DANKOOK UNIVERSITY

DANKOOK UNIVERSITY

REPORT

Project #2
Single-cycle
MIPS

과 목 : 컴퓨터 구조 및 모바일 프로세서
학 과 : 모바일 시스템공학과
학 번 : 32184801
이 름 : 하승원

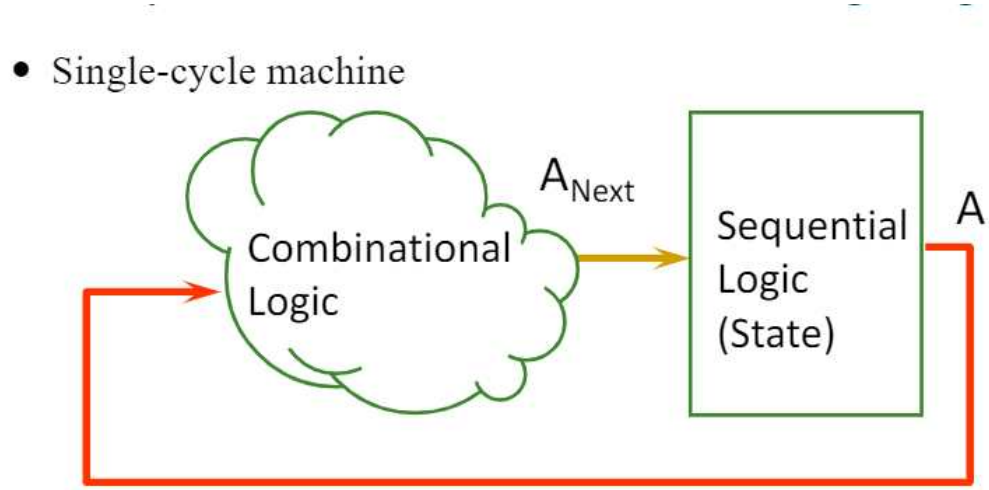
- 목차-

1. Intro

이번 프로젝트는 컴퓨터 작동 구조인 microarchitecture를 이해하고 또 구동되는 과정을 코드로 구현해 보면서 명령어들이 어떻게 CPU에서 처리되고 또 결과를 도출하는지를 알아보기 위한 프로젝트입니다. 먼저 microarchitecture 에서는 초기 방식인 single cycle 구조의 작동방식에 대한 이론과 그리고 single cycle에서 명령어가 datapath에서 어떻게 처리가 되는지 설명하겠습니다. 그리고 그 이후에는 실질적으로 single cycle에서 효율성과 그리고 처리 시간을 지정하는 방법에 관해 설명하겠습니다. 마지막으로 코드 분석과 결괏값에 대하여 말씀드리겠습니다.

2. What is single cycle?

2.1 Single cycle



-single cycle

sinle cycle 은 한번 clock cycle time에서 명령어가 모든 단계의 수행과정을 거치는 것을 말합니다. 그리고 모든 수행 과정이 끝나면 모든 상태들이 업데이트가 됩니다.

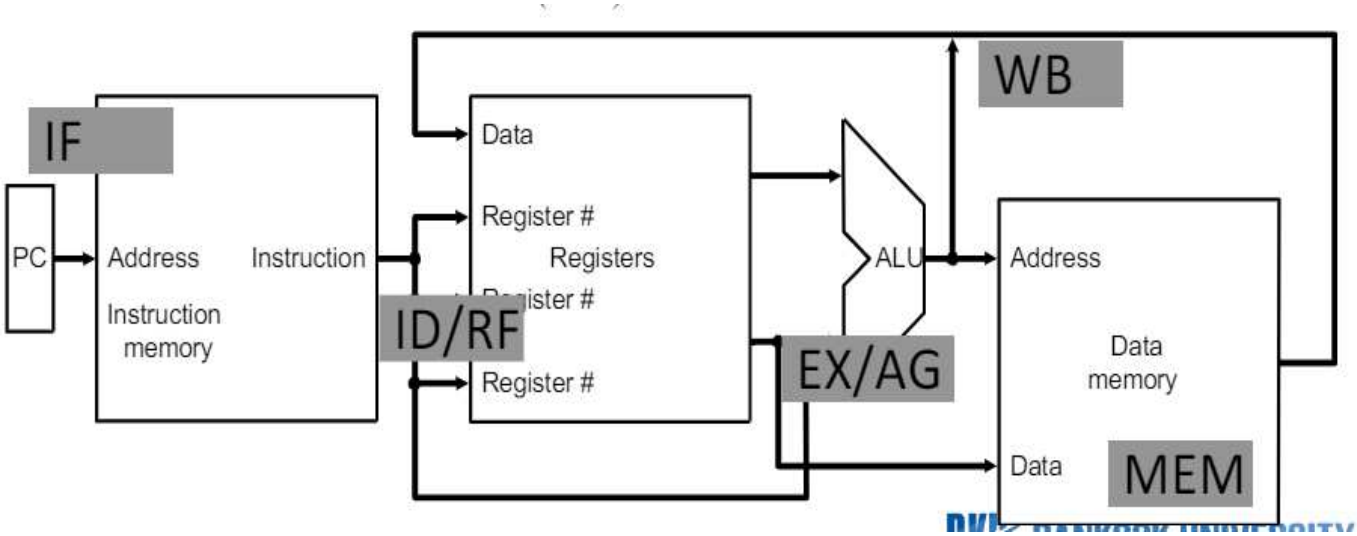
2.1.1 what is cycle??

사이클이란 명령어가 단계적으로 명령을 수행하는 것을 말합니다. 사이클은 아래와 같이 총 6가지 단계로 구분되며 모든 단계를 거치면 한 번의 clock cycle이 돌았다고 말합니다. 하지만 모든 명령어가 이 6가지 단계를 모두 필요한 것은 아니다.

1. Fetch - Fetch 단계에서는 pc(programm counter)을 이용하여 명령어들이 저장되어있는 메모리에서 pc 주소에 있는 명령어들이 저장되어있는 메모리에서 pc 명령어를 가지고 오는 단계이다. 그다음 다음 명령어를 실행하기 위해 pc에 4(주소 값은 4bit씩 이동)를 더한다.
2. Decode - Decode 단계에서는 읽어온 명령어를 통해서 먼저 opcode를 decode 시킨다. 그리고 레지스터에 주솟값을 보내기 전에 명령어 종류에 따라 어떤 레지스터 주소가 들어가는 것들을 구분 지어준다. 그리고 여기서 opcode를 이용해서 각각 데이터를 보내는 걸 결정하는 control signal 들을 초기화해준다.
3. Evaluate Address - 이 단계에서는 명령어 저장 메모리에서 받아온 주솟값을 이용해서 레지스터에 값들을 읽고 그 값들을 다음 단계로 보내준다.
4. Fetch Operands - 실행 전 실행하기 위해 ALU(Arithmetic logic unit)에 데이터가 들어가기 전에 어 떤 값이 ALU로 들어가야 하는지 정해준다.
5. Execute - 전해진 데이터값들을 이용하여 ALU를 실행하고 결괏값을 다음 단계로 보낸다.
6. Store Result - 이 단계에서는 모든 실행을 마친 값을 메모리나 아니면 레지스터에 저장한다.

2.2 Data path

data patch에서는 근본적인 6단계 사이클이 아닌 5단계로 나뉜다.



1.Instruction fetch (IF)□

2.Instruction decode and register operand fetch (ID/RF)□

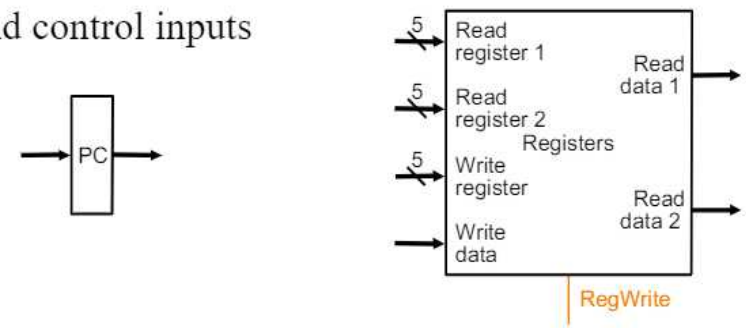
3.Execute/Evaluate memory address (EX/AG)

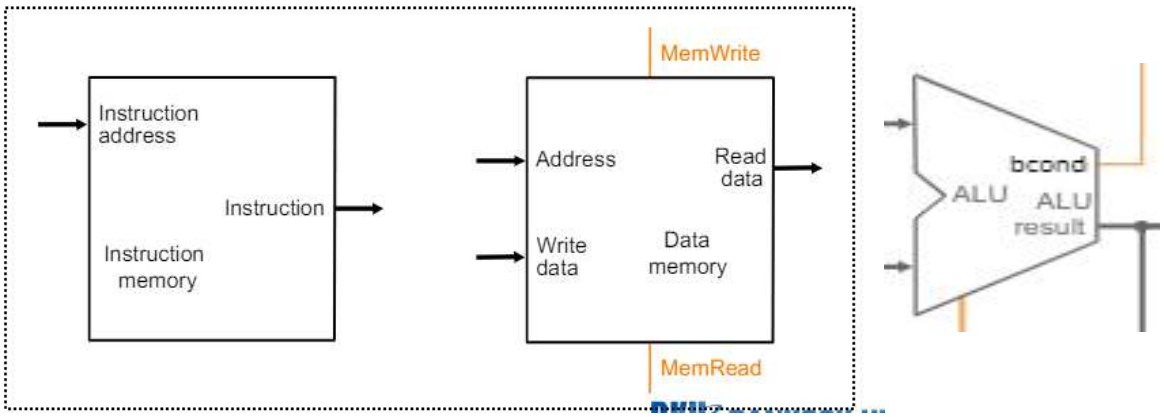
4.Memory operand fetch (MEM)

5. Store/write back result (WB

2.2.1 state element

- Data and control inputs





-PC: PC(program counter)는 현재 진행 중인 명령어의 위치를 가리킨다. 그리고 명령어에 따라 PC값은 바뀔 수 있다.

