컴퓨터구조 Project #2

(MIPS Multi Cycle CPU Implementation)

수업명: 컴퓨터구조

교수님: 이성원 교수

수업 시간: 월 3교시, 수 4교시

소속: 컴퓨터정보공학부

학번: 2018202074

이름: 김상우

Introduction

위 프로젝트에서 MIPS Instructions SUBU, XOR, SRA, BNE, BLEZ, JALR, ANDI, SLTIU, SB, LBU에 대해 Distpatch ROM filename을 의미하는 ROM_DISP.txt와 Micro-Program ROM filename을 의미하는 ROM_MICRO.txt의 값을 변경하는 것을 통해 구현하는 것을 목표로 하고 있다. (이 때 little endian을 사용하며 ALU function은 1cycle, 곱하기나 나누기는 2cycle을 소비한다. 또한 state number로 128미만의 0과 1을 제외한 수를 사용할 수 있다. 다른 명령어들은 구현되어 있다고 가정한다.

위 프로젝트에서 ROM의 경우 txt파일의 형태로 구현되며 Dispatch ROM은 ROM_DISP.txt, Micro-Program ROM filename의 경우 ROM_MICRO.txt로 구성된다. ROM_DISP.txt의 경우 undef를 나타내는 1bit와 raddr을 나타내는 8bit로 구성되며 각각 명령어,Undefined Instruction, Microprogram ROM의 주소값을 의미하게 된다. 해당 프로젝트에서 Raddr를 제어하여 state를 조절하게 될 것이다.

-설계

위 설계에서 Action은 종류별로 총 5개의 Stage로 분류될 것이며 다음의 Stage들은 1clock을 기준으로 나뉘게 될 것이다. 각각을 Clock으로 나눈 후 다음과 같이 구분하도록 한다.

1clock -Memory로부터 32bit word의 크기로 값을 받아 Instruction Register의 값을 적어준다. 읽어온 값들을 바탕으로 PC의 값을 ALU의 ADD기능으로 4를 더해, nextPC, 즉 다음 줄의 명령어를 가리킬 주소를 계산하고 다음 clock에서 넣어줄 준비를 한다.**(IF)**

2clock -Instruction Decode와 Register Fetch를 진행한다. 이 과정에서 Branch 사용시 갈 주소를 받기 위해 PC값과 immediate value<<2 값을 더해주게 된다.(ALU ADD기능 이용) 또한 EXEC에서 사용하게 될 Register A,B값을 받아오게 된다.**(ID)**

3clock -Execution 혹은 Branch/Jump Completion을 진행하게 된다. R-type의 경우 ALU control과 ALU operation Code를 바탕으로 Register A와 B의 ALU의 결과를 받게 되며, **(R-EXEC)** Memory instruction의 경우 ALU를 통한 덧셈을 통해 접근하게 될 Memory의 주소를 받게 된다.**(Memory access)** Branch의 경우 ALU를 통해 조건을 확인하고 해당 결과에 따라 PC값에 ID에서 만들어 놓은 주소를 사용한다.**(BRANCH)** J-type의 경우 값을 받고 PC값과 shift left 2된 Immediate 값을 연결하여 다음 PC값을 만들어 낸다.**(JUMP)** 기타 I-type의 경우(addi, addiu, slti, sltiu, andi, ori, xori등의 immediate value를 alu의 input으로 사용하는 경우로 이후 IR-type이라 칭하겠다.) Register 값과 immediate 값의 ALU control과 ALU operation Code를 바탕으로 둘의 input에 대한 ALU계산 결과를 받게 된다.**(I-EXEC)**

4clock -Memory Access와 R-Type Completion(IR-Type Completion)이 일어난다. Memory Access가 일어나는 경우, Memory로부터 값을 읽어오는 경우 MDR을 EXEC에서 구한 주소 값을 바탕으로 Memory에서 가져온 값으로 한다.**(Load)** 값을 쓰는 경우에는 Register B의 값을 Memory의 EXEC에서 구한 주소 값을 바탕으로 구해진 위치로 넣는다.**(Store)** R-type(혹은 IR-type)일 경우,

EXEC에서 구하게 된 ALU의 결과를 지정된 Register에 넣게 된다**.(R-WB, I-WB)**

5clock- Memory Read Completion가 일어나며 Memory에서 값을 읽어오게 되는 경우, MDR의 값을 지정된 Register에 넣는다. **(Load-WB)**

위 결과를 바탕으로 명령어 별로 지나게 되는 stage는 다음과 같게 될 것이다.

R-type, IR-type Instructions: IF-ID-(R-EXEC/I-EXEC)-(R-WB/I-WB)

Memory instructions Load:IF-ID-Memory access-Load-(Load-WB)

Memory instructions Store:IF-ID-Memory access-Load-(Load-WB)

Branch & Jump Instructions:IF-ID-(Branch/JUMP)

위의 내용에서 사용하게 되는 Stage 들에서 ALU등의 사용되는 부분들이 겹치지 않으므로 5개의 stage로 나뉜 작동 단계들은 Multi cycle에서 겹침 없이 잘 작동하게 될 것이다.

위의 나온 기능들을 구현하기 위해 Field들이 사용될 것이다. 이에 대해 우리는 다음과 같은 field에서의 signal들이 필요할 것이다.

-Memory접근 관련

lord-메모리 접근에 대해 Instruction인지 Data인지 판단, Memory로 들어가는 Mux에 연결된다.

MemRead-Memory 읽기 가능여부, Memory에 연결된다.

MemWrite-Memory 쓰기 가능여부,Memory에 연결된다.

DatWidth-Memory 접근에 대한 Data 너비(32bit, 16bit, 8bit Sign 16bit, sign 8 bit), Memory와 연결된다.

-Instruction Register 관련

IRwrite-Instruction Register 쓰기 가능여부, Instruction Register와 연결된다.

-Registers 관련

RegDst-Register가 써지는 곳(rt,rd,rs,31(pc)), Write Register로 들어가는 MUX와 연결된다.

RegDatSel-Register가 값을 받아오는 곳(ALUOUT,MDR,LO,HI,PC),Write data로 들어가는 MUX와 연결된다.

RegWrite-레지스터 쓰기 가능여부, Registers와 연결된다.

-Extension

EXTmode-Immediate Data의 확장 옵션(zero extension/sign extension), Sign Extender와 연결된다.

-ALU 관련

ALUsrcA-ALU input A에 들어가는 값(register A, 0x4, 0x0, PC, MDR), ALU input A로 들어가는 MUX와 연결된다.

ALUscrB-ALU input B에 들어가는 값(register B, 0x4, 0x0, SEU, SEU << 2), ALU input B로 들어가는 MUX와 연결된다.

ALUop-ALU가 실행할 동작, ALU와 연결된다.

ALU op	ALU Operation Code		
00000	Bitwise AND	01001	axb
00001	BitwiseOR	01010	Unsigned axb
00010	BitwiseNOR	01011	a/b
00011	BitwiseXOR	01100	Unsigned a/b
00100	a+b	01101	b<<a
00101	Unsigned a+b	01110	b>>a
00110	a-b	01111	b>>>a
00111	Unsigned a-b	10000	Set Less Than
01000	Zero	10001	Unsigned SLT
10010	HI=a	10011	LO=a

ALUctrl-ALU의 추가 제어 신호(ALU input 순서 / Use Shift amount or rs), ALU와 연결된다.

Branch-Branch 옵션 제어(조건없음/보존/음수 여부/동일 여부/양수 여부), PC로 향하는 부품에 연결되어 PC값에 Branch 계산 결과를 넣을지 말지 결정.

-PC관련

PCsrc-PC data를 받아오는 곳(ALU out, ALUOUT register, JUMP address, Current PC), PC로 향하는 Mux에 연결된다.

PCwrite-PC register쓰기 여부, PC에 연결된다.

-State 관련

StateSel-다음 State 결정 신호(0, Instruction에 의해 결정, 보존, 다음 state), ROM에 연결되어 ROM_DISP를 기준으로 진행된다.

이들은 제어신호로서 사용될 것이며 각 clock에서 연속적으로 동작해야하는 만큼 Multi Cycle간 부품들끼리의 제어를 위해 필요하다. 이들은 하나의 Control에서 위에 서술한 대로 각 부품들에 연결되어 신호를 보내며 이들을 제어하게 될 것이다.

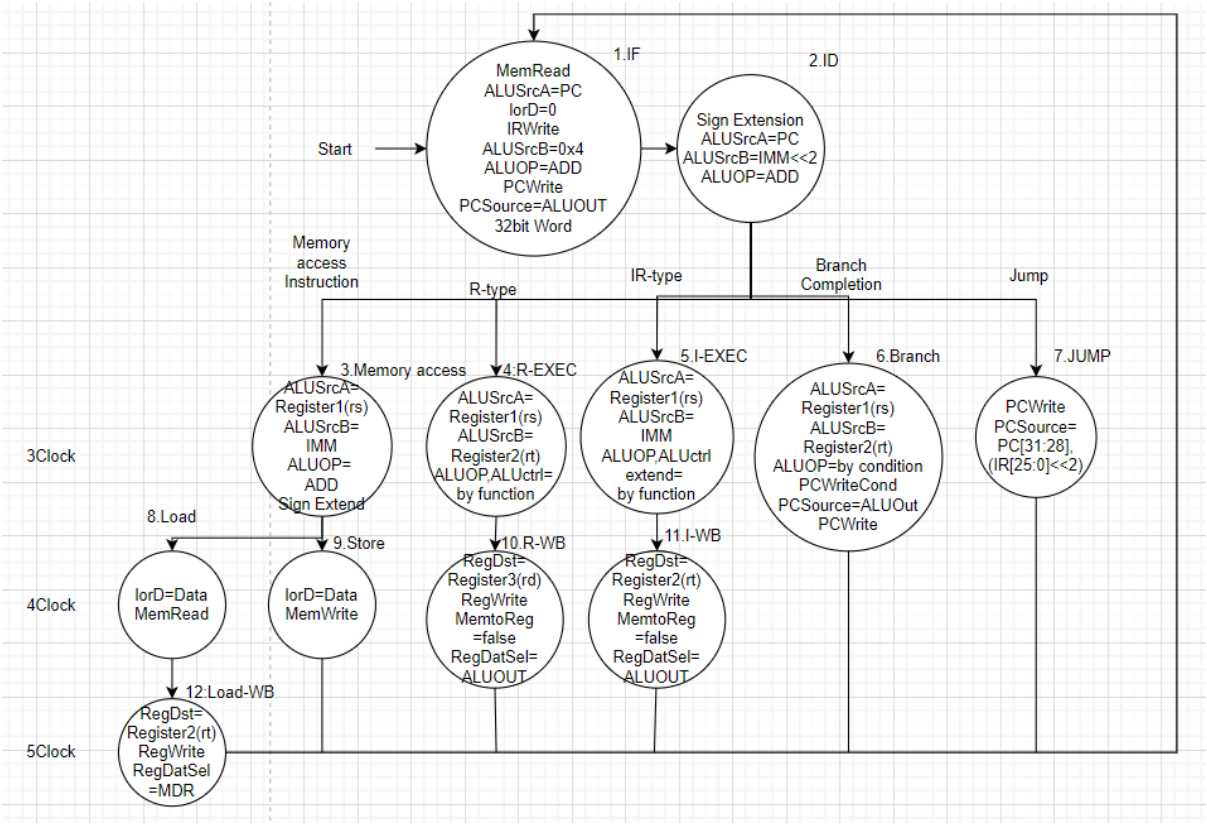
(ex 예를 들어 MemWrite나 RegWrite가 제어가 안 될 경우, Memory나 register에 적으면 안되는 타이밍에 data들이 계속해 들어가거나 적어야 하는 타이밍에 적히지 않을 수 있다.

해당 값들은 ROM_MICRO.txt를 통해 제어 될 것이다.(Control Signals for MCPU.pdf 기반)

위 내용을 바탕으로 Multi Cycle CPU FSM Diagram을 그리면 다음과 같이 그려질 것이다.

위 내용들을 바탕으로 반복되는 Micro-Instruction들을 확인할 수 있다. 대표적으로 IF의 경우 모든 명령어에 대해 각 명령어 종류에 따라 필요한 초기값이 호환되고 이 때문에 같은 동작을 취해도 문제가 없다. 이때의 signal은 MemRead, ALUScrA=PC, ALUScrB=0x4, IRWrite, ALUOp=ADD, PCWrite, IorD=0, PCSource=ALUOUT이 된다. 또한 ID의 경우도 마찬가지로 모든 명령어에 대해 같은 동작을 취해도 문제가 없을 것이다. 이때의 signal은 ALUSrcA는 input A, ALUSrcB는 IMM<<2, ALUOp=ADD, Sign Extend가 된다. 이들은 모든 명령어에서 반복되므로 ROM_MICRO.txt의 0x00, 0x01로 기술하고 각각 다음 state로, instruction에 맞는 state로 이동하게 하여 ROM의 효율을 높일 수 있도록 하였다.

추가적으로 3clock를 기준으로 명령어의 종류(Memory access, Rtype, IRtype, Branch Completion, Jump)에 따라 3clock의 공통적인 signal이 존재한다. (예를 들어 Memory access를 필요로 하는 Instruction의 경우 ALUSrcA로 rs의 값을 받으며 ALuSrcB의 값으로는 immediate value를 받고, ALUOP로 ADD를 받게 된다.) 이러한 반복되는 signal을 기준으로 FSM Diagram을 작성할 수 있을 것이다.



(기존에 제시된 Figure3와 ANDI를 위한 IR-Type을 추가한 모습. (Over flow는 따로 기재되지 않아 제거함.)) 해당 그림에서 우리가 구현해야 하는 명령어들의 경우 다음과 같은 회로를 따라갈 것을 알 수 있다.

SUBU-IF-ID-(R-EXEC)-(R-WB)

XOR-IF-ID-(R-EXEC)-(R-WB)

SRA- IF-ID-(R-EXEC)-(R-WB)

BNE- IF-ID-Branch

BLEZ- IF-ID-Branch

JALR- IF-ID-JUMP

ANDI-IF-ID-(I-EXEC)-(I-WB)

SLTIU- IF-ID-(I-EXEC)-(I-WB)

SB- IF-ID-Memory access-Store

LBU- IF-ID-Memory access-Load-(Load-WB)

위 Diagram을 보면 알 수 있듯, 명령어를 받고 IF-ID를 거쳐 명령어에 따라 이후 Stage(EXEC, MEM, Load)를 거치고 다시 IF로 돌아와 명령어를 실행하고 있는 형태로 동작하고 있다. 이는 명

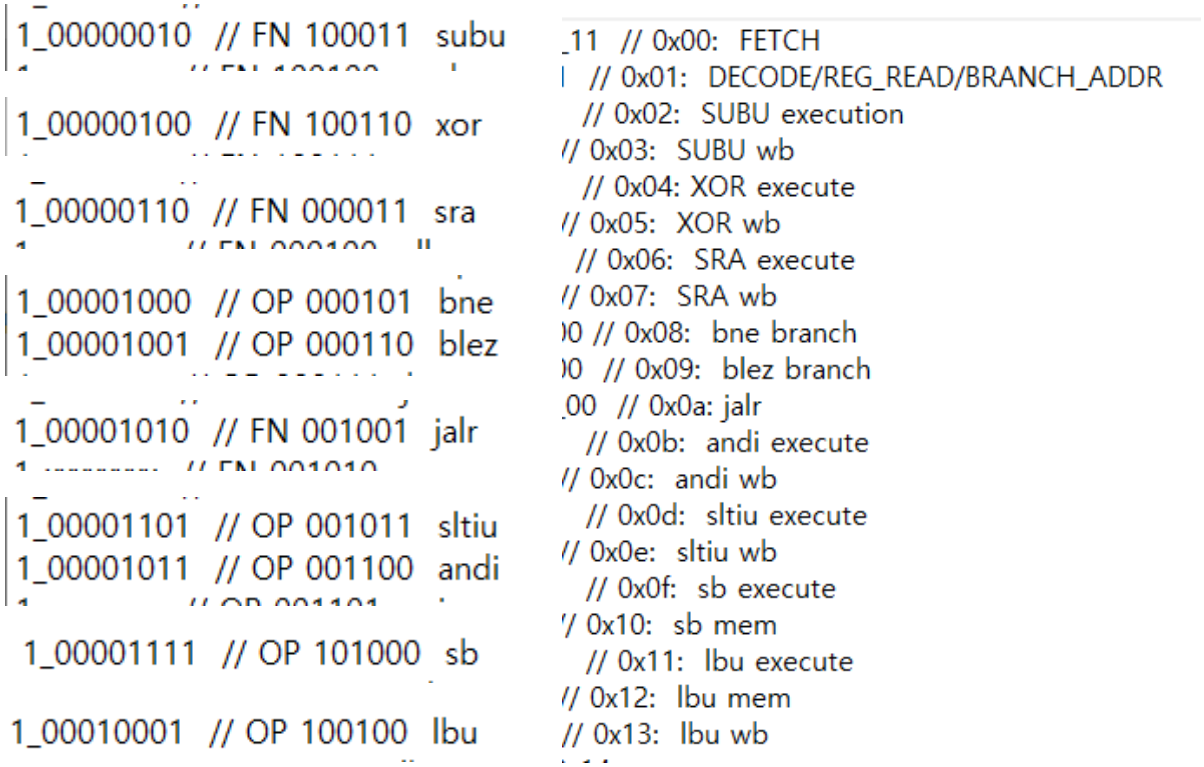
령어당 하나의 clock이 아닌 stage당 하나의 clock을 사용하는 해당 프로젝트의 특징을 활용해서 구현한 것으로 이 덕분에 해당 Multi Cycle CPU는 명령어당 사용되는 Clock의 수가 유동적이게 구현할 수 있었다.

