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

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.

2. Requirements

Index	Requirement
1	Run MIPS binary program
2	Before the execution, the binary file is loaded into the memory. Read all the file into the memory (data structure)
3	Assume that all register values are all zero, except for r31 and r29. Initial r31 value is 0xFFFF:FFFF Initial r29 value is 0x0100:0000
4	Single-cycle MIPS processor performs all the below five stages. MIPS executes instructions in following stages: 1. Instruction Fetch 2. Instruction Decode 3. Execution 4. Load / Store (or memory) 5. Write back
5	To begin the machine, it moves PC value to the very beginning point of the program, which is address zero.
6	At the end of each cycle, the simulator prints out the changed architectural state from the previous cycle. User visible architectural state: general-purpose registers, PC, and memory
7	At the end of the execution, we need to pint out the calculated result value. We know the end of execution by moving PC to 0xFFFF:FFFF Final return value is stored in V0 (or r2) register.
8	Return the proper result of the programs that are in “test_prog”. test_prog: simple.c, simple2.c, simple3.c, simple4.c, gcd.c, fib.c, input.c
9	Return the output including the information below. Output includes: 1. For each instruction execution – changed architectural state. 2. Final return value (value in 2) 3. Number of executed instructions 4. Number of executed R-type instructions 5. Number of executed I-type instructions 6. Number of executed J-type instructions 7. Number of executed memory access instructions 8. Number of executed taken branches instructions
Optional	1. We can implement separated control signal from data path 2. We can put additional instruction JALR 3. We must make our own sample programs: consider jal to jalr in the ”fib.bin”

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.

3. 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.

3-1. Von Neumann Computer

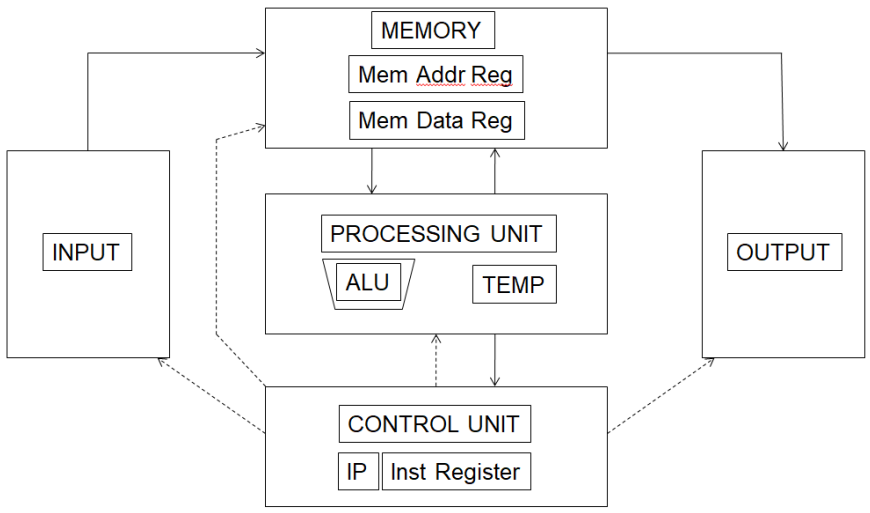


Figure 2 - Von Neumann Model (*Project1, CHANGYOON*)

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*).

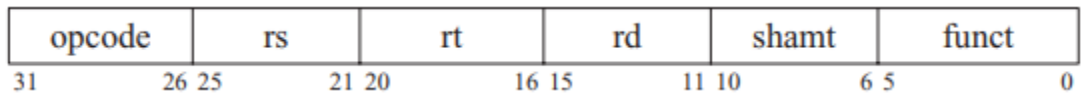
3-2. ISA (Instruction Set Architecture)

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.

3-3. MIPS (Microprocessor without Interlocked Pipeline Stages)

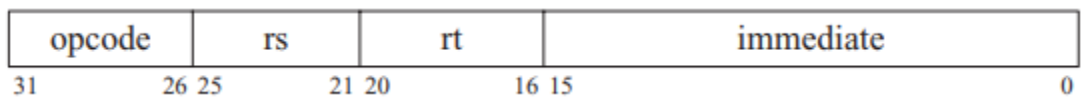
MIPS (Microprocessor without Interlocked Pipeline Stages) is the RISC (Reduced Instruction Set Computer) ISA which has been built by MIPS Technology. The reason is that MIPS ISA is RISC type ISA and all the lengths of instructions are set to be 32bit. According to the different information for CPU needed 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:



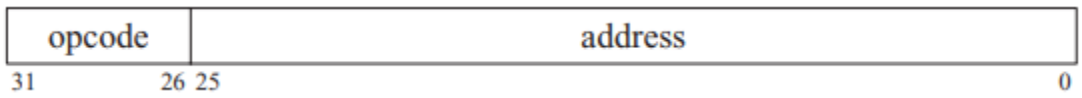
opcode (6bit): opcode is always zero in R-type instructions. MIPS distinguishes the operation type due to the funct field.
rs (5bit): Fist source register index.
rt (5bit): Second source register index.
rd (5bit): Write register index.
shamt (5bit): Shift amount. This field is only used for shift operations.
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).

3-4. Data Paths of MIPS ISA

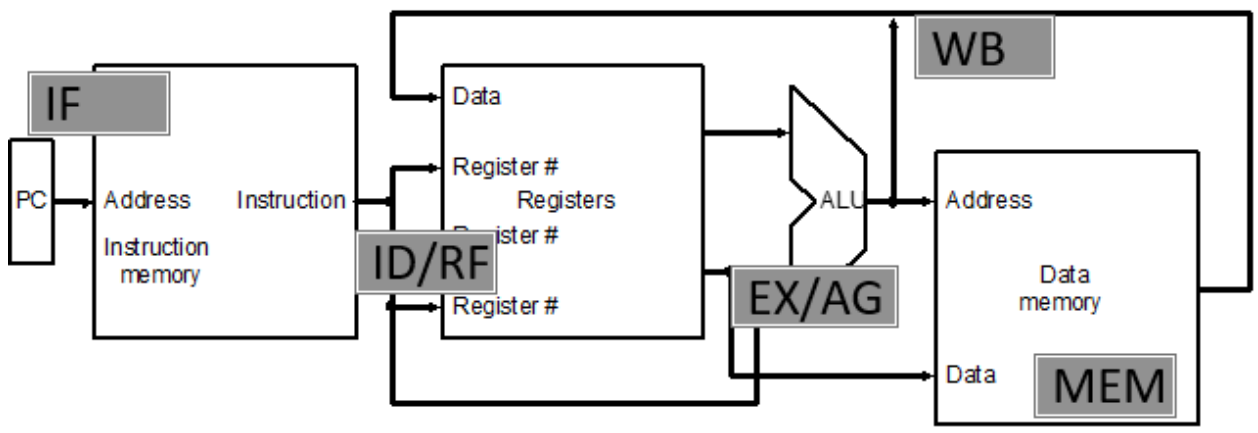
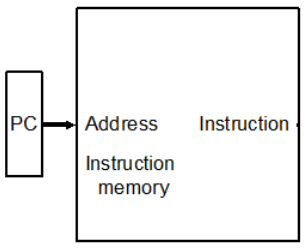


Figure 3 - 5 Generic Steps of MIPS Data Path

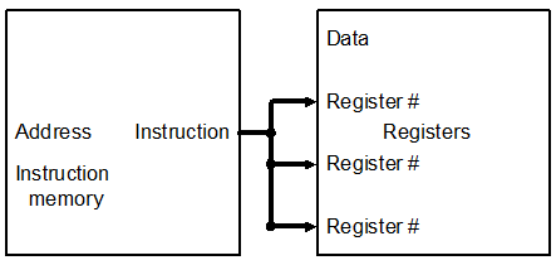
Datapath is the structure that refers to all elements that process and operate the data, address, and registers in the CPU, and we can broadly divide it into 5 phases: IF – ID/RF – EX/AG – MEM – WB. Figure 3 shows the data path for each phase.

- Instruction Fetch (IF)



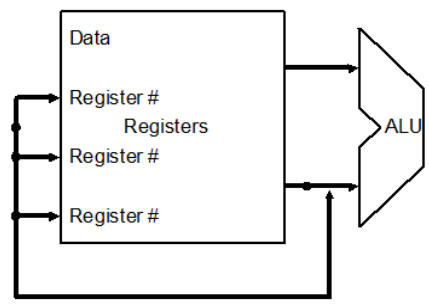
Reads the instruction memory address from the PC and update the PC.

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



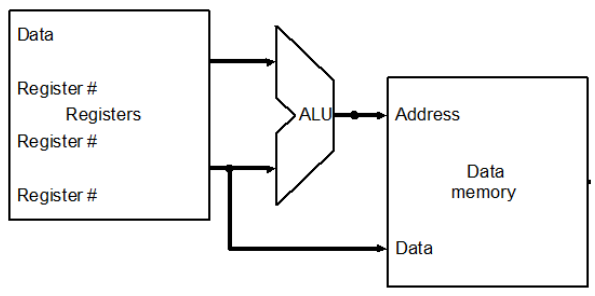
Decode the instruction and read the data that stored in following registers.

- Execute / Evaluate memory address (EX / AG)



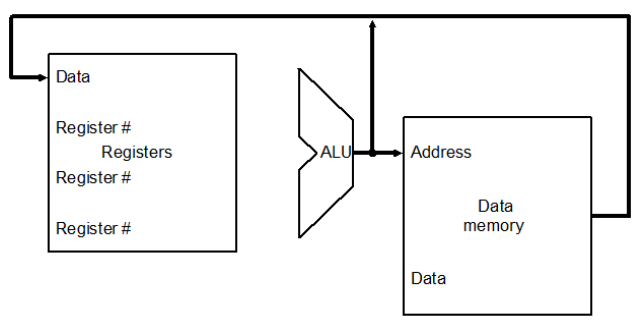
Execute the operation for the following instruction and if it needs to get the memory address, it calculates the memory address.

