자동 생성된 설명

Figure 17 - Multiplexer

Figure 17 shows a multiplexer (MUX). We could use 3-input MUX rather than 2-input MUX, however, because we have to add an additional signal for increasing the input of MUX, we decided to implement only 2-input MUX.

* 1. **Main**

텍스트이(가) 표시된 사진

자동 생성된 설명

Figure 18 - Main

The main module follows the sequence of instruction execution in Von Neumann architecture and single-cycle MIPS architecture: IF - ID/RF - EX/AG – MEM - WEB. Figure 18 shows this sequence of execution, and from now on, we will check each phase one by one.

* + 1. **Load Program and Initialize**

텍스트이(가) 표시된 사진

자동 생성된 설명

Figure 19 - Load and Init

Figure 19 shows how we loaded the program to the instruction memory. In the initial phase, we read all of the programs as we represented in 5-1, IM\_Init. IM\_Init function is in the init\_all function. Init\_all function is explained in the next section.

* + 1. **Initialize**

텍스트이(가) 표시된 사진

자동 생성된 설명

Figure 20 - Init

Figure 20 show the init\_all function. It initializes all of the components and variables.

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

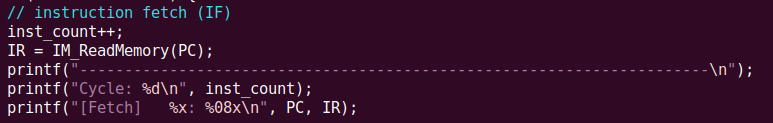


Figure 21 - Instruction Fetch

Figure 21 shows the instruction fetch phase. By using the IM\_ReadMemory function, we can get the instructions in the address of the PC.

* + 1. **Instruction Decode / Register Operand Fetch (ID / RF)**

텍스트이(가) 표시된 사진

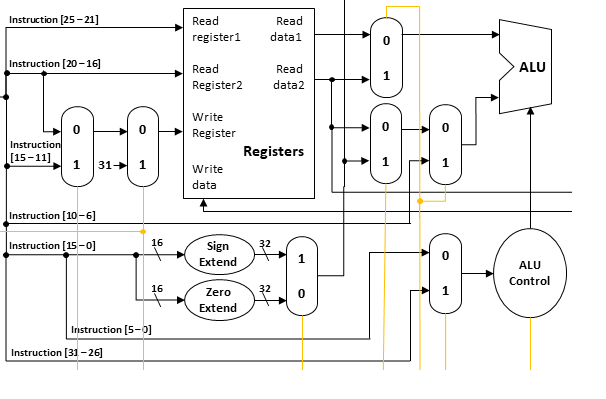
자동 생성된 설명

Figure 21 – Instruction Decode and Register Operand Fetch

Figure 21 shows the ID / RF phase of the program. We decode the instruction by using the Inst\_Decode function and print the result of the decode by using Print\_Decode. Then, we fetch the operand from the first and second GPR read register according to the decoded instruction and return the value of the first and second GPR read data. Then, we set the control unit bits by using the CU\_Operation function.

* + 1. **Execute / Evaluate Memory Address (EX / AG)**

**3.**



**2.**

**1.**

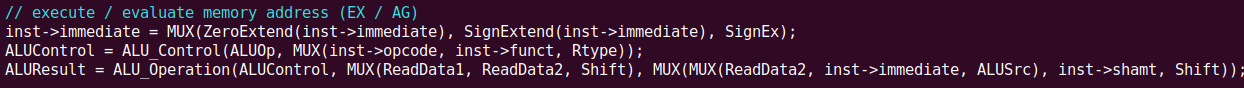


Figure 22 - EX / AG

Figure 22 shows the data paths and their implementation in a C code of EX / AG phases. From top to bottom, each red square shows each implementation of the parts. First, we control whether we will use sign-extended or zero-extended immediate. Second, we decide whether opcode or funct filed flows into the ALU Control unit. Third, we decide the input variables that will flow into the ALU unit and be operated. By implementing the simulator in this way, we can execute the instructions such as R-type and I-type arithmetic operations, and shift operations.

* + 1. **Memory Operand Fetch (MEM)**

텍스트이(가) 표시된 사진

자동 생성된 설명

Figure 23 - MEM

Figure 23 shows the implementation of the memory access function of data memory. The interesting thing is that the number ‘32’ is in the function variable. This refers to the size of the data or memory that we are trying to access. In this project, because the given programs only use load word (LW) and store word (SW) instructions, we only use the size of 32 bits of the data. We just fixed them in the number of 32 bits. However, if there comes a new program that uses instructions that access less than 32 bits, such as store byte (SB) or load half-word unsigned (LHU), then we will implement those in the later project by using this size field.