-ROM_DISP

이는 해당 값들의 State위치를 나타내는 것으로 StateSel signal의 01형태인 State Indicated by Instruction에 사용된다. 이곳에는 해당 명령어의 3번째 clock때 오는 state의 ROM_MICRO에서의 line number를 적게 된다.

이유는 해당 project에서 StateSel signal의 01형태는 모든 명령어의 2clock에서 진행되는 ID에서 사용되며 이후 ID에서 3clock으로 향할 때 알맞은 동작을 하기 위해서는 이 값들이 다음 행동을 정확하게 가리킬 필요가 있기 때문이다.

다음과 같은 사실들을 바탕으로 해당 문서는 다음과 같이 작성되었다.



-ROM_MICRO

ROM_MICRO는 위에서 언급된 field에 따른 signal을 control하는 txt파일이다. 해당 파일에 조건 등에 따른 값들을 넣어 명령어들을 구현하게 된다.

-FETCH, DECODE/REG_READ/BRANCH_ADDR

명령어들을 설명하기에 앞서 이번에 구현하는 모든 명령어에 대해 stage IF와 ID를 지나는 것을 우리는 확인했다. 이들은 ROM_MICRO에 다음과 같이 구현될 것이다.

```
0_1_0_000_1_xx_xxx_0_x_011_001_00100_00_000_00_1_xxxxxxxx_11  // 0x00:  FETCH
x_x_0_xxx_0_xx_xxx_0_1_011_100_00100_00_xxx_xx_0_xxxxxxxx_01  // 0x01:  DECODE/REG_READ/BRANCH_ADDR
```

IF에 해당하는 Fetch의 경우 다음과 같은 진행을 가진다.

IorD-Instruction Fetch인 만큼 Memory에 instruction를 위해 접근

MemRead-Instruction을 읽어오기 위해 Memory Read를 한다.

MemWrite-Memory에 Write하지 않는다.

DatWidth-32bit word로 받는다

IRwrite-Instruction값을 보내줘야 하므로 Instruction Register에 Write할 수 있다.

RegDst-상관없다.

RegDatSel-상관없다.

RegWrite-Register file에 write하지 않는다.

EXTmode-상관없다.

ALUsrcA-PC값이 들어온다.

ALUsrcB-0x4값이 들어온다 .이후 ALUsrcA의 PC값과 더해져 다음 명령어 주소를 만들게 된다.

ALUop-ALUsrcA와 ALUsrcB의 값을 더해 다음 명령어 주소를 만든다.

ALUctrl-Normal ALU input, Shift=Shift Amount

Branch-조건이나 jump없이 branch한다.

PCsrc-ALU output으로부터 받는다.

PCwrite-PC에 write한다.

StateSel-ID로의 이동을 위해 Next State = CurrentState +1이다.

ID에 해당하는 DECODE/REG_READ/BRANCH_ADDR 의 경우 다음과 같은 진행을 가진다.

lord-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite-Instruction Register에 Write할 수 없다.

RegDst-상관없다.

RegDatSel-상관없다.

RegWrite-Register file에 write하지 않는다.

EXTmode-Sign Extension이 필요하다.

ALUsrcA-PC값이 들어온다.

ALUsrcB-MDR값이 들어온다 .이후 ALUsrcA의 PC값과 더해져 Branch될 주소를 만들게 된다.

ALUop-ALUsrcA와 ALUsrcB의 값을 더해 Branch될 주소를 만든다.

ALUctrl-Normal ALU input, Shift=Shift Amount

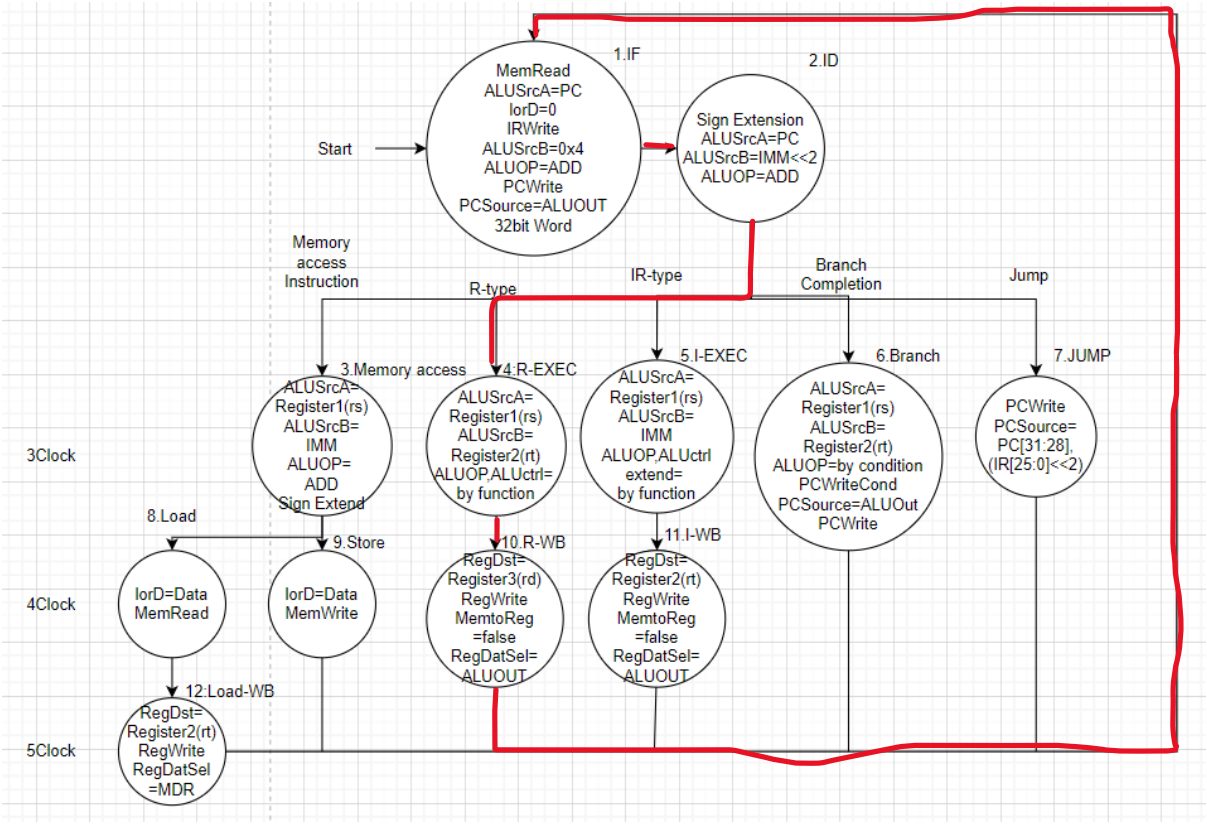
Branch-Don't Care

PCsrc- Don't care

PCwrite-PC에 write하지 않는다.

StateSel-명령어에 맞춰 State를 정한다.(ROM_DISP 기반)

-SUBU



SUBU의 경우 r-type 명령어로서 IF-ID-(R-EXEC)-(R-WB)로의 4cycle을 가지게 된다. SUBU는 R-EXEC에서 alu를 통해 해당 계산을 진행하고 R-WB에서 Register에 값을 저장하게 될 것이다. 이를 바탕으로 아래와 같이 작성할 수 있다.

IF, ID는 위에서 확인하였으므로 우리는 EXEC와 MEM(R complete)에서의 확인만 확인하면 된다.

```
x_x_0_xxx_0_xx_xxx_0_x_000_000_00111_0x_xxx_xx_0_xxxxxxxx_11  // 0x02:  SUBU execution
x_x_0_xxx_0_01_000_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00  // 0x03:  SUBU wb
```

우선 R-EXEC에 대한 field의 정리이다. R-EXEC에선 기본적으로 ALU에서 Unsigned A-B를 하는 것을 중점으로 진행된다.

IorD-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite- Instruction Register에 Write하지 않는다.

RegDst-상관없다.(이후 (R-WB)에서 register에 넣어준다.)

RegDatSel- Don't care

RegWrite-Register file에 write하지 않는다.

EXTmode-상관없다.

ALUsrcA-Register A의 값을 ALU input A로 한다.

ALUsrcB- Register B의 값을 ALU input B로 한다.

ALUop-Unsigned a-b 를 진행한다.

ALUctrl-Normal ALU input, No Shift

Branch-Branch나 jump가 발생하지 않는다.

PCsrc-Don't Care

PCwrite-No PC Register Write

StateSel-ID로의 이동을 위해 Next State = CurrentState +1이다.

R-WB의 경우 Register의 값을 저장하기 위해 다음과 같은 진행을 가진다.

lord-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite-Instruction Register에 Write할 수 없다.

RegDst-$rd에 저장된다.

RegDatSel-ALUOUT의 값을 Register file에 적는다.

RegWrite-Register file에 write한다.

EXTmode- Don't care

ALUsrcA- Don't care

ALUsrcB- Don't care

ALUop- Don't care.

ALUctrl- Don't care

Branch-Don't Care

PCsrc- Don’t care

PCwrite-PC에 write하지 않는다.

StateSel-명령어의 마지막 stage이므로 IF로 돌아가기 위해 Next State =0

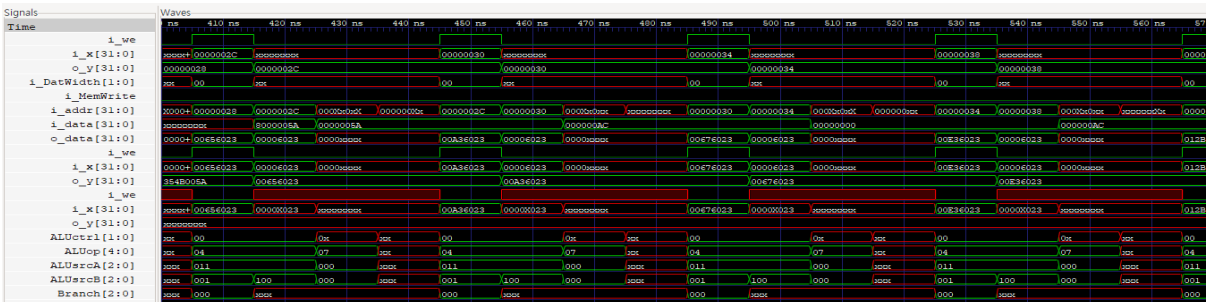
Register 3=00011(2):0x000000AC Register 00101(2):0x0000005A

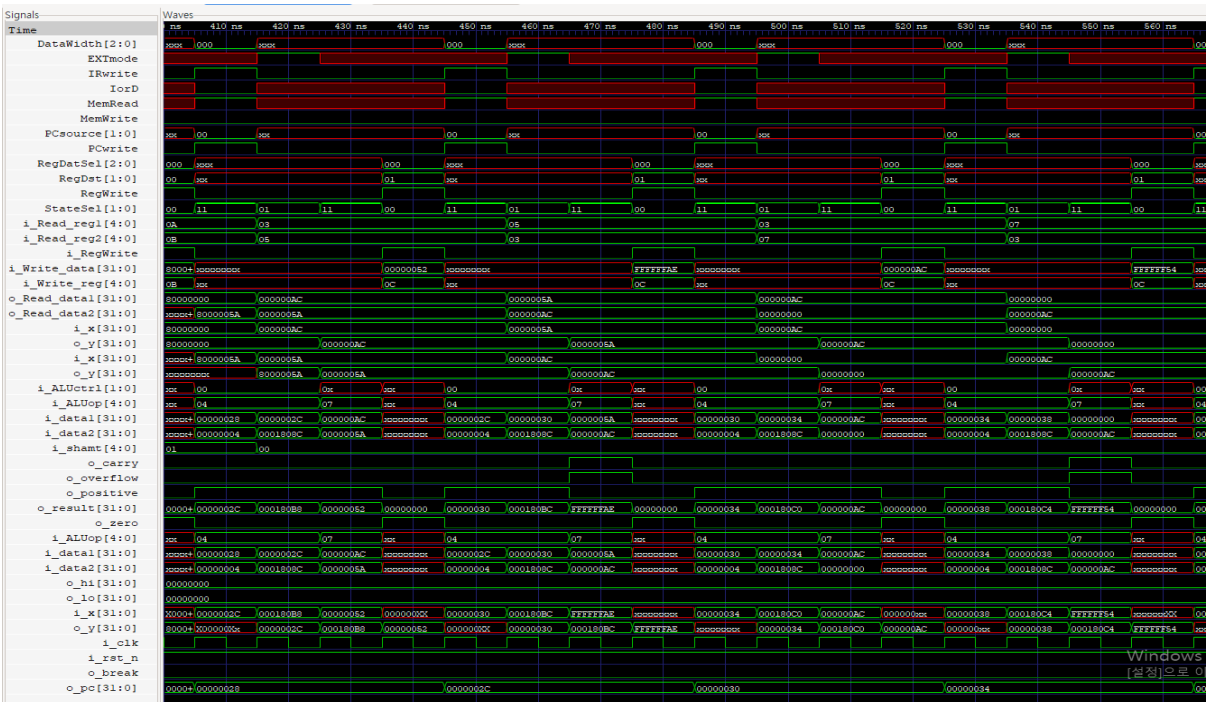
Register 00111(2):0x00000000 Register 01001(2):0x800000AC

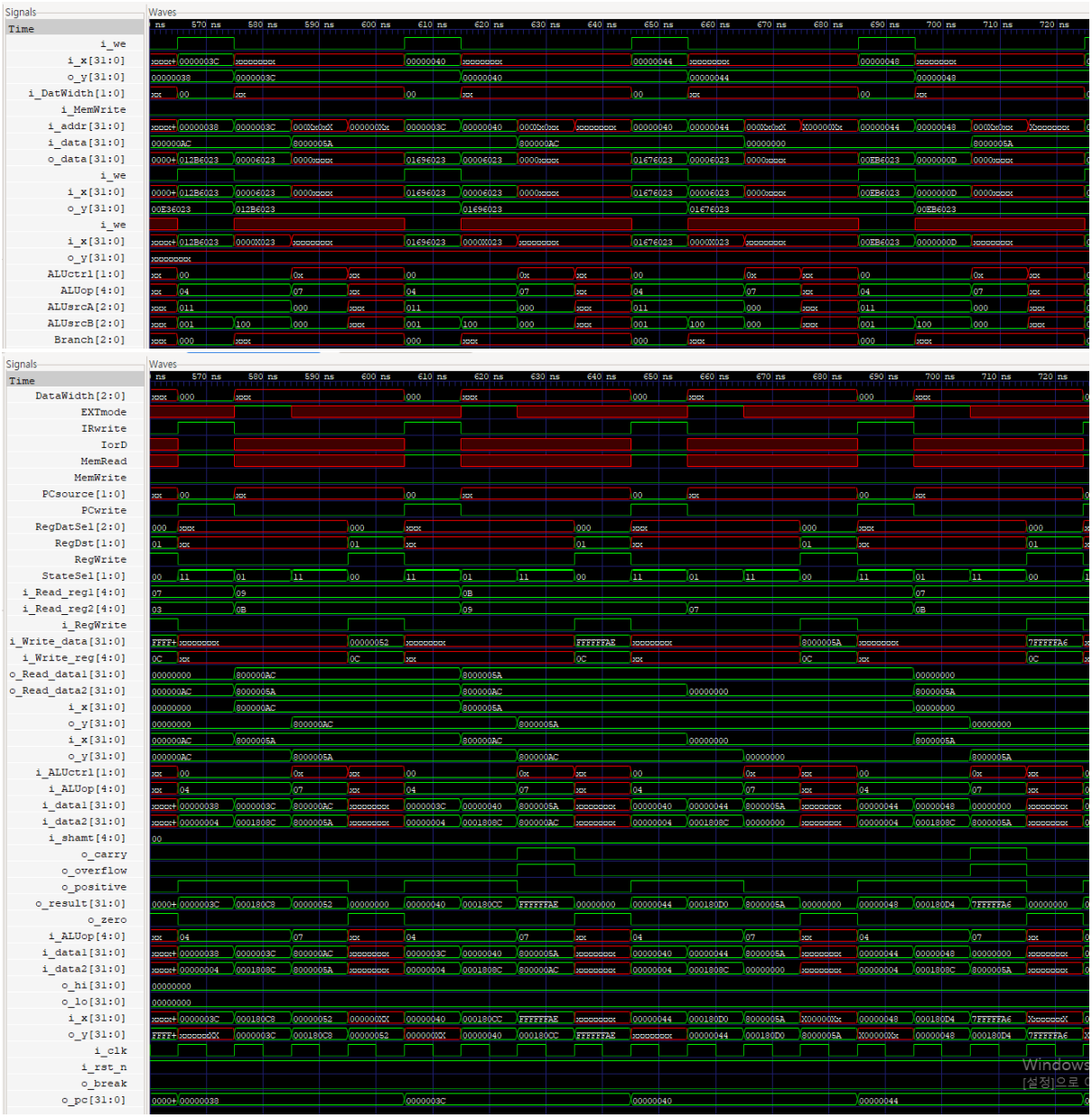
Register 01011(2):0x8000005A

위와 같이 레지스터에 lui와 ori를 이용해 값들을 지정한 후 테스트 벤치를 돌렸다. 이때 테스트 벤치는 아래와 같이 잡았다. (양수-양수 결과로 양수/음수가 나올 때 양수-0, 0-양수일 때, MSB가 1일때의 계산으로 결과가 양수/음수가 나올때, MSB가 1일 때 0을 뺄 때, 0에서 MSB가 1인 값을 뺄 때.) 해당 코드를 돌리기 전, SUBU는 unsigned에서 진행되는 만큼, 결과가 0아래로 내려갈 경우 음수를 표현하지 못해 오류가 생길 것이라 예상했다.