- 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.

3-5. L1 Cache

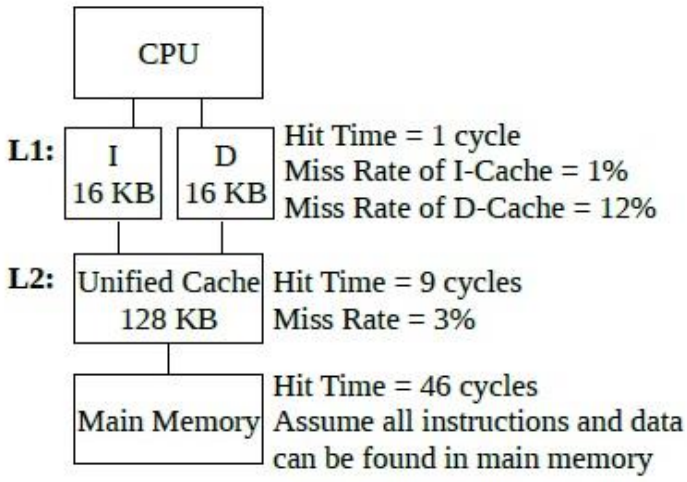


Figure 4 – Cache Hierarchy

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.

3-6. Byte Order

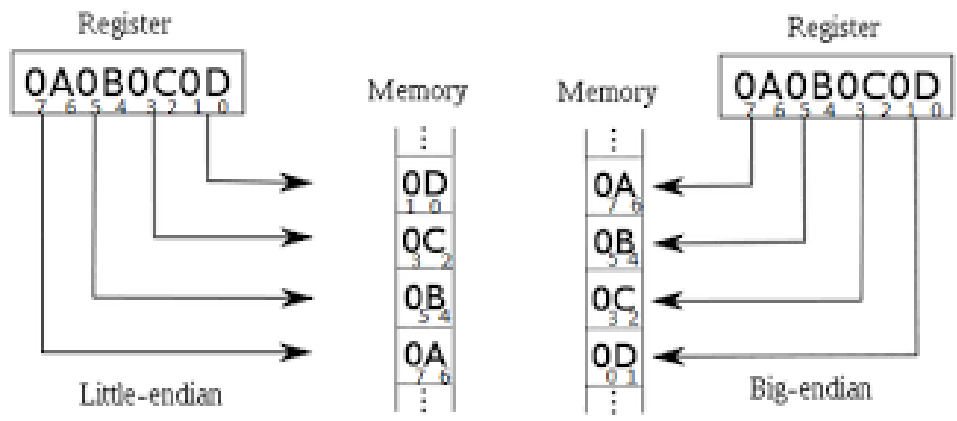


Figure 5 - Byte Order

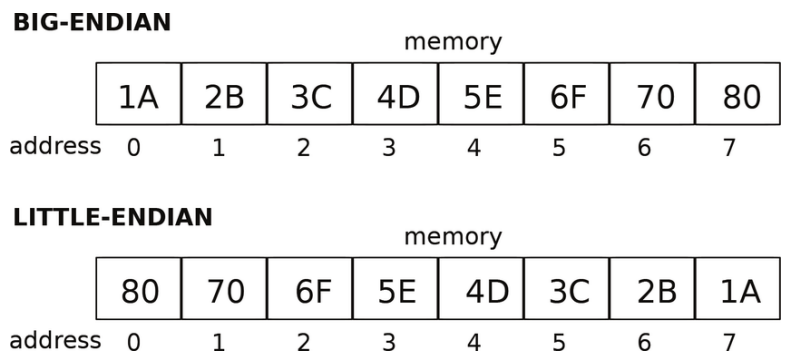


Figure 6 - Endian

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.

4. 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.

4-1. Implemented Datapath

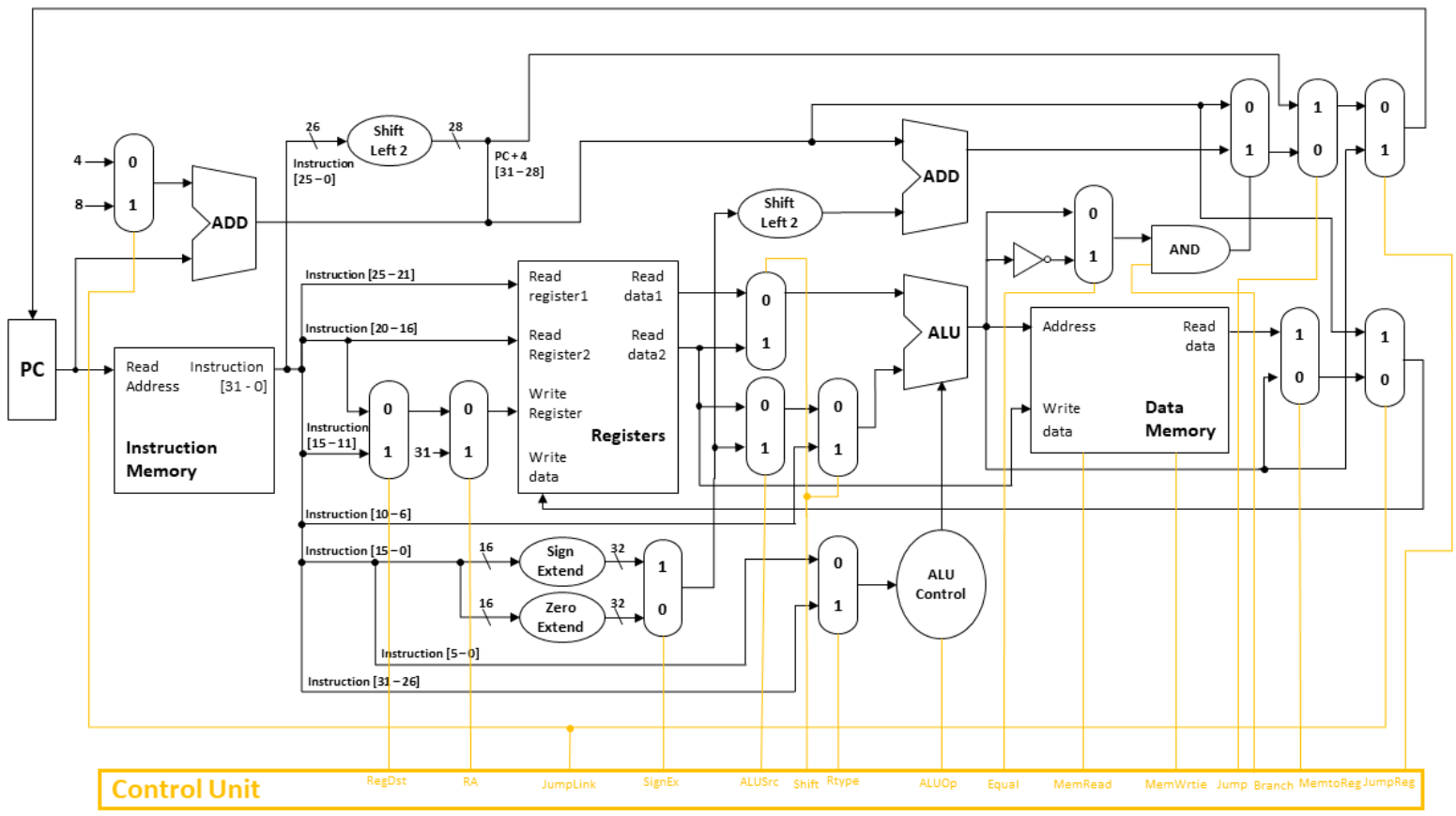


Figure 7 - Single-Cycle MIPS Architecture

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

4-2. Control Signal Table

Control Signal	R-Type (Except JR, JALR)	ANDI / ORI	SLL / SRL	BEQ	BNE	LW	SW	J	JR	JAL	JALR	I-Type
RegDst	1	0	1	0	0	0	X(0)	0	1	0	1	0
SignEx	X(0)	0	X(0)	1	1	1	1	X(0)	X(0)	X(0)	X(0)	1
Shift	0	0	1	0	0	0	0	0	0	0	0	0
ALUSrc	0	1	0	1	1	1	1	X(0)	0	X(0)	0	1
MemtoReg	0	0	0	0	0	1	X(0)	X(0)	X(0)	X(0)	X(0)	0
RegWrite	1	1	1	1	1	1	0	0	0	1	1	1
MemRead	0	0	0	0	0	1	0	0	0	0	0	0
MemWite	0	0	0	0	0	0	1	0	0	0	0	0
Brach	0	0	0	1	1	0	0	X(0)	X(0)	X(0)	X(0)	0
Jump	0	0	0	0	0	0	0	1	0	1	0	0
JumpReg	0	0	0	0	0	0	0	0	1	0	1	0
JumpLink	0	0	0	0	0	0	0	0	0	1	1	0
Rtype	1	0	1	0	0	0	0	0	1	0	1	0
Equal	0	X(0)	X(0)	1	0	X(0)	X(0)	X(0)	X(0)	X(0)	X(0)	0
RA	X(0)	X(0)	X(0)	X(0)	X(0)	X(0)	X(0)	X(0)	X(0)	1	X(0)	X(0)
ALUOp	10	11	10	01	01	00	00	X(00)	10	X(00)	10	11

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

4-3. 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

Variable	Data type	Definition
PROGRAM	char*	Program name
REGSIZE	32	Size of the register
MEMSIZE	0x0100:0000	Size of the memory: 16MB
Inst_type	enum	Enum type for instruction types
Instruction	struct	Contains all the field in R, I, and J type instruction
Control unit variables	bool and uint8_t for ALUOp	Check explanation in 4-2
Inst_count	Int	Instruction counter
R_count	Int	R-type instruction counter
I_count	Int	I-type instruction counter
J_count	Int	J-type instruction counter
M_count	Int	Memory access instruction counter
B_count	int	Branch taken instruction counter
bcond	bool	Branch counter
IR	uint32_t	Instruction register
PC	uint32_t	Program counter
Isnt	instruction*	Instruction
ALUConrol	uint8_t	ALU Control bit
ALUResult	uint32_t	GPR ALU result
ReadData	uint32_t	GPR Read data from data memory
ReadData1	uint32_t	First GPR read data from register
ReadData2	uint32_t	Second GPR red data from register
Register	uint32_t[32]	Register array
InstMemory	uint32_t[MEMSIZE]	Instruction memory
DataMemory	uint32_t[MEMSIZE]	Data memory

