컴퓨터구조실험 보고서

Project #2 - MIPS Multi Cycle CPU

과 목 컴퓨터구조실험

담당교수 이성원교수님

학 과 컴퓨터정보공학부

학 번 2021202058

이 름 송채영

제 출 일 2021. 05. 07

1. Introduction

이번 프로젝트는 MIPS Architecture 를 기반으로 하는 CPU 인 MIPS Multi Cycle CPU 를 설계하는 것이다. 데이터를 여러 사이클에 처리하며 각각의 사이클마다 다른 작업을 수행한다. 이전 프로젝트에서 진행한 single cycle 보다는 구현이 복잡하지만, clock frequency 를 높일 수 있다는 점에서 성능 향상을 이룰 수 있다는 장점이 있다. MIPS Multi Cycle CPU 는 Instruction Fetch(IF), Instruction Decode(ID), Execution(EX), Memory Access(MEM), Write Back(WB)의 단계를 거쳐 수행한다. 먼저 Instruction Fetch, IF 는 명령어를 메모리에서 가져온다. Instruction Decode, ID 는 명령어를 해독하고 레지스터에서 필요한 데이터를 가져온다. Execution, Ex 는 명령어를 실행한다. Memory Access, MEM 은 데이터를 메모리에서 읽거나 쓰기 위해 메모리에 액세스한다. Write back, WB 는 결과를 레지스터에 쓴다.

MIPS Multi Cycle CPU 는 Instruction Memory, Register File, ALU, Data Memory, Control Unit 으로 구성되어 있다. 먼저 Instruction Memory 는 명령어를 저장하는 메모리이고, Register File 은 명령어에서 사용되는 레지스터 값을 저장하는 메모리이다. 다음으로 ALU 는 산술 연산과 논리연산을 수행하고, Data Memory 는 데이터를 저장하는 메모리이다. 마지막으로 Control Unit 은 명령어 실행을 제어한다. 동작방식에 대해서도 설명해보면 다음과 같다. 먼저 명령어를 Instruction Memory 에서 가져와 Register File 에서 필요한 레지스터 값을 가져온다. 필요한 연산은 ALU 에서 수행하고 결과 값을 Register File 에 저장한다. 명령어에서 필요한 데이터를 Data Memory 에서 가져오고 필요한 데이터를 ALU 에서 처리한 후 결과 값을 Data Memory 에 저장한다. Control Unit 은 각 명령어를 실행할 때마다 다음 단계를 결정한다. 마지막으로 프로젝트에서 사용하는 MIPS Instruction 에 대해 설명해보겠다.

ADDU – Add Unsigned 로, 두개의 레지스터 값인 rs 와 rt 를 더하고 결과를 레지스터 rd 에 저장한다. 부호 없는 정수를 더하는데 사용되는 명령어이다.

OR – Bitwise Or 로, 두개의 레지스터 값 rs 와 rt 에 대해 비트 OR 연산을 수행하고 결과를 레지스터 rd 에 저장한다. 두 비트열의 비트 OR 연산을 수행하는데 사용된다.

ADDIU – Add Immediate Unsigned 로, 레지스터 rs 와 16 비트 상수값의 합을 레지스터 rt 에 저장한다. rs 와 상수 값을 더하는데 사용된다.

XORI – XOR Immediate 로, 레지스터 rs 와 16 비트 상수 값의 비트 XOR 결과를 레지스터 rt 에 저장한다. rs 와 상수 값을 XOR 연산하는데 사용된다.

SLL – Shift Left Logical 로, 레지스터 rt 의 비트열을 왼쪽으로 shift 하고 shift 된 결과를 레지스터 rd 에 저장한다. 레지스터의 비트열을 왼쪽으로 shift 하는데 사용된다.

SRAV – Shift Right Arithmetic Variable 로, 레지스터 rt 의 비트열을 레지스터 rs 에서 지정한 비트 수만큼 오른쪽으로 shift 하고 shift 된 결과를 레지스터 rd 에 저장한다. 레지스터의 비트열을 오른쪽으로 shift 하는데 사용된다.

SH – Store Halfword 로, 레지스터 rt 의 하위 16bit 를 메모리 주소에서 2byte 로 저장한다. 변수의 하위 16bit 값을 메모리에 저장하는데 사용된다.

LH – Load Halfword 로, 메모리의 주소에서 2byte 값을 읽고 그 값을 레지스터 rt 에 load 한다. 메모리에서 하위 16bit 값을 읽어 레지스터에 저장하는데 사용된다.

BLTZ – Branch on Less Than Zero 로, 레지스터 rs 의 값이 0 보다 작으면 label 로 분기한다. 조건 분기를 수행하는데 사용된다.

JAL – Jump and Link 로, PC 를 목표 주소로 변경하면서 다음 명령어의 주소를 레지스터 $31 에 저장한다. 함수 호출에서 많이 사용되며 함수 호출 이전의 주소를 저장하는데 사용된다.

2. 결과 화면

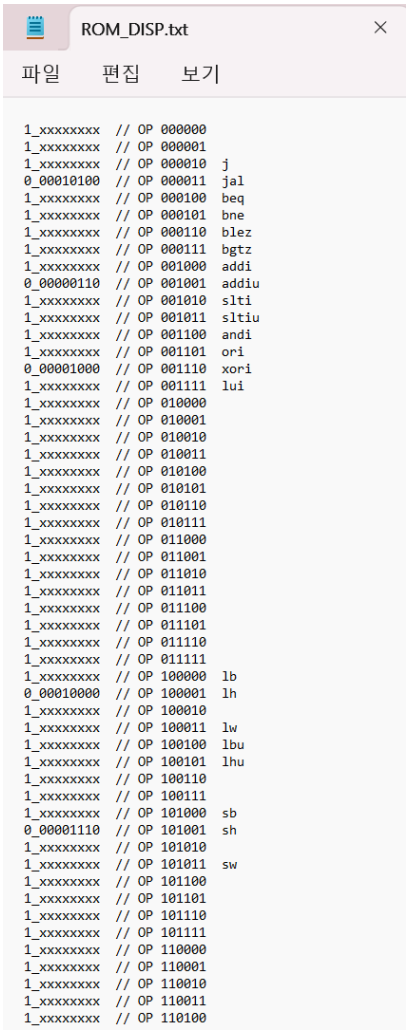
Micro-Instruction 의 Field 구분 및 Field

```
        FSM signal
     ALUctrl[1:0]=00
       ALUop[4:0]=04
     ALUsrcA[2:0]=011
     ALUsrcB[2:0]=001
      Branch[2:0]=000
  DataWidth[2:0]=000
         EXTmode=x
         IRwrite=1
            IorD=0
         MemRead=1
        MemWrite=0
   PCsource[1:0]=00
         PCwrite=1
  RegDatSel[2:0]=xxx
     RegDst[1:0]=xx
        RegWrite=0
    StateSel[1:0]=11
```

위의 사진을 바탕으로 설명해보면, 먼저 Field 는 ALU, Register, Memory, PC write control, sequencing 으로 나뉜다. 먼저 ALU field 는 ALU 연산을 제어하며 ALU 에서 수행할 수 있는 다양한 연산(덧셈, 뺄셈, AND, OR, XOR, Shift)를 지정한다. 이에 해당하는 signal 에는 ALUctrl[1:0], ALUop, ALUsrcA, ALUsrcB, EXTmode 가 있다. 다음으로 Register field 는 레지스터에 접근하기 위해 레지스터 번호를 지정한다. 이에 해당하는 signal 에는 RegDst, RegDatSel, RegWrite 가 있다. 다음으로 Memory field 는 메모리에서 데이터를 읽거나 쓸

때 사용하는 주소를 지정한다. 이에 해당하는 signal 에는 DatWidth, , IRwrite, IorD, MemRead, MemWrite 가 있다. PC write control 은 명령어를 수행한 후 PC 에 쓰이는 값을 결정한다. 이에 해당하는 signal 에는 Branch, PCsorce, PCwrite 가 있다. 마지막으로 Sequencing field 는 다음에 실행할 명령어의 주소를 결정하는데 사용하며 이에 해당하는 signal 에는 StateSel 이 있다.

ROM_DISP

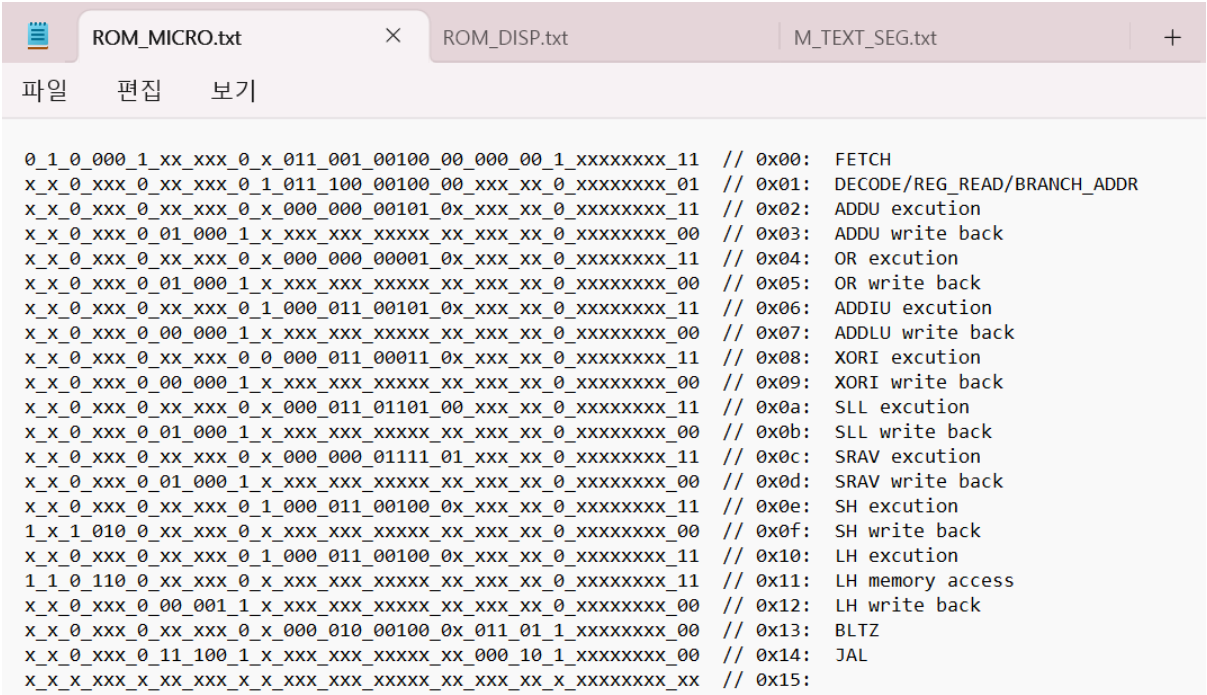


ROM_DSIP.txt 파일이다. ROM_MICRO.txt 파일에서 설정한 Signal 의 Address 를 설정한다. 구현한 10 개의 Instruction 을 표로 나타내면 다음과 같다. 이때 사용할 명령어는 1 에서 0 으로 바꿔주었다.