(reg3-reg5, reg5-reg3, reg3-reg7 ,reg7-reg3,reg9-reg11,reg11-reg9,reg11-reg7,reg7-reg11)







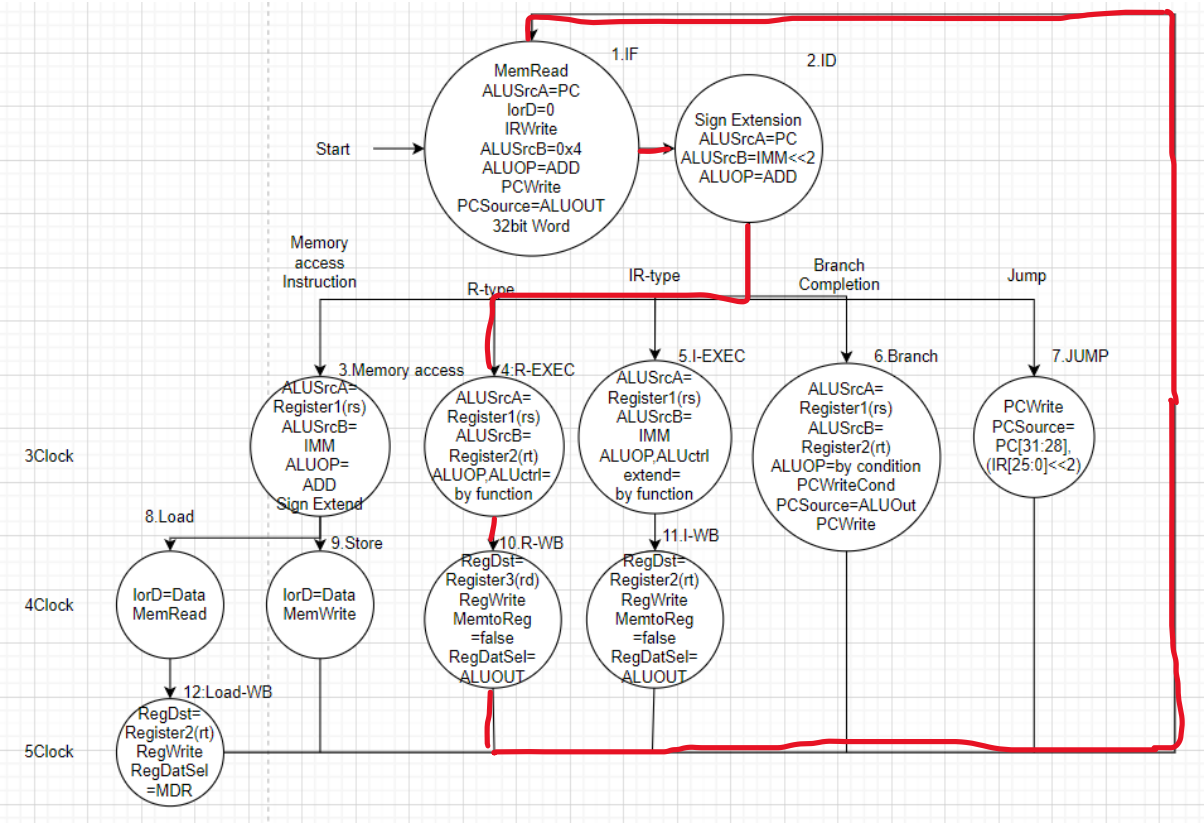
다음은 결과이다. 위와 같이 clock이 subu한번당 4번에 걸쳐 (IF-ID-(R-EXEC)-(R-WB)) 진행되는 것을 확인할 수 있었다. i_data1, 2의 값으로 알맞은 레지스터에서 값이 내보내짐을 확인할 수 있었다. 또한 i_Write_reg의 값들을 통해 지정했던 레지스터에 저장되고 있음도 확인했다. R-EXEC의 O_result와 R-WB의 i_Write_data를 통해 레지스터에 저장되는 ALU의 결과를 확인할 수 있었다.

Input1(hex)	Input2(hex)	Expected result(hex)	Result(hex)
000000AC	0000005A	00000052	00000052
0000005A	000000AC	FFFFFFAE(error)	FFFFFFAE(error)
000000AC	00000000	000000AC	000000AC
00000000	000000AC	FFFFFF54(error)	FFFFFF54(error)
800000AC	8000005A	00000052	00000052

8000005A	800000AC	FFFFFFAE(error)	FFFFFFAE(error)
8000005A	00000000	8000005A	8000005A
00000000	8000005A	7FFFFFAE(error)	7FFFFFAE(error)

예상대로 SUBU는 unsigned에서 진행되는 만큼 음수가 나올 시 적절한 값을 출력해주지 못했다. 허나 결과값이 0이상의 값일 경우 올바르게 표현되고 있음을 확인할 수 있었다.

-XOR



XOR의 경우 r-type 명령어로서 IF-ID-(R-EXEC)-(R-WB)로의 4cycle을 가지게 된다. XOR는 R-Exec에서 alu를 통해 해당 계산을 진행하고 R-WB에서 Register에 값을 저장하게 될 것이다. 이를 바탕으로 아래와 같이 작성할 수 있다.

IF, ID는 위에서 확인하였으므로 우리는 EXEC와 MEM(R complete)에서의 확인만 확인하면 된다.

```
x_x_0_xxx_0_xx_xxx_0_x_000_000_00011_0x_xxx_xx_0_xxxxxxxx_11  // 0x04: XOR execute
x_x_0_xxx_0_01_000_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00  // 0x05:  XOR wb
```

우선 R-EXEC에 대한 field의 정리이다. R-EXEC에선 기본적으로 ALU에서 Bitwise XOR를 하는 것을 중점으로 진행된다.

lorD-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite- Instruction Register에 Write하지 않는다.

RegDst-상관없다.(이후 R-WB에서 register에 넣어준다.)

RegDatSel- Don't care

RegWrite-Register file에 write하지 않는다.

EXTmode-상관없다.

ALUsrcA-Register A의 값을 ALU input A로 한다.

ALUsrcB- Register B의 값을 ALU input B로 한다.

ALUop-Bitwise XOR 를 진행한다.

ALUctrl-Normal ALU input, No Shift

Branch-Branch나 jump가 발생하지 않는다.

PCsrc-Don't Care

PCwrite-No PC Register Write

StateSel-ID로의 이동을 위해 Next State = CurrentState +1이다.

R-WB의 경우 Register의 값을 저장하기 위해 다음과 같은 진행을 가진다.

lord-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite-Instruction Register에 Write할 수 없다.

RegDst-$rd에 저장된다.

RegDatSel-ALUOUT의 값을 Register file에 적는다.

RegWrite-Register file에 write한다.

EXTmode- Don't care

ALUsrcA- Don't care

ALUsrcB- Don't care

ALUop- Don't care.

ALUctrl- Don't care

Branch-Don't Care

PCsrc- Don't care

PCwrite-PC에 write하지 않는다.

StateSel-명령어의 마지막 stage이므로 IF로 돌아가기 위해 Next State =0

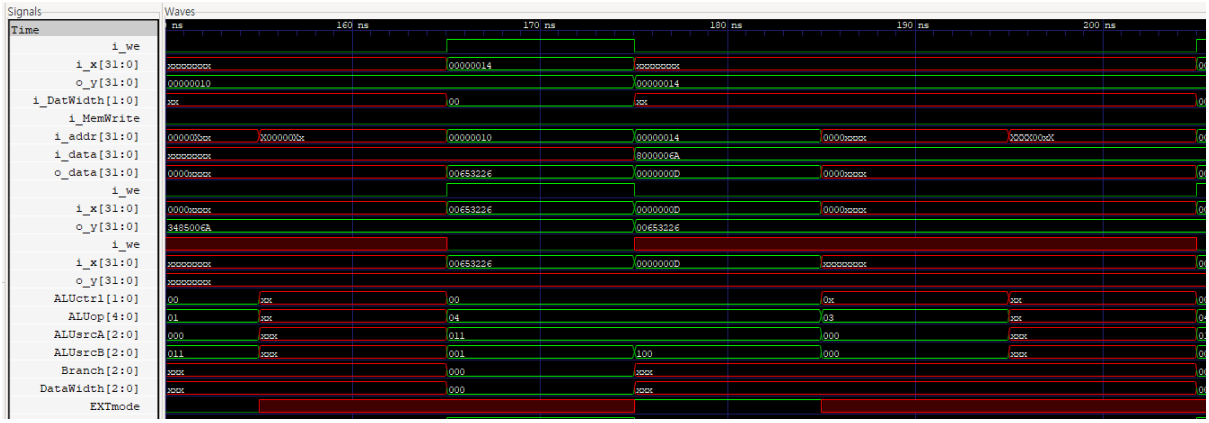
이들이 잘 동작하는지 확인하기 위해 두개의 input은 다음과 같이 주었다.

10000000_00000000_00000000_10101100(2) / 100000000_00000000_00000000_01101010(2)

이 둘을 각각 reg3, reg5에 저장 후 XOR하여 reg6에 값을 저장하는 테스트벤치로 진행하였다.

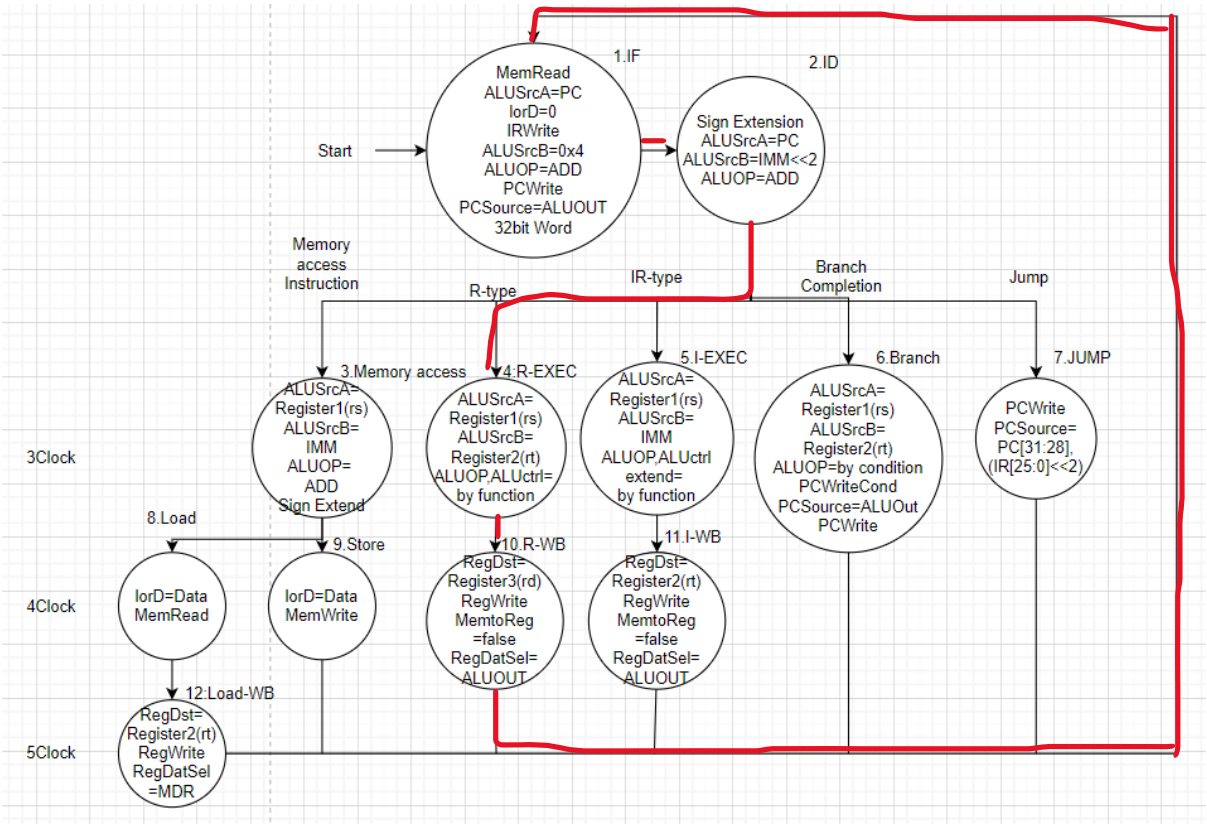
이들은 n번째의 bit가 다를 경우 reg6의 nbit에 1을, 같을 경우 0을 줄것이므로 예측 결과값은 00000000_00000000_00000000_11000110(2)이다.

Testbench의 경우에는 두 input의 bit가 각각 (0,0)(1,0)(0,1)(1,1)일 때를 고려해서 input값을 주었다.



위와 같은 결과가 나왔다. 결과를 보면 R-Exec에서의 i_data1,2의 값을 통해 input으로 register3, 5에 0x800000AC, 0x8000006A값이 들어감을 확인했다.(해당 hex값은 위의 input으로 하려했던 binary값과 같다.) 또한 XOR가 사용된 부분을 보면 00110(2)에 해당하는 곳에 잘 저장되고 있음을 알 수 있다. 또한 R-Exec의 o_result와 R-wb의 i_Write_data를 통해 예측된 값과 같은 결과가 나왔다는 것을 확인했다. 이는 R-WB에서 i_write_reg가 06인 것을 확인하는 것으로서 알맞은 register에 들어가게 되었다는 것 또한 확인할 수 있었다.

SRA



SRA의 경우 r-type 명령어로서 IF-ID-(R-EXEC)-(R-WB)로의 4cycle을 가지게 된다. SRA는 R-Exec에서 alu를 통해 해당 계산을 진행하고 MEM에서 Register에 값을 저장하게 될 것이다. 이를 바탕으로 아래와 같이 작성할 수 있다.

IF, ID는 위에서 확인하였으므로 우리는 R-EXEC와 R-WB에서의 확인만 확인하면 된다.

```
x_x_0_xxx_0_xx_xxx_0_x_xxx_000_01111_00_xxx_xx_0_xxxxxxxx_11  // 0x06:  SRA execute
x_x_0_xxx_0_01_000_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00  // 0x07:  SRA wb
```

우선 SRA에 대한 field의 정리이다. R-EXEC에선 기본적으로 ALU에서 Shift Right Arithmetic를 하는 것을 중점으로 진행된다.

IorD-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite- Instruction Register에 Write하지 않는다.

RegDst-상관없다.(이후 (R-WB)에서 register에 넣어준다.)

RegDatSel- Don't care

RegWrite-Register file에 write하지 않는다.

EXTmode-상관없다.

ALUsrcA-B값을 shamt만큼 shift right 해주므로 상관없다.

ALUsrcB- Register B의 값을 ALU input B로 한다.

ALUop-b>>>a를 진행한다.

ALUctrl-Normal ALU input, Shift= Shift Amount

Branch-Branch나 jump가 발생하지 않는다.

PCsrc-Don't Care

PCwrite-No PC Register Write

StateSel-ID로의 이동을 위해 Next State = CurrentState +1이다.

R-WB의 경우 Register의 값을 저장하기 위해 다음과 같은 진행을 가진다.

lord-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite-Instruction Register에 Write할 수 없다.

RegDst-$rd에 저장된다.

RegDatSel-ALUOUT의 값을 Register file에 적는다.

RegWrite-Register file에 write한다.