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

Module	Functions	Definition
Main	init_all	Initialize all state of the CPU
	terminate	Print the result of the program
	Inst_Decode	Decode the instruction in the format of MIPS ISA
	Print_Decode	Print the decoded instruction according to the type of the instruction
	Print_Execute	Print the changed architectural state from the previous cycle
	MUX	If the given signal is true, it returns the second input data. Else, it returns first input data.
CU	CU_Init	Initialize the state of the control unit. Set all the control bits to zero.
	CU_Operation	Set control bits due to the opcode or funct field.
IM	IM_Init	Initialize the instruction memory to the zero.
	IM_ReadMemory	Read the instruction in instruction memory of the given address.
RF	RF_init	Initialize the general-purpose registers (r0 ~ r31) to the zero except r31 and r29. Initial r31 value is 0xFFFF:FFFF. Initial r29 value is 0x0100:0000.
	RF_Read	Read the data from first and second read register. Each of the read data are stored in first and second GPR read data.
	RF_Write	Write the data into the following write register if the RegWrite control bit is 1.
ALU	ALU_Control	Generates the ALU control bits due to the ALUOp control bit, and opcode or funct field.
	ALU_Operation	Operates the calculation due to the ALU Control bits and returns the result of the calculation.
DM	DM_Init	Initialize data memory to the zero.
	DM_MemoryAccess	If MemRead bit is 1, it reads the data from the following data memory address and returns it. If MemWrite bit is 1, it stores the data into the following data memory address.
ADDR	PCAddr	Returns PC + [4 | 8]
	SignExtend	Returns { 16{immediate[15]}, immediate }
	ZeroExtend	Returns { 16{1b'0}, immediate }
	JumpAddr	Returns { PC+4[31:28], address, 2'b0 }
	BranchAddr	Returns { 14{immediate[15]}, immediate, 2'b0 }

5. Implementation

5-1. Control Unit (CU)

```
// initialize variables in control unit
void CU_Init() {
    RegDst      = 0;
    SignEx      = 0;
    ALUSrc      = 0;
    MemtoReg    = 0;
    RegWrite    = 0;
    MemRead     = 0;
    MemWrite    = 0;
    Branch      = 0;
    Jump        = 0;
    JumpReg     = 0;
    JumpLink    = 0;
    Rtype       = 0;
    Equal       = 0;
    RA          = 0;
    ALUOp       = 0b00;
}
```

Figure 9 - CU_Init

```
// control unit
void CU_Operation(uint8_t opcode,  uint32_t funct) {
    RegDst      = (opcode == 0x00);
    SignEx      = (opcode != ANDI) && (opcode != ORI);
    Shift       = (opcode == 0x00) && ((funct == SLL) || (funct == SRL));
    ALUSrc      = (opcode != 0x00) && (opcode != BEQ) && (opcode != BNE);
    MemtoReg    = (opcode == LW);
    RegWrite    = (opcode != SW) && (opcode != BEQ) && (opcode != BNE) && (opcode != J) && !((opcode == 0x00) && (funct == JR));
    MemRead     = (opcode == LW);
    MemWrite    = (opcode == SW);
    Branch      = (opcode == BEQ) || (opcode == BNE);
    Jump        = (opcode == J) || (opcode == JAL);
    JumpReg     = ((opcode == 0x00) && (funct == JR)) || ((opcode == 0x00) && (funct == JALR));
    JumpLink    = (opcode == JAL) || ((opcode == 0x00) && (funct == JALR));
    Rtype       = (opcode == 0x00);
    Equal       = (opcode == BEQ);
    RA          = (opcode == JAL);

    // load type
    if ((opcode == LW) || (opcode == LL) || (opcode == LHU) || (opcode == LBU))
        ALUOp = 0b00;
    // store type
    else if ((opcode == SW) || (opcode == SH) || (opcode == SC) || (opcode == SB))
        ALUOp = 0b00;
    // branch type
    else if ((opcode == BEQ) || (opcode == BNE))
        ALUOp = 0b01;
    // R type
    else if (opcode == 0x00)
        ALUOp = 0b10;
    // I type
    else
        ALUOp = 0b11;
}
```

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.

5-2. Instruction Memory (IM)

```c
// initialize the instruction memory by loading the program into it
void IM_Init() {
    FILE* fp_in;
    uint32_t bin;
    uint32_t counter = 0x00000;

    fp_in = fopen(PROGRAM, "rb");
    if (fp_in == NULL) {
        printf("program not found\n");
        exit(0);
    }

    // store the instruction into memory in big-edian
    printf("********************mips**********************\n");
    printf("--------------------------------------------------------------------------------\n");
    while (fread(&bin, 1, sizeof(int), fp_in) == 4) {
        printf("%x:\t", counter);
        for (int i = 3; i >= 0; i--) {
            InstMemory[counter + i] = bin & 0xff;
            printf("%02x", InstMemory[counter + i]);
            bin = bin >> 8;
            }
        printf("\n");
        counter += 4;
    }
    printf("--------------------------------------------------------------------------------\n\n");
    fclose(fp_in);
}
```

Figure 11 - IM_Init

```c
// fetch the instruction from the instruction memory
uint32_t IM_ReadMemory(uint32_t ReadAddress) {
    return
        (InstMemory[ReadAddress + 3] << 24) |
        (InstMemory[ReadAddress + 2] << 16) |
        (InstMemory[ReadAddress + 1] <<  8) |
        (InstMemory[ReadAddress + 0] <<  0);
}
```

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.

5-3. Registers (RF)

```c
// initialize the registers
// decreased tha values of data that initially stored in registers because of the limitation of static array size in c
void RF_Init() {
    Register[sp] = 0x1000000;
    Register[ra] = 0xffffffff;
}

// read data from the registers
void RF_Read(uint8_t RdReg1, uint8_t RdReg2) {
    ReadData1 = Register[RdReg1];
    ReadData2 = Register[RdReg2];
}

// write data into the registers
void RF_Write(uint8_t WrtReg, uint32_t WrtData, bool WrtEnable) {
    if ((WrtReg > 0) && (WrtEnable == true))
        Register[WrtReg] = WrtData;
}
```

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.

5-4. Arithmetic Logic Unit (ALU)

```
// ALU control
uint8_t ALU_Control(uint8_t ALUOp, uint8_t opcode) {
    switch (ALUOp)
    {
    // LW or SW
    case 0b00: return 0b0010;

    // BEQ or BNE
    case 0b01: return 0b0110;

    // R type
    case 0b10:
        switch (opcode)
        {
        case ADD:   return 0b0010;
        case ADDU:  return 0b1010;
        case SUB:   return 0b0110;
        case SUBU:  return 0b1110;
        case AND:   return 0b0000;
        case OR:    return 0b0001;
        case NOR:   return 0b1001;
        case SLT:   return 0b0111;
        case SLTU:  return 0b1111;
        case SLL:   return 0b0100;
        case SRL:   return 0b0101;
        }

    // I type
    case 0b11:
        switch (opcode)
        {
        case ADDI:  return 0b0010;
        case ADDIU: return 0b1010;
        case ANDI:  return 0b0000;
        case ORI:   return 0b0001;
        case SLTI:  return 0b0111;
        case SLTIU: return 0b1111;
        case LUI:   return 0b1000;
        }
    }
}
```

Figure 12 - ALU Control

ALUOp	opcode or funct	ALU Control	Operation
00	00100011	0010	ADD
00	00101011	0010	ADD
01	00000100	0110	SUB
01	00000101	0110	SUB
10	00100000	0010	ADD
10	00100001	1010	ADDU
10	00100010	0110	SUB
10	00100011	1110	SUBU
10	00100100	0000	AND
10	00100101	0001	OR
10	00100111	1001	NOR
10	00101010	0111	SLT
10	00101011	1111	SLTU
10	00000000	0100	SLL
10	00000010	0101	SRL
11	00001000	0010	ADD
11	00001001	1010	ADDU
11	00001100	0000	AND
11	00001101	0001	OR
11	00001010	0111	SLT
11	00001011	1111	SLTU
11	00001111	1000	LUI

Figure 13 - ALU Control Table

```
// ALU operation
uint32_t ALU_Operation(uint8_t ALU_Control, uint32_t operand1, uint32_t operand2) {
    switch (ALU_Control)
    {
    case 0b0010: return (long)operand1 + (long)operand2;    // add
    case 0b1010: return operand1 + operand2;                // addu
    case 0b0110: return (long)operand1 - (long)operand2;    // sub
    case 0b1110: return operand1 - operand2;                // subu
    case 0b0000: return operand1 & operand2;                // and
    case 0b0001: return operand1 | operand2;                // or
    case 0b1001: return ~(operand1 | operand2);             // nor
    case 0b0111: return (long)operand1 < (long)operand2;    // slt
    case 0b1111: return operand1 < operand2;                // sltu
    case 0b0100: return operand1 << operand2;               // sll
    case 0b0101: return operand1 >> operand2;               // srl
    case 0b1000: return operand2 << 16;                     // lui
    }
}
```

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.

5-5. Data Memory (DM)

```
// data memory access
void DM_MemoryAccess(uint32_t Address, int size, uint32_t WriteData, bool MemRead, bool MemWrite) {

    // memory read operation
    if ((MemRead == true) && (MemWrite == false)) {
        uint32_t result = 0;
        for (int i = 0; i < (size / 8); i++) {
            result = result << 8;
            result = result | DataMemory[Address + i];
        }
        ReadData = result;
    }

    // memory write operation
    if ((MemRead == false) && (MemWrite == true)) {
        for (int i = (size / 8) - 1; i >= 0; i--) {
            DataMemory[Address + i] = WriteData & 0xff;
            WriteData = WriteData >> 8;
        }
    }
}
```

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.