Instruction	Address
ADDU	0_00000010
OR	0_00000100

ADDIU	0_00000110
XORI	0_00001000
SLL	0_00001010
SRAV	0_00001000
SH	0_00001110
LH	0_00010000
BLTZ	0_00010011
JAL	0_00010100

ROM_MICRO



다음으로 ROM_MICRO.txt 파일이다. 해당 파일에서는 Signal 을 설정한다. 구현한 10 개의 Instruction 과 FETCH, DECODE/REG_READ/BRANCH_ADDR 을 표로 나타내면 다음과 같다.

FETCH

FETCH	Signal

IorD	0
MemRead	1
MemWrite	0
DataWidth	000
IRwrite	1
RegDst	Xx
RegDatSel	Xxx
RegWrite	0
EXTmode	X
ALUsrcA	011
ALUsrcB	001
ALUop	00100
ALUctr[1:0]	00
Branch	000
PCsrc	00
PCwrite	1
8 bit	xxxxxxxx
StateSel	11

Decode/REG_READ/Branch_ADDR

Decode/REG_READ/Branch_ADDR	Signal
IorD	X
MemRead	X
MemWrite	0
DataWidth	Xxx

IRwrite	0
RegDst	Xx
RegDatSel	Xxx
RegWrite	0
EXTmode	X
ALUsrcA	011
ALUsrcB	100
ALUop	00100
ALUctr[1:0]	00
Branch	Xxx
PCsrc	Xx
PCwrite	0
8 bit	xxxxxxxx
StateSel	01

Instruction execution

Instruction /Signal	ADDU	OR	ADDIU	XORI	SLL	SRAV	SH	LH
IorD	X	X	X	X	X	X	X	X
MemRead	X	X	X	X	X	X	X	X
MemWrite	0	0	0	0	0	0	0	0
DataWidth	Xxx	Xxx	Xxx	Xxx	Xxx	Xxx	Xxx	Xxx
IRwrite	0	0	0	0	0	0	0	0
RegDst	xx	Xx	Xx	Xx	Xx	Xx	Xx	Xx
RegDatSel	xxx	Xxx	Xxx	Xxx	xxx	Xxx	Xxx	Xxx

RegWrite	0	0	0	0	0	0	0	0
EXTmode	X	X	1	0	X	X	1	1
ALUsrcA	000	000	000	000	000	000	000	000
ALUsrcB	000	000	011	011	011	000	011	011
ALUop	00101	00001	00101	00011	01101	01111	00100	00100
ALUctr[1:0]	0x	0x	0x	0x	00	01	0x	0x
Branch	Xxx	Xxx	Xxx	Xxx	Xxx	Xxx	Xxx	Xxx
PCsrc	Xx	Xx	Xx	Xx	Xx	Xx	Xx	Xx
PCwrite	0	0	0	0	0	0	0	0
8 bit	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
StateSel	11	11	11	11	11	11	11	11

Instruction Write back

Instruction /Signal	ADDU	OR	ADDIU	XORI	SLL	SRAV	SH	LH
IorD	X	X	X	X	X	X	1	X
MemRead	X	X	X	X	X	X	X	X
MemWrite	0	0	0	0	0	0	1	0
DataWidth	Xxx	Xxx	Xxx	Xxx	Xxx	Xxx	010	Xxx
IRwrite	0	0	0	0	0	0	0	0
RegDst	01	01	00	00	01	01	Xx	00
RegDatSel	000	000	000	000	000	000	Xxx	001
RegWrite	1	1	1	1	2	1	0	1
EXTmode	X	X	X	X	X	X	X	X
ALUsrcA	Xxx	Xxx	Xxx	Xxx	Xxx	Xxx	Xxx	Xxx

ALUsrcB	Xxx	Xxx	Xxx	Xxx	Xxx	Xxx	Xxx	Xxx
ALUop	Xxxxx	Xxxxx	Xxxxx	Xxxxx	Xxxxx	Xxxxx	Xxxxx	Xxxxx
ALUctr[1:0]	Xx	Xx	xx	xx	Xx	Xx	xx	Xx
Branch	Xxx	Xxx	Xxx	Xxx	Xxx	Xxx	Xxx	Xxx
PCsrc	Xx	Xx	Xx	Xx	Xx	Xx	Xx	Xx
PCwrite	0	0	0	0	0	0	0	0
8 bit	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
StateSel	00	00	00	00	00	00	00	00

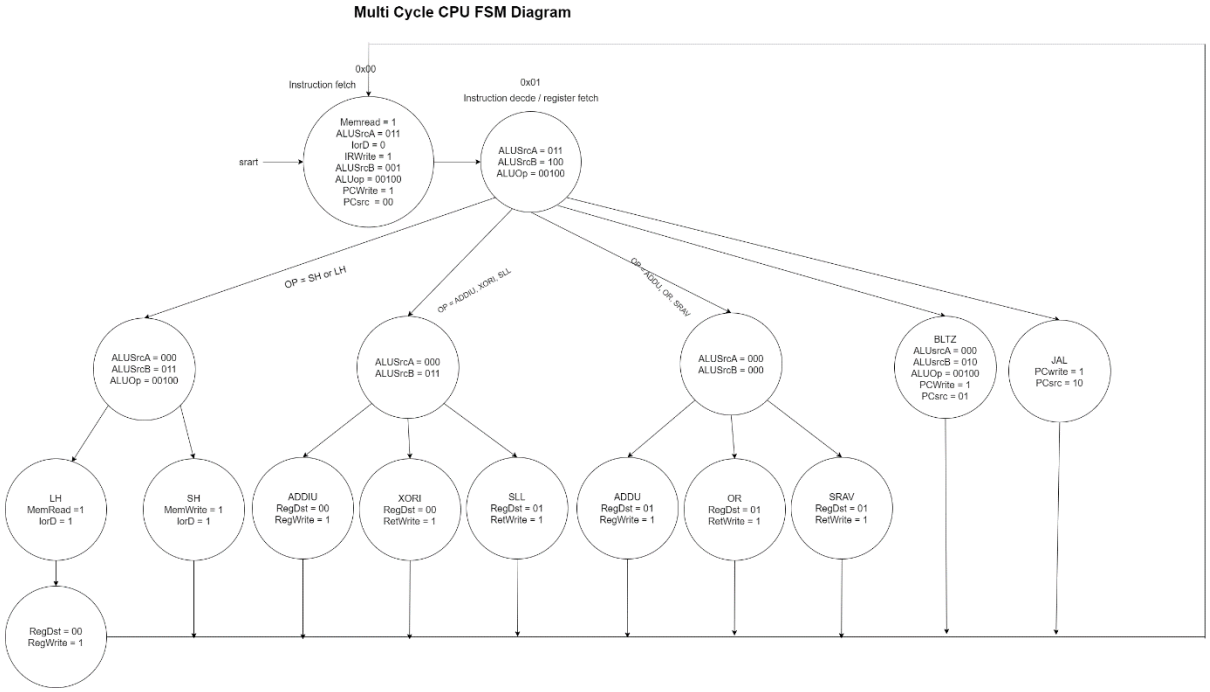
LH 의 경우 memory access 의 부분이 추가되었다. LH 명령어는 MDR 에 넣었다가 register 로 접근한다. 따라서 메모리 접근 시간이 길기 때문에, Excution 후 Memory access 가 필요하다.

BLTZ & JAL

Instruction /Signal	BLTZ	JAL
IorD	X	X
MemRead	X	X
MemWrite	0	0
DataWidth	Xxx	Xxx
IRwrite	0	0
RegDst	xx	11
RegDatSel	xxx	100
RegWrite	0	1
EXTmode	X	X
ALUsrcA	000	Xxx
ALUsrcB	010	Xxx

ALUop	00100	Xxxxx
ALUctr[1:0]	0x	Xx
Branch	011	000
PCsrc	01	10
PCwrite	1	1
8 bit	xxxxxxxx	xxxxxxxx
StateSel	00	00

다음으로 구현한 Multi Cycle CPU 의 FSM 에 대해 설명하겠다.



ROM_MICRO.txt 파일에 정의된 signal 과 프로젝트 제안서에 명시된 signal 을 바탕으로 state 를 정의하였다. ROM_MICRO.txt 파일에 명시된 것처럼 0x00 에 instruction fetch 와 0x01 에 decode/ register fetch 가 있으며 나머지 명령어들은 ALUsrcA 와 ALUsrcB 가 같은 걸 기준으로 나누어 설계하였다. 각 명령어들이 단계에 맞게 수행된 후 수행이 끝나면 다시 instruction fetch, 즉 0x00 으로 돌아가 다음 명령어를 수행한다.

다음으로 각 명령어 별 기능과 동작에 대해 설명하겠다.

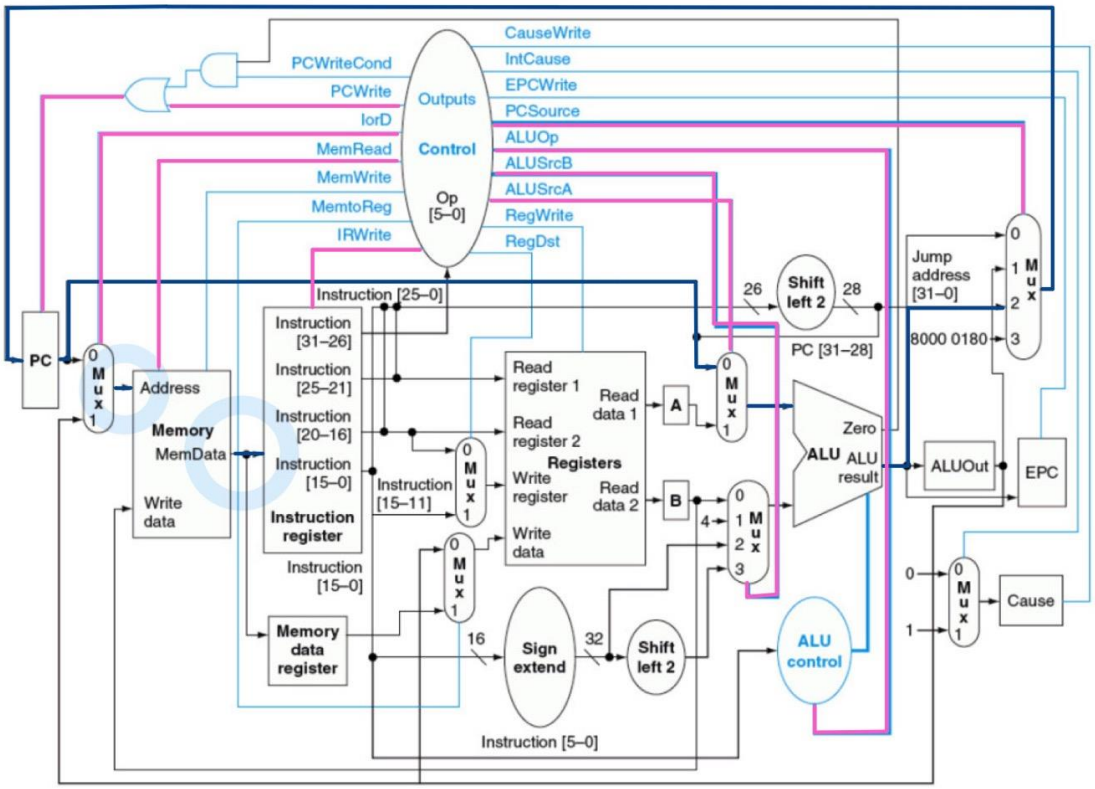


Figure 1 - The multi-cycle datapath together with the necessary control lines

먼저 instruction fetch 부분이다. 파란색 선 중 동그라미 친 부분은 instruction fetch 가 일어나는 부분으로, pc 의 값에 해당하는 memory 의 주소에 값을 IR 에 쓴다. 파란색 선은 pc+4 를 의미하며, instruction fetch 에 해당하는 signal 은 핑크색으로 표시하였다.