* + 1. **Store / Writeback Result (WB)**

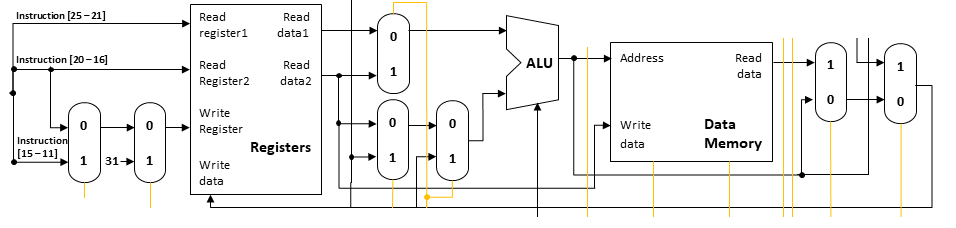




Figure 24 - WB

Figure 24 shows the data paths and their implementation in the c code of the WB phase. From left to right, each red square shows the implementations of each part. First, it determines which register should go into the write register: rt, rd, and r31. Next, it chooses which data should flow into the write data port of the register’s component. By implementing this method, we can write the following data into the given register if the RegWrite bit is 1.

* + 1. **PC Update**



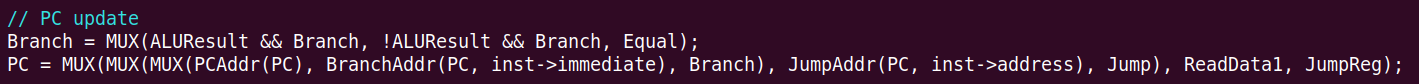


Figure 25 - PC Update

Figure 25 shows the data flow and implementation in the c code of the PC update phase. From left to right, each red square shows the implementations of each part. First, it distinguishes whether the given signal is branch equal or branch not equal. If it is branch equal, it chooses the NAND gate. Else, it chooses AND gate. Second, it decides the PC update value in PC adder, branch adder, jump adder, and First GPR read data, which is data from rs register in instructions such as JR or JALR.

# **Build Environment**

* Build Environments:

1. Linux environment – Vi editor, GCC complier
2. Program is built by using the Makefile.

* Make command:

1. $make main -> build the execution program
2. $make clean -> clean all the object files that builds main

# **Results**

* Results:

1. $ls -l

텍스트이(가) 표시된 사진

자동 생성된 설명

1. $make main

텍스트이(가) 표시된 사진

자동 생성된 설명

1. After $make main

텍스트이(가) 표시된 사진

자동 생성된 설명

1. $./main test\_prog/simple.bin

텍스트이(가) 표시된 사진

자동 생성된 설명

1. $./main test\_prog/simple2.bin

텍스트이(가) 표시된 사진

자동 생성된 설명

1. $./main test\_prog/simple3.bin

텍스트이(가) 표시된 사진

자동 생성된 설명

1. $./main test\_prog/simple4.bin

텍스트이(가) 표시된 사진

자동 생성된 설명

1. $./main test\_prog/gcd.bin

텍스트이(가) 표시된 사진

자동 생성된 설명

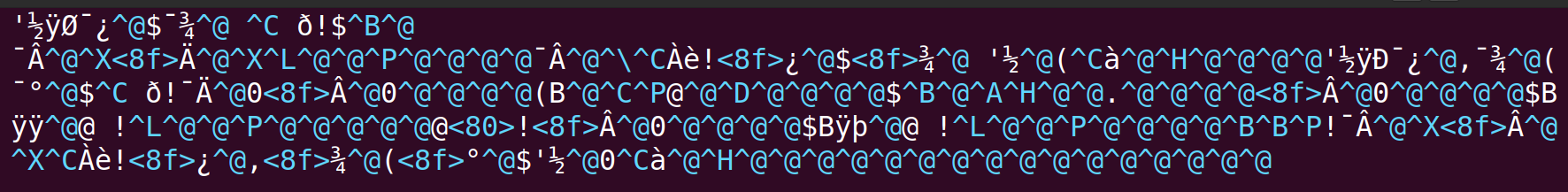
1. $./main test\_prog/input4.bin

텍스트이(가) 표시된 사진

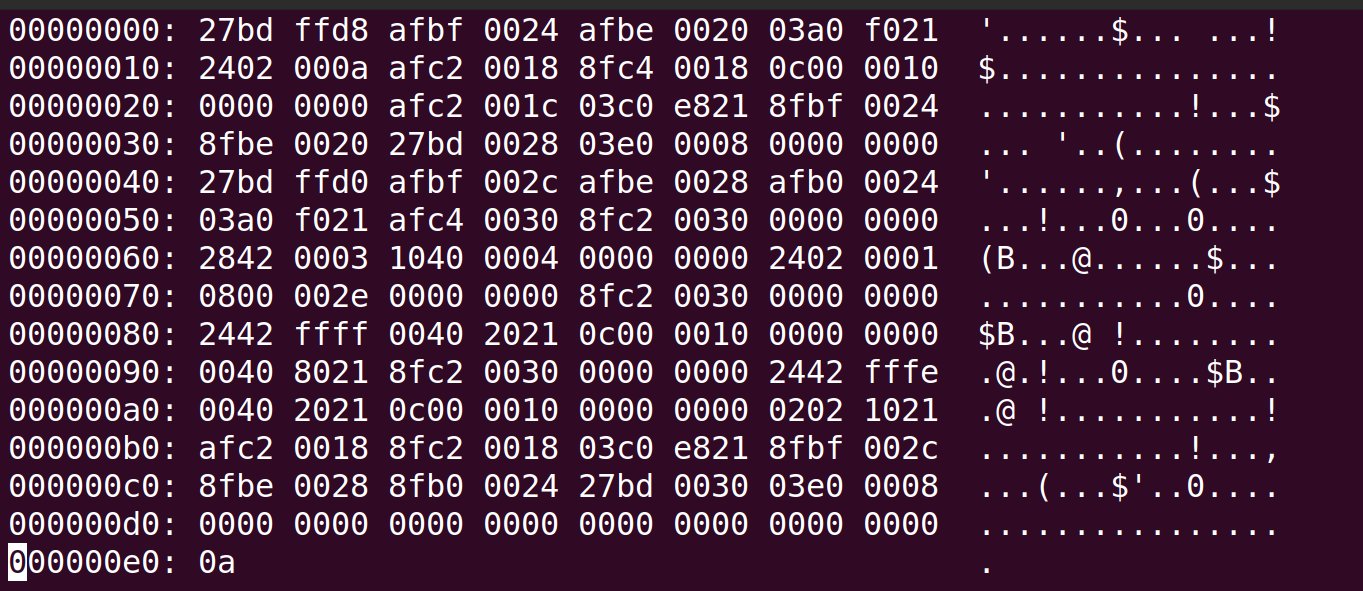
자동 생성된 설명

1. Swapping JAL to JALR in fib.bin

* fib.bin

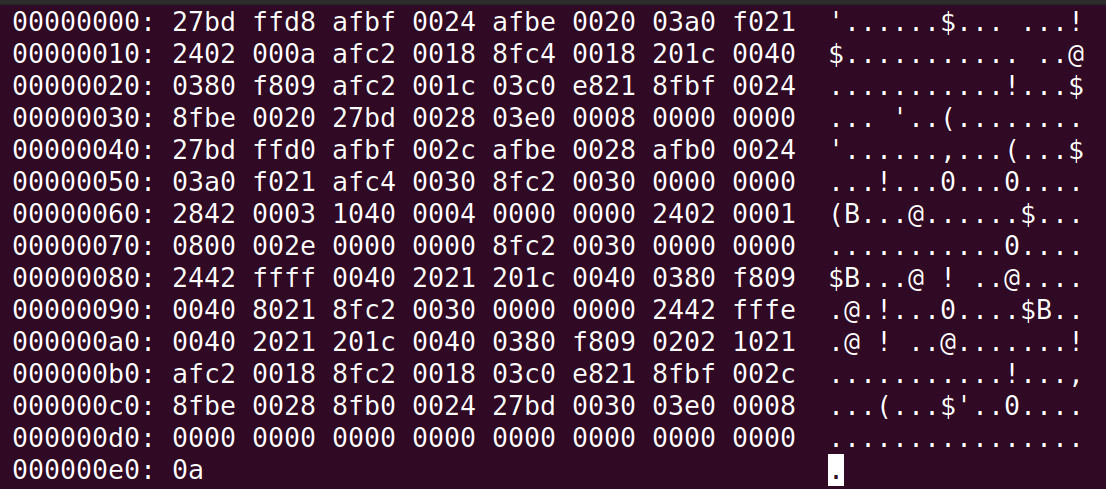


* fib.bin by using :%!xxd command in vim editor



By using the “:%!xxd” command in vim while opening the fib.bin file, we can get the instructions represented in hexadecimal numbers like the figure above. The instructions with red squares are jump and link (JAL) instructions. When the simulator reads this instruction, it stores PC + 8 into the r31 register and jumps to the given address. We can swap this instruction into the set of jump and link register (JALR) instructions.