5-6. ADDR

```
// sign extend
uint32_t SignExtend(uint16_t immediate) {
    if (!(immediate & 0x8000)) return 0x0000ffff & immediate;
    else return 0xffff0000 | immediate;
}

// zero extend
uint32_t ZeroExtend(uint16_t immediate) {
    return 0x0000ffff & immediate;
}

// branch addr
uint32_t BranchAddr(uint32_t PC, uint32_t immediate) {
    return PC + 4 + (immediate << 2);
}

// jump addr
uint32_t JumpAddr(uint32_t PC, uint32_t address) {
    return ((PC + 4) & 0xf0000000) | (address << 2);
}

// PC addr
uint32_t PCAddr(uint32_t PC) {
    return PC + MUX(4, 8, JumpLink);
}
```

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

5-7. MUX

```
// multiplexer (MUX)
uint32_t MUX(uint32_t IN1, uint32_t IN2, bool S) {
    return S ? IN2 : IN1;
}
```

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.

5-8. Main

```c
#include "mips.h"


int main(int argc, char* argv[]) {
        // initialize
    PROGRAM = argv[1];
    init_all();
    printf("***************single cycle**************\n");

    while (PC < 0xffffffff) {
        // instruction fetch (IF)
        inst_count++;
        IR = IM_ReadMemory(PC);
        printf("--------------------------------------------------------------------------------\n");
        printf("Cycle: %d\n", inst_count);
        printf("[Fetch]   %x: %08x\n", PC, IR);

        // instruction decode and register operand fetch (ID / RF)
        Inst_Decode();
        Print_Decode();
        RF_Read(inst->rs, inst->rt);
        CU_Operation(inst->opcode, inst->funct);

        // execute / evaluate memory address (EX / AG)
        inst->immediate = MUX(ZeroExtend(inst->immediate), SignExtend(inst->immediate), SignEx);
        ALUControl = ALU_Control(ALUOp, MUX(inst->opcode, inst->funct, Rtype));
        ALUResult = ALU_Operation(ALUControl, MUX(ReadData1, ReadData2, Shift), MUX(MUX(ReadData2, inst->immediate, ALUSrc), inst->shamt, Shift));

        // memory operand fetch (MEM)
        DM_MemoryAccess(ALUResult, 32, ReadData2, MemRead, MemWrite);

        // store / writeback result (WB)
        RF_Write(MUX(MUX(inst->rt, inst->rd, RegDst), ra, JumpLink), MUX(MUX(ALUResult, ReadData, MemtoReg), PCAddr(PC), JumpLink), RegWrite);

        // PC update
        Branch = MUX(ALUResult && Branch, !(ALUResult && Branch), Equal);
        PC = MUX(MUX(MUX(PCAddr(PC), BranchAddr(PC, inst->immediate), Branch), JumpAddr(PC, inst->address), Jump), ReadData1, JumpReg);

        // show user visible architecture state
        Print_Execute();
    }
    terminate();
    return 0;
}
```

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.

5-7-1. Load Program and Initialize

```c
// initialize
PROGRAM = argv[1];
init_all();
printf("***************single cycle**************\n");
```

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.

5-7-2. Initialize

```c
// init CPU
void init_all() {
    RF_Init();
    IM_Init();
    CU_Init();
    DM_Init();
    inst_count = 0;
    IR = 0x00000000;
    PC = 0x00000000;
    inst = (instruction*)malloc(sizeof(instruction));
}
```

Figure 20 - Init

Figure 20 show the init_all function. It initializes all of the components and variables.

5-7-3. Instruction Fetch (IF)

```
// instruction fetch (IF)
inst_count++;
IR = IM_ReadMemory(PC);
printf("--------------------------------------------------------------------------------\n");
printf("Cycle: %d\n", inst_count);
printf("[Fetch]   %x: %08x\n", PC, IR);
```

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.

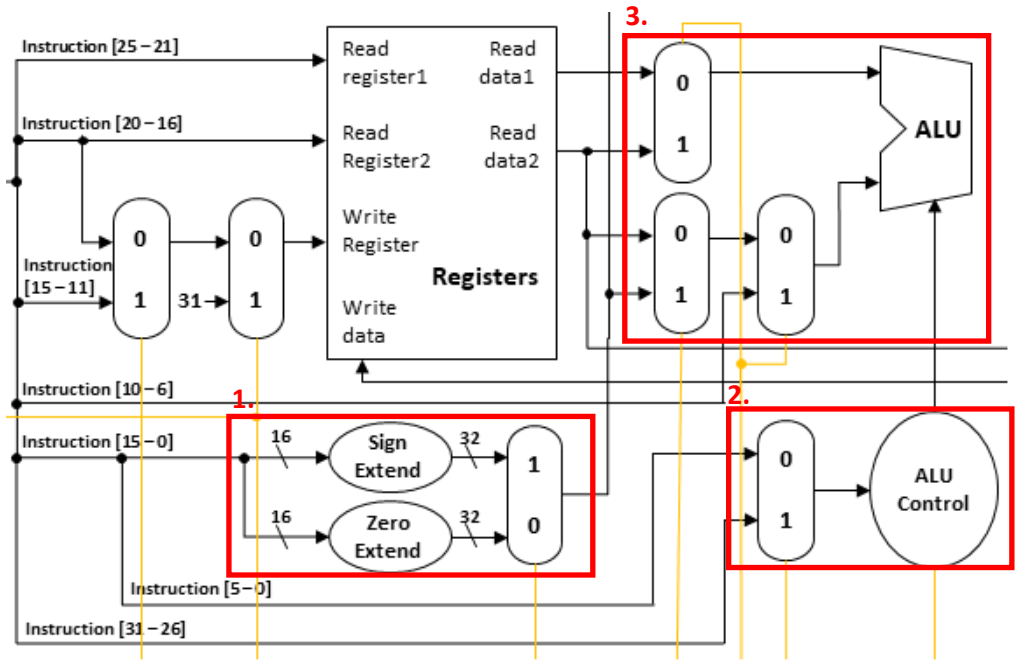
5-7-4. Instruction Decode / Register Operand Fetch (ID / RF)

```
// instruction decode and register operand fetch (ID / RF)
Inst_Decode();
Print_Decode();
RF_Read(inst->rs, inst->rt);
CU_Operation(inst->opcode, inst->funct);
```

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.

5-7-5. Execute / Evaluate Memory Address (EX / AG)




```
// execute / evaluate memory address (EX / AG)
inst->immediate = MUX(ZeroExtend(inst->immediate), SignExtend(inst->immediate), SignEx);
ALUControl = ALU_Control(ALUOp, MUX(inst->opcode, inst->funct, Rtype));
ALUResult = ALU_Operation(ALUControl, MUX(ReadData1, ReadData2, Shift), MUX(MUX(ReadData2, inst->immediate, ALUSrc), inst->shamt, Shift));
```

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.

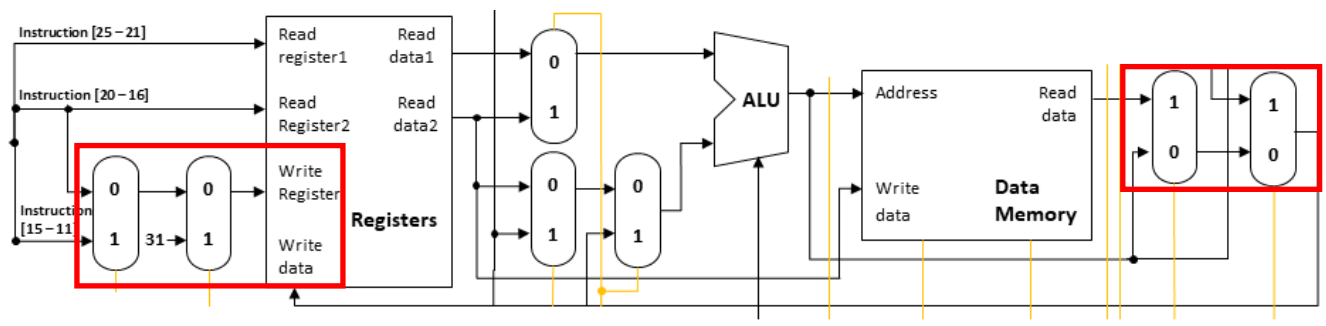
5-7-6. Memory Operand Fetch (MEM)

```
// memory operand fetch (MEM)
DM_MemoryAccess(ALUResult, 32, ReadData2, MemRead, MemWrite);
```

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.

5-7-7. Store / Writeback Result (WB)

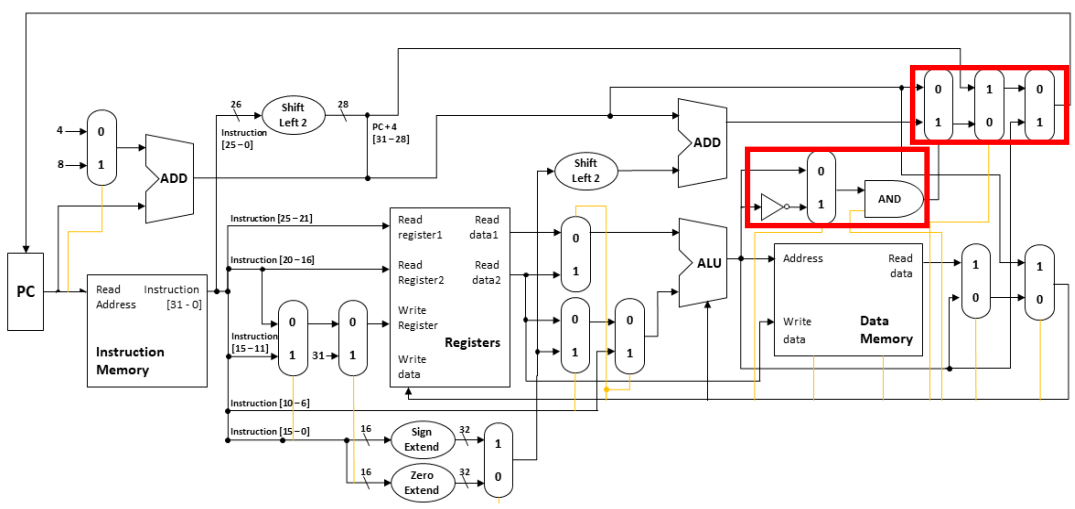



```
// store / writeback result (WB)
RF_Write(MUX(MUX(inst->rt, inst->rd, RegDst), ra, RA), MUX(MUX(ALUResult, ReadData, MemtoReg), PCAddr(PC), JumpLink), RegWrite);
```

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.