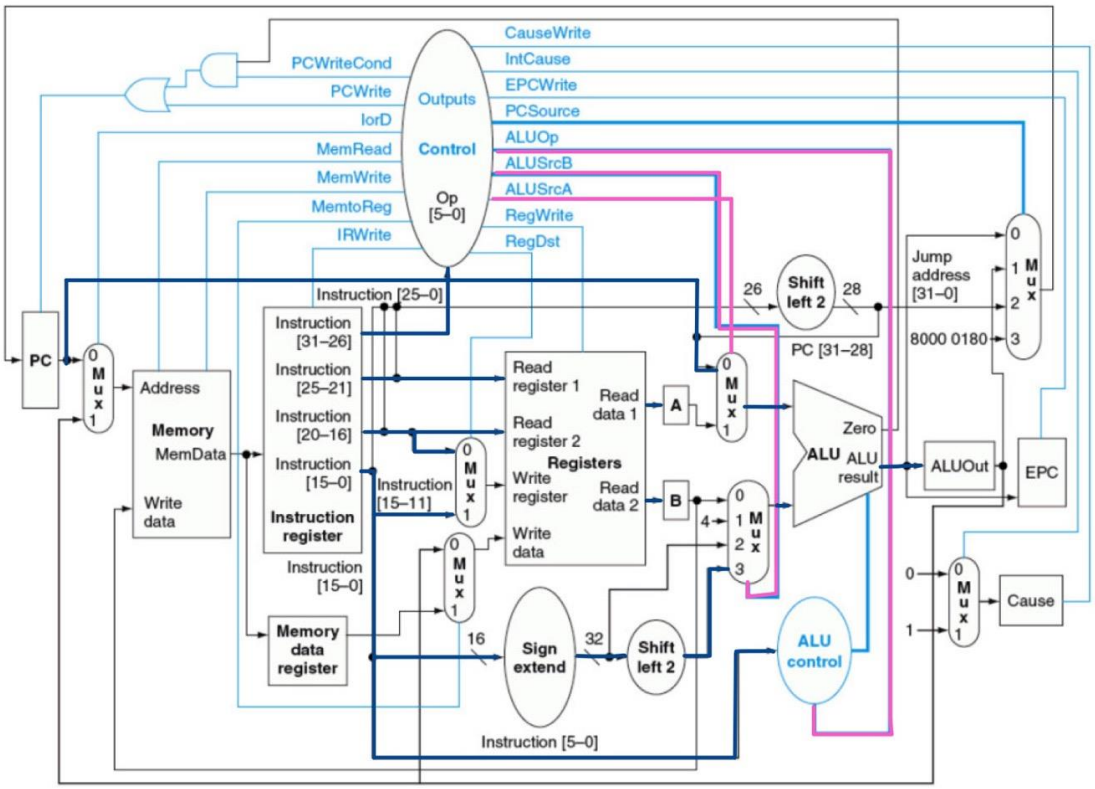


Figure 1 - The multi-cycle datapath together with the necessary control lines

다음으로 decode/register fetch/branch address 부분이다. 파란색 선은 register fetch 와 branch address 가 일어나는 부분을 나타내며, 분홍색 선은 decode/register fetch/branch address 에 해당하는 signal 을 표시하였다. register fetch 부분은 register 에 저장된 A 와 B 에 값을 write 한다. 또한 branch address 부분은 branch 할 주소를 만들어 ALUOUT 에 넣어주었다.

10 개의 명령어를 바탕으로 설명해보겠다. Instructio Fetch 와 decode/register fetch/branch address 는 모든 명령어의 공통적인 부분이고, 위에서 설명했으므로 생략하여 그림을 그려 표현하였다.

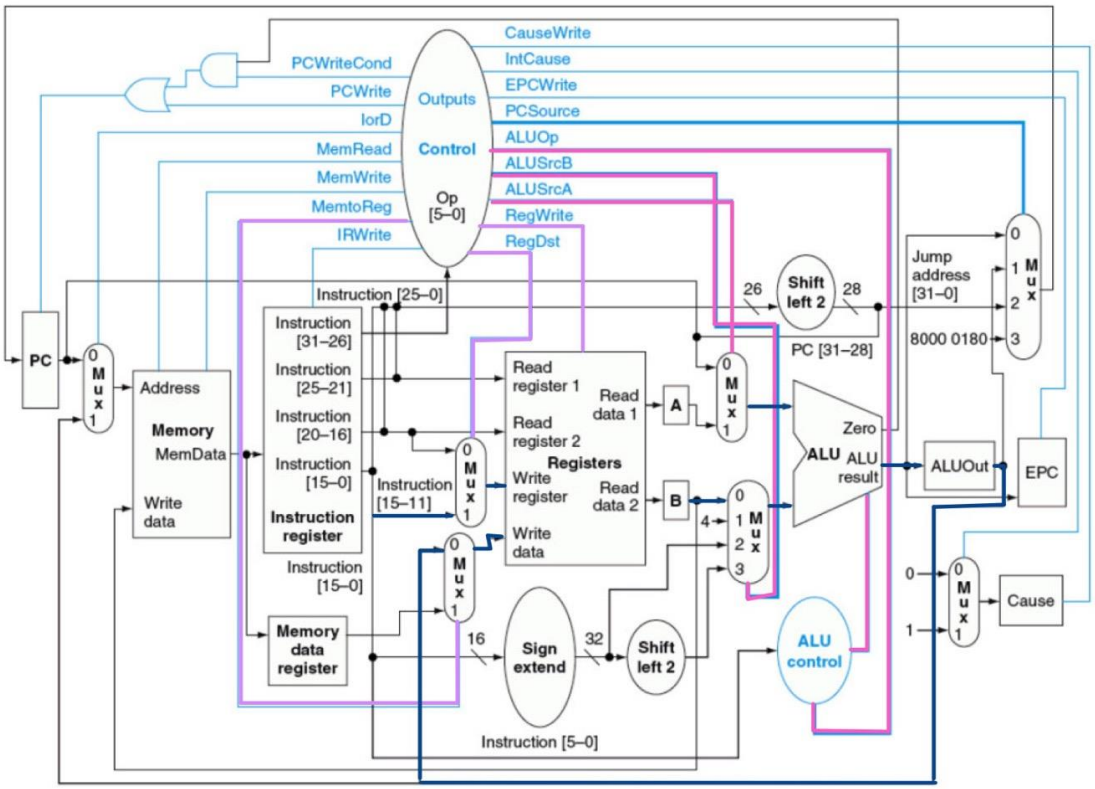


Figure 1 - The multi-cycle datapath together with the necessary control lines

ADDU 명령어로, Rtype 의 명령어이고, 부호 없는 두 개의 레지스터 값을 더하는 연산을 수행한다. addu $rd, $rs, $rt 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rd 는 결과를 저장할 대상, rt 와 rs 는 더할 값이 들어 있는 레지스터이다. 핑크색 선은 ADDU 의 execution signal 을 의미하며 보라색 선은 ADDU 의 write back signal 을 의미한다. 파란색 선은 흐름을 나타낸다.

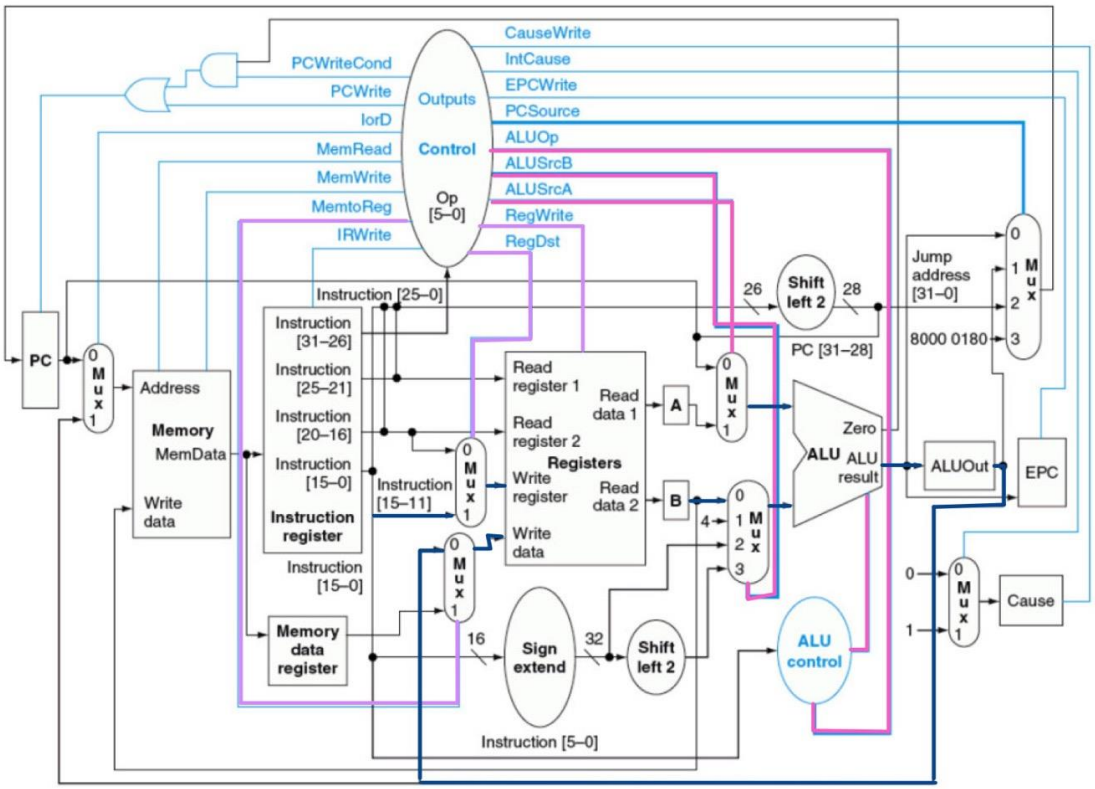


Figure 1 - The multi-cycle datapath together with the necessary control lines

OR 명령어로, Rtype 의 명령어이고 두 개의 레지스터 값을 bit 별로 or 연산을 수행한다. or $rd, $rs, $rt 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rd 는 결과를 저장할 대상, rt 와 rs 는 더할 값이 들어 있는 레지스터이다. 핑크색 선은 OR 의 execution signal 을 의미하며 보라색 선은 OR 의 write back signal 을 의미한다. 파란색 선은 흐름을 나타낸다.