EXTmode- Don't care

ALUsrcA- Don't care

ALUsrcB- Don't care

ALUop- Don't care.

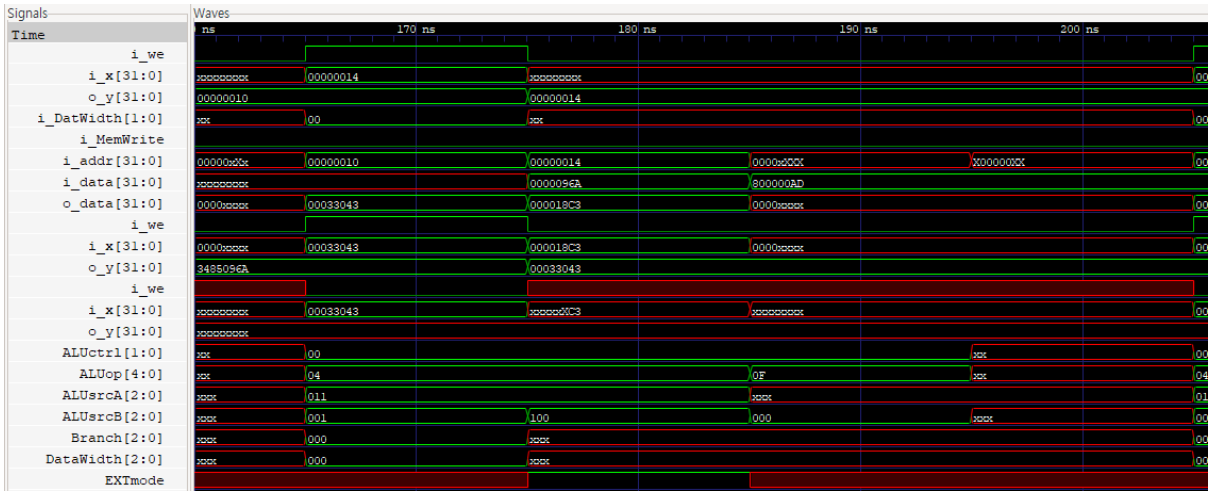
ALUctrl- Don't care

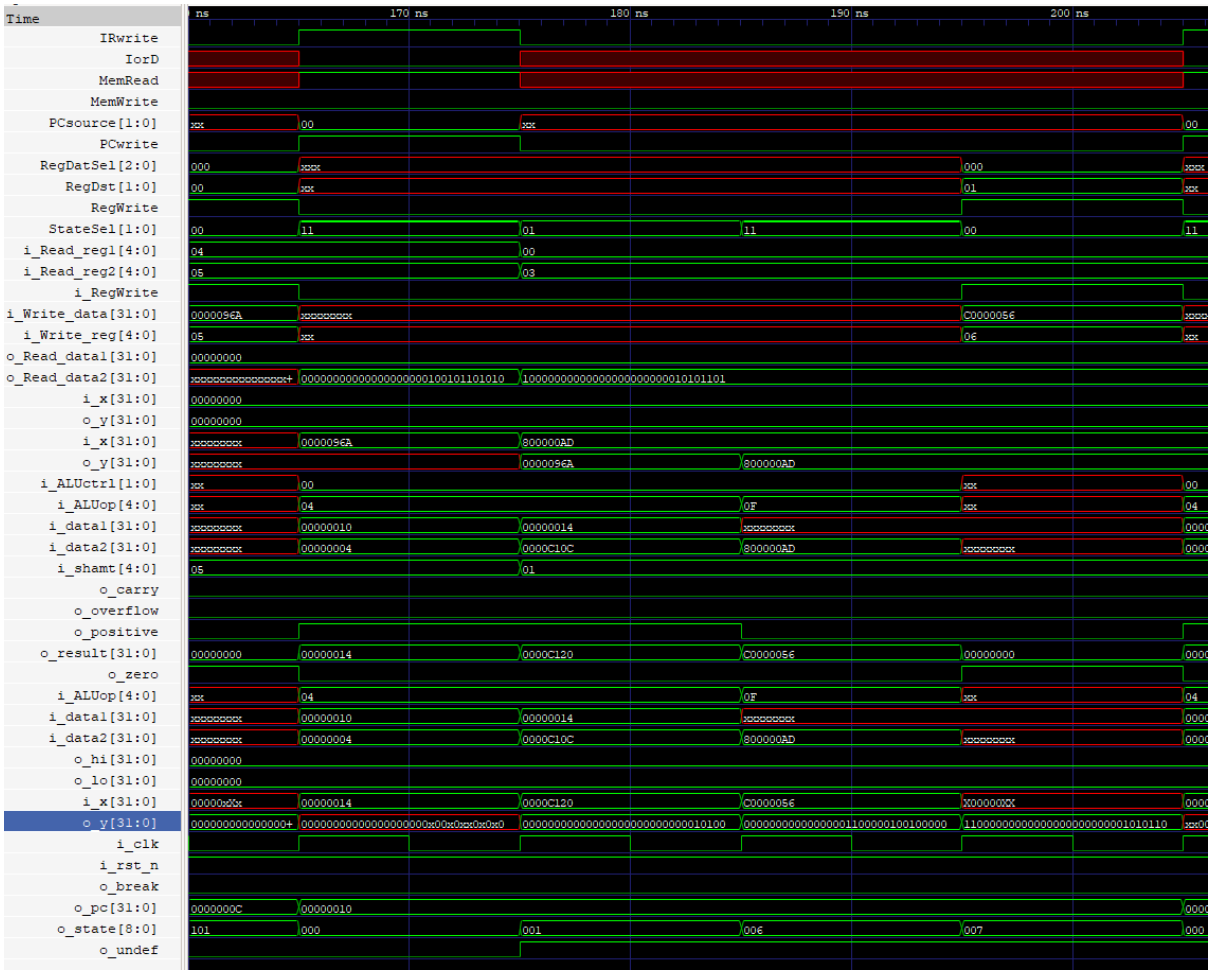
Branch-Don't Care

PCsrc- Don't care

PCwrite-PC에 write하지 않는다.

StateSel-명령어의 마지막 stage이므로 IF로 돌아가기 위해 Next State =0





위는 i_shamt값에서 알 수 있듯, shamt가 1인 경우에서의 모습이다.

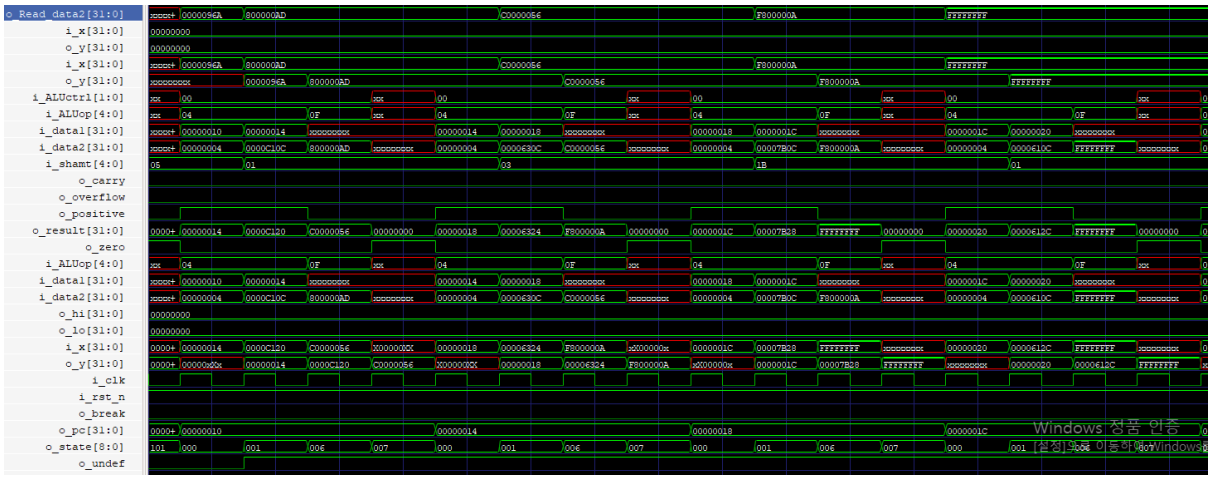
Register 3에 10000000_00000000_00000000_10101101(2)인 값을 넣고 실행한 것으로 R-EXEC에서 i_data2의 값($t)으로 넣어진 것을 확인할 수 있었다. 이는 R-exec의 o_result를 통해 shamt값이 1만큼 오른쪽으로 이동한 0xC0000056=11000000_00000000_00000000_01010110(2)을 확인할 수 있었으며 이는 R-WB의 o_y와 i_Write_reg를 통해 다음 레지스터(0x06)에 해당 값이 잘 들어갔음을 알 수 있었다.

이를 위해 MSB가 0,1일 때 shift 후의 상태를 확인하는 test bench를 준비했다. 이는 아래의 레지스터들이 각각 1,3,27,1만큼 shamt를 갖고 SRA를 실행하게 된다. Sra가 정상적으로 작동한다면 다음과 같이 작동할 것이다.

Register 3: 10000000_00000000_00000000_10101101(2)

Register 5: 00000000_00000000_00001001_01101010(2)

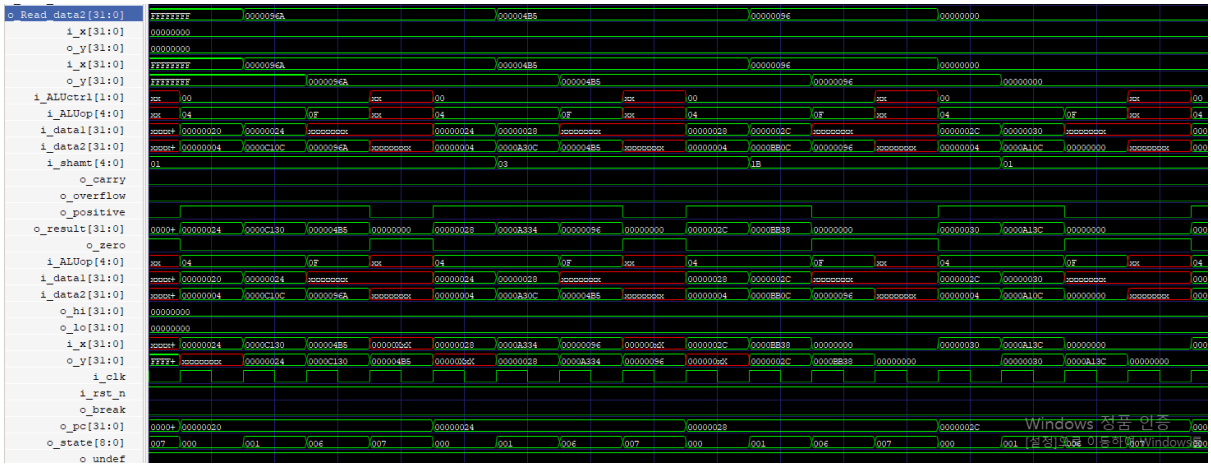
Shamt	Value
0	10000000000000000000000010101101(2)
1	11000000000000000000000001010110(2)
3	11111000000000000000000000001010(2)
27	11111111111111111111111111111111(2)
1	11111111111111111111111111111111(2)
0	00000000_00000000_00001001_01101010(2)
1	000000000_00000000_00001001_0110101(2)
3	000000000000_00000000_00001001_0110(2)
27	00000000000000000000000000000000(2)
1	00000000000000000000000000000000(2)



위는 register3(MSB가 1)가 1,3,27,1만큼 shamt를 갖고 SRA된 것이다. 결과는 예측한 값과 같게

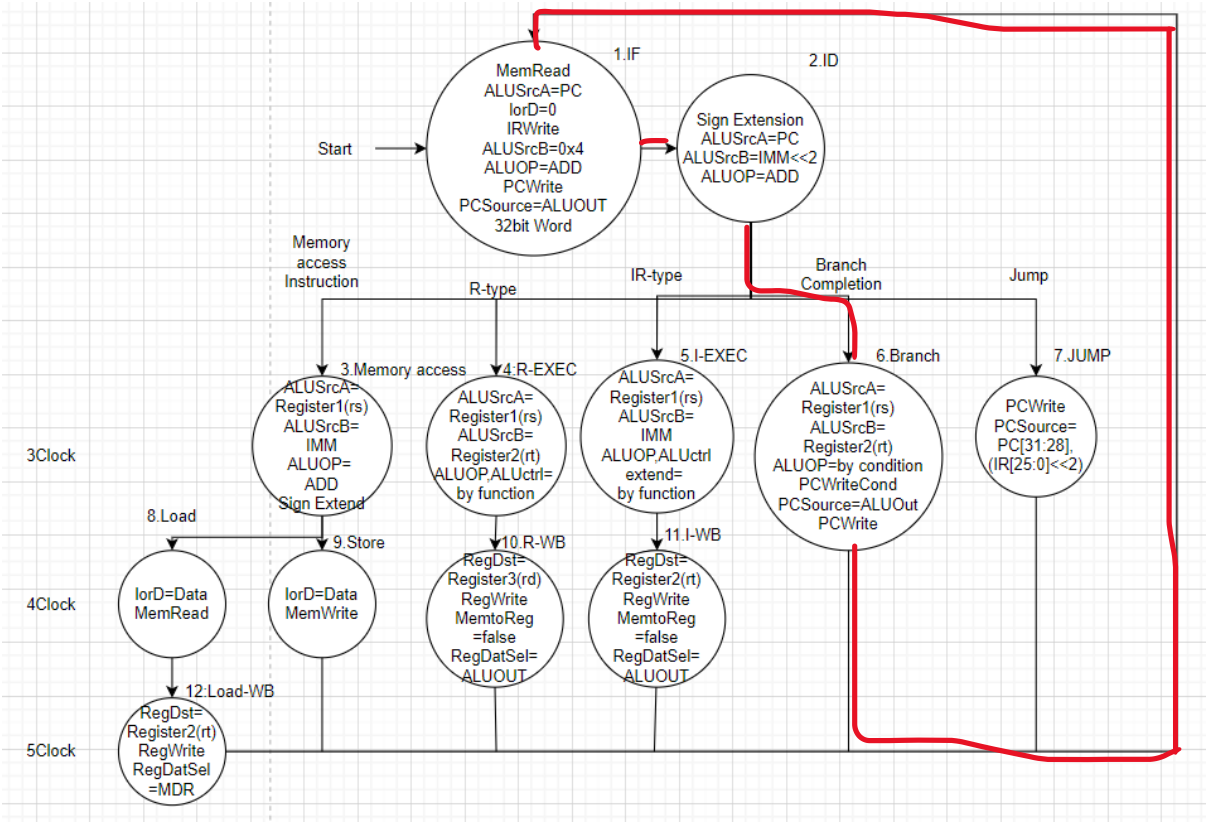
나왔다.

Shamt	Value
0	10000000000000000000000010101101(2)
1	11000000000000000000000001010110(2)
3	11111000000000000000000000001010(2)
27	11111111111111111111111111111111(2)
1	11111111111111111111111111111111(2)



위는 register5(MSB가 0)가 1,3,27,1만큼 shamt를 갖고 SRA된 것이다. 해당 결과 또한 예측한 값과 같게 나왔다.

-Bne



BNE는 Branch 명령어로 IF-ID-Branch이 이루어 진다. 즉, 3 clock을 소비할 것이다.

BNE는 Branch에서 alu를 통해 해당 들어온 값들에 대해 뺄셈을 하고 이를 바탕으로 같은 지 같지 않은 지 판단하여 Branch여부를 정하게 될 것이다. 이를 바탕으로 아래와 같이 작성할 수 있다.

IF, ID는 위에서 확인하였으므로 우리는 Branch에서의 확인만 확인하면 된다.

```
x_x_0_xxx_0_xx_xxx_0_1_000_000_00110_0x_101_01_1_xxxxxxxx_00 // 0x08:  bne branch
```

우선 BNE에 대한 field의 정리이다. Branch에선 기본적으로 ALU에서 뺄셈을 진행하고 이를 바탕으로 Branch 조건이 참인지 거짓인지를 판단하고 참일 경우 ID에서 계산된 ALU register에 저장된 값을 PC값에 넣는 동작이 진행된다.

IorD-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite- Instruction Register에 Write하지 않는다.

RegDst-상관없다.

RegDatSel- Don't care

RegWrite-Register file에 write하지 않는다.

EXTmode-sign Extension을 진행한다.

ALUsrcA- Register A의 값을 ALU input A로 한다.

ALUsrcB- Register B의 값을 ALU input B로 한다.

ALUop-A와 B의 Sub를 진행한다.