5-7-8. PC Update




```
// PC update
Branch = MUX(ALUResult && Branch, !ALUResult && Branch, Equal);
PC = MUX(MUX(MUX(PCAddr(PC), BranchAddr(PC, inst->immediate), Branch), JumpAddr(PC, inst->address), Jump), ReadData1, JumpReg);
```

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.

6. 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

7. Results

- Results:
  1. $ls -l

```
changyoon18@assam:~/camp/hw2/single_cycle$ ls -l
total 52
-rw-rw-r-- 1 changyoon18 changyoon18  617  3월 31 20:10 ADDR.c
-rw-rw-r-- 1 changyoon18 changyoon18 1751  3월 31 09:27 ALU.c
-rw-rw-r-- 1 changyoon18 changyoon18 1587  3월 31 20:16 CU.c
-rw-rw-r-- 1 changyoon18 changyoon18  752  3월 31 09:03 DM.c
-rw-rw-r-- 1 changyoon18 changyoon18 1174  3월 31 09:29 IM.c
-rw-rw-r-- 1 changyoon18 changyoon18  673  4월  1 21:10 instruction.h
-rw-rw-r-- 1 changyoon18 changyoon18 6691  4월  1 10:34 main.c
-rw-rw-r-- 1 changyoon18 changyoon18  405  3월 30 16:11 Makefile
-rw-rw-r-- 1 changyoon18 changyoon18 2443  4월  1 10:14 mips.h
-rw-rw-r-- 1 changyoon18 changyoon18  505  3월 31 09:07 register.h
-rw-rw-r-- 1 changyoon18 changyoon18  594  4월  2 15:41 RF.c
drwxrwxrwx 2 changyoon18 changyoon18 4096  4월  2 01:37 test_prog
```

  2. $make main

```
changyoon18@assam:~/camp/hw2/single_cycle$ make main
gcc -O2 -c -o main.o main.c
gcc -O2 -c -o IM.o IM.c
gcc -O2 -c -o CU.o CU.c
gcc -O2 -c -o RF.o RF.c
gcc -O2 -c -o ALU.o ALU.c
gcc -O2 -c -o DM.o DM.c
gcc -O2 -c -o ADDR.o ADDR.c
gcc -O2 -o main main.o IM.o CU.o RF.o ALU.o DM.o ADDR.o
```

  3. After $make main

```
changyoon18@assam:~/camp/hw2/single_cycle$ ls -l
total 120
-rw-rw-r-- 1 changyoon18 changyoon18   690  4월  3 16:57 ADDR.c
-rw-rw-r-- 1 changyoon18 changyoon18  3056  4월  3 19:36 ADDR.o
-rw-rw-r-- 1 changyoon18 changyoon18  1751  3월 31 09:27 ALU.c
-rw-rw-r-- 1 changyoon18 changyoon18  5352  4월  3 19:36 ALU.o
-rw-rw-r-- 1 changyoon18 changyoon18  1643  4월  3 16:54 CU.c
-rw-rw-r-- 1 changyoon18 changyoon18  3960  4월  3 19:36 CU.o
-rw-rw-r-- 1 changyoon18 changyoon18   752  3월 31 09:03 DM.c
-rw-rw-r-- 1 changyoon18 changyoon18  2872  4월  3 19:36 DM.o
-rw-rw-r-- 1 changyoon18 changyoon18  1174  3월 31 09:29 IM.c
-rw-rw-r-- 1 changyoon18 changyoon18  4248  4월  3 19:36 IM.o
-rw-rw-r-- 1 changyoon18 changyoon18   673  4월  1 21:10 instruction.h
-rwxrwxr-x 1 changyoon18 changyoon18 19312  4월  3 19:36 main
-rw-rw-r-- 1 changyoon18 changyoon18  6677  4월  3 18:45 main.c
-rw-rw-r-- 1 changyoon18 changyoon18 15928  4월  3 19:36 main.o
-rw-rw-r-- 1 changyoon18 changyoon18   405  3월 30 16:11 Makefile
-rw-rw-r-- 1 changyoon18 changyoon18  2453  4월  3 16:52 mips.h
-rw-rw-r-- 1 changyoon18 changyoon18   505  3월 31 09:07 register.h
-rw-rw-r-- 1 changyoon18 changyoon18   594  4월  2 23:32 RF.c
-rw-rw-r-- 1 changyoon18 changyoon18  2920  4월  3 19:36 RF.o
drwxrwxrwx 2 changyoon18 changyoon18  4096  4월  3 16:55 test_proc
```

4. $./main test_prog/simple.bin

```
[Execute] R[29]: 00fffff8 = R[29] + ffffffff8
--------------------------------------------------------------------
Cycle: 2
[Fetch]   4: afbe0004
[Decode]  opcode(2b) rs: 1d rt: 1e immediate: 4
[Execute] M[R[29] + 00000004]: 00000000 = R[30]
--------------------------------------------------------------------
Cycle: 3
[Fetch]   8: 03a0f021
[Decode]  opcode(00) rs: 1d rt: 1e rd: 0 shamt: 0 funct(21)
[Execute] R[30]: 00fffff8 = R[29] + R[0]
--------------------------------------------------------------------
Cycle: 4
[Fetch]   c: 00000000
[Decode]  opcode(00) rs: 0 rt: 0 rd: 0 shamt: 0 funct(0)
[Execute] R[0]: 00000000 = R[0] << R[0]
--------------------------------------------------------------------
Cycle: 5
[Fetch]   10: 03c0e821
[Decode]  opcode(00) rs: 1e rt: 1d rd: 0 shamt: 0 funct(21)
[Execute] R[29]: 00fffff8 = R[30] + R[0]
--------------------------------------------------------------------
Cycle: 6
[Fetch]   14: 8fbe0004
[Decode]  opcode(23) rs: 1d rt: 1e immediate: 4
[Execute] R[30]: 00000000 = M[R[29] + 00000004]
--------------------------------------------------------------------
Cycle: 7
[Fetch]   18: 27bd0008
[Decode]  opcode(09) rs: 1d rt: 1d immediate: 8
[Execute] R[29]: 01000000 = R[29] + 00000008
--------------------------------------------------------------------
Cycle: 8
[Fetch]   1c: 03e00008
[Decode]  opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[Execute] PC: ffffffff = R[31]

**********result**********
(1) Final return value: 0
(2) Number of executed instruction: 8
(3) Number of executed R type instruction: 3
(4) Number of executed I type instruction: 4
(5) Number of executed J type instruction: 0
(6) Number of executed memory access instruction: 2
(7) Number of executed taken branches instruction: 0
******************************************
```

5. $./main test_prog/simple2.bin

```
[Execute] R[30]: 00fffe8 = R[29] + R[0]
--------------------------------------------------------------------
Cycle: 4
[Fetch]   c: 24020064
[Decode]  opcode(09) rs: 0 rt: 2 immediate: 64
[Execute] R[2]: 00000064 = R[0] + 00000064
--------------------------------------------------------------------
Cycle: 5
[Fetch]   10: afc20008
[Decode]  opcode(2b) rs: 1e rt: 2 immediate: 8
[Execute] M[R[30] + 00000008]: 00000064 = R[2]
--------------------------------------------------------------------
Cycle: 6
[Fetch]   14: 8fc20008
[Decode]  opcode(23) rs: 1e rt: 2 immediate: 8
[Execute] R[2]: 00000064 = M[R[30] + 00000008]
--------------------------------------------------------------------
Cycle: 7
[Fetch]   18: 03c0e821
[Decode]  opcode(00) rs: 1e rt: 1d rd: 0 shamt: 0 funct(21)
[Execute] R[29]: 00fffe8 = R[30] + R[0]
--------------------------------------------------------------------
Cycle: 8
[Fetch]   1c: 8fbe0014
[Decode]  opcode(23) rs: 1d rt: 1e immediate: 14
[Execute] R[30]: 00000000 = M[R[29] + 00000014]
--------------------------------------------------------------------
Cycle: 9
[Fetch]   20: 27bd0018
[Decode]  opcode(09) rs: 1d rt: 1d immediate: 18
[Execute] R[29]: 01000000 = R[29] + 00000018
--------------------------------------------------------------------
Cycle: 10
[Fetch]   24: 03e00008
[Decode]  opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[Execute] PC: ffffffff = R[31]

**********result**********
(1) Final return value: 100
(2) Number of executed instruction: 10
(3) Number of executed R type instruction: 3
(4) Number of executed I type instruction: 7
(5) Number of executed J type instruction: 0
(6) Number of executed memory access instruction: 4
(7) Number of executed taken branches instruction: 0
******************************************
```