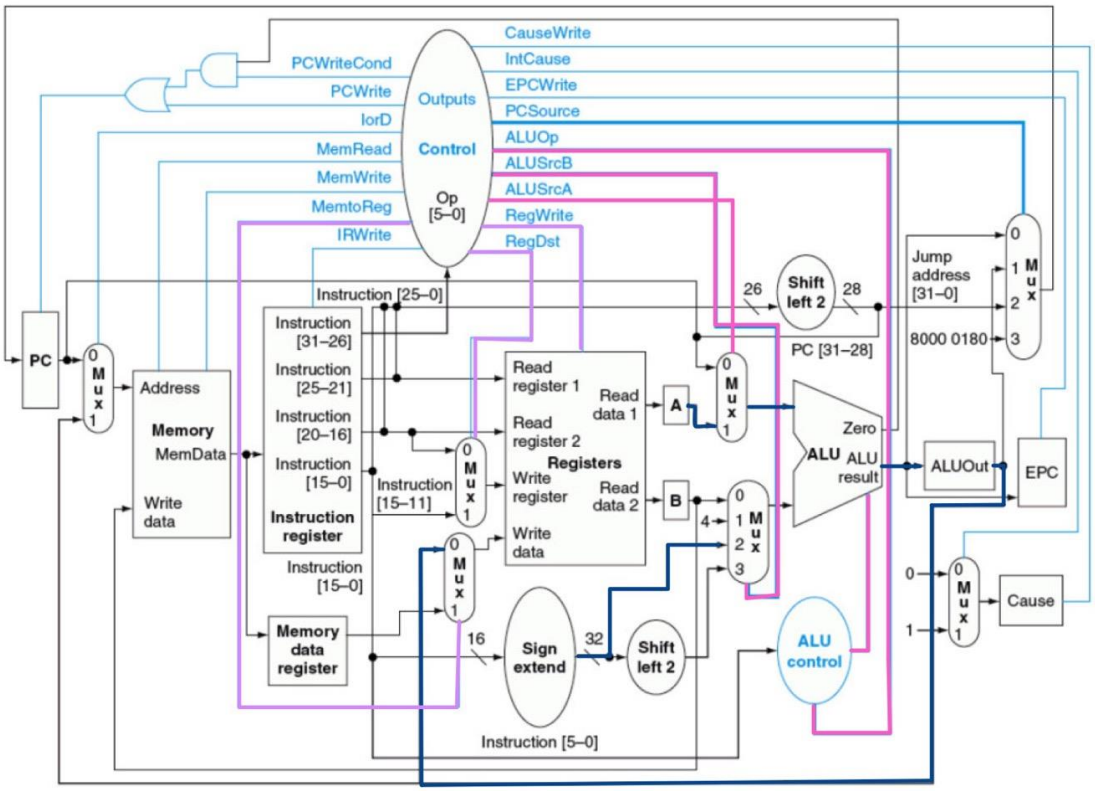


Figure 1 - The multi-cycle datapath together with the necessary control lines

ADDIU 명령어로, 레지스터와 16bit 의 상수 값을 더한다. addiu $rt, $rs, imm 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rt 는 결과를 저장할 대상 레지스터, rs 는 더해질 값이 들어 있는 레지스터이며 immediate value 를 사용하여 연산한다. 핑크색 선은 ADDIU 의 execution signal 을 의미하며 보라색 선은 ADDIU 의 write back signal 을 의미한다. 파란색 선은 흐름을 나타낸다.

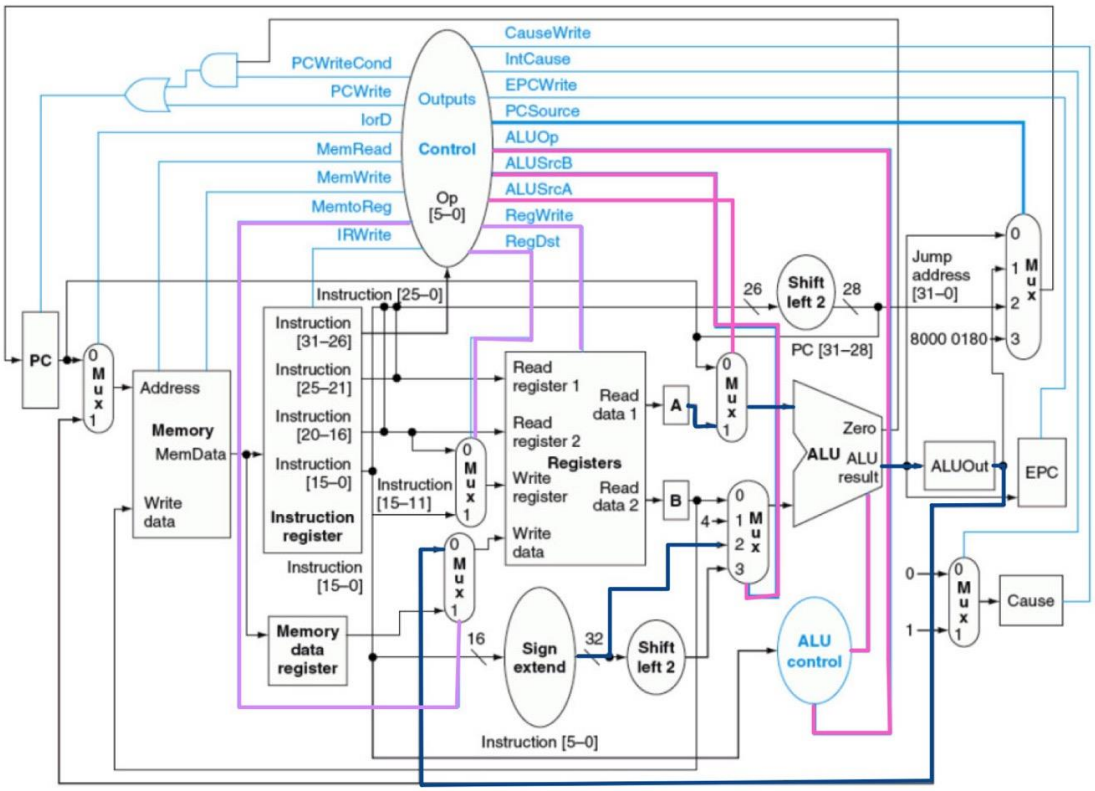


Figure 1 - The multi-cycle datapath together with the necessary control lines

XORI 명령어로, 레지스터와 16bit 의 상수 값을 xor 한다. xor $rt, $rs, imm 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rt 는 결과를 저장할 대상, rs 는 더해질 값이 들어 있는 레지스터이며 immediate value 를 사용하여 연산한다. 핑크색 선은 XORI 의 execution signal 을 의미하며 보라색 선은 XORI 의 write back signal 을 의미한다. 파란색 선은 흐름을 나타낸다.

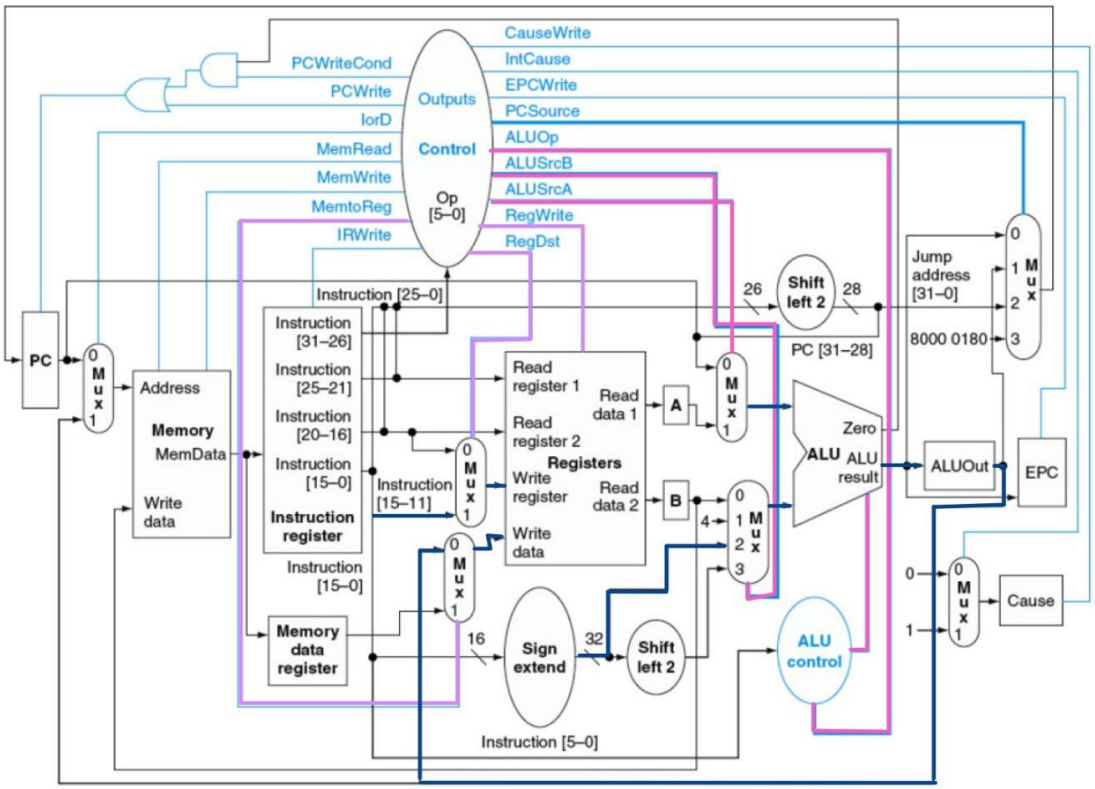


Figure 1 - The multi-cycle datapath together with the necessary control lines

SLL 명령어로, 레지스터 값의 비트를 왼쪽으로 amount 만큼 shift 한다. sll $rd, $rt, sa 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rd 는 결과를 저장할 대상, rt 는 시프트 할 값이 들어 있는 레지스터이며 sa 는 shift amount 를 의미한다. sa 는 5bit 이기 때문에 sign-extend 를 거쳐 32bit 로 확장해준다. 핑크색 선은 SLL 의 execution signal 을 의미하며 보라색 선은 SLL 의 write back signal 을 의미한다. 파란색 선은 흐름을 나타낸다.