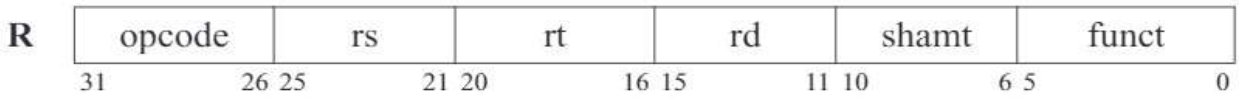
-Register : 명령어 종류의 따라서 값을 받아오고 레지스터의 값을 반환한다. 이 부분은 control signal에 의하여 결정된다.

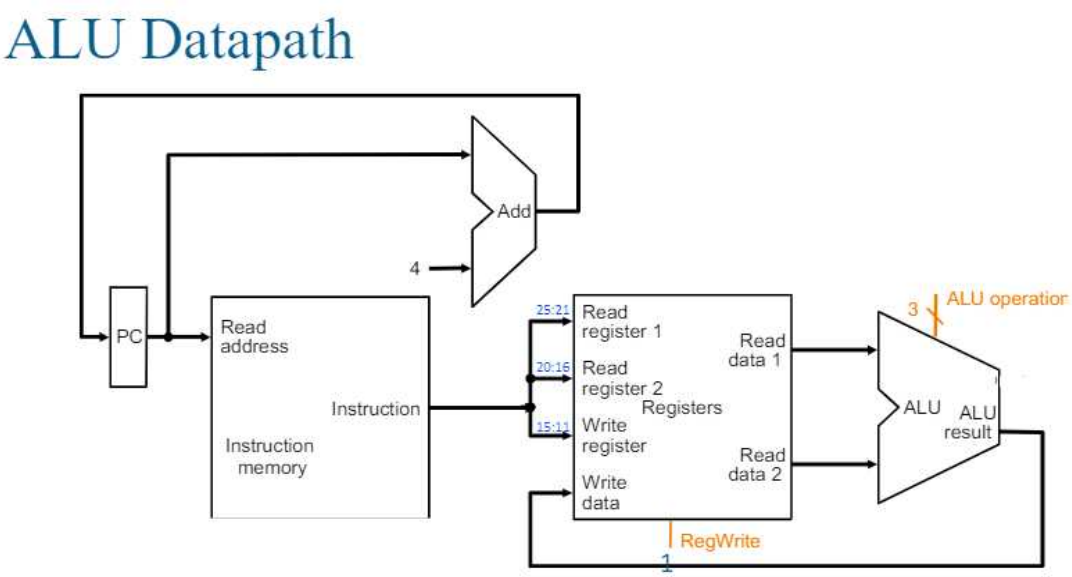
-Instruction memory: 명령어들이 저장되어있는 메모리에서 PC값을 통해 명령어를 도출한다.

-data memory : 모든 실행을 완료하고 데이터를 저장하거나 반환하는 메모리 부분이다. 이 data memory에서는 signal 신호를 이용해서 값을 저장하거나 아니면 반환한다.

-ALU(Arithmetic logic unit): ALU는 연산을 실행하는 부분으로 명령어의 종류에 따라 각각 다른 연산을 한다.

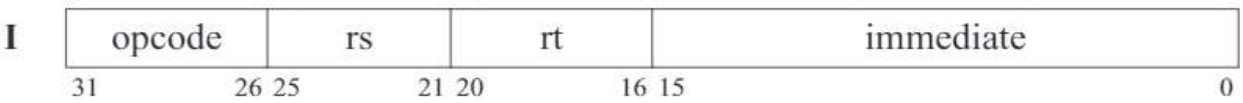
2.2.2 R-type

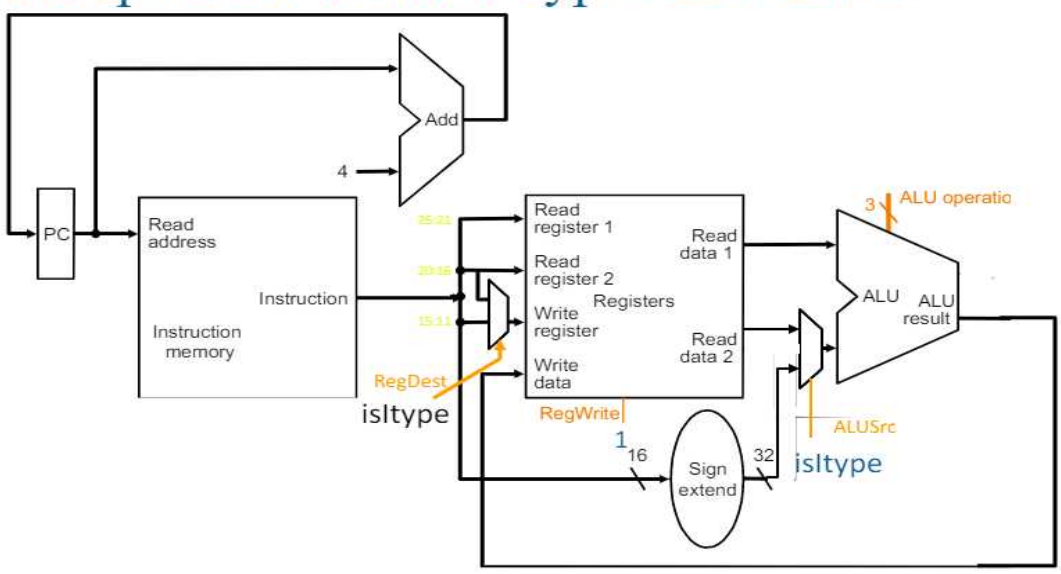




R-type 명령어는 총 32bit로 구성되어있으며 각각의 bit 자릿수마다 다른 값들을 담고 있다. 보통 opcode는 R-type에서는 0x0 값이어서 명령어 구분은 funct에서 해줄 수 있다. 그래서 R-type들은 연산 처리를 할 때 opcode와 funct를 이용한 ALUoperation signal을 통해 연산한다. 여기서 R-type의 특별한 점은 shamt(shift amout) 부분이다. R-type 명령어에서는 shamt 명령어를 통해서 연산과 bit 시프트가 한 명령어에 처리하는데 가능하다.
R-type data path에서는 먼저 2가지의 control signal이 있다. Regwirte 와 ALU operation signal이다. 보통 R-type에서는 opcode가 모두 같아서 ALU operation은 funct을 통해서 신호를 처리해주고 또 연산 후의 결괏값을 RegWrite 신호를 통해서 레지스터에 저장할지 안 할지 결정해준다.

2.2.3 I-type



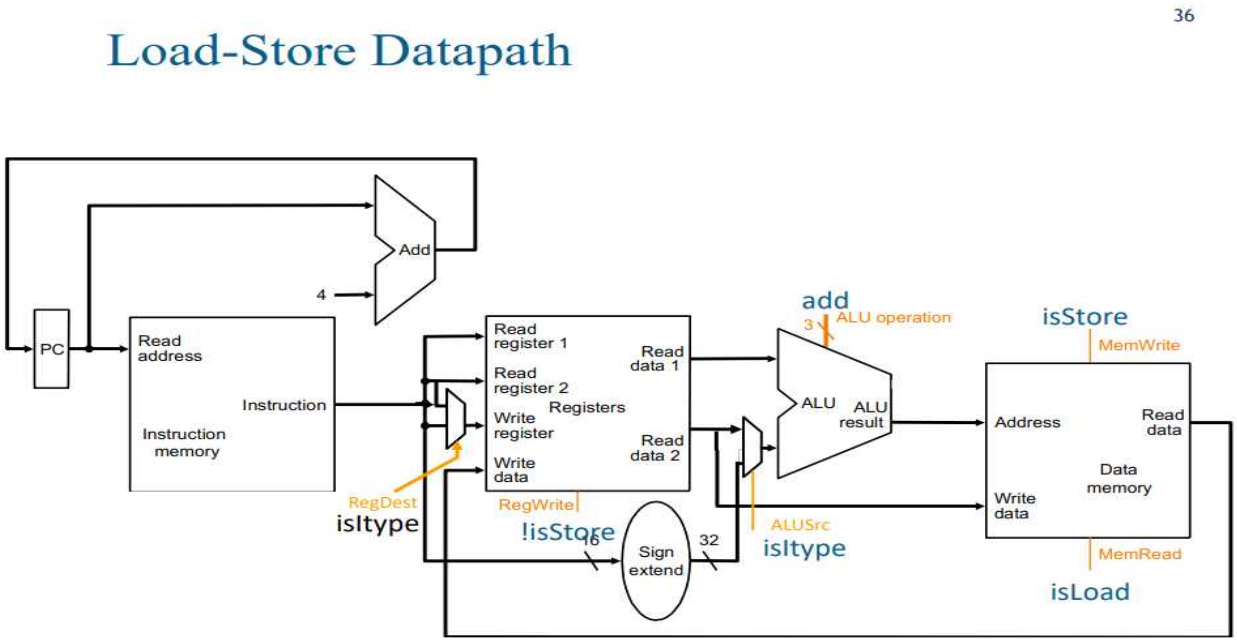


I-type 명령어는 R-type과는 다르게 immdiate 이라는 부분이 있다. I-type 명령어는 보통 연산 처리를 하면서 상숫값이 필요한 경우 사용한다. 그리고 I-type의 특별한 점은 보통 다른 단계에서 실행할 때 32bit 기준으로 값들을 처리해야 하는데 I-type immediate