6. $./main test_prog/simple3.bin

```
[Execute] R[2]: 0 = (R[2] < 00000065)? 1 : 0
--------------------------------------------------------------------
Cycle: 1324
[Fetch]   50: 1440fff3
[Decode]  opcode(05) rs: 2 rt: 0 immediate: fff3
[Execute]  Jump to  00000054 = R[2] != R[0]
--------------------------------------------------------------------
Cycle: 1325
[Fetch]   54: 00000000
[Decode]  opcode(00) rs: 0 rt: 0 rd: 0 shamt: 0 funct(0)
[Execute] R[0]: 00000000 = R[0] << R[0]
--------------------------------------------------------------------
Cycle: 1326
[Fetch]   58: 8fc2000c
[Decode]  opcode(23) rs: 1e rt: 2 immediate: c
[Execute] R[2]: 000013ba = M[R[30] + 0000000c]
--------------------------------------------------------------------
Cycle: 1327
[Fetch]   5c: 03c0e821
[Decode]  opcode(00) rs: 1e rt: 1d rd: 0 shamt: 0 funct(21)
[Execute] R[29]: 00fffe8 = R[30] + R[0]
--------------------------------------------------------------------
Cycle: 1328
[Fetch]   60: 8fbe0014
[Decode]  opcode(23) rs: 1d rt: 1e immediate: 14
[Execute] R[30]: 00000000 = M[R[29] + 00000014]
--------------------------------------------------------------------
Cycle: 1329
[Fetch]   64: 27bd0018
[Decode]  opcode(09) rs: 1d rt: 1d immediate: 18
[Execute] R[29]: 01000000 = R[29] + 00000018
--------------------------------------------------------------------
Cycle: 1330
[Fetch]   68: 03e00008
[Decode]  opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[Execute] PC: ffffffff = R[31]

*****************result********************
(1) Final return value: 5050
(2) Number of executed instruction: 1330
(3) Number of executed R type instruction: 104
(4) Number of executed I type instruction: 920
(5) Number of executed J type instruction: 1
(6) Number of executed memory access instruction: 613
(7) Number of executed taken branches instruction: 102
*******************************************
```

7. $./main test_prog/simple4.bin

```
[Execute] R[30]: 00fffe0 = M[R[29] + 00000020]
--------------------------------------------------------------------
Cycle: 237
[Fetch]   a0: 27bd0028
[Decode]  opcode(09) rs: 1d rt: 1d immediate: 28
[Execute] R[29]: 00fffe0 = R[29] + 00000028
--------------------------------------------------------------------
Cycle: 238
[Fetch]   a4: 03e00008
[Decode]  opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[Execute] PC: 0000001c = R[31]
--------------------------------------------------------------------
Cycle: 239
[Fetch]   1c: 03c0e821
[Decode]  opcode(00) rs: 1e rt: 1d rd: 0 shamt: 0 funct(21)
[Execute] R[29]: 00fffe0 = R[30] + R[0]
--------------------------------------------------------------------
Cycle: 240
[Fetch]   20: 8fbf001c
[Decode]  opcode(23) rs: 1d rt: 1f immediate: 1c
[Execute] R[31]: ffffffff = M[R[29] + 0000001c]
--------------------------------------------------------------------
Cycle: 241
[Fetch]   24: 8fbe0018
[Decode]  opcode(23) rs: 1d rt: 1e immediate: 18
[Execute] R[30]: 00000000 = M[R[29] + 00000018]
--------------------------------------------------------------------
Cycle: 242
[Fetch]   28: 27bd0020
[Decode]  opcode(09) rs: 1d rt: 1d immediate: 20
[Execute] R[29]: 01000000 = R[29] + 00000020
--------------------------------------------------------------------
Cycle: 243
[Fetch]   2c: 03e00008
[Decode]  opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[Execute] PC: ffffffff = R[31]

*****************result********************
(1) Final return value: 55
(2) Number of executed instruction: 243
(3) Number of executed R type instruction: 60
(4) Number of executed I type instruction: 153
(5) Number of executed J type instruction: 11
(6) Number of executed memory access instruction: 100
(7) Number of executed taken branches instruction: 10
*******************************************
```

8. $./main test_prog/gcd.bin

```
[Execute] R[29]: 00ffffd0 = R[29] + 00000020
--------------------------------------------------------------------
Cycle: 1055
[Fetch]   f4: 03e00008
[Decode]  opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[Execute] PC: 00000030 = R[31]
--------------------------------------------------------------------
Cycle: 1056
[Fetch]   30: afc20020
[Decode]  opcode(2b) rs: 1e rt: 2 immediate: 20
[Execute] M[R[30] + 00000020]: 00000001 = R[2]
--------------------------------------------------------------------
Cycle: 1057
[Fetch]   34: 03c0e821
[Decode]  opcode(00) rs: 1e rt: 1d rd: 0 shamt: 0 funct(21)
[Execute] R[29]: 00ffffd0 = R[30] + R[0]
--------------------------------------------------------------------
Cycle: 1058
[Fetch]   38: 8fbf002c
[Decode]  opcode(23) rs: 1d rt: 1f immediate: 2c
[Execute] R[31]: ffffffff = M[R[29] + 0000002c]
--------------------------------------------------------------------
Cycle: 1059
[Fetch]   3c: 8fbe0028
[Decode]  opcode(23) rs: 1d rt: 1e immediate: 28
[Execute] R[30]: 00000000 = M[R[29] + 00000028]
--------------------------------------------------------------------
Cycle: 1060
[Fetch]   40: 27bd0030
[Decode]  opcode(09) rs: 1d rt: 1d immediate: 30
[Execute] R[29]: 01000000 = R[29] + 00000030
--------------------------------------------------------------------
Cycle: 1061
[Fetch]   44: 03e00008
[Decode]  opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[Execute] PC: ffffffff = R[31]

**********result**********
(1) Final return value: 1
(2) Number of executed instruction: 1061
(3) Number of executed R type instruction: 222
(4) Number of executed I type instruction: 637
(5) Number of executed J type instruction: 65
(6) Number of executed memory access instruction: 486
(7) Number of executed taken branches instruction: 73
**********
```

9. $./main test_prog/input4.bin

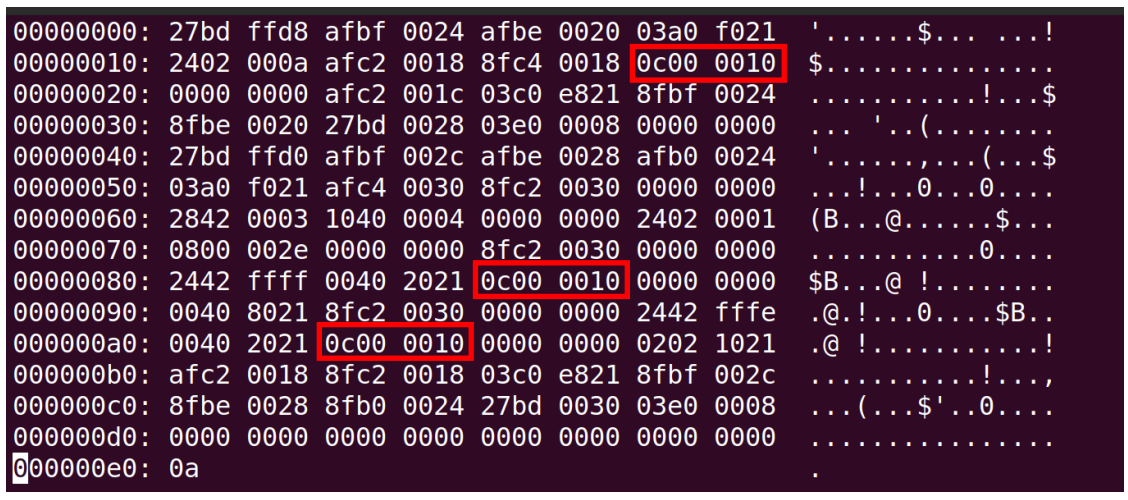
```
[Execute] R[2]: 00000055 = M[R[30] + 000001ac]
--------------------------------------------------------------------
Cycle: 23372700
[Fetch]   18ed8: 00000000
[Decode]  opcode(00) rs: 0 rt: 0 rd: 0 shamt: 0 funct(0)
[Execute] R[0]: 00000000 = R[0] << R[0]
--------------------------------------------------------------------
Cycle: 23372701
[Fetch]   18edc: afc20014
[Decode]  opcode(2b) rs: 1e rt: 2 immediate: 14
[Execute] M[R[30] + 00000014]: 00000055 = R[2]
--------------------------------------------------------------------
Cycle: 23372702
[Fetch]   18ee0: 8fc20014
[Decode]  opcode(23) rs: 1e rt: 2 immediate: 14
[Execute] R[2]: 00000055 = M[R[30] + 00000014]
--------------------------------------------------------------------
Cycle: 23372703
[Fetch]   18ee4: 27dd1c78
[Decode]  opcode(09) rs: 1e rt: 1d immediate: 1c78
[Execute] R[29]: 00ff8010 = R[30] + 00001c78
--------------------------------------------------------------------
Cycle: 23372704
[Fetch]   18ee8: 8fbe7fec
[Decode]  opcode(23) rs: 1d rt: 1e immediate: 7fec
[Execute] R[30]: 00000000 = M[R[29] + 00007fec]
--------------------------------------------------------------------
Cycle: 23372705
[Fetch]   18eec: 27bd7ff0
[Decode]  opcode(09) rs: 1d rt: 1d immediate: 7ff0
[Execute] R[29]: 01000000 = R[29] + 00007ff0
--------------------------------------------------------------------
Cycle: 23372706
[Fetch]   18ef0: 03e00008
[Decode]  opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[Execute] PC: ffffffff = R[31]

**********result**********
(1) Final return value: 85
(2) Number of executed instruction: 23372706
(3) Number of executed R type instruction: 5076368
(4) Number of executed I type instruction: 13219741
(5) Number of executed J type instruction: 103
(6) Number of executed memory access instruction: 7116606
(7) Number of executed taken branches instruction: 2029699
**********
```