* fib2.bin by swapping JAL into JALR



We have changed all the jump and link (JAL) instructions to the jump and link register (JALR) instructions. Jump and link register (JALR) instruction stores PC + 8 into rs and jumps to the address that is stored in rd. We swapped the JAL instructions into JALR instructions, but there were some problems. One problem that comes with this swap is that when we change the instruction 0x0c00:0010 into the set of 0x201c:0040 and 0x0380:f809, we must additionally add nop instruction, which is 0x0000:0000, next to the 0x0380:f809. When we add this nop instruction next to it, we must fix all the jump or branch taken instruction’s address due to the additional instructions. For this reason, changing these addresses is much hassle to fix. Therefore, we have added some implementation to the PC adder.

텍스트이(가) 표시된 사진

자동 생성된 설명

For only this time, we added a condition to add only 4 for JALR instruction. It seems weird to put conditions like that. However, in the real situation, the compiler will add nop instructions after the JALR instruction. At this time, we do not have to put the following conditions on the PC adder.

1. $./main test\_prog/fib.bin

텍스트이(가) 표시된 사진

자동 생성된 설명

1. $./main test\_prog/fib2.bin

텍스트이(가) 표시된 사진

자동 생성된 설명

# **Evaluating the Single-Cycle Microarchitecture**

* 1. **Analysis**

테이블이(가) 표시된 사진

자동 생성된 설명

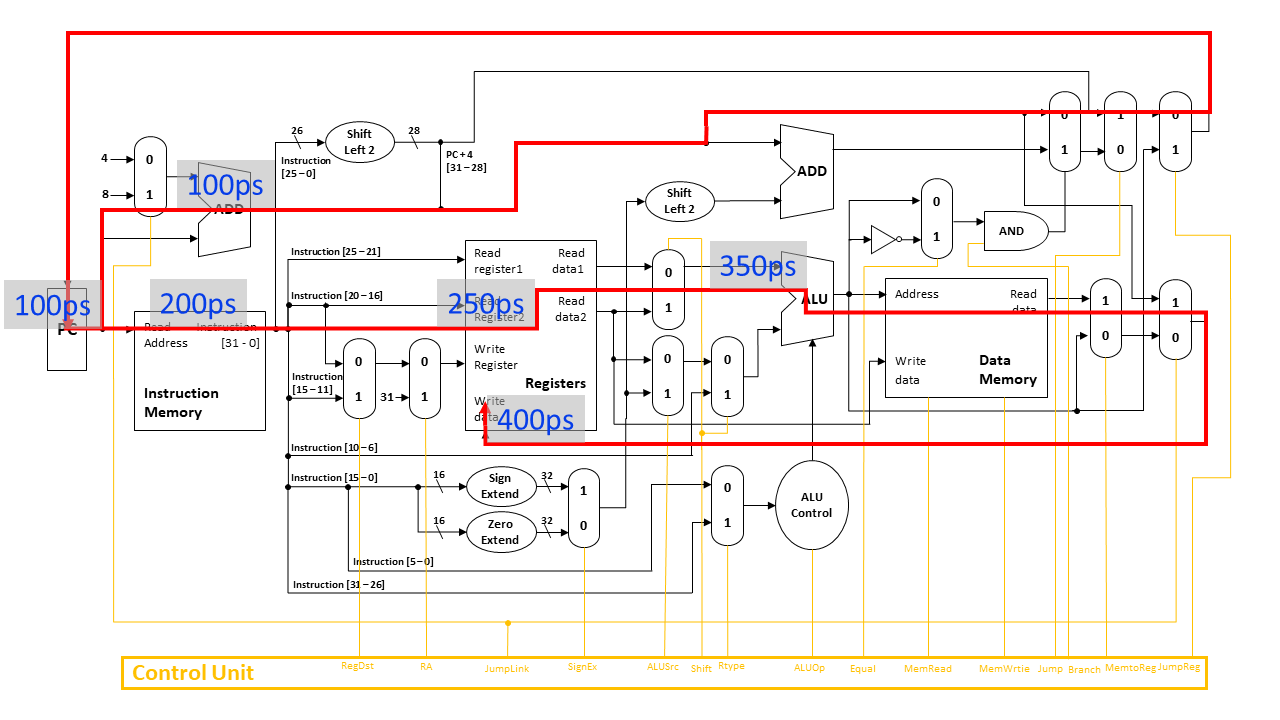
Figure 25 - Instruction Delay Timetable

In short, we have built a single-cycle MIPS simulator with the implementations above. Even though we built a great design for the single-cycle instruction simulator, we still have some questions: Is this a good design? Is it good architecture? It seems like reasonable architecture. However, we cannot say that it is a good design. Here is the reason why. Each instruction takes one cycle to execute. Cycle per instruction (CPI) is strictly 1. Therefore, how long each instruction takes is determined by how long the slowest instruction takes to execute. Even though many instructions do not need that long to execute, they follow the cycle time that the longest instruction has. That means the clock cycle time of the microarchitecture is determined by how long it takes to complete the slowest instruction. It also means that the critical path of the design is determined by the processing time of the slowest instruction. Then, let’s check the MIPS instructions. All size phases (IF – ID/RF – EX/AG – MEM – WB) of the instruction processing cycle take a single machine clock cycle to complete. Does each of the above phases take the same time (latency) for all instructions? Let’s assume the processing time for each unit. Let us assume that reading or writing data in memory takes 200ps (picosecond), ALU and ADDRs take 100ps, reading or writing data in the register file takes 50ps, and other combinational logic takes 0ps. Then we can get the delay time for each unit like in Figure 25. The answer to the question “Do each of the following phases take the same time for all instruction?” is “No they don’t”.