숫자는 16bit임으로 연산 처리를 하려면 자릿수를 늘려서 계산을 해야 한다.
위에서 설명한 내용처럼 I-type 명령어에서는 immediate number가 16bit여서 다른 단계에서 사용을 위해서 sign extend를 통해서 다른 단계에서도 이용할 수 있도록 32bit로 만들어준다. 이때 비트 signed int에서 음수와 양수를 구분해서 extend를 하기 위해서 맨 앞자리 숫자를 확인하고 sign extend를 해준다.
그리고 이제 여기서부터 각각 단계별로 명령어마다 전해지는 값들이 다른데 이걸 정해주기 위해서 mux 가 필요하다. mux 은 각각의 control 신호들로 통제가 되며 0과 1 혹은 2 값으로 나뉜다.
여기서는 추가로 2개의 control signal이 추가되었다. 먼저 RegDest 는 레지스터에서 값을 저장하려는 레지스터를 지정해주는 신호이다. I-type 명령어에서는 R-type 과 다르게 rt 레지스터에 저장된다. 다음은 ALUSrc 신호이다. 이 신호는 I-type 명령어에서 사용되는 immediate number를 ALU에서 사용하기 위한 신호이다.

2.2.4 LW/SW

Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$

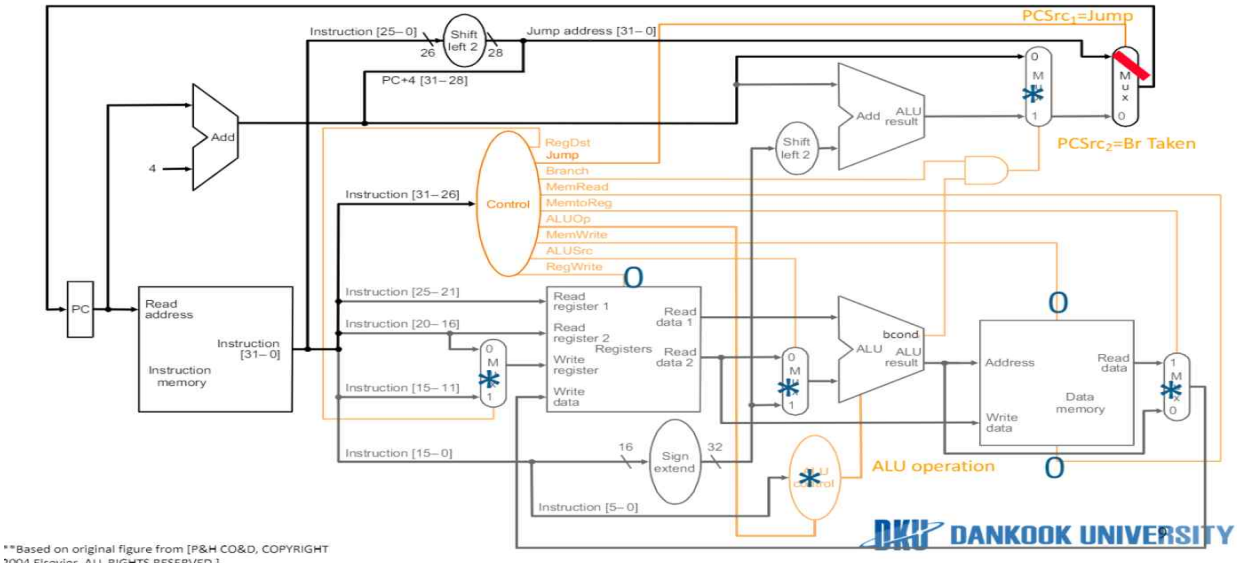


LW/SW에서는 둘 다 모두 I-type의 명령어지만 메모리를 이용한다는 특징이 있다. 그래서 LW/SW 명령어 data path에서는 data memory를 구현해야 하며 메모리에 데이터를 저장하는지 불러오는지를 정하는 신호들을 초기화 해주어야 한다. 그래서 signal 초기화를 해줄 때 메모리에 저장할지, 불러올지 구분해주는 MemRead, MemWrite 신호를 추가를 해야 한다.

2.2.5 J-type

Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}

Jump



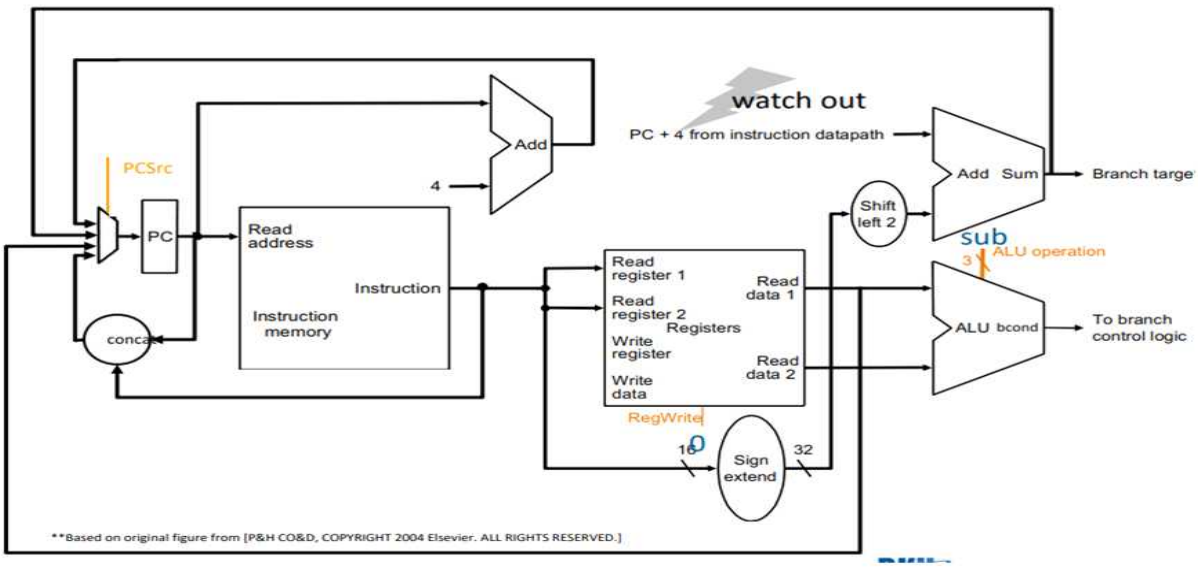
**Based on original figure from [P&H CO&D, COPYRIGHT

J-type 명령어에서는 opcode와 address로만 이루어져 있는 명령어이다. 여기서 중요한 점은 J-type 명령어에서도 주솟값이 26bit로 표현된다는 것이다. 그래서 32bit로 구동되는 구조에서는 반드시 이 주소를 32bit로 만들어서 사용해야 한다. J-type명령어 에서는 주솟값을 먼저 2만큼 bit 연산을 해준다. 왜냐하면, J-type 명령어 주소에서 PC값을 업데이트해주려면 4bit 단위로 바뀌는 PC값에 맞춰주어야 한다. 그래서 bit 연산을 해줘서 단위를 맞춰준다. 하지만 bit 연산을 하더라도 아직 주솟값은 28bit이다. 그래서 32bit로 만들어주기 위해서 현재 PC값의 최상위 4bit를 가지고 와서 주소에 더해준다. 이렇게 32bit jump 주소를 만들어주고 PCSrc1(J-type 명령어이면 1 아니면 0)signal 에 따라서 PC값을 업데이트해준다. 그리고 주솟값이 큰 값을 사용해야 한다면 레지스터게 주솟값을 저장해서 사용한다.

2.2.6 Branch

Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}

Conditional Branch Datapath (For You to Fix)



Branch 명령어들은 signextend 된 숫자에서 bit 연산을 해주어야 한다. 여기서 bit 연산을 해주는 이유는 PC값 때문이다. PC값은 4씩 더해지면서 마지막 2bit는 00으로 채워진다. 그래서 이 규칙을 맞춰주기 위해 bit를 2를 시프트 해주는 것이다. 이렇게 된다면 컴퓨터가 구동되면서 조금 더 많은 주소를 표현할 수 있는 장점이 있다. Branch 명령어에서는 그래서 왼쪽으로 2만큼 bit 연상된 숫자와 그리고 4가 더해진 PC값을 더해서 Branch targe 주소를 만든다. 그 이후에는 control signal 들에 따라서 PC값을 업데이트해준다.