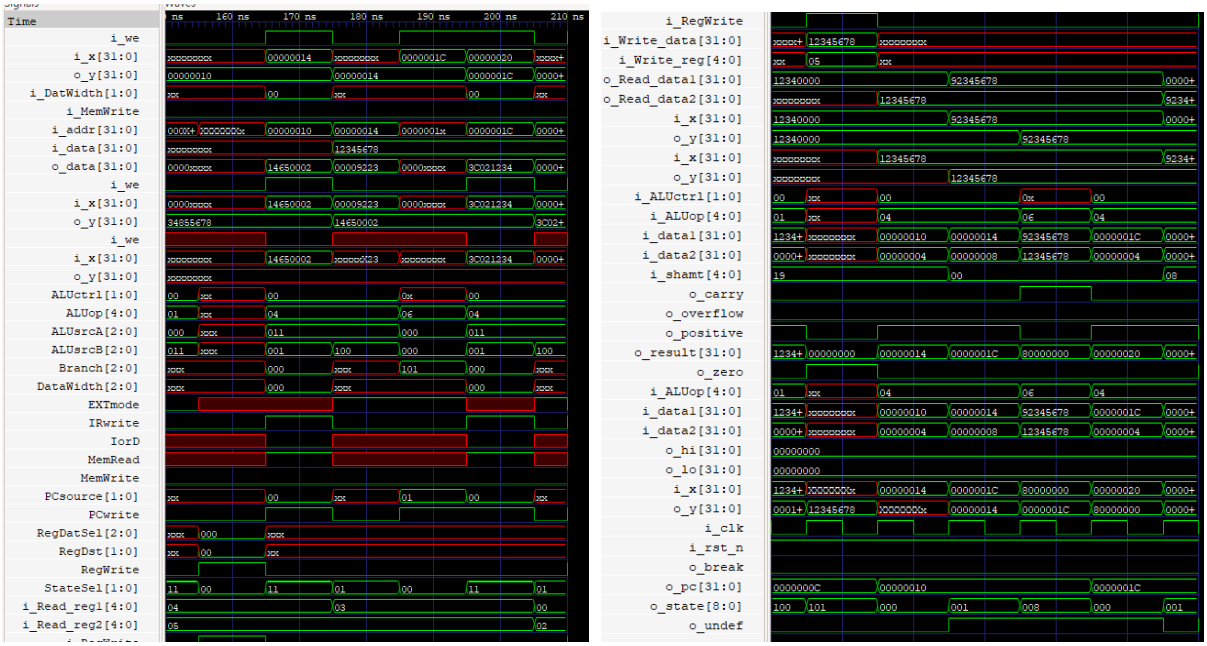
ALUctrl-Normal ALU input, No Shift

Branch-같지 않을 때 Branch

PCsrc-ALUOUT Register로부터 값을 받는다.

PCwrite-PC Register에 Write하게 된다.

StateSel-명령어의 마지막 stage이므로 IF로 돌아가기 위해 Next State =0



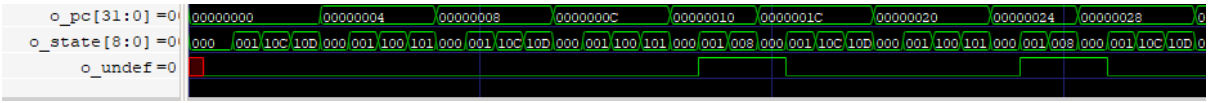
위는 입력된 값들이 다를 때의 경우이다. 3clock의 i_data1,2가 서로 다르단 것을 확인 가능하다. 최종적으로 3clock의 o_y와 i_x와 다음 o_pc가 0x0000001C인 것을 확인하여 다음 branch가 0x0000001C로 변경되었다는 것을 확인할 수 있다.

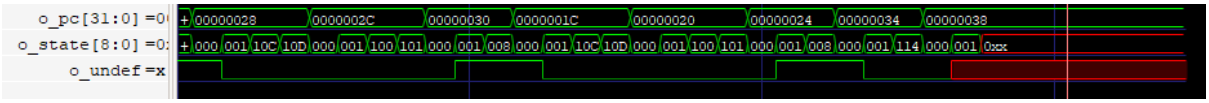
위는 입력된 값들이 같을 때의 경우이다. 3clock의 i_data1,2가 서로 같다는 것을 확인 가능하다. 최종적으로 3clock의 o_y와 i_x를 통해 BNE에서 입력값이 다른 경우인 0x0000001C는 생성했지만, Branch가 101(not equal)이므로 다음 branch가 0x00000028로 변경되었다는 것을 확인할 수 있다.

다양한 경우에서의 테스트를 위해 다음과 같은 테스트 벤치를 준비했다.

```
00111100_00000010_10010010_00110100
00110100_01000011_01010110_01111000
00111100_00000100_00010010_00110100
00110100_10000101_01010110_01111000

000101_00011_00101_00000000_00000010
001111_00000_00100_10010010_00100011
001101_00100_00101_01010110_01111000
001111_00000_00010_00010010_00110100

00110100_01000011_01010110_01111000
000101_00011_00101_00000000_00000011
001111_00000_00100_00010001_00110100
00110100_10000101_01010110_01111000

000101_00011_00101_11111111_11111010
```

예측대로 라면 register3, 5에 0x92345678, 0x12345678을 input으로 받고, 서로 다르므로 현재 위치에서 3(기존에 주어지는 +1포함)만큼 이동한다. 이동한 곳에서 register3의 값이 0x12345678로 변경되고 이들은 같으므로 다음 BNE를 무시한다. 이후 register5의 값을 0x11345678로 변경한다. 이때문에 다음 BNE에서 -5(기존에 주어지는 +1포함)만큼 이동하게 된다. 이후 register3에 0x12345678값을 넣고 BNE에 걸려 +4(기존에 주어지는 +1포함)만큼 이동하게 된다.

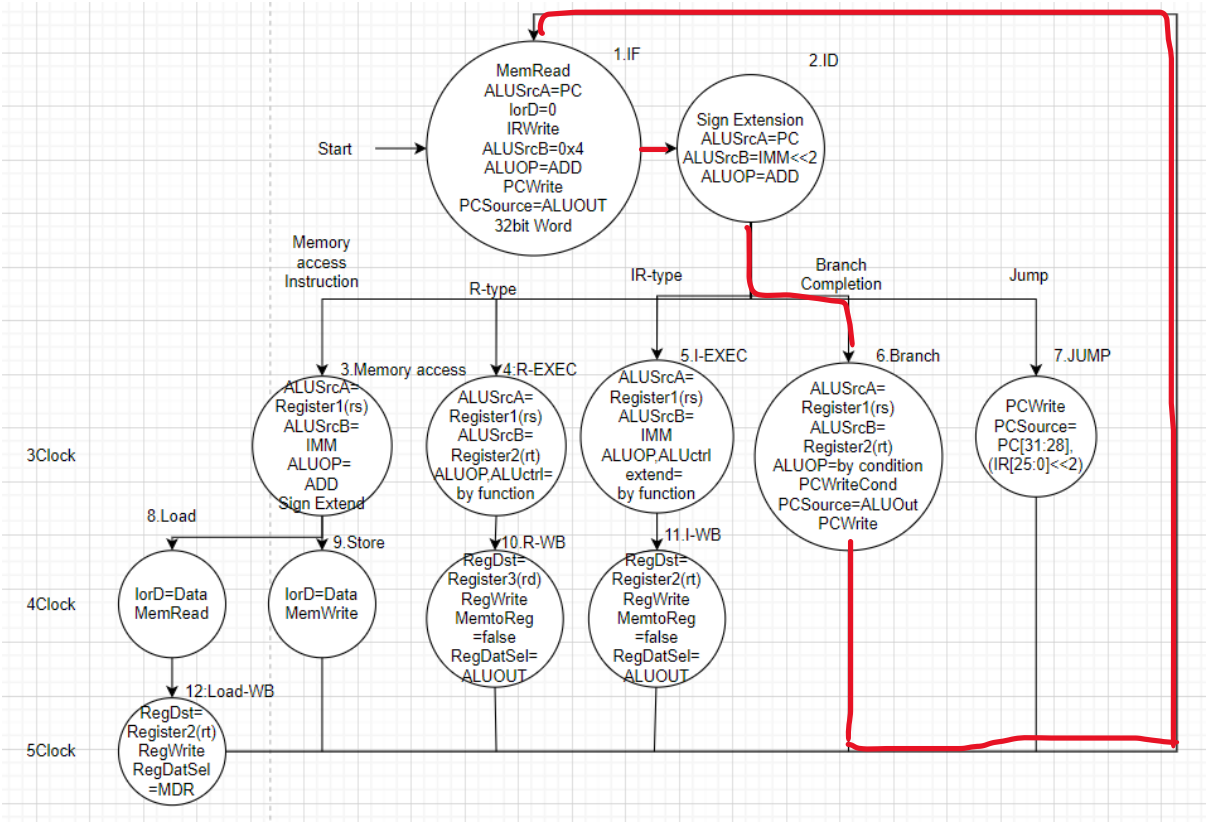




다음은 테스트 벤치를 돌린 결과이다.

위의 값 변화는 예측과 같다는 것을 확인할 수 있다. 실제로 input2개가 다를 때 BNE가 걸린 부분은 입력된 값만큼 이동한 것을 확인할 수 있었다.(0x10->0x1C, 0x30->0x1C,0x24->0x34) 또한 두 input값이 같을 때는 작동하지 않는 것을 확인할 수 있었다.(0x24->0x28)

-blez



BLEZ는 Branch 명령어로 IF-ID-Branch이 이루어 진다. 즉, 3 clock을 소비할 것이다.
BLEZ는 Branch에서 alu를 통해 해당 들어온 값들에 대해 뺄셈을 하고 이를 바탕으로 0이하의 값인지 판단하여 Branch여부를 정하게 될 것이다. 이를 바탕으로 아래와 같이 작성할 수 있다.

IF, ID는 위에서 확인하였으므로 우리는 EXEC에서의 확인만 확인하면 된다.

```
x_x_0_xxx_0_xx_xxx_0_1_000_010_00110_0x_110_01_1_xxxxxxxx_00  // 0x09:  blez branch
```

우선 BLEZ에 대한 field의 정리이다. Branch에선 기본적으로 ALU에서 뺄셈을 진행하고 이를 바탕으로 Branch 조건이 참인지 거짓인지를 판단하여 조건이 참일 경우 ALU register내의 값을 PC로 옮기는 작동을 중점으로 진행된다.

IorD-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite- Instruction Register에 Write하지 않는다.

RegDst-상관없다.

RegDatSel- Don't care

RegWrite- Register file에 write하지 않는다.

EXTmode-sign Extension을 진행한다.

ALUsrcA- Register A의 값을 ALU input A로 한다.

ALUsrcB- 0과의 비교가 진행되므로 0x0의 값을 ALU input B로 한다.

ALUop- A(Register A)-B(0x0)를 진행한다.

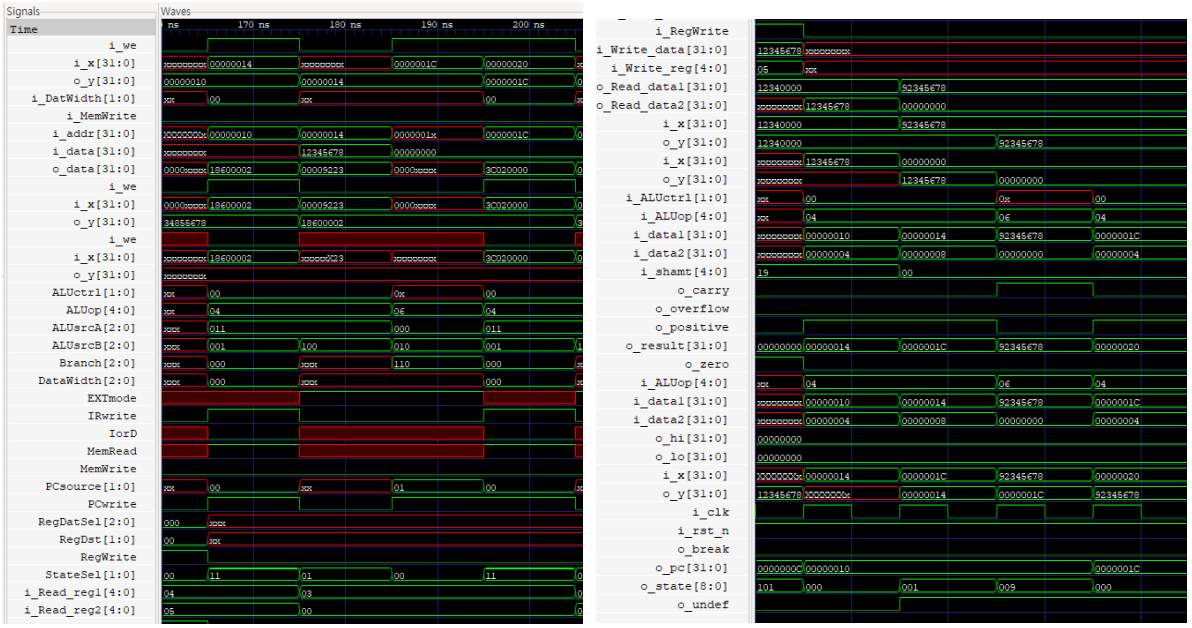
ALUctrl-Normal ALU input, No Shift

Branch-ALU를 통해 뺄셈된 결과가 not positive할때 Branch

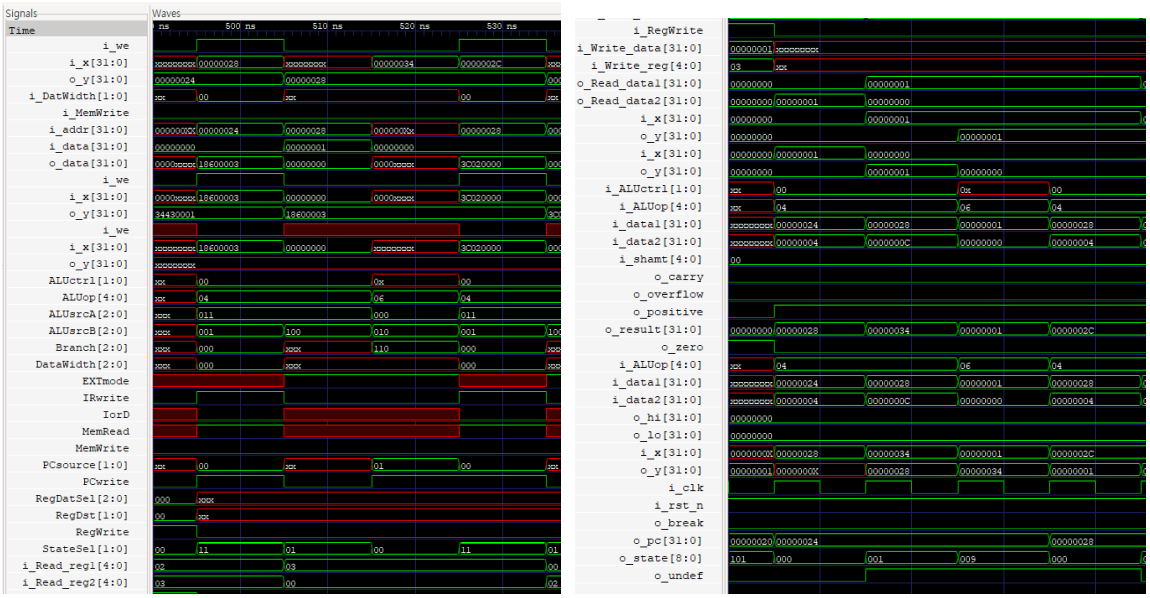
PCsrc-ALUOUT Register로부터 값을 받는다.

PCwrite-PC Register에 Write하게 된다.

StateSel-명령어의 마지막 stage이므로 IF로 돌아가기 위해 Next State =0



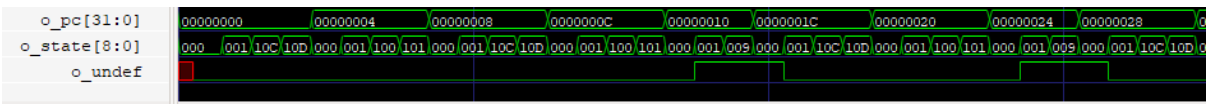
위는 입력된 값이 0보다 작거나 같을 경우이다. 3clock의 i_data1로 ($s)가 들어가며 비교를 위해 i_data2에 0값이 들어가는 것을 확인 가능하다. 이후 o_result의 결과가 음수가 나오며 최종적으로 3clock의 2clock에서 i_data1과 i_data2의 더해진 값인 0x0000001C의 값이 들어간 o_y와 i_x를 바탕으로 다음 o_pc가 0x0000001C인 것을 확인하여 다음 branch가 0x0000001C로 변경되었다는 것을 확인할 수 있다.