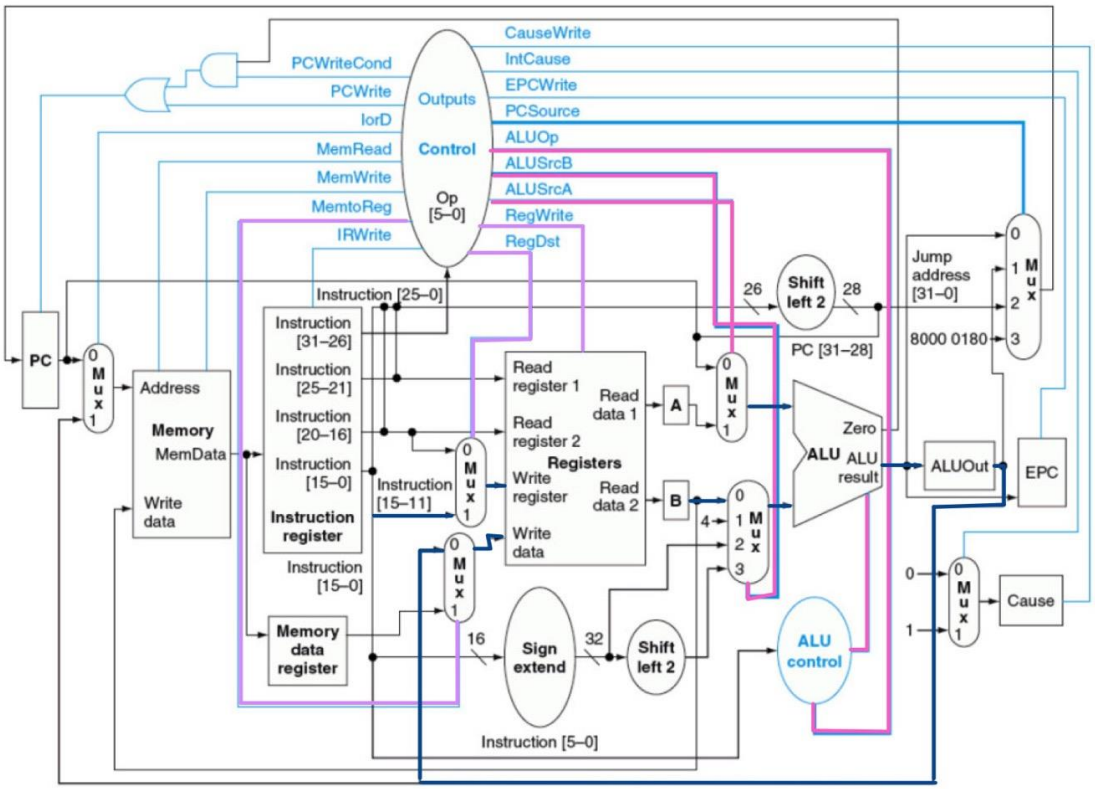


Figure 1 - The multi-cycle datapath together with the necessary control lines

SRAV 명령어로, 레지스터 값의 비트를 오른쪽으로 레지스터 rs 만큼 shift 한다. srav $rd, $rt, $rs 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rd 는 결과를 저장할 대상, rt 는 더해질 값이 들어 있는 레지스터이며 rs 은 shift 할 값을 말한다. 핑크색 선은 SRAV 의 execution signal 을 의미하며 보라색 선은 SRAV 의 write back signal 을 의미한다. 파란색 선은 흐름을 나타낸다.

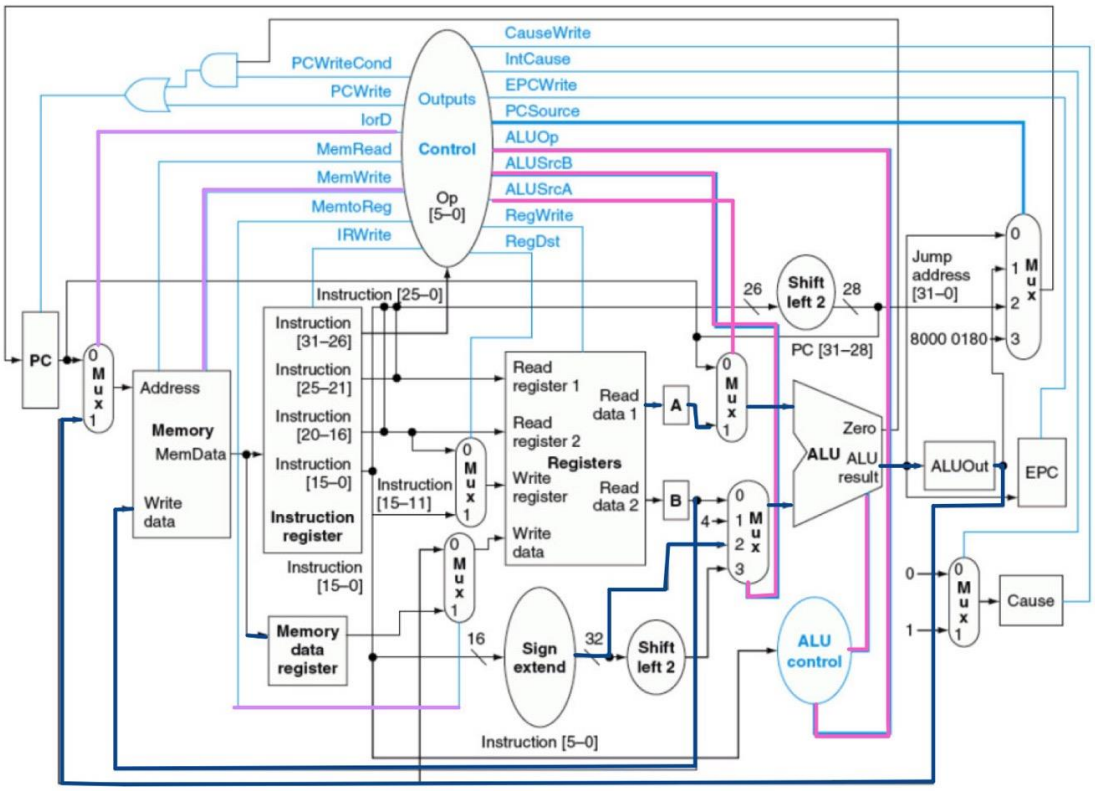


Figure 1 - The multi-cycle datapath together with the necessary control lines

SH 명령어로, 하위 16bit 를 메모리에 저장한다. sh $rt, imm 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rt 는 결과를 저장할 대상, imm 은 immediate value 를 말한다. immediate value 를 sign-extend 하여 32bit 로 만들어야 하기 때문에 확장해준다. 핑크색 선은 SH 의 execution signal 을 의미하며 보라색 선은 SH 의 write back signal 을 의미한다. 파란색 선은 흐름을 나타낸다.

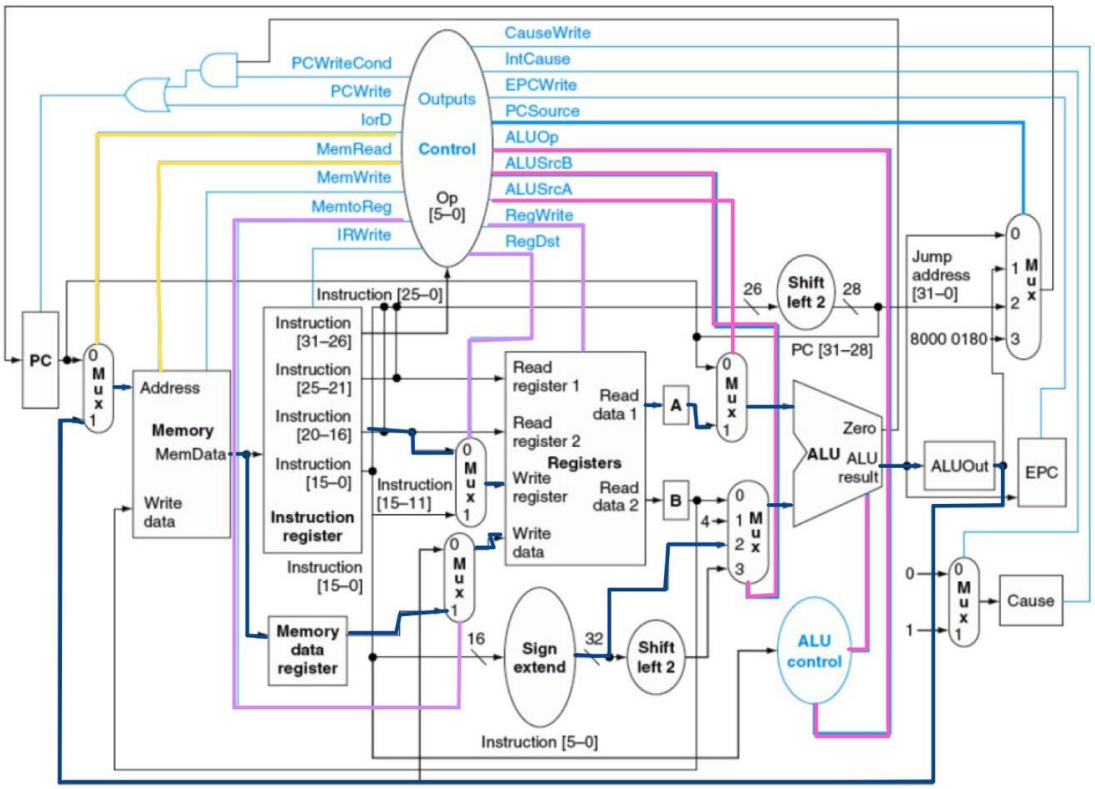


Figure 1 - The multi-cycle datapath together with the necessary control lines

LH 명령어로, 하위 16bit 를 load 한다. lh $rt, imm 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rt 는 결과를 저장할 대상, imm 은 immediate value 를 말한다. immediate value 를 sign-extend 하여 32bit 로 확장해준다. 핑크색 선은 LH 의 execution signal 을 의미하며 노란색 선은 LH 의 memory execution signal, 보라색 선은 LH 의 write back signal 을 의미한다. 파란색 선은 흐름을 나타낸다.

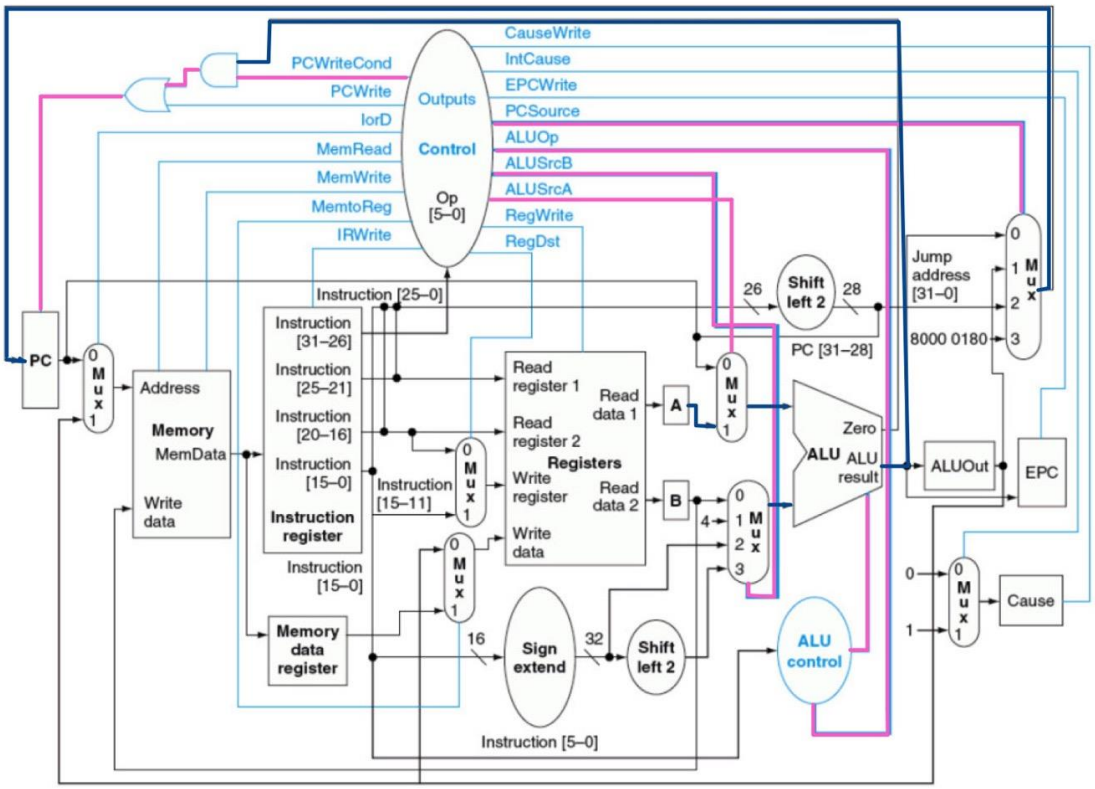


Figure 1 - The multi-cycle datapath together with the necessary control lines

BLTZ 명령어로, 레지스터의 값이 0 보다 작으면 label 로 분기한다. bltz $rs, label 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rs 는 분기 조건으로 사용될 레지스터이며, label 은 분기할 목적지 주소를 말한다. label 을 sign-extend 하여 32bit 로 확장하여 만든다. 핑크색 선은 BLTZ 의 signal 을 의미하며 파란색 선은 흐름을 나타낸다.