2.2.5 control-signal

	When De-asserted	When asserted	Equation
RegDest	GPR write select according to rt, i.e., inst[20:16]	GPR write select according to rd, i.e., inst[15:11]	opcode==0
ALUSrc	2nd ALU input from 2nd GPR read port	2nd ALU input from sign-extended 16-bit immediate	(opcode!=0) && (opcode!=BEQ) && (opcode!=BNE)
MemtoReg	Steer ALU result to GPR write port	steer memory load to GPR wr. port	opcode==LW
RegWrite	GPR write disabled	GPR write enabled	(opcode!=SW) && (opcode!=Bxx) && (opcode!=J) && (opcode!=JR))

	When De-asserted	When asserted	Equation
MemRead	Memory read disabled	Memory read port return load value	opcode==LW
MemWrite	Memory write disabled	Memory write enabled	opcode==SW
$PCSrc_1$	According to $PCSrc_2$	next PC is based on 26-bit immediate jump target	(opcode==J) || (opcode==JAL)
$PCSrc_2$	next PC = PC + 4	next PC is based on 16-bit immediate branch target	(opcode==Bxx) && "bcond is satisfied"

각각의 control signal 들은 모든 명령어마다 초기화가 되어야 한다. 그래서 신호들을 초기화할 때 opcode나 funct 같은 명령어들을 구분 지어서 신호들을 통해서 신호들을 초기화해준다. Datapath를 통해서 명령어가 들어갈 때 명령어의 값들은 단계별로 전해지지만, 신호들을 통해서 값들이 저장될지 아니면 읽어올지를 결정한다.

3. Implementation

3.1 Instruction fetch (IF)

```
//fetch=->explain
memset(&instr, 0, sizeof(instr));
inst = memory[pc / 4];
```

명령어들이 저장되어있는 메모리에서 4byte 크기의 명령어를 읽고 변수에 저장한다. 그리고 메모리를 읽을 때 PC값에 4를 나누어 주는 이유는 PC는 구현상으로 4bit씩 더해진다. 그래서 4를 나눠서 PC값을 메모리의 인덱스 값으로 이용한다.

3.2 Instruction decode and register operand fetch (ID/RF)

```
//begin execution loop
while (1) {
    clock++;

    if (pc == -1)break;
    //in the loop..

    //fetch=->explain
    memset(&instr, 0, sizeof(instr));
    inst = memory[pc / 4];

    //decode
    instr.inst = inst;
    instr.opcode = (inst >> 26) & 0x3f;  // get some opcode bit from the instruction -> explain

    if (instr.opcode == 0x0) {// r type instruction
        instr.rs = (inst >> 21) & 0x1f;
        instr.rt = (inst >> 16) & 0x1f;
        instr.rd = (inst >> 11) & 0x1f;
        instr.shamt = (inst >> 6) & 0x1f;
        instr.funct = inst & 0x3f;
    }
    else {// not r type instruction
        if (instr.opcode == 0x2 || instr.opcode == 0x3) {//jump and jump and link( j type instruction)
            instr.jtarget = inst & 0x3fffffff;
        }
        else {//i type instrution
            instr.rs = (inst >> 21) & 0x1f;
            instr.rt = (inst >> 16) & 0x1f;
            instr.imm = inst & 0xffff;
        }
    }
```

명령어 메모리에 저장되어있는 명령어를 읽어오고 그 명령어를 decode 해준다. 이때 명령어의 종류와 명령어에서 사용되는 값들을 초기화해준다.

2.3 Execute/Evaluate memory address (EX/AG)

```
//execute
if (RegDst == 1) {// r-type
    reg(instr.rs, instr.rt, instr.rd);
}
else if (RegDst == 0) {//not r-type


    reg(instr.rs, instr.rt, instr.rt);
}
else if (RegDst == 2) {//initialize the R[31] for the register destination
    reg(instr.rs, instr.rt, 31);

}
```

명령어의 종류에 따라서 저장되는 레지스터가 다르다. 그래서 저장되는 위치를 control signal에 따라서 다르게 설정하고 reg 함수를 호출한다.

```
int ALU(int data1, int data2) {
    int result = 0;
    int temp = instr.imm;

    //opcode
    if (temp >> 15 == 0) {//imm sign extend
        instr.simm = 0xffffffff & instr.imm;
    }
    else {
        instr.simm = (0xffff << 16) | instr.imm;
    }

    if (zeroex) {//zeroextend of ori
        int zeroextend = instr.imm;
        instr.simm = 0xffffffff & instr.imm;



    }




    if (ALUSrc) {//not R type and not branch
      switch (ALUOp)
        {
        case 0x8://addi
            result = data1 + instr.simm;
            break;

        case 0x9://addiu
            result = data1 + instr.simm;
            break;

        case 0x2://jump here we just initlized the control signals
            break;

        case 0x3://jump and link
            break;
        case 0x2b://sw
            result = data1 + instr.simm;
            break;
        case 0x23:
            result = data1 + instr.simm;
```

```
else {//R type and branch
    switch (ALUOp)
    {
    case 0x4://beq
        if (data1 == data2) {
            becond = 1;
            PCSrc2 = becond & branch;

        }
        else {
            becond = 0;
            PCSrc2 = becond & branch;
        }
        break;
    case 0x5://bne
        if (data1 != data2) {
            becond = 1;
            PCSrc2 = becond & branch;

        }
        else {
            becond = 0;
            PCSrc2 = becond & branch;
        }
        break;
    case 0x0:       //r type instruction
        if (instr.funct == 0x20) {//add
            result = data1 + data2;
        }
        else if (instr.funct == 0x21) {//addu
            result = data1 + data2;
        }
        else if (instr.funct == 0x08) {//jump register
            result = 0;
        }
        else if (instr.funct == 0x2a) {//slt
            if (data1 < data2)result = 1;

        }
        else if (instr.funct == 0x22) {//sub
            result = data1 - data2;
```

이제 연산 처리 부분이다. 각 명령어의 종류 opcode에 따라서 먼저 다르게 초기화가 된 ALUSrc 신호를 통해서 구분을 지어준다. ALUSrc 값이 1이면 R-type 이거나 branch가 아닌 명령어들을 처리한다. 그리고 만약 opcode가 0x0 즉 R-type 명령어인 경우에는 funct을 통해서 명령어 연산을 구분 지어서 처리한다.

2.4 Memory operand fetch (MEM)

```
int mem(int address, int wirteData) {//memory
    if (memoryWrite == 1) {//sw
        memory[address / 4] = wirteData;
        return;
    }
    else if (memoryRead == 1) {//lw
        if (memtoReg) {
            return memory[address / 4];
        }
        else {
            return address;
        }
    }
    else if (memoryRead == 0 && memoryRead == 0) {
        return 0;
    }
    else if (j_and_l_toReg == 0) {
        return pc;
    }
}
```

명령어가 연상된 후에 메모리로 값이 넘어가면서 메모리에 저장될 것인지 아니면 메모리값을 읽을 것인지 아니면 메모리에서 아무 작업도 하지 않을 것인지를 control signal을 통해서 각각 다르게 처리해준다.

2.5 Store/writebackresult (WB)

```
int result_m = 0;
result_m = mem(result, data2);
if (memtoReg) {//return the value depends on memtoReg signal
    return result_m;
}
else {
    return result;
}
//writeback
```

```
    instr.j_r = data1;
    if (writeReg == 31) {
        Reg[writeReg] = pc;
    }

    if (RegWrite) {//register update
        Reg[writeReg] = ALU(data1, data2);
    }
    else {//memory update
        ALU(data1, data2);
    }
    return;
}
//execution
```

이제 모든 단계를 거친 후에 결괏값을 어디에 저장할지 정하고 그 상태들을 업데이트해준다. 각 단계에서 control signal 들로 저장되는 위치를 다르게 해준다.

2.6 추가 설명

-Byte order

```
while (1) {
    unsigned char in[4];
    int inst;
    int* ptr;


    for (int i = 0; i < 4; i++) {
        ret = fread(&in[i], 1, 1, fp);
        if (ret != 1) {
            fin = 1;
        }
    }

   // change the position of the instruction by 1byte-> explain
    unsigned tmp;
    tmp = in[1];
    in[1] = in[2];
    in[2] = tmp;
    tmp = in[3];
    in[3] = in[0];
    in[0] = tmp;
    ptr = &in[0];
    inst = *ptr;
    if (fin == 1) break;

    memory[index] = inst;
    index++;

}

fclose(fp);
```