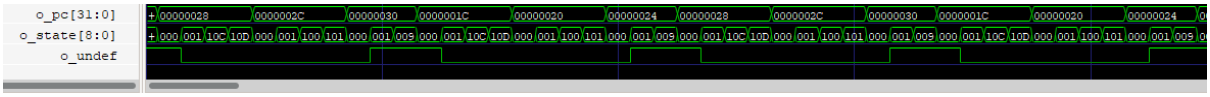


위는 입력된 값이 0보다 클 경우이다. 3clock의 i_data1로 ($s)가 들어가며 비교를 위해 i_data2에 0값이 들어가는 것을 확인 가능하다. 이후 o_result의 결과가 양수가 나오며 최종적으로 3clock의 2clock에서 i_data1과 i_data2의 더해진 값인 0x000020이 있지만 다음 o_pc가 0x0000001C인 것을 확인하여 다음 branch가 0x0000001C로 변경되었다는 것을 확인할 수 있다.

```
00111100_00000010_10010010_00110100
00110100_01000011_01010110_01111000
00111100_00000100_00010010_00110100
00110100_10000101_01010110_01111000

000110_00011_00000_00000000_00000010
001111_00000_00100_10010010_00100011
001101_00100_00101_01010110_01111000
001111_00000_00010_00000000_00000000

001101_00010_00011_00000000_00000001
000110_00011_00000_00000000_00000011
001111_00000_00010_00000000_00000000
001101_00010_00011_00000000_00000000

000110_00011_00000_11111111_11111010
```

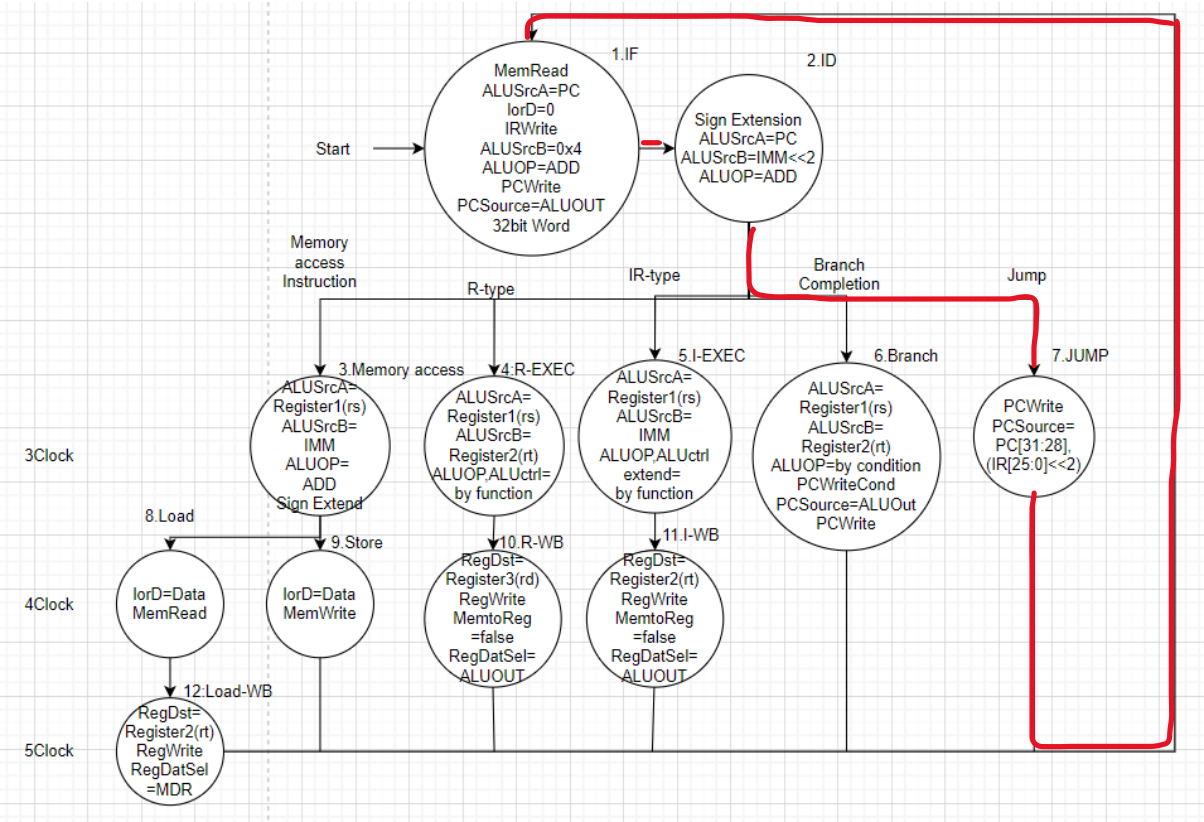
예측대로 라면 register3에 0x92345678을 input으로 받고, register3에 MSB가 1인 음수 값이 들어갔으므로 현재 위치에서 3(기존에 주어지는 +1포함)만큼 이동한다. 이동한 곳에서 register3의 값이 0x00000001로 변경되고 이는 0보다 크므로 다음 BNE를 무시한다. 이후 register3의 값을 0x00000000로 변경한다. 이때문에 다음 BNE에서 -5(기존에 주어지는 +1포함)만큼 이동하게 된다. 이후 register3에 0x92235678값을 넣고 이후 0x00000001값을 넣어준다. 이 때문에 0x30에서 0x1C로 계속해서 무한루프가 발생하게 될 것이다.





해당 테스트벤치를 돌려본 결과는 다음과 같다. 위의 값 변화는 예측과 같다는 것을 확인할 수 있다. 실제로 input의 값이 음수일 때와 0일 때 BLEZ가 걸린 부분은 입력된 값만큼 이동한 것을 확인할 수 있었다.(0x10->0x1C, 0x30->0x1C) 또한 두 input값이 같을 때는 작동하지 않는 것을 확인할 수 있었다.(0x24->0x28) 끝으로 의도된 대로 0x30->0x1C가 계속해 발생하게 되어 무한루프가 생겨남을 확인할 수 있었다.

-jalr



JALR는 JUMP 명령어로 IF-ID-JUMP이 이루어 진다. 즉, 3 clock을 소비할 것이다.
JALR는 JUMP에서 현재 PC값을 $31에 저장하고 통해 해당 들어온 값들에 대해 뺄셈을 하고 rs 레지스터가 가지는 값에 해당하는 주소로 jump하게 될 것이다. 이를 바탕으로 아래와 같이 작성할 수 있다.

IF, ID는 위에서 확인하였으므로 우리는 JUMP에서의 확인만 확인하면 된다.

```
x_x_0_xxx_0_11_100_1_x_000_010_00110_0x_000_00_1_xxxxxxxx_00  // 0x0a: jalr
```

우선 JALR에 대한 field의 정리이다. JUMP에선 기본적으로 register의 경우 현재의 PC값을 받고, ALU에서는 0과의 뺄셈을 통해 Jump할 주소를 유지하고 ALUOUT을 PC로 옮기는 작동을 중점으로 진행된다.

lorD-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite- Instruction Register에 Write하지 않는다.

RegDst-$31에 Write된다.

RegDatSel- PC의 값이 지정된 Register에 Write된다.

RegWrite- Register file에 write한다.

EXTmode- Don't care

ALUsrcA- Register A의 값을 ALU input A로 한다.

ALUsrcB- 0과의 뺄셈이 진행되므로 0x0의 값을 ALU input B로 한다.

ALUop- A(Register A)-B(0x0)를 진행한다.

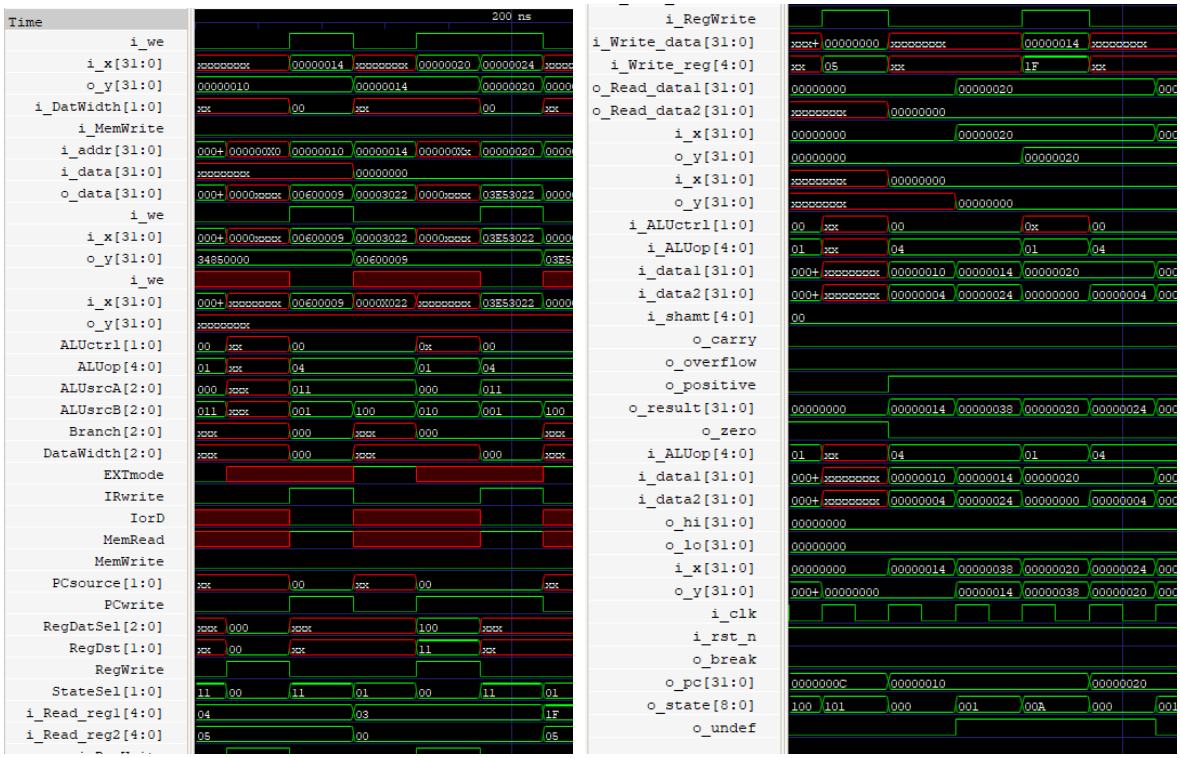
ALUctrl-Normal ALU input, No Shift

Branch-조건 없이 Jump한다.

PCsrc-ALUOUT로부터 값을 받는다.

PCwrite-PC Register에 Write하게 된다.

StateSel-명령어의 마지막 stage이므로 IF로 돌아가기 위해 Next State =0



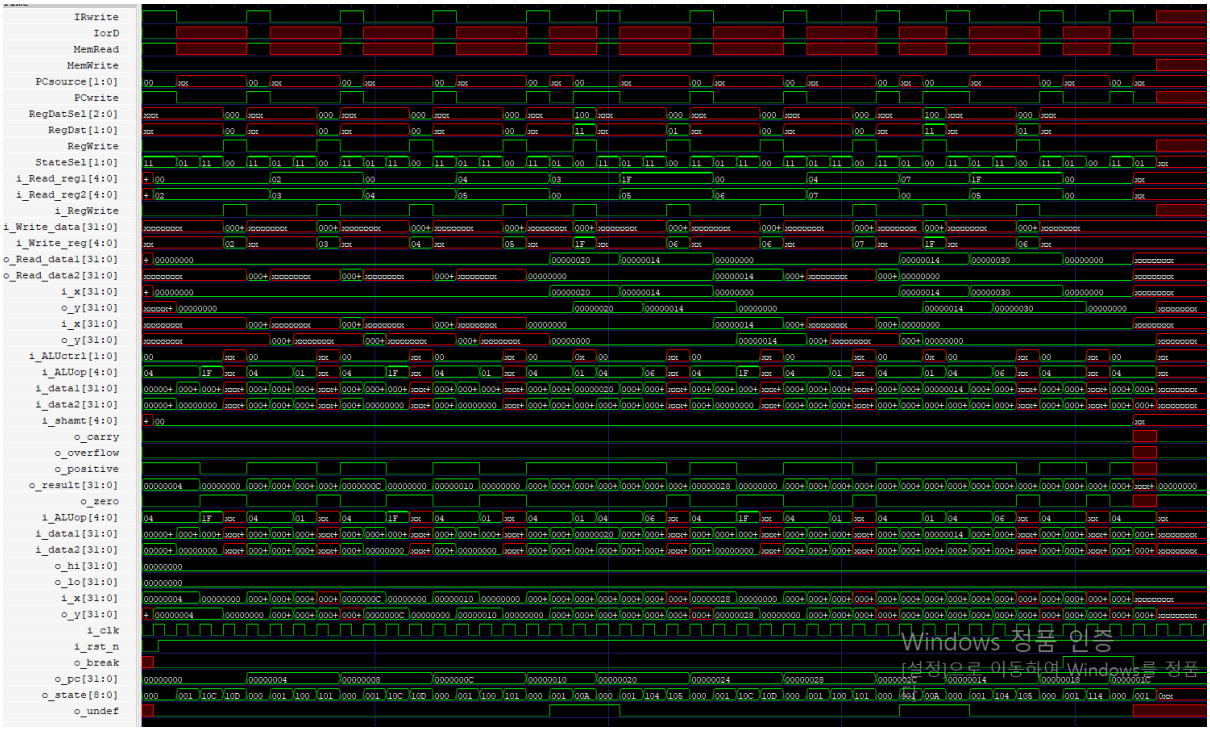
위는 0x00000020을 지닌 register를 $rs로 받은 모습이다. JUMP에서의 o_result결과와 다음 O_pc를 통해 ALUOUT을 통해 잘 JUMP되었음을 확인할 수 있다. 또한 JUMP에서 RegDatSel이 100이고 i_Write_data가 0x00000014이며 RegDst가 11임과 i_Write_reg가 1F(0x00011111=>$31)을 확인하여 통해 PC값을 $31 Register에 넣었음을 확인할 수 있다.

M_TEXT_SEG - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

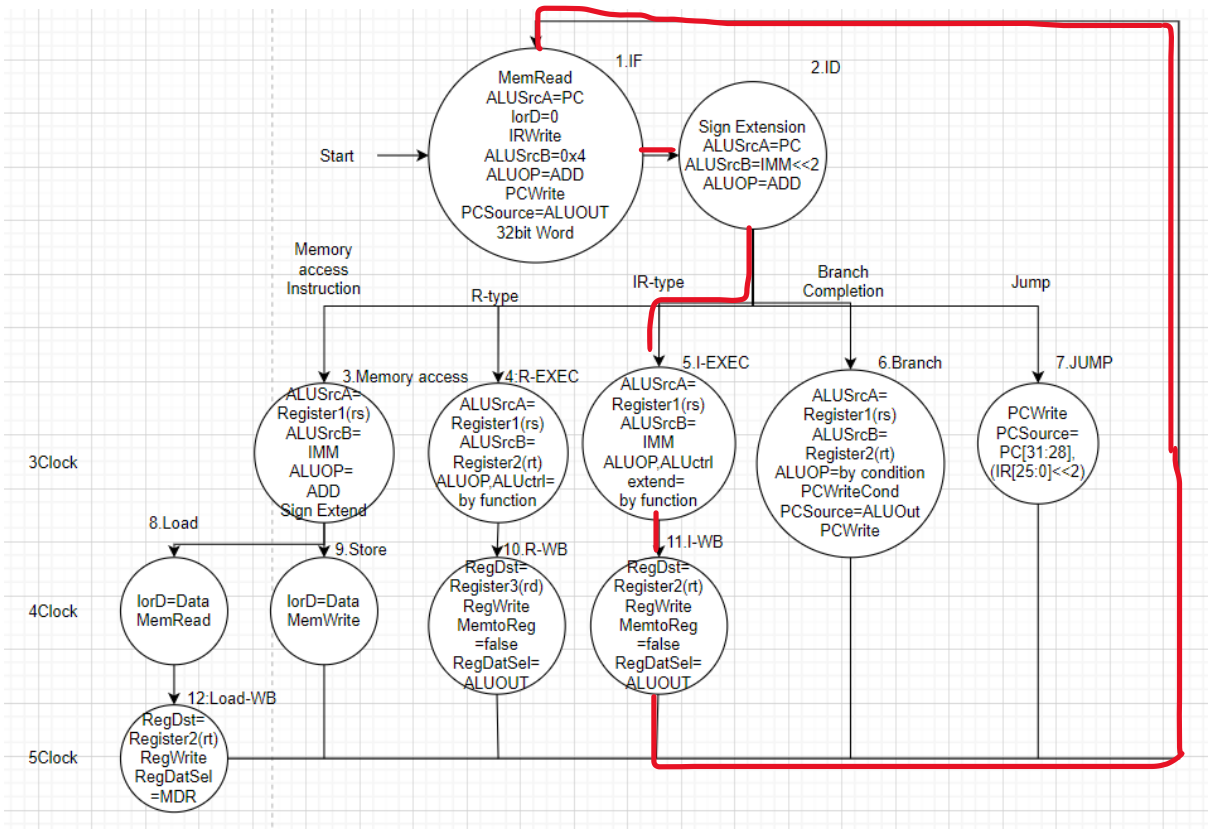
```
00111100_00000010_00000000_00000000
00110100_01000011_00000000_00100000
00111100_00000100_00000000_00000000
00110100_10000101_00000000_00000000

00000000_01100000_00000000_00001001
00000011_11100101_00110000_00100010
00000000_00000000_00000000_00001101          //break;
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx

00000011_11100101_00110000_00100010
```

다음과 같은 테스트벤치를 통해, 추가적인 확인을 진행하였다. 예상대로라면 0x00000010명령어에서 Jalr가 일어나 rs레지스터에 저장된 0x00000020명령어로 jump하게 될 것이다. 0x00000020에서 명령어는 rs레지스터로 0x00011111 레지스터, 즉 직전 JALR을 통해 저장된 0x00000014로 Jump시킬 것이다. 그렇게 0x00000014로 점프한 후 해당 명령어에서 0을 뺀 다음 0x00000018에서 break될 것이다.