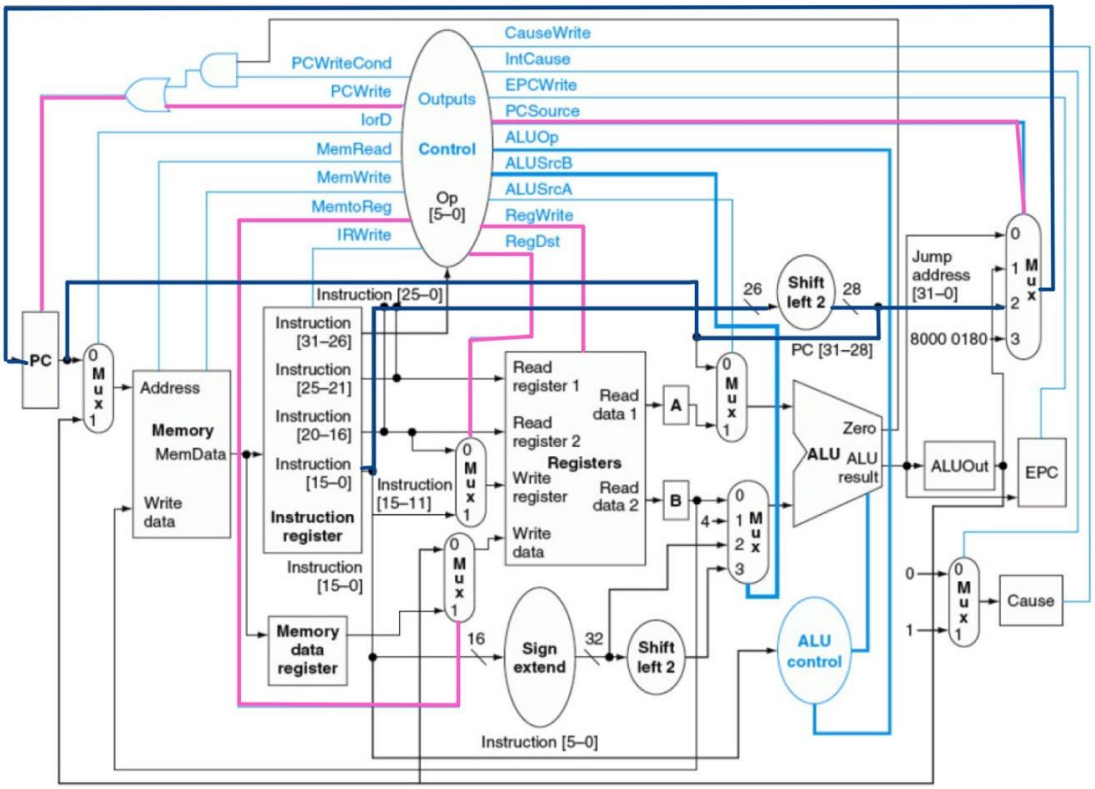


Figure 1 - The multi-cycle datapath together with the necessary control lines

JAL 명령어로, 현재 pc 값을 레지스터에 저장하고 지정된 주소로 분기한다. jar label 의 syntax 를 가지고, label 은 분기할 목적지 주소를 말한다. pc+4 와 pc + 4 + shift left 2 label 을 파란색 선으로 표현하였다. JAL 명령어의 signal 을 분홍색으로 표현 하였으며, 회로도에는 표현할 수 없지만, write register 에 $31 을 연결해 주어야 하고, write data 에 pc+4 도 연결해주어야 한다.

다음으로 구현한 내용에 대해서 설명해보겠다.

ROM_MICRO.txt | ROM_DISP.txt | M_TEXT_SEG.txt

파일 편집 보기

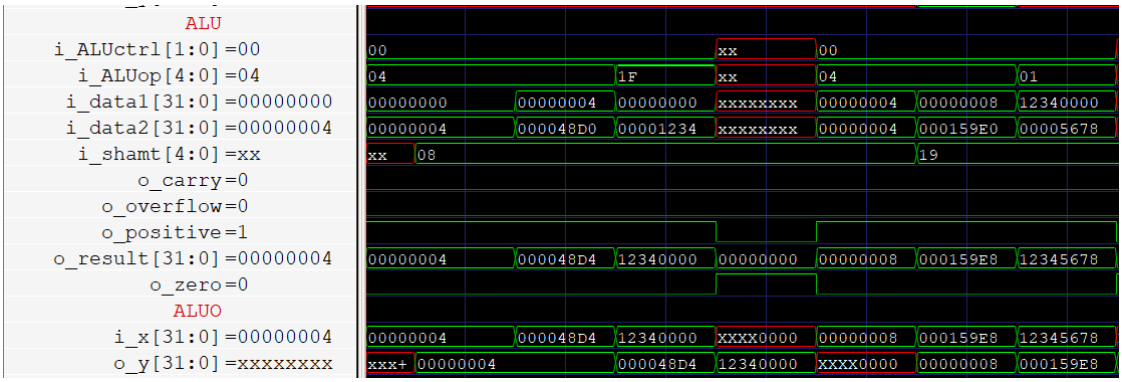
```
001111_00000_00010_0001001000110100 //lui r2 0x1234 -> r2,0x12340000
001101_00010_00011_0101011001111000 //ori r2 r3 0x5678 -> r3,0x12345678
001111_00000_00100_1111111111111111 //lui r4 0xffff -> r4,0xffff0000
001101_00000_00101_0000000000000011 //ori r0 r5 0x0003 -> r5,0x00000003

000000_00010_00011_00110_00000_100001 //addu r2 r3 r6 -> r6,24686578
000000_00010_00011_00111_00000_100101 //or r2 r3 r7 -> r7,12345678
001001_00010_01000_0001001000110100 //addiu r2 r8 0x1234 -> r8, 0x12341234
001110_00010_01001_0001001000110100 //xori r2 r9 0x1234 -> r9, 0x12341234
000000_00000_00010_01010_00001_000000 //sll r2 r10 shift 1 -> r10, 0x24680000
000000_00101_00010_01011_00000_000111 //srav r5 r2 r11 -> r11, 0x02468000
101001_00000_00011_0000000000000001 //sh r3 1 해당 주소 값 넣음
100001_00000_00011_0000000000000001 //lh r3 1 해당 주소 값을 가져옴
000001_00010_00000_0000_0000_0000_0010 //bltz r2 branch -> pc + 4
000001_00100_00000_0000_0000_0000_0010 //bltz r4 branch -> pc+4 + label
001111_00000_00010_0001001000110100 //lui r2 0x1234 -> r2,0x12340000
001101_00010_00011_0101011001111000 //ori r2 r3 0x5678 -> r3,0x12345678
000011_0000_0000_0000_0000_0000_000000 //jal
```

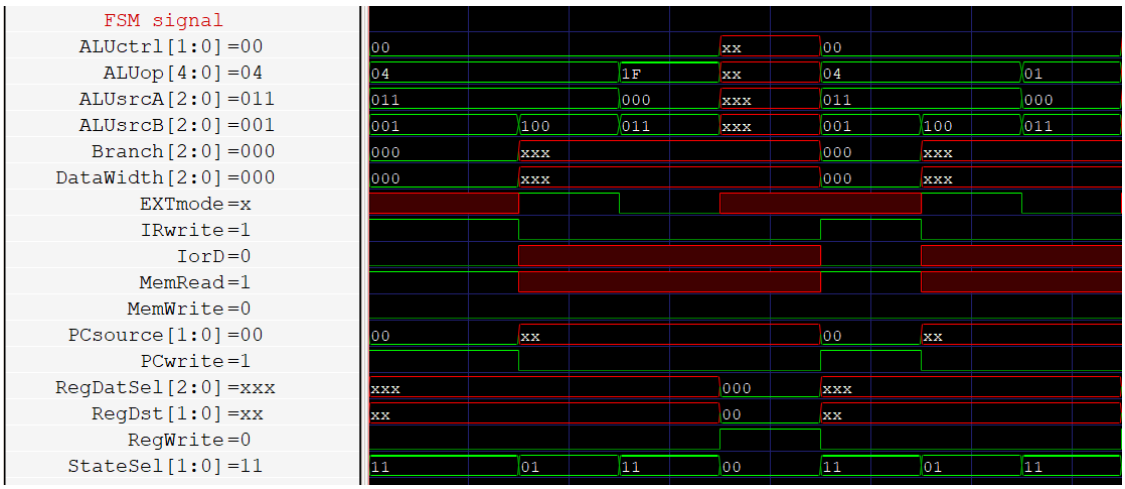
먼저 위의 사진은 M_TEXT_SEG.txt 파일로 작성한 명령어가 잘 동작하는지 확인하기 위한 txt 파일이다. 사용한 레지스터와 명령어, 그리고 기대값이 어떻게 나올지 주석으로 작성해 두었다.

실행 결과로 하나씩 살펴보겠다.

먼저 Instruction Fecth 부분이다. Instruction Fetch 단계에서는 pc=pc+4 를 올바르게 수행하는지, 그리고 pc 의 값에 해당하는 memory 의 주소에 값을 IR 에 잘 쓰는지 확인하였다. pc 의 0x00000000 이 가리키는 값이 i0x3C021234 이고 IR 에 쓰는 것이 확인되므로 값을 잘 읽은 것을 알 수 있다.