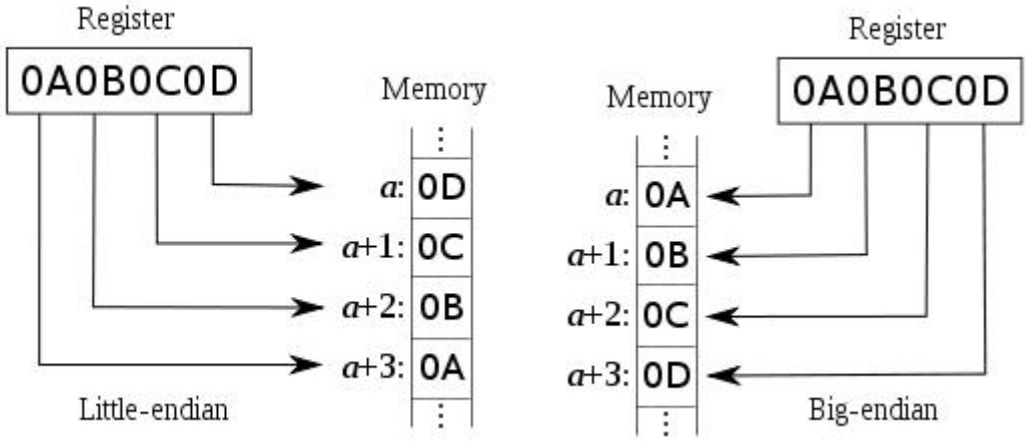
Big-Endian → 메모리 주소 증가

메모리 주소	....	0x100	0x101	0x102	0x103	...
변수 값		0x12	0x34	0x56	0x78	

Little-Endian → 메모리 주소 증가

메모리 주소	....	0x100	0x101	0x102	0x103	...
변수 값		0x78	0x56	0x34	0x12	

먼저 코드 bin 파일을 읽어와서 메모리에 저장하려면 byte order를 생각을 해주어야 한다. 그 이유는 먼저 MIPS에서 명령어를 처리할 때는 빅엔디안 구조를 사용하여 명령어들을 처리한다. 그래서 메모리에 명령어들을 저장할 때 먼저 리틀엔디안을 빅엔디안으로 바꾸는 작업이 필요하다. 2가지 방법이 있는데 첫 번째는 비트 연산을 통해서 8비트씩이 읽어온 숫자를 이어 나가는 것이다. 그리고 두 번째 방법은 32bit 배열에 저장하고 각각의 인덱스를 이용해서 자리로 바꿔준 다음 처리하는 방법이다. 저는 두 번째 방법을 사용하여 코드를 구현했습니다.




```
if (instr.opcode == 0x2 || instr.opcode == 0x3) {// shift 2bits and add the current upper pc 4bit to use jump address-
    int tempj = instr.jtarget << 2;
    instr.jtarget = (tempj + (pc & (0xf << 28)));

}
```

J-type 주소 주소를 위한 조건문이다. 위에서 설명한 내용처럼 26bit 주소를 32로 사용하기 위해서 먼저 2bit를 시프트 해주고 PC 앞자리 4개의 bit를 (0xf 값을 시프트 하여 and 연산) 더해주어서 32bit 주솟값을 만들었다.

```
if (temp >> 15 == 0) {//imm sign extend
    instr.simm = 0xffffffff & instr.imm;
}
else {
    instr.simm = (0xffff << 16) | instr.imm;
}

if (zeroex) {//zeroextend of ori
    int zeroextend = instr.imm;
    instr.simm = 0xffffffff & instr.imm;


}
```

R-type 이 아닌 명령어에서 16bit immediate 숫자를 사용하기 위해서 bit 시프트 연산을 통해서 32bit로 만들어준다.

```
int control(int opcode) {//function to initialize the control signal->explain
    RegDst = opcode == 0;//r type instrution
    if (opcode == 0x3) {// jump and link
        RegDst = 2;
    }
    ALUSrc = (opcode != 0) && (opcode != 0x4) && (opcode != 0x5);
    memtoReg = opcode == 0x23;
    RegWrite = (opcode != 0x2b) && (opcode != 0x4) && (opcode != 0x5) && (opcode != 0x2) && (opcode != 0x3);
    memoryRead = opcode == 0x23;//load word
    memoryWrite = opcode == 0x2b;//store word
    PCSrc1 = (opcode == 0x2) || (opcode == 0x3); //jump or not
    PCSrc2 = (opcode == 0x4) && (opcode == 0x5);//bcond statisfied
    branch = (opcode == 0x4) || (opcode == 0x5);//branch
    i_and_I = (opcode != 0x3);
    i_and_I_toReg = (opcode != 0x3);
    ir = (opcode == 0x0) && (instr.funct == 0x08);//ir mux signal
    ALUOp = instr.opcode;

    if (opcode == 0xd) {//or immediate
        zeroex = 1;
    }
}
```

이 코드는 명령어의 종류(opcode)에 따라서 control signal 들의 값들을 초기화해주는 부분이다. 이때 각각의 명령어마다 신호들이

모두 초기화되어야 하고 또 0과1 값으로 초기화가 되어야 하므로 조건연산을 해주었다.

4. Result

```
}
printf("************result**************₩n");
printf("R[2]:%d₩n", Reg[2]);//print out result (result will be store in the register number 2)
printf("J-type: %d R-type: %d I-type: %d₩n", j, r, i);
printf("MemooryAccess: %d₩n", memoryAccess);
printf("BranchCount: %d₩n", branchCount);
printf("*********************************");
//end of exeution
return 0;
```

4.1 simple

```
1 00000000: 27bd fff8 afbe 0004 03a0 f021 0000 0000
2 00000010: 03c0 e821 8fbe 0004 27bd 0008 03e0 0008
3 00000020: 0000 0000 0000 0000 0000 0000 0000 0000
```

```
instruction: 0x3a0f021
-------------------------------------------------
clock:4
pc[0xc]
instruction: 0x0
-------------------------------------------------
clock:5
pc[0x10]
instruction: 0x3c0e821
-------------------------------------------------
clock:6
pc[0x14]
instruction: 0x8fbe0004
-------------------------------------------------
clock:7
pc[0x18]
instruction: 0x27bd0008
-------------------------------------------------
clock:8
pc[0x1c]
instruction: 0x3e00008
************result**************
R[2]:0
J-type: 0 R-type: 4 I-type: 4
MemooryAccess: 2
BranchCount: 0
*********************************
```

4.2 simple2

```
1 00000000: 27bd ffe8 afbe 0014 03a0 f021 2402 0064
2 00000010: afc2 0008 8fc2 0008 03c0 e821 8fbe 0014
3 00000020: 27bd 0018 03e0 0008 0000 0000 0000 0000
~
~
```

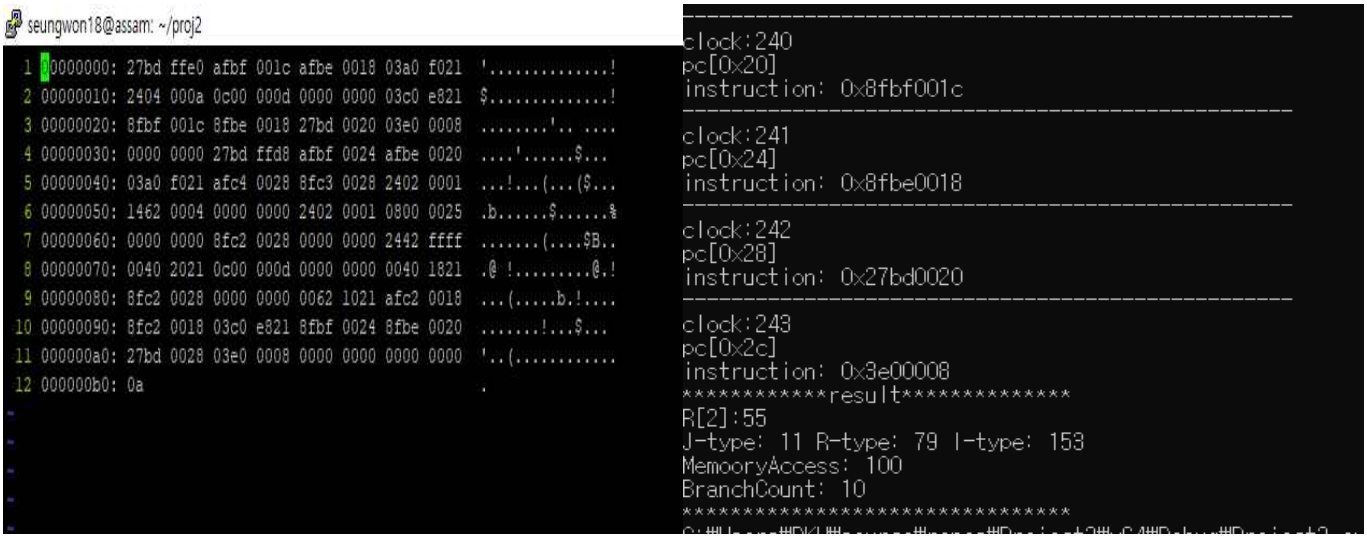
```
instruction: 0x3c0e821
------------------------------------------------
clock:8
pc[0x1c]
instruction: 0x8fbe0014
------------------------------------------------
clock:9
pc[0x20]
instruction: 0x27bd0018
------------------------------------------------
clock:10
pc[0x24]
instruction: 0x3e00008
**********result**********
R[2]:100
J-type: 0 R-type: 3 I-type: 7
MemooryAccess: 4
BranchCount: 0
```