위는 해당 테스트 벤치의 결과이다. 예상대로 Jump가 정상적으로 작동하고 있으며 Register에 저장되는 것 또한 잘 작동하고 있음을 확인할 수 있었다.

-Andi



ANDI의 경우 i-type(FSM diagram에서 IR type으로 분류) 명령어로서 IF-ID-(I-EXEC)-(I-WB)로의 4cycle을 가지게 된다. ANDI는 I-Exec에서 alu를 통해 해당 계산을 진행하고 I-WB에서 Register에 값을 저장하게 될 것이다. 이를 바탕으로 아래와 같이 작성할 수 있다.

IF, ID는 위에서 확인하였으므로 우리는 I-EXEC와 I-WB에서의 확인만 확인하면 된다.

```
x_x_0_xxx_0_xx_xxx_0_0_000_011_00000_0x_xxx_xx_0_xxxxxxxx_11  // 0x0b:  andi execute
x_x_0_xxx_0_00_000_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00  // 0x0c:  andi wb
```

우선 I-EXEC에 대한 field의 정리이다. I-EXEC에선 기본적으로 ALU에 register A의 값과 immediate value를 BITWISE AND를 하는 것을 중점으로 진행된다.

IorD-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite- Instruction Register에 Write하지 않는다.

RegDst-상관없다.(이후 I-WB에서 register에 넣어준다.)

RegDatSel- Don't care

RegWrite-Register file에 write하지 않는다.

EXTmode-크기가 다른 경우를 대비해 Zero Extension하게 된다.

ALUsrcA-Register A의 값을 ALU input A로 한다.

ALUsrcB- SEU output을(extend된 immediate value) ALU input B로 한다.

ALUop-Bitwise AND를 진행한다.

ALUctrl-Normal ALU input, No Shift

Branch-Branch나 jump가 발생하지 않는다.

PCsrc-Don't Care

PCwrite-No PC Register Write

StateSel-ID로의 이동을 위해 Next State = CurrentState +1이다.

I-WB의 경우 Register의 값을 저장하기 위해 다음과 같은 진행을 가진다.

lord-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite-Instruction Register에 Write할 수 없다.

RegDst-$rt에 저장된다.

RegDatSel-ALUOUT의 값을 Register file에 적는다.

RegWrite-Register file에 write한다.

EXTmode- Don't care

ALUsrcA- Don't care

ALUsrcB- Don't care

ALUop- Don't care.

ALUctrl- Don't care

Branch-Don't Care

PCsrc- Don't care

PCwrite-PC에 write하지 않는다.

StateSel-명령어의 마지막 stage이므로 IF로 돌아가기 위해 Next State =0

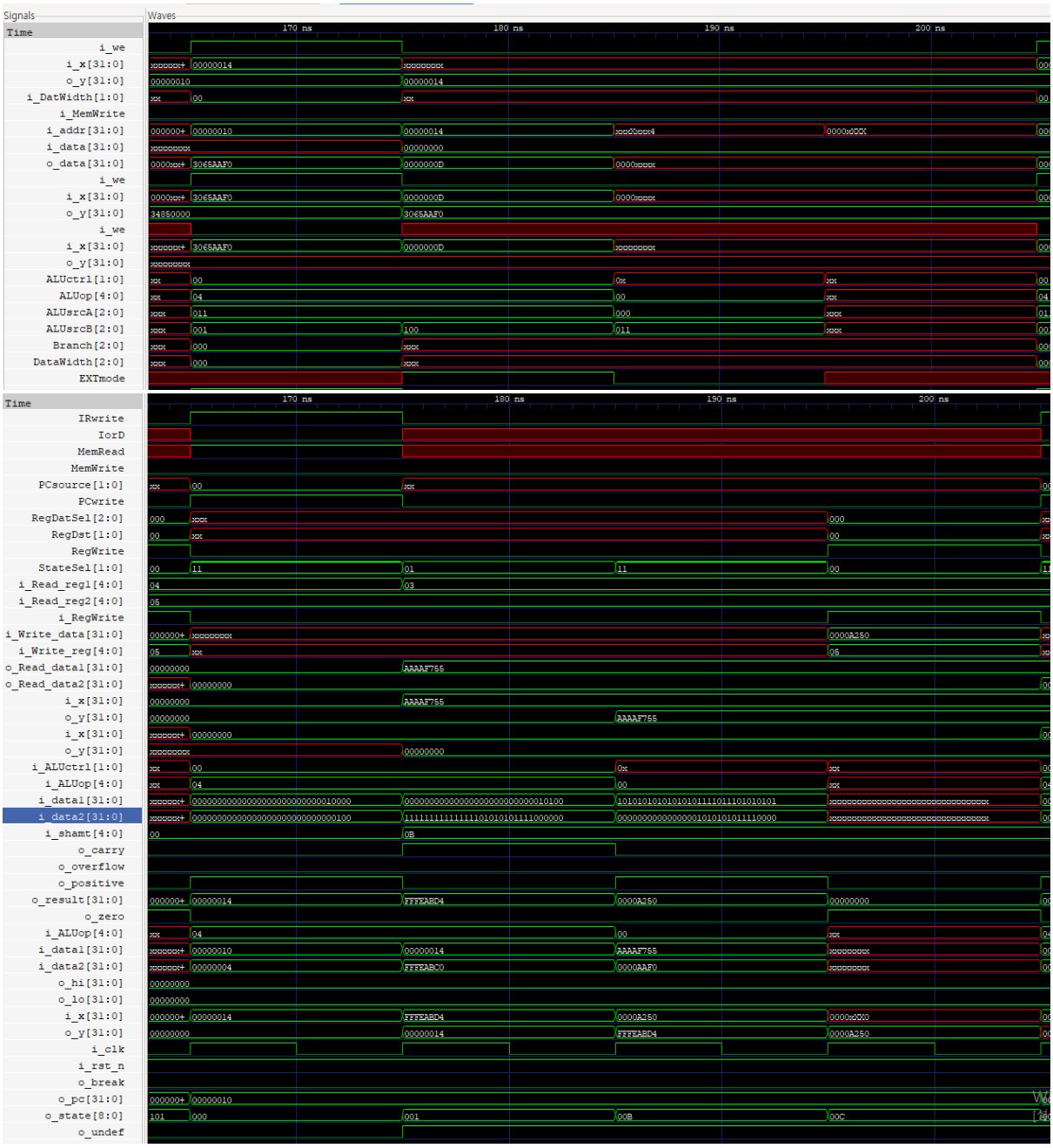
```
00111100_00000010_10101010_10101010
00110100_01000011_11110111_01010101
00111100_00000100_00000000_00000000
00110100_10000101_00000000_00000000

00110000_01100101_10101010_11110000
00000000_00000000_00000000_00001101        //break;
```

Andi를 테스트하기 위해 다음과 같은 값을 주었다.

Register3: 10101010_10101010_11110111_01010101(2)

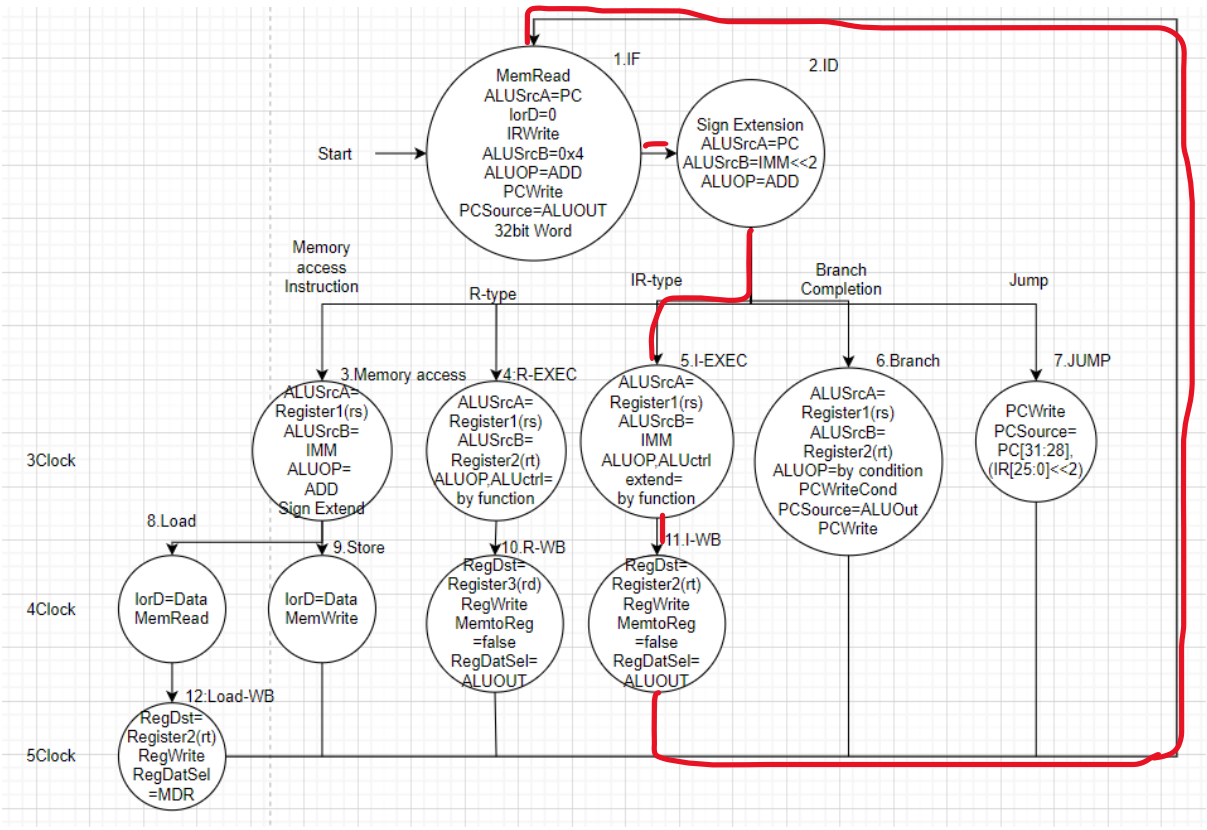
Immediate value:10101010_11110000(2)

Zero extension이므로 immediate value가 00000000_00000000_10101010_11110000으로 판정되어 예상되는 값은 00000000_00000000_10100010_01010000(2)이다.



위는 해당 test bench를 넣어본 결과이다. 예측된 값과 같이 I-EXEC의 i_data2에서 immediate value가00000000_00000000_10101010_11110000으로 판정되어 계산되었다.(zero extension) 이후 해당 결과 값은 o_result를 바탕으로 00000000_00000000_10100010_01010000(2)으로 확인되었으며 I-WB에서의 i_Write_data와 i_Write_reg를 바탕으로 register 5에 이 값이 들어갔음을 확인할 수 있다.

-SItiu



SItiu는 IF-ID-(I-EXEC)-(I-WB)을 거치는 4clock명령어가 될 것이다.

SItiu의 경우 i-type(FSM diagram에서 IR type으로 분류) 명령어로서 IF-ID-(I-EXEC)-(I-WB)로의 4cycle을 가지게 된다. SItiu는 I-Exec에서 alu를 통해 해당 계산을 진행하고 I-WB에서 Register에 값을 저장하게 될 것이다. 이를 바탕으로 아래와 같이 작성할 수 있다.

IF, ID는 위에서 확인하였으므로 우리는 I-EXEC와 I-WB에서의 확인만 확인하면 된다.

```
x_x_0_xxx_0_xx_xxx_0_0_000_011_10001_0x_xxx_xx_0_xxxxxxxx_11  // 0x0d:  sltiu execute
x_x_0_xxx_0_00_000_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00  // 0x0e:  sltiu wb
```

우선 I-EXEC에 대한 field의 정리이다. I-EXEC에선 기본적으로 ALU에 register A의 값과 immediate value를 Unsigned SLT를 하는 것을 중점으로 진행된다.

IorD-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite- Instruction Register에 Write하지 않는다.

RegDst-상관없다.(이후 I-WB에서 register에 넣어준다.)

RegDatSel- Don't care

RegWrite-Register file에 write하지 않는다.

EXTmode-unsigned에 대한 비교이므로 Zero Extension하게 된다.

ALUsrcA-Register A의 값을 ALU input A로 한다.

ALUsrcB- SEU output을(extend된 immediate value) ALU input B로 한다.

ALUop-Unsigned SLT를 진행한다.

ALUctrl-Normal ALU input, No Shift

Branch-Branch나 jump가 발생하지 않는다.

PCsrc-Don't Care

PCwrite-No PC Register Write

StateSel-ID로의 이동을 위해 Next State = CurrentState +1이다.

I-WB의 경우 Register의 값을 저장하기 위해 다음과 같은 진행을 가진다.

lord-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite-Instruction Register에 Write할 수 없다.

RegDst-$rt에 저장된다.

RegDatSel-ALUOUT의 값을 Register file에 적는다.

RegWrite-Register file에 write한다.

EXTmode- Don't care

ALUsrcA- Don't care

ALUsrcB- Don't care

ALUop- Don't care.

ALUctrl- Don't care

Branch-Don't Care

PCsrc- Don't care

PCwrite-PC에 write하지 않는다.

StateSel-명령어의 마지막 stage이므로 IF로 돌아가기 위해 Next State =0

```
00111100_00000010_00000000_00000000
00110100_01000011_01110111_01010101
00111100_00000100_00000000_00000000
00110100_10000101_00000000_00000000

00111100_00000110_10101010_10101010
00110100_11000111_11110111_01010101
001011_00011_01001_11111111_11111111
100011_01001_01011_0000000000000000

001011_00011_01001_01110111_01010101
100011_01001_01011_0000000000000000
001011_00011_01001_00000000_00000000
100011_01001_01011_0000000000000000

001011_00111_01001_00000000_00000000
100011_01001_01011_0000000000000000
001011_00111_01001_11110111_01010101
100011_01001_01011_0000000000000000

001011_00101_01001_10000000_00000111
100011_01001_01011_0000000000000000
00000000_00000000_00000000_00001101        //break;
```

테스트 벤치는 register값의 MSB가 0일경우 Immediate값이 더 클 경우, 0일 경우, 더 작을 경우와 register값의 MSB가 1일 경우에서 Immediate값이 0일 경우, MSB가 1일 경우, register값이 0, immediate값이 양수일 경우를 비교해 보았다. 해당 테스트를 위해 register들에는 다음과 같은 값들을 넣었다.

Register3:0x00007755 / Register5:0x00000000 / Register7:0xAAAAF755

Register3에 대해 0xFFFF, 0x7755, 0x0000를, Register7에 대해 0x0000,0xF755, Register5에 대해 0x8007을 대입해 주었다. 이 때 해당 명령어는 unsigned에서 작동하기 때문에 0x00000001, 0x00000000,0x00000000,0x00000000,0x00000000,0x00000001 순으로 register에 저장될 것이다. 이 때 저장되는 register는 9일 것이다.