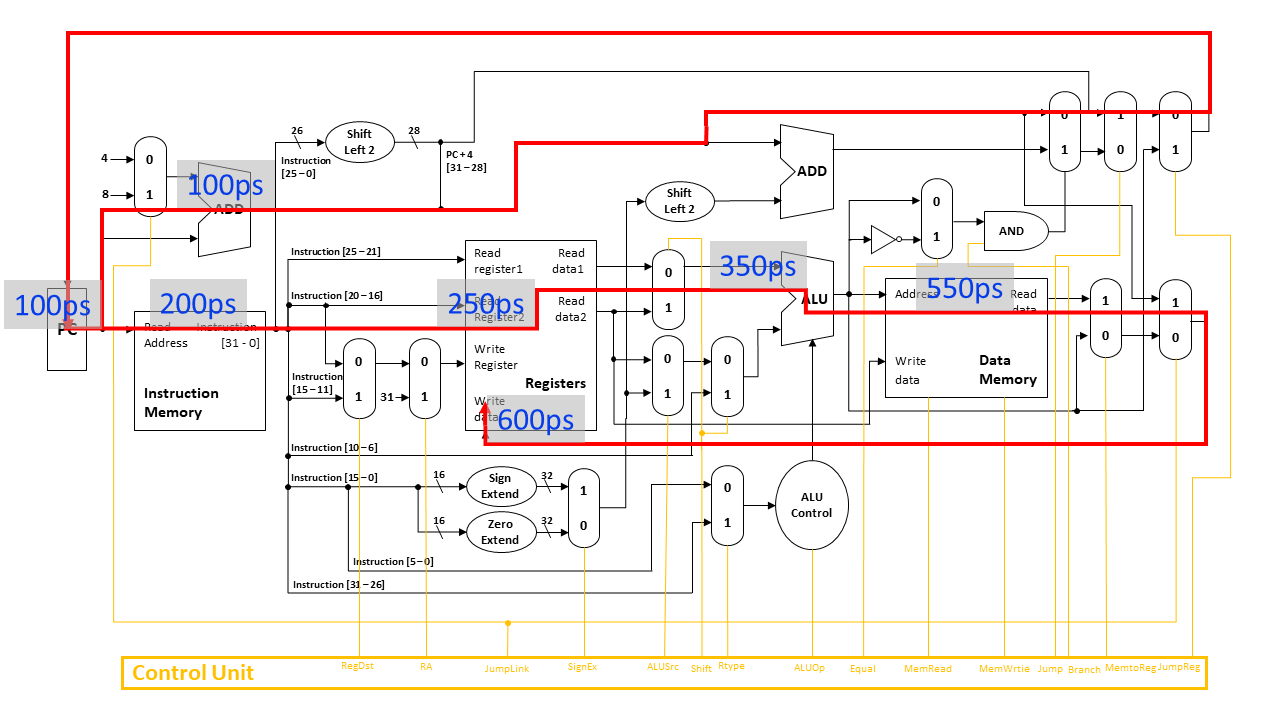
* 1. **Single-Cycle Datapath Analysis**

According to Figure 25, each instruction has a different execution time. By including the information in Figure 25, let’s find the critical path of our single-cycle architecture implementation.