4.3 simple3

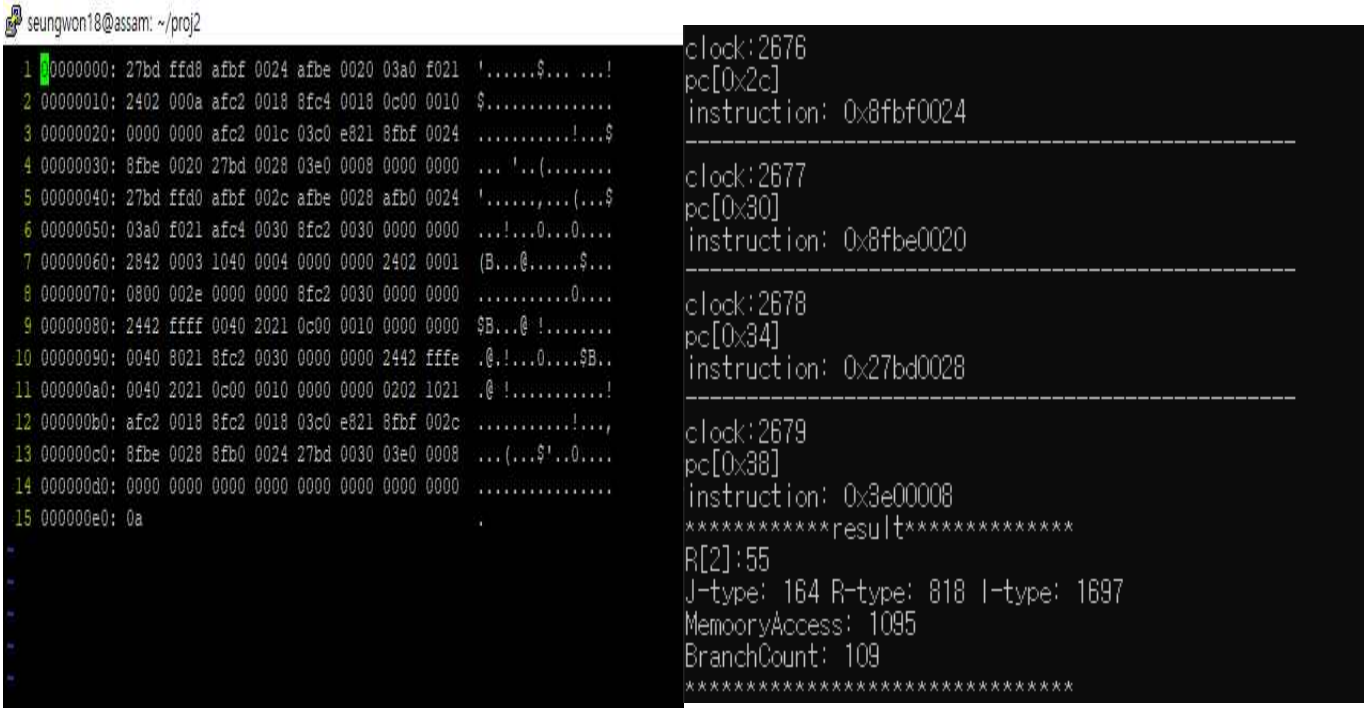
```
1 00000000: 27bd ffe8 afbe 0014 03a0 f021 afc0 0008
2 00000010: afc0 000c afc0 0008 0800 0011 0000 0000
3 00000020: 8fc3 000c 8fc2 0008 0000 0000 0062 1021
4 00000030: afc2 000c 8fc2 0008 0000 0000 2442 0001
5 00000040: afc2 0008 8fc2 0008 0000 0000 2842 0065
6 00000050: 1440 fff3 0000 0000 8fc2 000c 03c0 e821
7 00000060: 8fbe 0014 27bd 0018 03e0 0008 0000 0000
```

```
instruction: 0x3c0e821
------------------------------------------------
clock:1328
pc[0x60]
instruction: 0x8fbe0014
------------------------------------------------
clock:1329
pc[0x64]
instruction: 0x27bd0018
------------------------------------------------
clock:1330
pc[0x68]
instruction: 0x3e00008
**********result**********
R[2]:5050
J-type: 1 R-type: 409 I-type: 920
MemooryAccess: 613
BranchCount: 102
**********
```

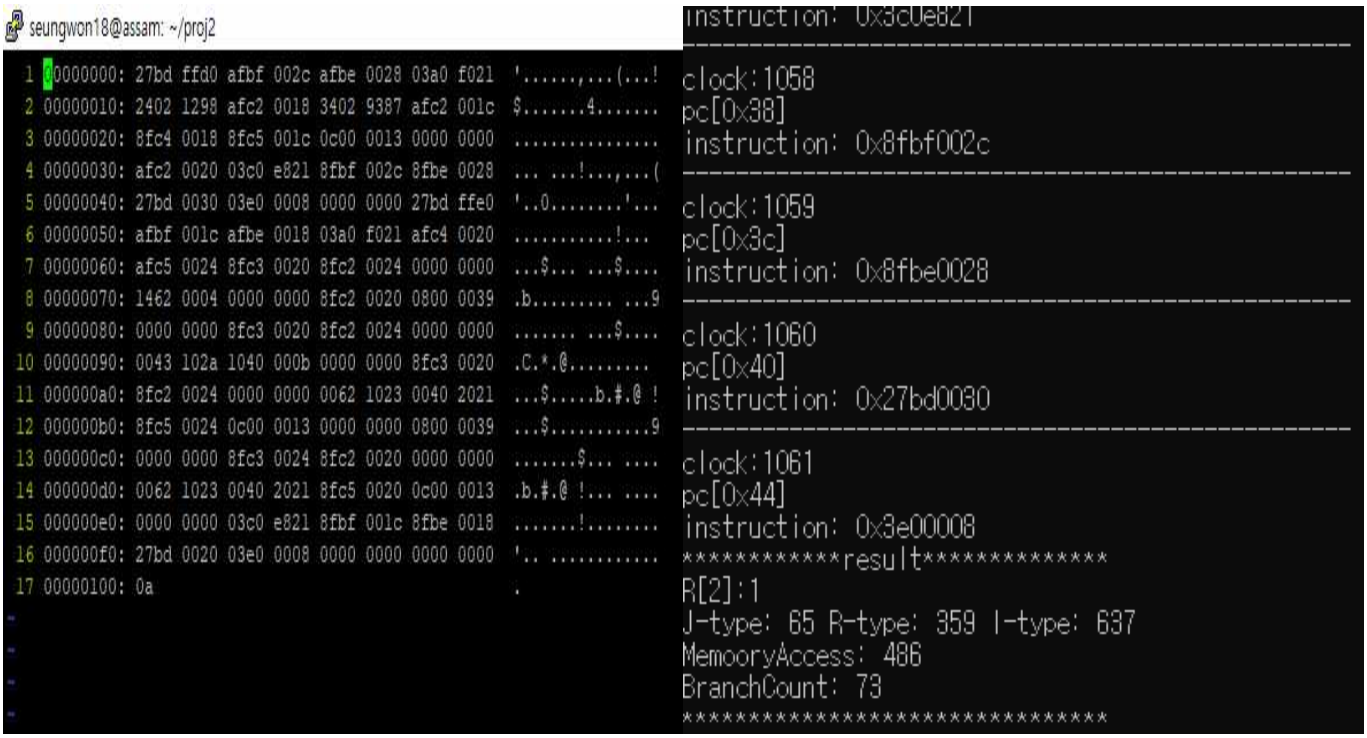
4.4 simple4


seungwon18@assam: ~/proj2
00000000: 27bd ffe0 afbf 001c afbe 0018 03a0 f021 '..............!
00000010: 2404 000a 0c00 000d 0000 0000 03c0 e821 $..............!
00000020: 8fbf 001c 8fbe 0018 27bd 0020 03e0 0008 ....'.. ....
00000030: 0000 0000 27bd ffd8 afbf 0024 afbe 0020 ....'......$...
00000040: 03a0 f021 afc4 0028 8fc3 0028 2402 0001 ...!...(...($...
00000050: 1462 0004 0000 0000 2402 0001 0800 0025 .b......$......%
00000060: 0000 0000 8fc2 0028 0000 0000 2442 ffff ....(....$B..
00000070: 0040 2021 0c00 000d 0000 0000 0040 1821 .@ !.........@.!
00000080: 8fc2 0028 0000 0000 0062 1021 afc2 0018 ...(.....b.!....
00000090: 8fc2 0018 03c0 e821 8fbf 0024 8fbe 0020 ....!...$...
000000a0: 27bd 0028 03e0 0008 0000 0000 0000 0000 '..(............
000000b0: 0a .
clock:240
pc[0x20]
instruction: 0x8fbf001c
clock:241
pc[0x24]
instruction: 0x8fbe0018
clock:242
pc[0x28]
instruction: 0x27bd0020
clock:243
pc[0x2c]
instruction: 0x3e00008
**********result**********
R[2]:55
J-type: 11 R-type: 79 I-type: 153
MemooryAccess: 100
BranchCount: 10