10. Swapping JAL to JALR in fib.bin

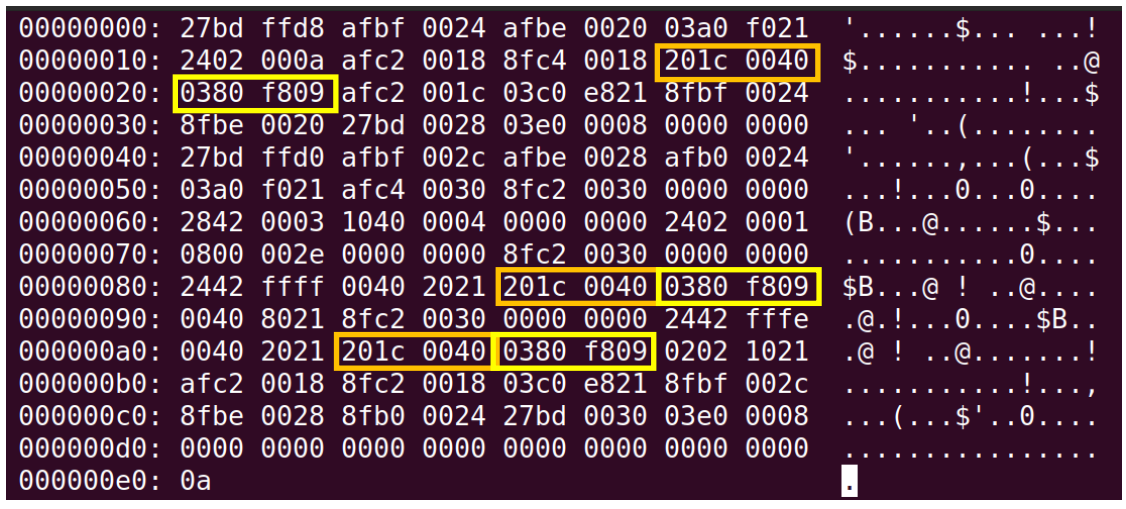
- fib.bin

- 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.

```
// PC addr
uint32_t PCAddr(uint32_t PC) {
    if ((inst->opcode == 0x00) && (inst->funct == JALR)) return PC + 4;
    return PC + MUX(4, 8, JumpLink);
}
```

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.

11. $./main test_prog/fib.bin

```
[Execute] R[29]: 00ffffd8 = R[29] + 00000030
--------------------------------------------------------------------
Cycle: 2673
[Fetch]   cc: 03e00008
[Decode]  opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[Execute] PC: 00000024 = R[31]
--------------------------------------------------------------------
Cycle: 2674
[Fetch]   24: afc2001c
[Decode]  opcode(2b) rs: 1e rt: 2 immediate: 1c
[Execute] M[R[30] + 0000001c]: 00000037 = R[2]
--------------------------------------------------------------------
Cycle: 2675
[Fetch]   28: 03c0e821
[Decode]  opcode(00) rs: 1e rt: 1d rd: 0 shamt: 0 funct(21)
[Execute] R[29]: 00ffffd8 = R[30] + R[0]
--------------------------------------------------------------------
Cycle: 2676
[Fetch]   2c: 8fbf0024
[Decode]  opcode(23) rs: 1d rt: 1f immediate: 24
[Execute] R[31]: ffffffff = M[R[29] + 00000024]
--------------------------------------------------------------------
Cycle: 2677
[Fetch]   30: 8fbe0020
[Decode]  opcode(23) rs: 1d rt: 1e immediate: 20
[Execute] R[30]: 00000000 = M[R[29] + 00000020]
--------------------------------------------------------------------
Cycle: 2678
[Fetch]   34: 27bd0028
[Decode]  opcode(09) rs: 1d rt: 1d immediate: 28
[Execute] R[29]: 01000000 = R[29] + 00000028
--------------------------------------------------------------------
Cycle: 2679
[Fetch]   38: 03e00008
[Decode]  opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[Execute] PC: ffffffff = R[31]

**********result**********
(1) Final return value: 55
(2) Number of executed instruction: 2679
(3) Number of executed R type instruction: 546
(4) Number of executed I type instruction: 1697
(5) Number of executed J type instruction: 164
(6) Number of executed memory access instruction: 1095
(7) Number of executed taken branches instruction: 109
**********
```

12. $./main test_prog/fib2.bin

```
[Execute] R[29]: 00ffffd8 = R[29] + 00000030
--------------------------------------------------------------------
Cycle: 2782
[Fetch]   cc: 03e00008
[Decode]  opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[Execute] PC: 00000024 = R[31]
--------------------------------------------------------------------
Cycle: 2783
[Fetch]   24: afc2001c
[Decode]  opcode(2b) rs: 1e rt: 2 immediate: 1c
[Execute] M[R[30] + 0000001c]: 00000037 = R[2]
--------------------------------------------------------------------
Cycle: 2784
[Fetch]   28: 03c0e821
[Decode]  opcode(00) rs: 1e rt: 1d rd: 0 shamt: 0 funct(21)
[Execute] R[29]: 00ffffd8 = R[30] + R[0]
--------------------------------------------------------------------
Cycle: 2785
[Fetch]   2c: 8fbf0024
[Decode]  opcode(23) rs: 1d rt: 1f immediate: 24
[Execute] R[31]: ffffffff = M[R[29] + 00000024]
--------------------------------------------------------------------
Cycle: 2786
[Fetch]   30: 8fbe0020
[Decode]  opcode(23) rs: 1d rt: 1e immediate: 20
[Execute] R[30]: 00000000 = M[R[29] + 00000020]
--------------------------------------------------------------------
Cycle: 2787
[Fetch]   34: 27bd0028
[Decode]  opcode(09) rs: 1d rt: 1d immediate: 28
[Execute] R[29]: 01000000 = R[29] + 00000028
--------------------------------------------------------------------
Cycle: 2788
[Fetch]   38: 03e00008
[Decode]  opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[Execute] PC: ffffffff = R[31]

**********result**********
(1) Final return value: 55
(2) Number of executed instruction: 2788
(3) Number of executed R type instruction: 655
(4) Number of executed I type instruction: 1806
(5) Number of executed J type instruction: 55
(6) Number of executed memory access instruction: 1095
(7) Number of executed taken branches instruction: 109
**********
```

8. Evaluating the Single-Cycle Microarchitecture

8-1. Analysis

Steps	IF	ID	EX	MEM	WB	Delay
Resources	Inst Mem	RF	ALU	Data Mem	RF	
R-type	200	50	100	X	50	400
I-type	200	50	100	X	50	400
LW	200	50	100	200	50	600
SW	200	50	100	200	X	550
Branch	200	50	100	X	X	350
Jump	200	X	100	X	X	200

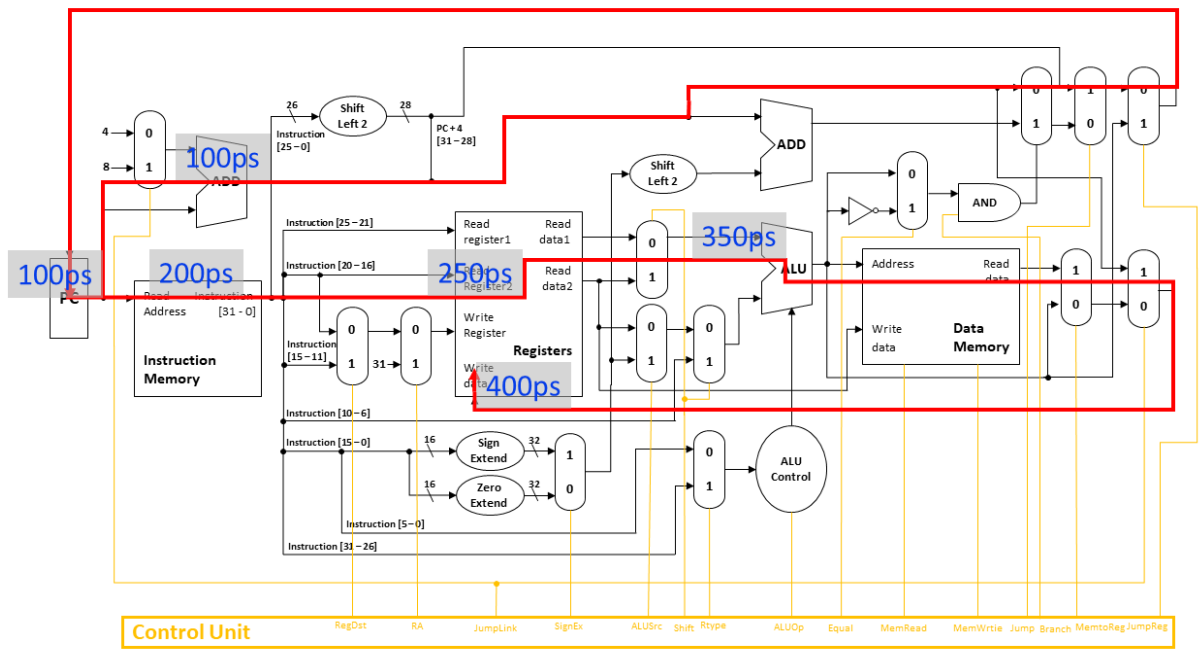
Figure 25 - Instruction Delay Timetable

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".