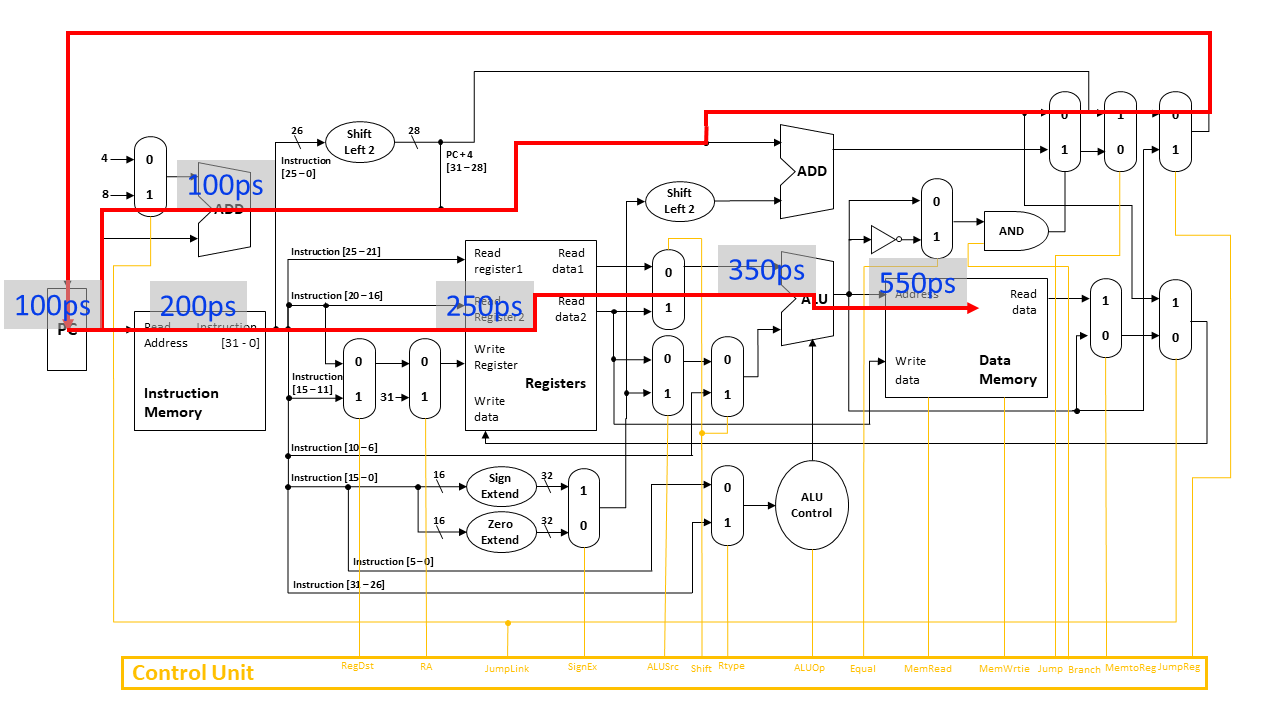
* + 1. **R-Type and I-Type**



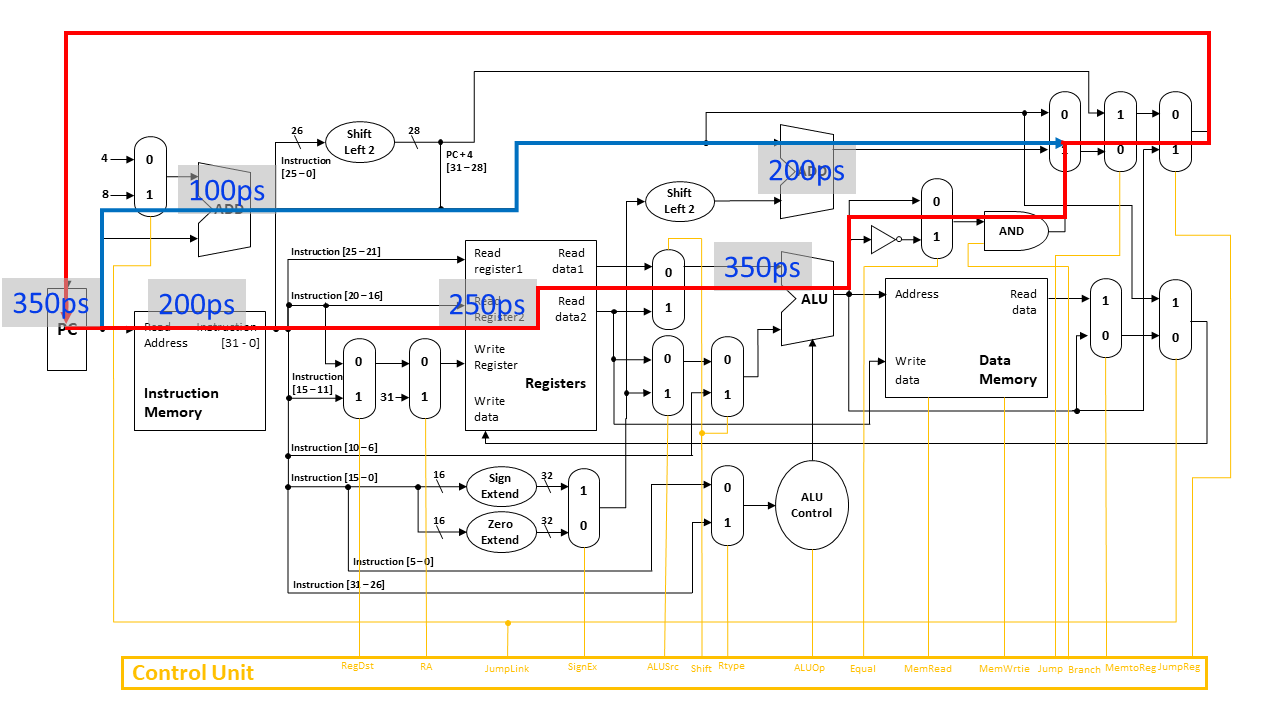
* + 1. **Load Word (LW)**



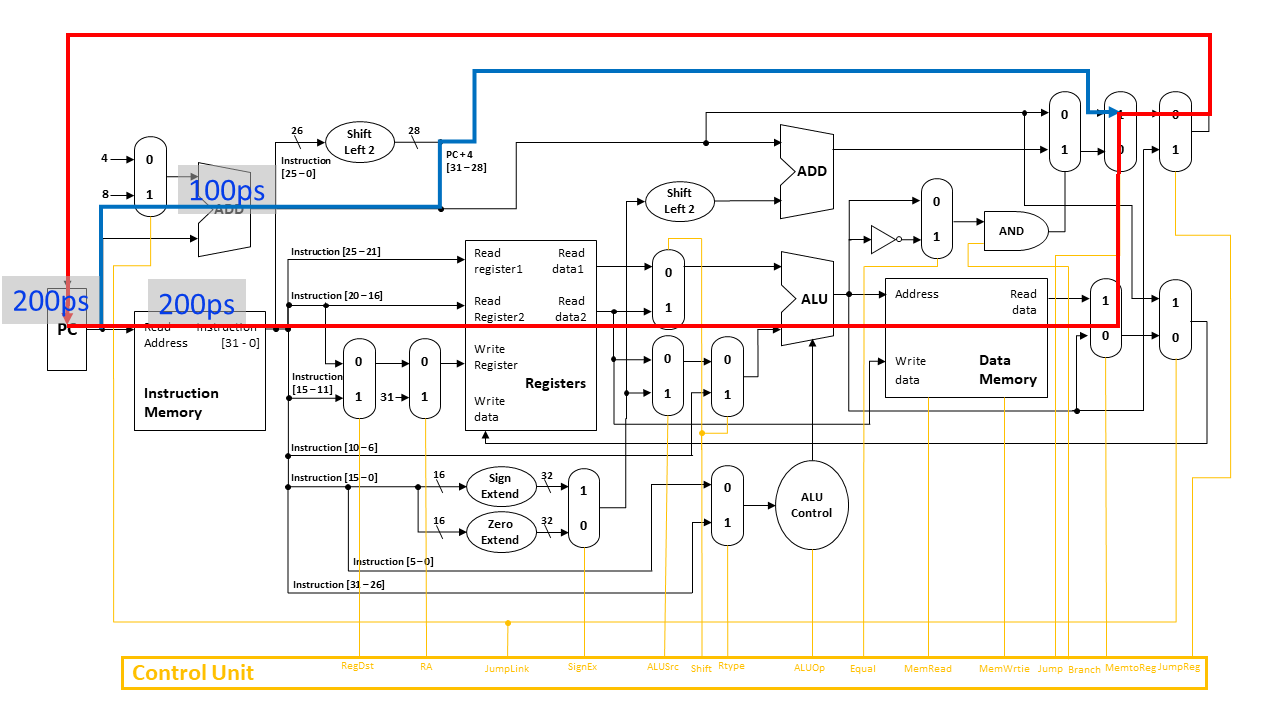
* + 1. **Store Word (SW)**



* + 1. **Branch Taken**



* + 1. **Jump**



* 1. **Conclusion**

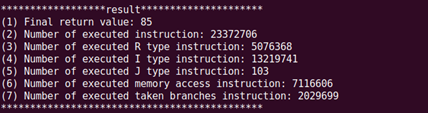


Figure 26 - Result of Input4.bin

테이블이(가) 표시된 사진

자동 생성된 설명

Figure 27 - Frequency of the Instructions in input4.bin

We have checked the cycle time for each instruction through the Figures above. Before we get into the analysis, we must mention the one component that we have passed, the control unit. After the control unit cycle is done, the single cycle is processed. This means that before every single cycle on the above Figures is executed, the control unit cycle, which has constant time complexity, is executed. Therefore, just for the comparison of the delay time for each execution time for different instructions, we can avoid that control unit cycle time complexity.

The instruction that has the slowest execution time is load word (LW) instruction. Therefore, in our single-cycle microarchitecture, each cycle time is set to be the cycle time of load word (LW) instruction, which