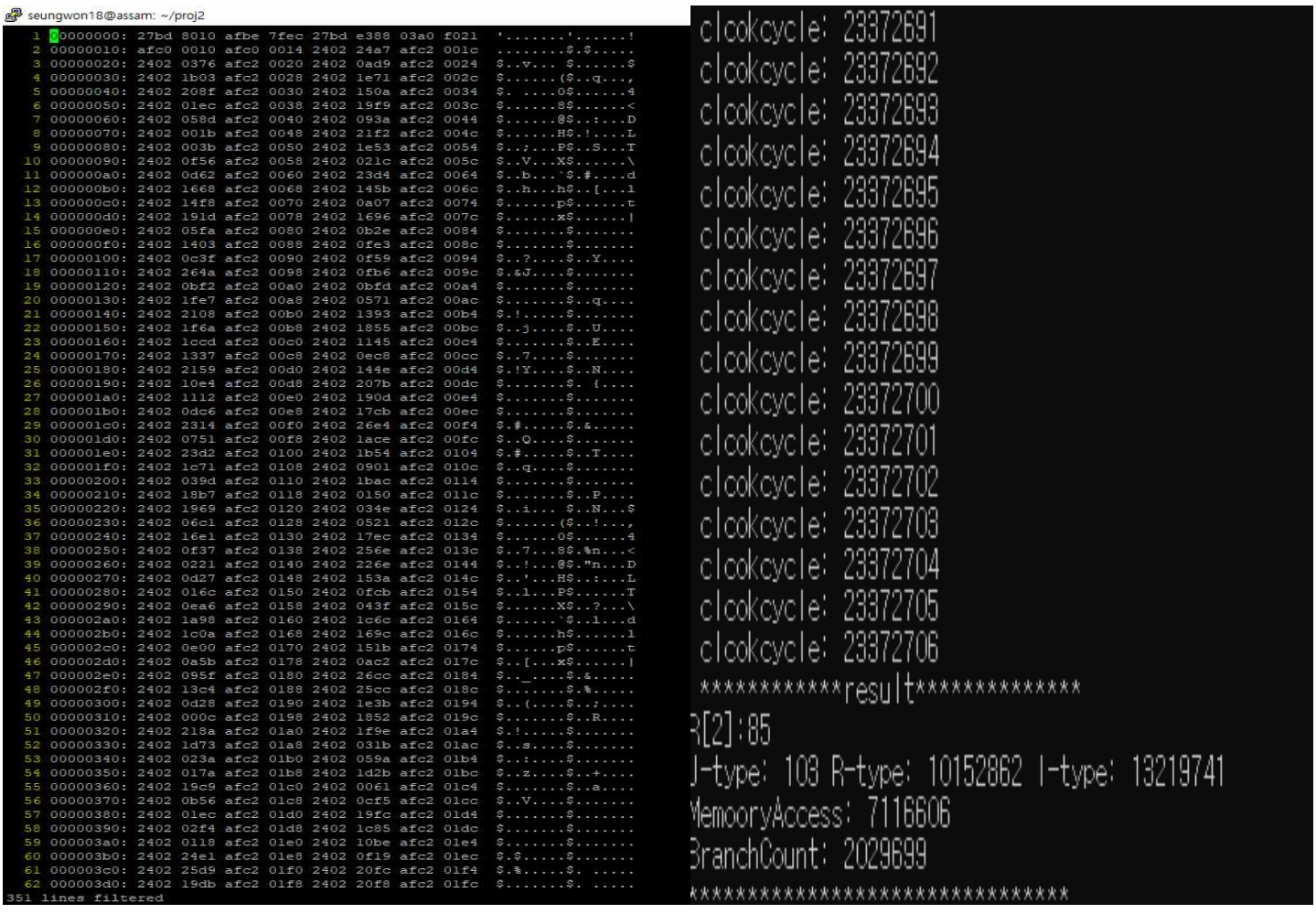
4.5 fib


seungwon18@assam: ~/proj2
00000000: 27bd ffd8 afbf 0024 afbe 0020 03a0 f021 '......$... ...!
00000010: 2402 000a afc2 0018 8fc4 0018 0c00 0010 $...............
00000020: 0000 0000 afc2 001c 03c0 e821 8fbf 0024 ....!...$
00000030: 8fbe 0020 27bd 0028 03e0 0008 0000 0000 ... '..(........
00000040: 27bd ffd0 afbf 002c afbe 0028 afb0 0024 '......,...(...$
00000050: 03a0 f021 afc4 0030 8fc2 0030 0000 0000 ...!...0...0....
00000060: 2842 0003 1040 0004 0000 0000 2402 0001 (B...@......$...
00000070: 0800 002e 0000 0000 8fc2 0030 0000 0000 ....0....
00000080: 2442 ffff 0040 2021 0c00 0010 0000 0000 $B...@ !........
00000090: 0040 8021 8fc2 0030 0000 0000 2442 fffe .@.!...0....$B..
000000a0: 0040 2021 0c00 0010 0000 0000 0202 1021 .@ !...........!
000000b0: afc2 0018 8fc2 0018 03c0 e821 8fbf 002c ....!...,
000000c0: 8fbe 0028 8fb0 0024 27bd 0030 03e0 0008 ...(...$'..0....
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000 ....
000000e0: 0a .
clock:2676
pc[0x2c]
instruction: 0x8fbf0024
clock:2677
pc[0x30]
instruction: 0x8fbe0020
clock:2678
pc[0x34]
instruction: 0x27bd0028
clock:2679
pc[0x38]
instruction: 0x3e00008
**********result**********
R[2]:55
J-type: 164 R-type: 818 I-type: 1697
MemooryAccess: 1095
BranchCount: 109

4.6 gcd


seungwon18@assam: ~/proj2
00000000: 27bd ffd0 afbf 002c afbe 0028 03a0 f021 '.....,...(...!
00000010: 2402 1298 afc2 0018 3402 9387 afc2 001c $.......4.....
00000020: 8fc4 0018 8fc5 001c 0c00 0013 0000 0000 ....
00000030: afc2 0020 03c0 e821 8fbf 002c 8fbe 0028 ... ...!...,...(
00000040: 27bd 0030 03e0 0008 0000 0000 27bd ffe0 '..0........'...
00000050: afbf 001c afbe 0018 03a0 f021 afc4 0020 ....!...
00000060: afc5 0024 8fc3 0020 8fc2 0024 0000 0000 ...$... ...$....
00000070: 1462 0004 0000 0000 8fc2 0020 0800 0039 .b......... ...9
00000080: 0000 0000 8fc3 0020 8fc2 0024 0000 0000 ....$....
00000090: 0043 102a 1040 000b 0000 0000 8fc3 0020 .C.*.@.........
000000a0: 8fc2 0024 0000 0000 0062 1023 0040 2021 ...$.....b.#.@ !
000000b0: 8fc5 0024 0c00 0013 0000 0000 0800 0039 ...$...........9
000000c0: 0000 0000 8fc3 0024 8fc2 0020 0000 0000 ....$... ....
000000d0: 0062 1023 0040 2021 8fc5 0020 0c00 0013 .b.#.@ !... ....
000000e0: 0000 0000 03c0 e821 8fbf 001c 8fbe 0018 ....!........
000000f0: 27bd 0020 03e0 0008 0000 0000 0000 0000 '.. ....
00000100: 0a .
instruction: 0x3c0e821
clock:1058
pc[0x38]
instruction: 0x8fbf002c
clock:1059
pc[0x3c]
instruction: 0x8fbe0028
clock:1060
pc[0x40]
instruction: 0x27bd0030
clock:1061
pc[0x44]
instruction: 0x3e00008
**********result**********
R[2]:1
J-type: 65 R-type: 359 I-type: 637
MemooryAccess: 486
BranchCount: 73