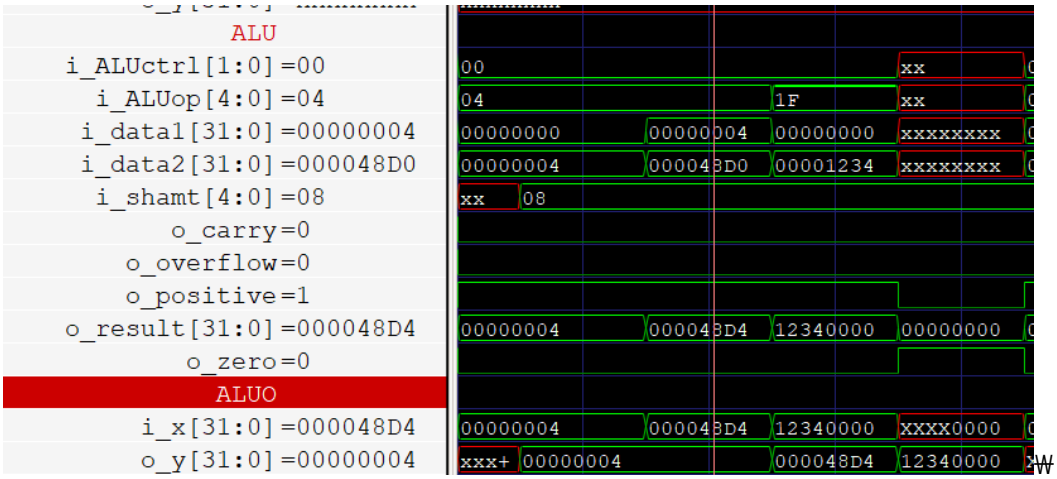


다음으로 ALU 에서 0x00000000 에 0x00000004 를 더해 0x00000004 가 되므로 pc+4 가 잘 작동하는 것을 알 수 있다.

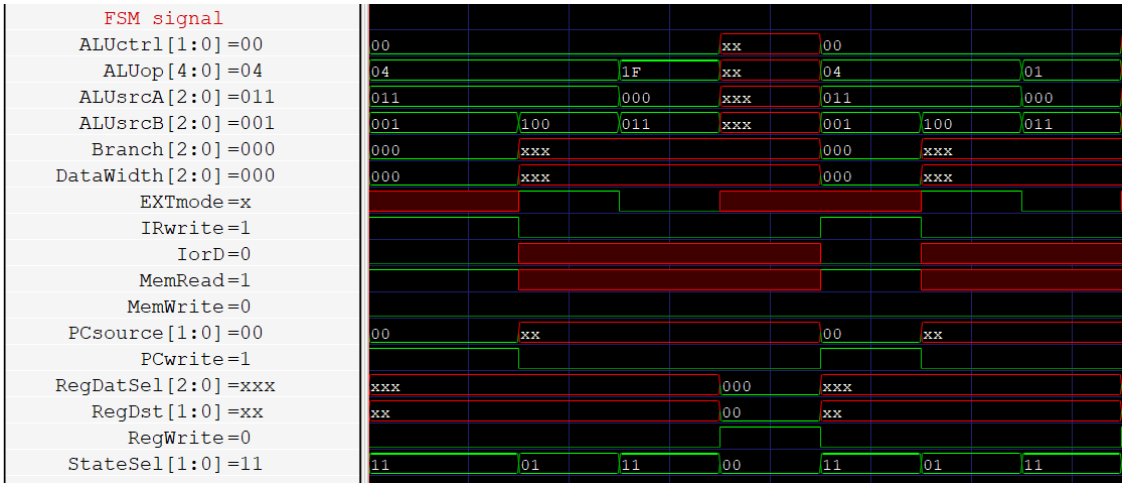


다음으로 Instruction fetch 의 FSM signal 이다.

다음으로 Decode/Register Fetch/Branch Address 부분이다. IR 에서 값을 읽어 비트단위로 쪼개어 00 이 rs 에, 02 가 rt 에 전달되었고 각 레지스터의 값을 A 와 B 에 전달해준 것을 확인할 수 있다.



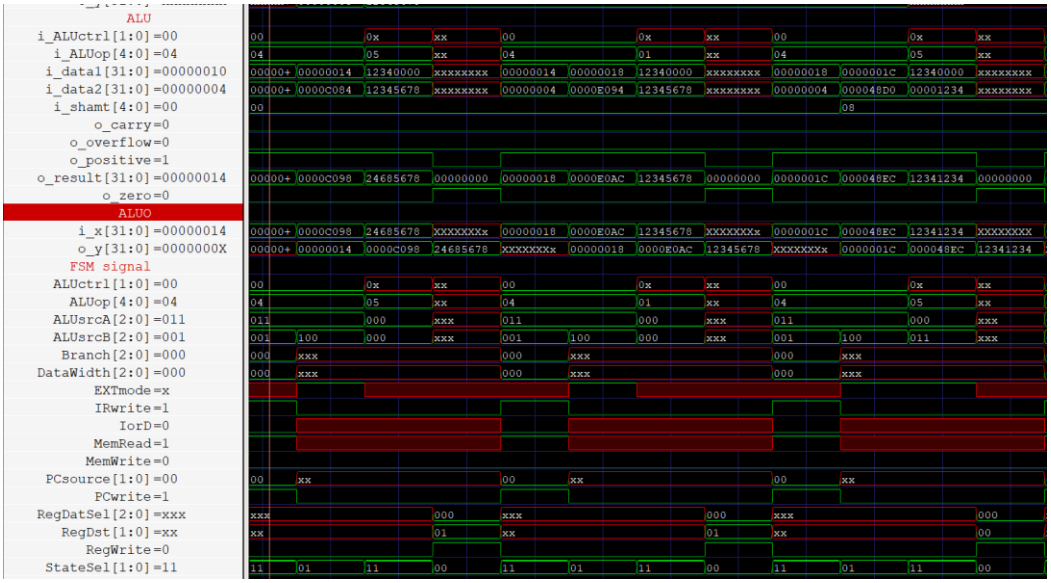
branch address 를 확인하는 부분으로 pc+4+shift amount 2 의 결과과 ALUO 의 input 으로 들어간 것을 알 수 있다.



다음으로 Decode/Register Fetch/Branch Address 의 FSM signal 이다.

각 명령어는 instruction fetch, Decode/Register Fetch/Branch Address 단계를 거치므로 설명은 생략하겠다.

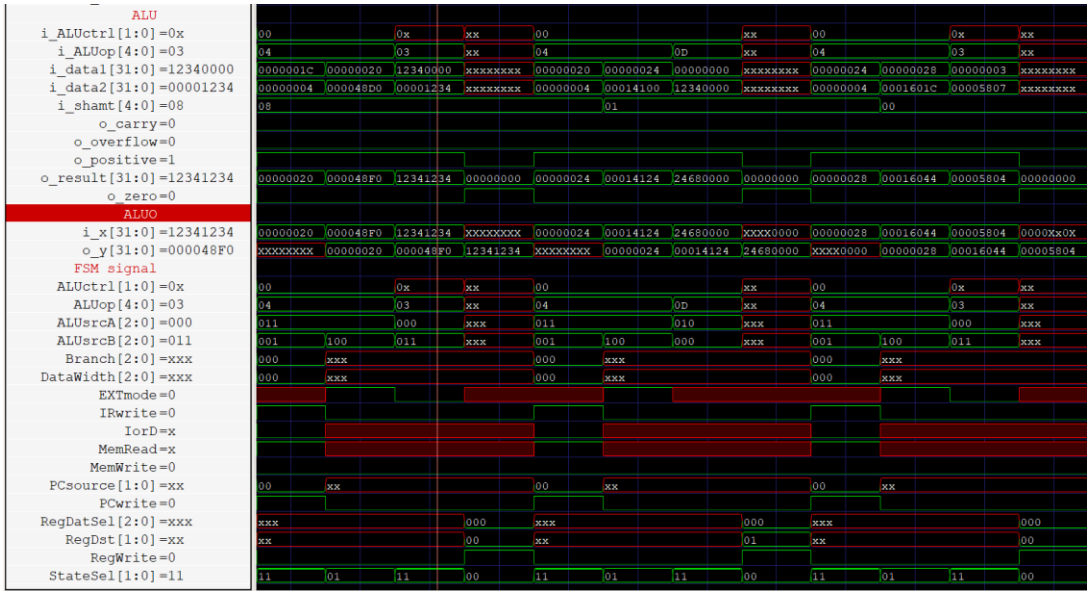
첫 두 단계는 lui 명령어와 ori 명령어를 사용하여 r2, r3, r4, r5 레지스터에 각각 0x12340000, 0x12345678, 0xffff0000, 0x00000003 을 넣어주었다.

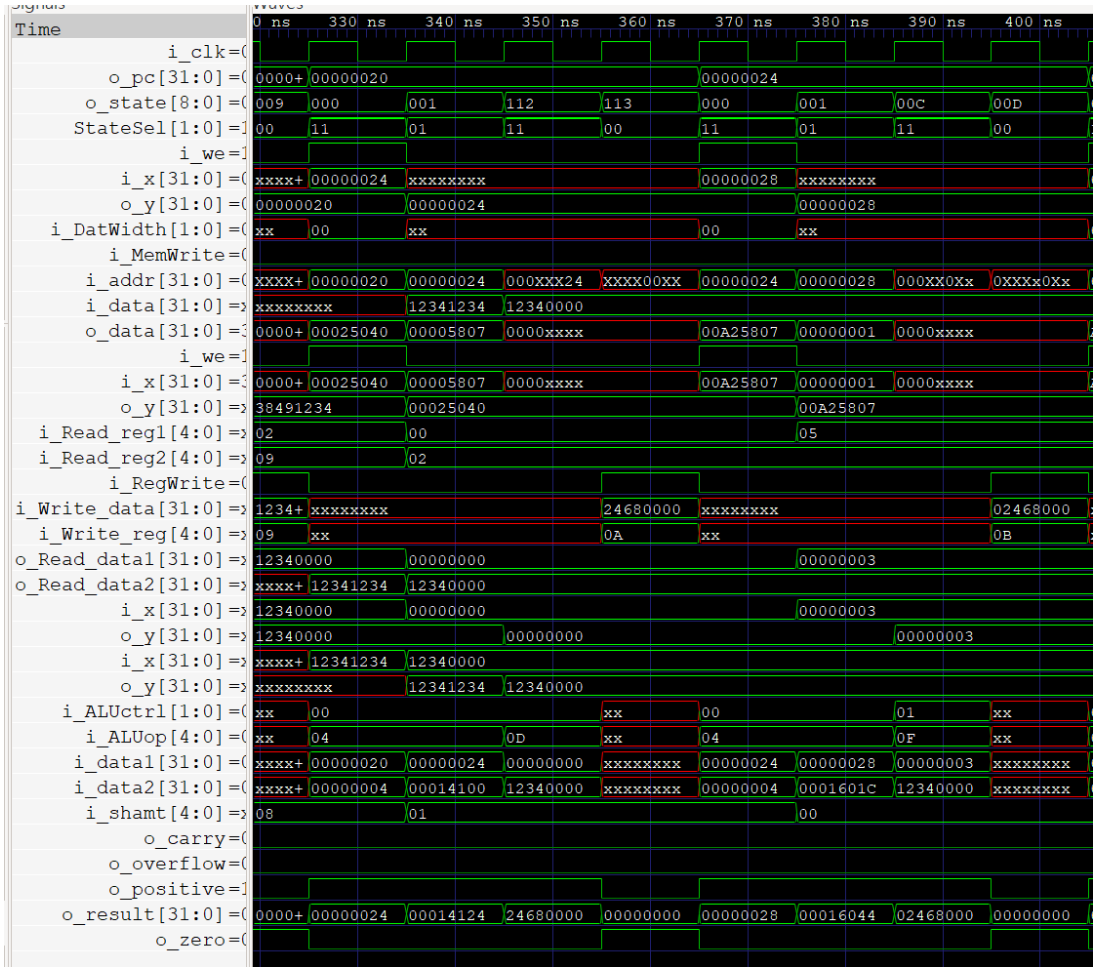


0x00000010 에서 addu 명령어를 fetch, decode, execution, write back 의 순서로 명령어를 수행한다. r2 와 r3 을 더해 r6 에 저장하였으므로, r6 에 0x24686578 이 저장된 것을 확인할 수 있다.