is 600ps. Then, let’s check the efficiency of this architecture. Figure 26 shows the result of executing the input4.bin program and Figure 27 shows the frequency of each instruction of the input4.bin execution. Let’s calculate the average time per instruction for a single-cycle data path and ideal-cycle data path.

* ·Single-Cycle data path: 600ps
* Ideal-Cycle data path: (39% \* 400ps) + (30% \* 600ps) + (0.4% \* 200ps) + (8.6% \* 350ps) + (22% \* 400ps) = 455ps

As a result, the single-cycle data path is about 1.31 times lower. By checking the comparison of these two cycles, we can say that the single-cycle data path is not that good design. However, this is a very optimistic assumption about memory latency. It can be hard to measure these factors in real life.

# **Conclusion**

From sections 1 to 8, we have shown the concepts that are used to implement our single-cycle MIPS simulator: Von Neumann Computer, Instruction Set Architecture (ISA), MIPS architectures, Data paths of MIPS ISA, L1 Cache, and Byte order (Big-Endian and Little-Endian). By applying these concepts, we implemented a real single-cycle MIPS simulator that follows the flow of the MIPS data path. Then, we evaluated our simulator and the single-cycle microarchitecture. In the reason of the disadvantage that the single cycle's clock of each instruction follows the slowest instruction execution time and that becomes the critical path of the microarchitecture, we evaluated that the single-cycle microarchitecture is not that good design to process the instructions. It is contrived, inefficient, and not necessarily the simplest way to implement an ISA, and it is not easy to optimize and improve the performance. In short, in the next project, we will add the additional implementation on this simulator, which is the pipeline, to build a multi-cycle microarchitecture to solve these limitations.

# **Citations**

* Lee, C. (2022). (rep.). Project1. Seoul: LEE CHANGYOON.
* “The Von Neumann Architecture.” Dive into systems. Accessed March 9, 2022. <https://diveintosystems.org/book/C5-Arch/von.html.>
* Mutlu, O. (2015, January 26). Computer Architecture Lecture 5: Intro to Microarchitecture: Single-Cycle. Carnegie Mellon University.
* A single-cycle MIPS processor - courses.cs.washington.edu. (n.d.). Retrieved April 4, 2022, from <https://courses.cs.washington.edu/courses/cse378/10sp/lectures/lec09-perf.pdf>