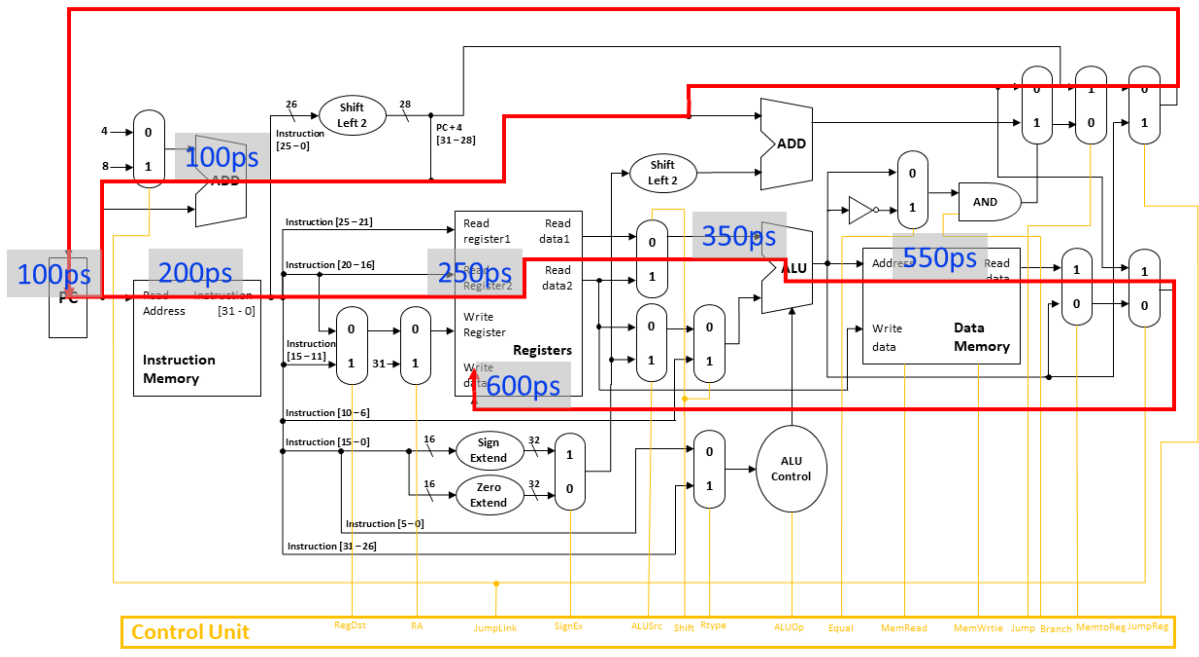
8-2. Single-Cycle Datapath Analysis

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.

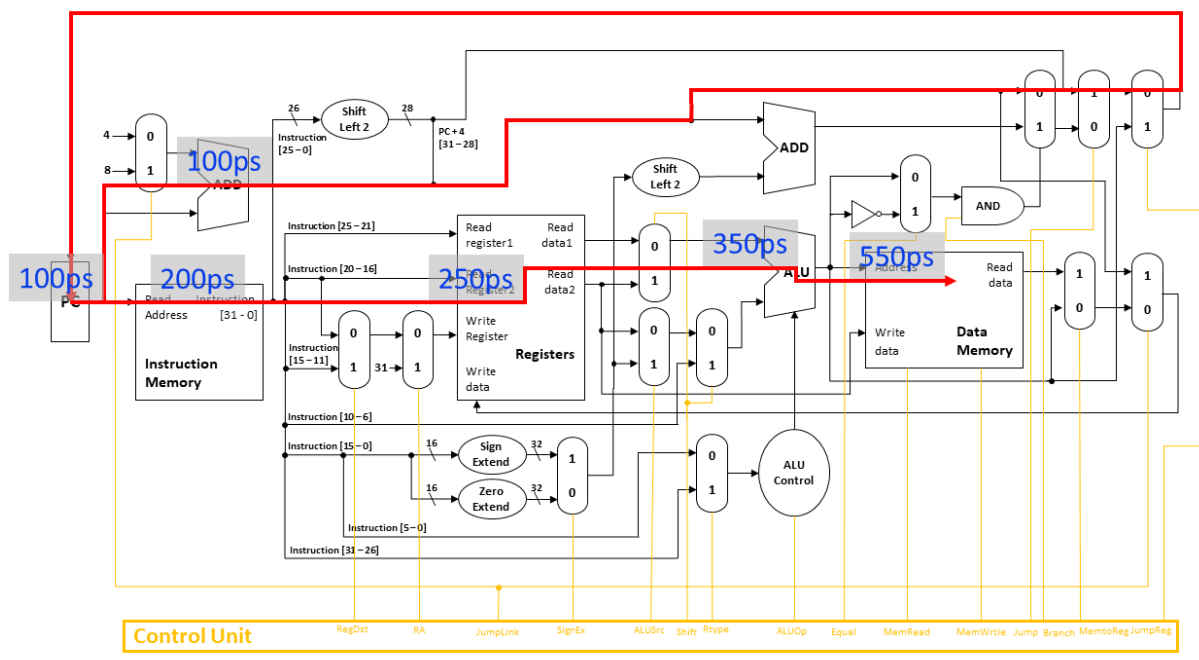
8-2-1. R-Type and I-Type



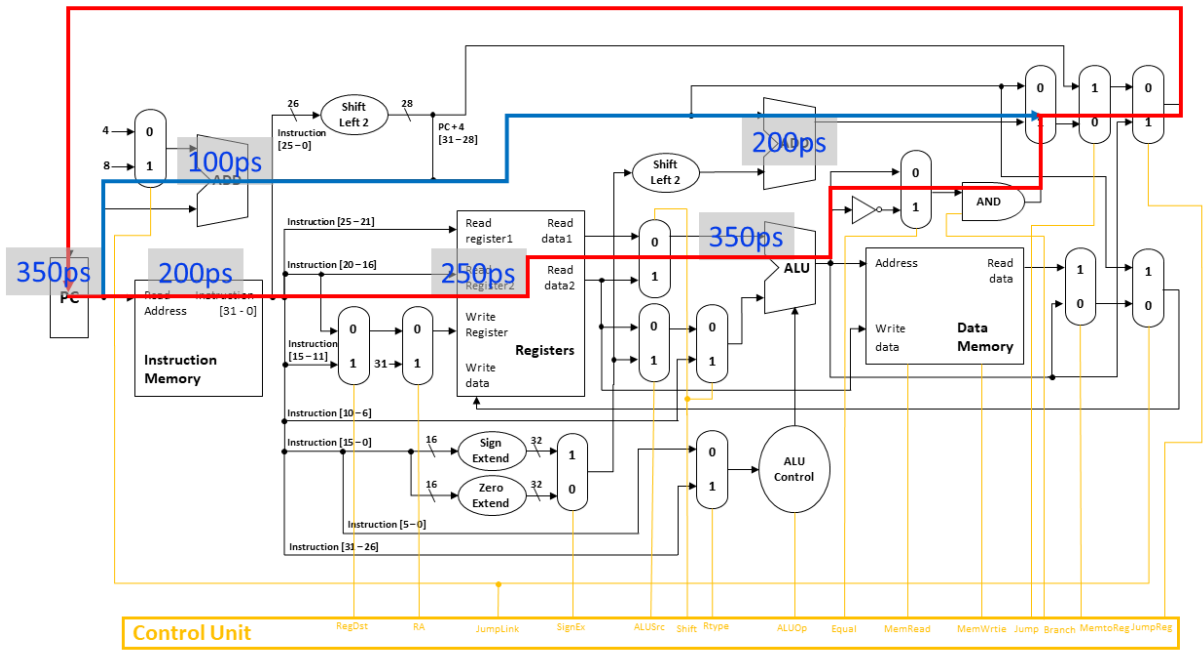
8-2-2.Load Word (LW)



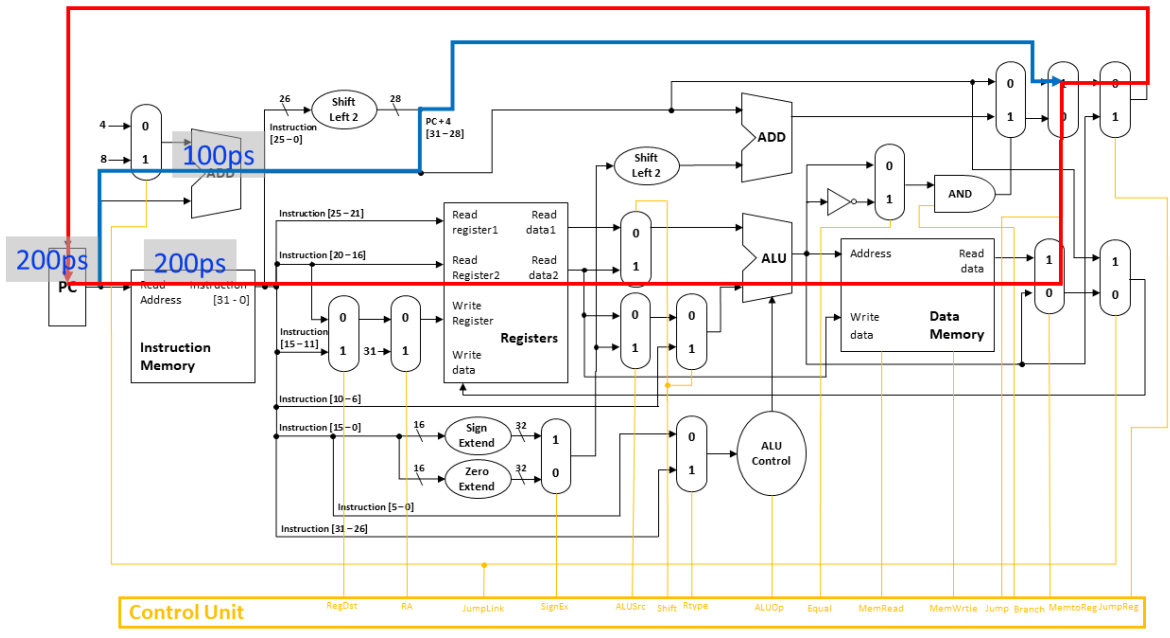
8-2-3.Store Word (SW)



8-2-4.Branch Taken



8-2-5.Jump



8-3. Conclusion

```
*****************result*********************
(1) Final return value: 85
(2) Number of executed instruction: 23372706
(3) Number of executed R type instruction: 5076368
(4) Number of executed I type instruction: 13219741
(5) Number of executed J type instruction: 103
(6) Number of executed memory access instruction: 7116606
(7) Number of executed taken branches instruction: 2029699
********************************************
```

Figure 26 - Result of Input4.bin

Instruction Type	Frequency
Arithmetic	39%
Memory Access	30%
Jump	0.4%
Branch	8.6%
Else	22%

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.

9. 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.

10. 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