4.7 input4


seungwon18@assam: ~/proj2
00000000: 27bd 8010 afbe 7fec 27bd e388 03a0 f021 '.......'......!
00000010: afc0 0010 afc0 0014 2402 24a7 afc2 001c ....$.$.....
00000020: 2402 0376 afc2 0020 2402 0ad9 afc2 0024 $..v... $......$
00000030: 2402 1b03 afc2 0028 2402 1e71 afc2 002c $......(...q...,
00000040: 2402 208f afc2 0030 2402 150a afc2 0034 $. ....0$......4
00000050: 2402 01ec afc2 0038 2402 19f9 afc2 003c $......8$......<
00000060: 2402 058d afc2 0040 2402 093a afc2 0044 $......@$..:...D
00000070: 2402 001b afc2 0048 2402 21f2 afc2 004c $......H$.!....L
00000080: 2402 003b afc2 0050 2402 1e53 afc2 0054 $..;...P$..S...T
00000090: 2402 0f56 afc2 0058 2402 021c afc2 005c $..V...X$......\
000000a0: 2402 0d62 afc2 0060 2402 23d4 afc2 0064 $..b...`$.#....d
000000b0: 2402 1668 afc2 0068 2402 145b afc2 006c $..h...h$..[...l
000000c0: 2402 14f8 afc2 0070 2402 0a07 afc2 0074 $......p$......t
000000d0: 2402 191d afc2 0078 2402 1696 afc2 007c $......x$......|
000000e0: 2402 05fa afc2 0080 2402 0b2e afc2 0084 $.......$.......
000000f0: 2402 1403 afc2 0088 2402 0fe3 afc2 008c $.......$.......
00000100: 2402 0c3f afc2 0090 2402 0f59 afc2 0094 $..?....$..Y....
00000110: 2402 264a afc2 0098 2402 0fb6 afc2 009c $.&J....$.......
00000120: 2402 0bf2 afc2 00a0 2402 0bfd afc2 00a4 $.......$.......
00000130: 2402 1fe7 afc2 00a8 2402 0571 afc2 00ac $.......$..q....
00000140: 2402 2108 afc2 00b0 2402 1393 afc2 00b4 $.!.....$.......
00000150: 2402 1f6a afc2 00b8 2402 1855 afc2 00bc $..j....$..U....
00000160: 2402 1ccd afc2 00c0 2402 1145 afc2 00c4 $.......$..E....
00000170: 2402 1337 afc2 00c8 2402 0ec8 afc2 00cc $..7....$.......
00000180: 2402 2159 afc2 00d0 2402 144e afc2 00d4 $.!Y....$..N....
00000190: 2402 10e4 afc2 00d8 2402 207b afc2 00dc $.......$. {....
000001a0: 2402 1112 afc2 00e0 2402 190d afc2 00e4 $.......$.......
000001b0: 2402 0dc6 afc2 00e8 2402 17cb afc2 00ec $.......$.......
000001c0: 2402 2314 afc2 00f0 2402 26e4 afc2 00f4 $.#.....$.&.....
000001d0: 2402 0751 afc2 00f8 2402 1ace afc2 00fc $..Q....$.......
000001e0: 2402 23d2 afc2 0100 2402 1b54 afc2 0104 $.#.....$..T....
000001f0: 2402 1c71 afc2 0108 2402 0901 afc2 010c $..q....$.......
00000200: 2402 039d afc2 0110 2402 1bac afc2 0114 $.......$.......
00000210: 2402 18b7 afc2 0118 2402 0150 afc2 011c $.......$..P....
00000220: 2402 1969 afc2 0120 2402 034e afc2 0124 $..i... $..N...$
00000230: 2402 06c1 afc2 0128 2402 0521 afc2 012c $......($..!...,
00000240: 2402 16e1 afc2 0130 2402 17ec afc2 0134 $......0$......4
00000250: 2402 0f37 afc2 0138 2402 256e afc2 013c $..7...8$.%n...<
00000260: 2402 0221 afc2 0140 2402 226e afc2 0144 $..!...@$."n...D
00000270: 2402 0d27 afc2 0148 2402 153a afc2 014c $..'...H$..:....L
00000280: 2402 016c afc2 0150 2402 0fcb afc2 0154 $..l...P$.......T
00000290: 2402 0ea6 afc2 0158 2402 043f afc2 015c $......X$..?...\
000002a0: 2402 1a98 afc2 0160 2402 1c6c afc2 0164 $......`$..l...d
000002b0: 2402 1c0a afc2 0168 2402 169c afc2 016c $......h$.......l
000002c0: 2402 0e00 afc2 0170 2402 151b afc2 0174 $......p$.......t
000002d0: 2402 0a5b afc2 0178 2402 0ac2 afc2 017c $..[...x$.......|
000002e0: 2402 095f afc2 0180 2402 26cc afc2 0184 $..._....$.&.....
000002f0: 2402 13c4 afc2 0188 2402 25cc afc2 018c $.......$.%.....
00000300: 2402 0d28 afc2 0190 2402 1e3b afc2 0194 $..(....$..;....
00000310: 2402 000c afc2 0198 2402 1852 afc2 019c $.......$..R....
00000320: 2402 218a afc2 01a0 2402 1f9e afc2 01a4 $.!.....$.......
00000330: 2402 1d73 afc2 01a8 2402 031b afc2 01ac $..s....$.......
00000340: 2402 023a afc2 01b0 2402 059a afc2 01b4 $..:....$.......
00000350: 2402 017a afc2 01b8 2402 1d2b afc2 01bc $..z....$..+....
00000360: 2402 19c9 afc2 01c0 2402 0061 afc2 01c4 $.......$..a....
00000370: 2402 0b56 afc2 01c8 2402 0cf5 afc2 01cc $..V....$.......
00000380: 2402 01ec afc2 01d0 2402 19fc afc2 01d4 $.......$.......
00000390: 2402 02f4 afc2 01d8 2402 1c85 afc2 01dc $.......$.......
000003a0: 2402 0118 afc2 01e0 2402 10be afc2 01e4 $.......$.......
000003b0: 2402 24e1 afc2 01e8 2402 0f19 afc2 01ec $.$.....$.......
000003c0: 2402 25d9 afc2 01f0 2402 20fc afc2 01f4 $.%.....$. .....
000003d0: 2402 19db afc2 01f8 2402 20f8 afc2 01fc $.......$. .....
351 lines filtered
clockcycle: 23372691
clockcycle: 23372692
clockcycle: 23372693
clockcycle: 23372694
clockcycle: 23372695
clockcycle: 23372696
clockcycle: 23372697
clockcycle: 23372698
clockcycle: 23372699
clockcycle: 23372700
clockcycle: 23372701
clockcycle: 23372702
clockcycle: 23372703
clockcycle: 23372704
clockcycle: 23372705
clockcycle: 23372706
**********result**********
R[2]:85
J-type: 103 R-type: 10152862 I-type: 13219741
MemooryAccess: 7116606
BranchCount: 2029699

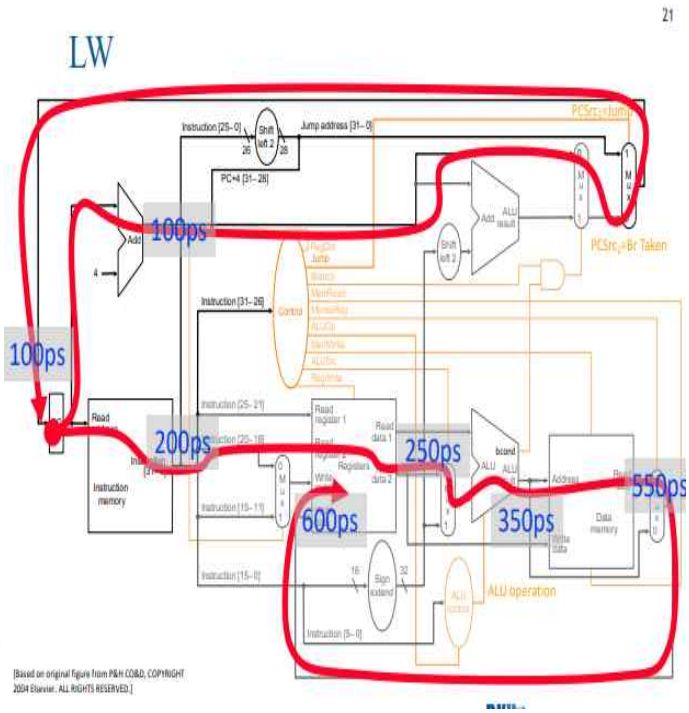
4. conclution

4.1 Single cycle 효율성

싱글 사이클에서는 모든 명령어가 1번의 사이클에서 6단계의 처리가 모두 완료된다. 그래서 CPI(Cycles per instruction)은 1로 고정이다. 그래서 명령어 중 가장 시간이 많이 들어가는 명령어를 기준으로 1번의 사이클 시간을 정하고 그 시간 안에 모든 명령어는 처리가 된다.

- Assume
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200



여기서 단계마다 걸리는 시간을 추정치로 정하고 표현한 표이다. 여기서 가장 시간이 오래 걸리는 명령어는 LW 명령어이다. 각 단계를 모두 거치는 명령어여서 시간이 가장 많이 걸린다. 그래서 single cycle에서는 LW 명령어를 기준으로 single cycle time을 지정해주고 그 시간 안에 모든 다른 명령어들은 처리할 수 있다.

Sigle cycle에 대한 효율성에 대하여 말해보겠습니다. 먼저 프로그램의 처리 시간(T)를 계산하려면 명령어의 수(I) 그리고 CPI(Clock cycle per instruction) 마지막으로 각각의 constant마다 처리 속도를 곱한 값이다. 이렇게 처리 속도를 구할 수 있는데 single cycle에서는 CPI가 1값으로 고정되어있다. 그래서 명령어가 모든 constant를 사용하지 않아도 CPI 값은 항상 1값으로 고정되어 한 cycle에서 한 개의 명령어만 처리한다. 그래서 single cycle에서는 짧은 명령어들을 처리하는데 많은 시간이 소유될 수 있으며 비효율적이라고 말할 수 있다. 추후 배울 이론인 pipline 구조와 multi cycle이론으로 이런 단점들을 보충하고 발전시켰다.

$$T = I_c \times CPI \times \tau$$

τ : constant cycle time 즉, $\frac{1}{clock\ frequency}$

4.2 느낀 점

먼저 코드를 작성하면서 기본 이론들을 정리하고 이해하고 코드를 짜기 시작했지만 큰 전체적인 틀 없이 코드를 짜기 시작해서 생각보다 많은 시간이 소모되었습니다. 그리고 코드를 작성하면서 bit 연산을 통한 주솟값을 구하는 방법에서 bit 시프팅에 대한 이해가 아직 부족했다고 느꼈습니다. 그래서 추가로 필요한 정보들을 인터넷을 찾아보며 다시 알아보았습니다.

구현상에서 어려웠던 부분은 data path를 구현하면서 PC값을 지정하는 부분이었습니다. 각 명령어마다 그리고 연산마다 각각 초기화되는 control signal 들이 모두 달랐으며 그 신호들에 따라 값을 처리하는데 생각보다 복잡했고 어렵게

느껴졌습니다. 각각 명령어들이 실행될 때마다 신호들을 초기화 해야 해서 생각보다 오류를 찾는 시간이 오래 걸렸습니다고 생각한다. 앞으로 이런 조금 큰 코드를 짤 때는 큰 틀과 그리고 이론적인 부분을 많이 숙지한 상태에서 코드를 짜야겠다고 생각했습니다.