다음으로 0x00000014 에서 or 명령어를 fetch, decode, execution, write back 의 순서로 명령어를 수행한다. r2 와 r3 을 or 연산을 하여 r7 에 저장하였으므로 r7 에 0x12345678 이 저장된 것을 확인할 수 있다.

다음으로 0x00000018 에서 addiu 명령어를 fetch, decode, execution, write back 의 순서로 명령어를 수행한다. r2 와 0x1234 를 더해 r8 에 저장하였으므로, r8 에 0x12341234 가 저장된 것을 확인할 수 있다.

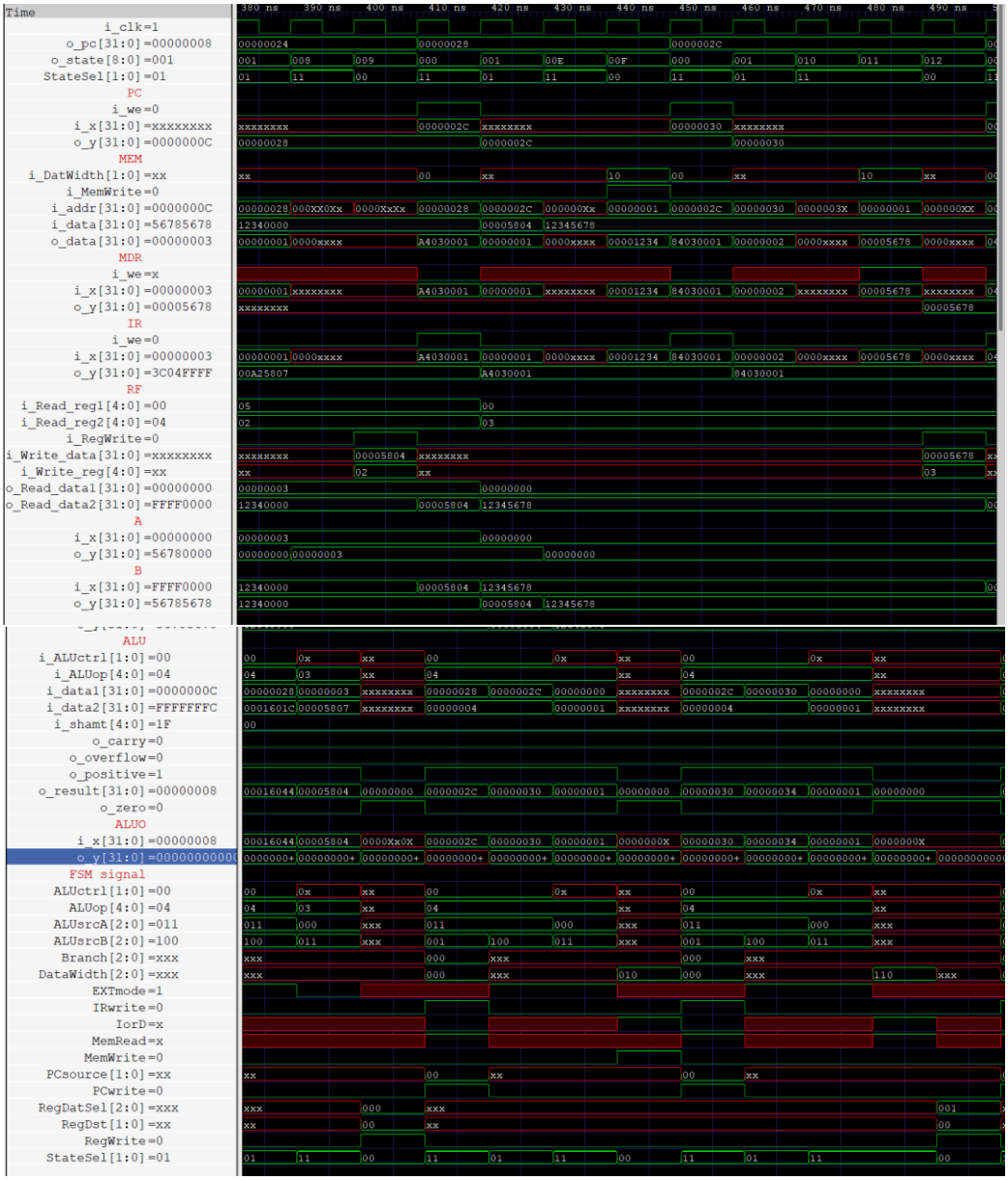




다음으로 0x0000001c 에서 xori 명령어를 fetch, decode, execution, write back 의 순서로 명령어를 수행한다. r2 와 0x1234 를 xor 연산을 하여 r9 에 저장하였으므로, r9 에 0x12341234 가 저장된 것을 확인할 수 있다.

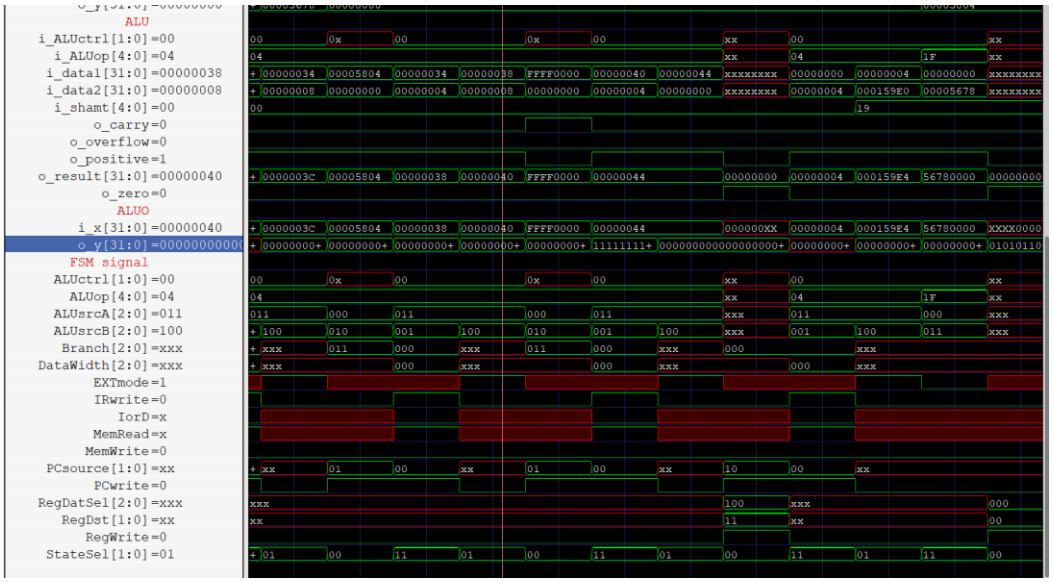
다음으로 0x00000020 에서 sll 명령어를 fetch, decode, execution, write back 의 순서로 명령어를 수행한다. r2 를 shift 1 해서 r10 에 저장하였으므로, r10 에 0x24680000 이 저장된 것을 확인할 수 있다.

다음으로 0x00000024 에서 srav 명령어를 fetch, decode, execution, write back 의 순서로 명령어를 수행한다. r2 를 r5 만큼 shift 해서 r11 에 저장하였으므로, r11 에 0x02468000 이 저장된 것을 확인할 수 있다.



다음으로 0x00000028 에서 sh 명령어를 fetch, decode, execution, write back 의 순서로 명령어를 수행한다. r3 에 저장된 값을 1 에 해당하는 주소의 메모리에 넣으므로 하위 16bit 에 1234 가 저장된 것을 알 수 있다.

다음으로 0x0000002c 에서 lh 명령어를 fetch, decode, execution, memory access, write back 의 순서로 명령어를 수행한다. 1 에 해당하는 메모리에서 값을 가져와 r3 에 저장한다. 따라서 0x00005678 이 저장된 것을 확인할 수 있다.



다음으로 0x00000030 에서 bltz 명령어를 수행하는데 r2 와 branch 값을 비교해서 r2 가 0 보다 크므로 pc+4 만 진행하므로 next_pc 가 0x00000034 가 된다..

다음으로 0x00000034 에서 bltz 명령어를 수행하는데 r4 와 branch 값을 비교해서 r4 가 0 보다 작으므로 pc+4+label 을 진행해하므로 next_pc 가 0x00000040 이 된다. 따라서 jal 명령어를 수행해 다음 pc 가 0x00000000 이 된 것을 알 수 있다.

```
00000000 : 00000000_00000000_00000000_00000000 : 00000000
00000001 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000002 : 01010110_01111000_00000000_00000000 : 56780000
00000003 : 01010110_01111000_01010110_01111000 : 56785678
00000004 : 11111111_11111111_00000000_00000000 : ffff0000
00000005 : 00000000_00000000_00000000_00000011 : 00000003
00000006 : 10101100_11110000_01010110_01111000 : acf05678
00000007 : 01010110_01111000_01010110_01111000 : 56785678
00000008 : 01010110_01111000_00010010_00110100 : 56781234
00000009 : 01010110_01111000_00010010_00110100 : 56781234
0000000a : 10101100_11110000_00000000_00000000 : acf00000
0000000b : 00001010_11001111_00000000_00000000 : 0acf0000
```

0 부터 10 까지의 레지스터이다. 명령어를 수행한 후의 값이 저장된 것을 확인할 수 있다.

3. 고찰

Micro instruction 을 반복해서 사용하면서 명령어를 다시 사용할 수 있다는 점이 좋았던 것 같다. single cycle 프로젝트 때는 한 번에 하나에 명령어만 수행할 수 있었지만 이번 프로젝트를 통해 multicycle 에 대해 더 잘 이해하고 multicycle processor 의 장점에 대해서도 알 수 있었던 것 같다. 기존의 single cycle 에서 구현했던 명령어를 바탕으로 이번 프로젝트를 진행했기에, 좀 더 수월했던 것 같다. 또한 회로도를 그리며 깨달았는데 jal 의 signal 을 fsm 에는 2 개만 표현했지만 그림에 표현한 것처럼 써줬어야 한다는 것을 알게되었다. 또한 fsm 을 그릴 때 type 이 같은 것 끼리 묶어서 그리려고 했지만, SLL 명령어의 경우 나머지 R-type 명령어와 달라 어려움이 있었다. 따라서 ALUsrcA 와 ALUsrcB 가 같은 것끼리 묶어 설계하였다.

4. Reference

컴퓨터구조실험 강의자료
컴퓨터구조 강의자료