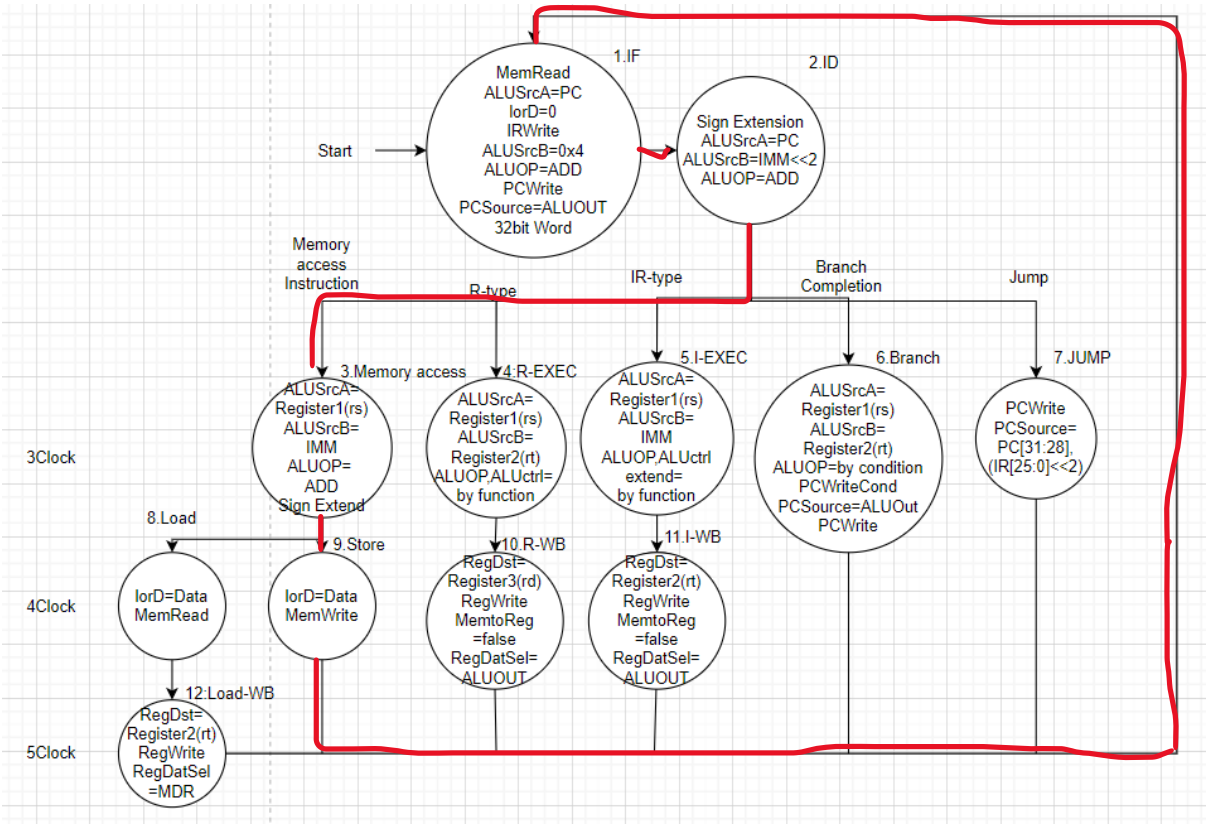
(register3(0x00007755)와 0xFFFF, 0x7755과의 비교)

(register3(0x00007755)와 0x0000, Register7(0xAAAAF755)와 0x0000)

(Register7(0xAAAAF755)와 0xF755, Register5(0x00000000)와 0x8007 비교)

위 testbench로 결과를 본 결과 다음과 같이 나오게 되었다. 예측된 대로 sltiu에서의 모든 4clock에서의 i_Write_reg가 0x09임을 통해 모든 결과가 register 9에 적히게 되었음을 확인할 수 있었다. 또한 모든 3clock에서 ALU에 i_data1, 2를 통해 input이 잘 들어갔음을 확인할 수 있었고, o_result와 4clock의 i_Write_data를 통해, 0x00000001, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000001 순으로 저장된 것을 확인할 수 있었다.

-sb, lbu



SB의 경우 MemoryAccess 명령어 중 Store에 해당한다. 때문에 IF-ID-Memory access-Store로의 4 clock을 사용하게 된다. SB는 Memory access에서 alu를 통해 접근할 Memory의 주소를 계산하여 진행하고 Store에서 Memory에 값을 저장하게 될 것이다. 이를 바탕으로 아래와 같이 작성할 수 있다.

```
x_x_0_xxx_0_xx_xxx_0_1_000_011_00100_0x_xxx_xx_0_xxxxxxxx_11  // 0x0f:  sb execute
1_0_1_111_0_xx_xxx_0_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00  // 0x10:  sb mem
```

IF, ID는 위에서 확인하였으므로 우리는 Memory access와 Store에서의 확인만 확인하면 된다.

우선 Memory access에 대한 field의 정리이다. Memory access에선 기본적으로 ALU에 register A의 값과 immediate value를 ADD를 하여 Memory의 주소를 구하는 것을 중점으로 진행된다.

IorD-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite- Instruction Register에 Write하지 않는다.

RegDst-상관없다.

RegDatSel- Don't care

RegWrite-Register file에 write하지 않는다.

EXTmode-sign Extension을 진행한다.

ALUsrcA- Register A의 값을 ALU input A로 한다.

ALUsrcB- SEU(extend된 immediate value)을 ALU input B로 한다.

ALUop-A와 B의 ADD를 진행한다.

ALUctrl-Normal ALU input, No Shift

Branch-Branch나 Jump가 일어나지 않는다.=>Don't care

PCsrc-Don't care

PCwrite-PC Register에 Write하지 않는다.

StateSel-다음 State로 가기위해 Current State+1

Store의 경우 Memory access에서 구해진 ALUout의 값을 바탕으로 Memory에 접근하고 해당주소에 Sign 8bit형태로 저장하기 위해 다음과 같은 진행을 가진다.

lord-Data로서의 Memory Access

MemRead- Memory에서 Read하지 않는다.

MemWrite-Memory에 Write한다.

DatWidth- 8bit Byte / Sign Ext 로서 접근된다.

IRwrite-Instruction Register에 Write하지 않는다.

RegDst-Don't Care

RegDatSel- Don't Care

RegWrite-Register file에 write하지 않는다.

EXTmode- Don't care

ALUsrcA- Don't care

ALUsrcB- Don't care

ALUop- Don't care.

ALUctrl- Don't care

Branch-Don't Care

PCsrc- Don't care

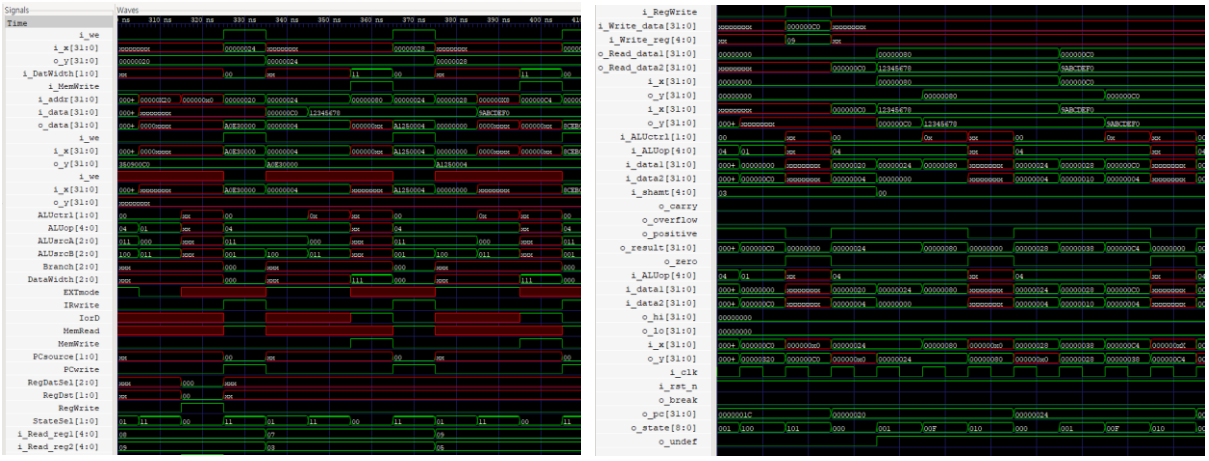
PCwrite-PC에 write하지 않는다.

StateSel-명령어의 마지막 stage이므로 IF로 돌아가기 위해 Next State =0

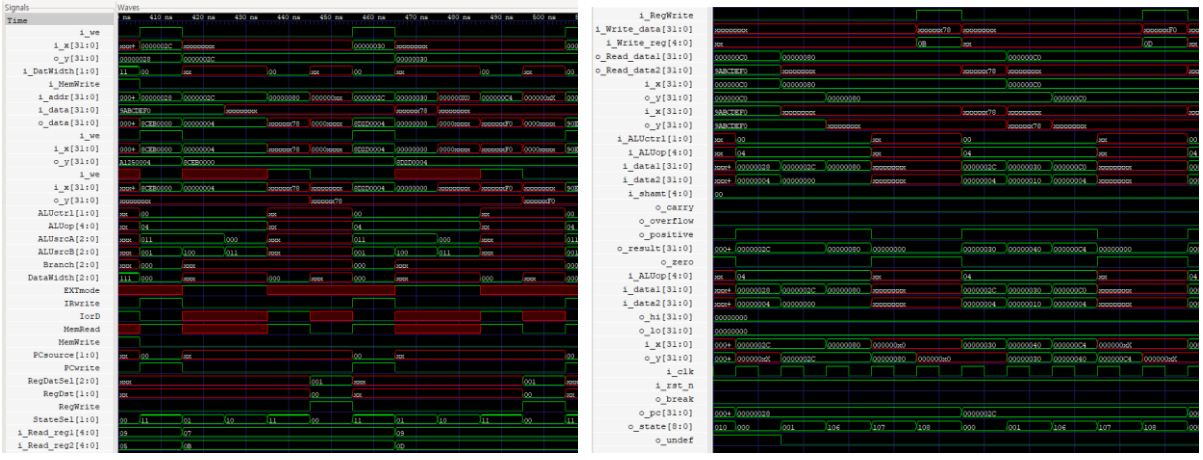
```
00111100_00000010_00010010_00110100
00110100_01000011_01010110_01111000
00111100_00000100_10011010_10111100
00110100_10000101_11011110_11110000

00111100_00000110_00000000_00000000
00110100_11000111_00000000_10000000
00111100_00001000_00000000_00000000
00110101_00001001_00000000_11000000

101000_00111_00011_0000000000000000
101000_01001_00101_0000000000000100
100011_00111_01011_00000000_00000000
100011_01001_01101_00000000_00000100

00000000_00000000_00000000_00001101        //break;
```

SB를 테스트하기 위해 다음과 같은 테스트를 진행하였다.

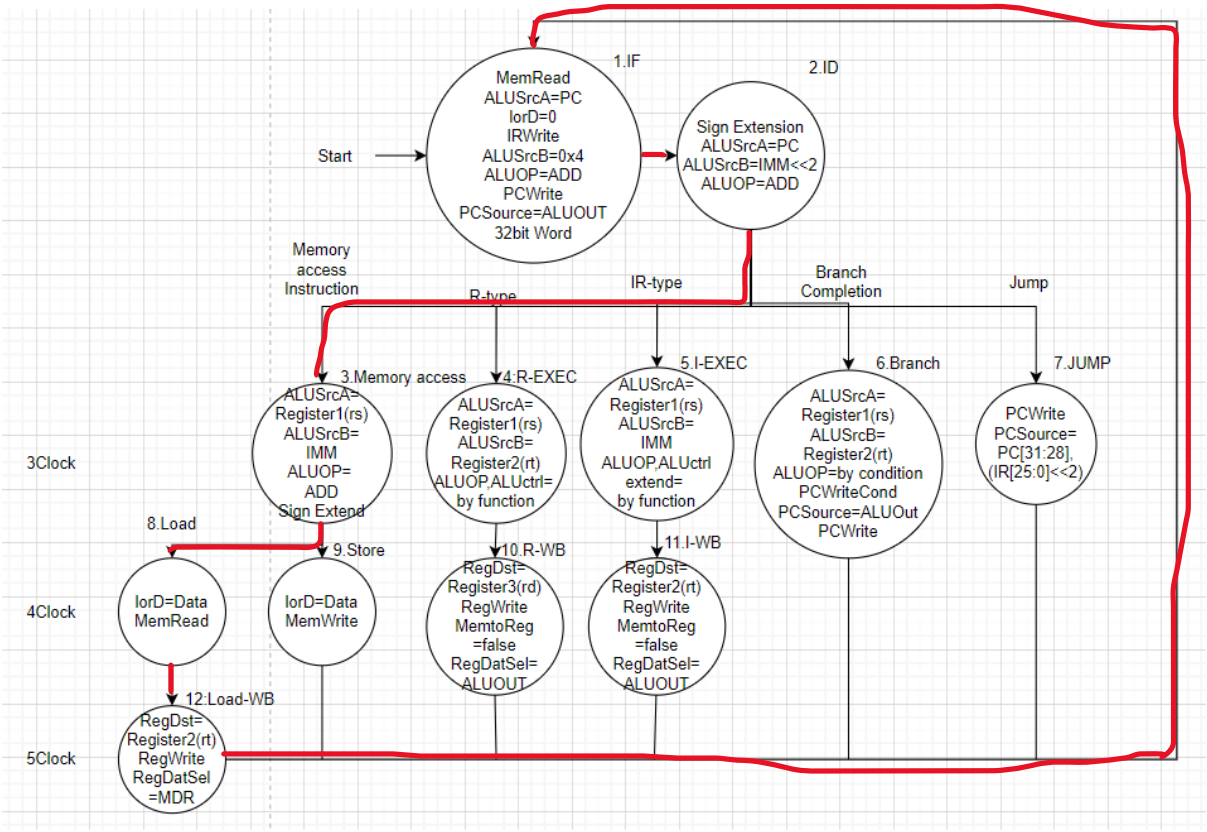
해당 명령어는 Memory의 지정된 위치에 $rs에 있는 값을 8bit 값으로 저장하는 역할을 한다. 이를 위해 Register3=0x12345678 Register4=0x9ABCDEF0을 주고 해당 값들을 MEM($s+i):1에 sb를 통해 저장하게 해주었다. 이때 예측된 값은 little endian이기 때문에 하위 8bit만을 남긴 0xxxxxxx78, 0xxxxxxxF0이 받아질 것이라 예측했다. (해당 프로그램은 없는 값을 x로 감지해주기 때문에 lw로 받으면 하위 8bit는 받아져 보이지만 나머지 memory는 비어서 x로 출력될 것이다.)이는 돌아온 값은 sub를 통해 i_data1를 확인했다. (해당 SB는 하위 8bit를 저장한다.)



위는 SB 명령어가 작동하는 모습이다. Memory access에서 i_data1,2를 통해 레지스터($rs) 내의 값과 immediate value가 더해지고 있음을 확인할 수 있으며, 이 결과는 i_addr로서 Store에서 작동함을 확인할 수 있다. 또한 Store에서의 i_data가 register($rt)내의 값이 들어가는 것 또한 확인할 수 있다. DataWidth 또한 맞게 들어가고 있다.



위는 LW를 통해 해당 위치의 값을 받아본 것이다. LW의 결과로 0xxxxxxx78과 0xxxxxxxF0가 나오는 것을 확인할 수 있는데,(4clock의 i_Write_data) 이는 예상대로 SB를 통해 하위 8bit만이 메모리에 저장되었다는 것이다.



LBU의 경우 Memory access를 값을 불러오기 위해 하는 명령어이므로 IF-ID-Memory access-Load-(Load-WB)의 과정을 거쳐 5 clock을 소비하게 될 것이다.

LBU는 Memory access에서 ALU를 통해 접근할 Memory의 주소를 계산하여 진행하고 Load에서 해당주소의Memory에 접근하여 값을 Load하게 될 것이다. 또한 이렇게 받아온 값을 Load-WB에서 Register에 값을 저장해줄 것이다. 이를 바탕으로 아래와 같이 작성할 수 있다.

```
x_x_0_xxx_0_xx_xxx_0_1_000_011_00100_0x_xxx_xx_0_xxxxxxxx_11  // 0x11:  lbu execute
1_1_0_011_0_xx_xxx_0_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_11  // 0x12:  lbu mem
x_x_0_xxx_0_00_001_1_0_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00  // 0x13:  lbu wb
```

IF, ID는 위에서 확인하였으므로 우리는 Memory access와 Load, Load-WB에서의 확인만 진행하면 된다.

우선 Memory access에 대한 field의 정리이다. Memory access에선 기본적으로 ALU에 register A의 값과 immediate value를 ADD를 하여 Memory의 주소를 구하는 것을 중점으로 진행된다.

IorD-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite- Instruction Register에 Write하지 않는다.

RegDst-상관없다.

RegDatSel- Don't care

RegWrite-Register file에 write하지 않는다.

EXTmode-sign Extension을 진행한다.

ALUsrcA- Register A의 값을 ALU input A로 한다.

ALUsrcB- SEU(extend된 immediate value)을 ALU input B로 한다.

ALUop-A와 B의 ADD를 진행한다.

ALUctrl-Normal ALU input, No Shift

Branch-Branch나 Jump가 일어나지 않는다.=>Don't care

PCsrc-Don't care

PCwrite-PC Register에 Write하지 않는다.

StateSel-다음 State로 가기 위해 Current State+1

Load의 경우 EXEC에서 구해진 ALUout의 값을 바탕으로 Memory에 접근하고 해당주소에 Sign 8bit형태로 저장하기 위해 다음과 같은 진행을 가진다.

lord-Data로서의 Memory Access

MemRead- Memory에서 Read한다.

MemWrite-Memory에 Write하지 않는다.

DatWidth- 8bit Byte(Unsigned) 로서 접근된다.

IRwrite-Instruction Register에 Write하지 않는다.

RegDst-Don't Care

RegDatSel- Don't Care

RegWrite-Register file에 write하지 않는다.

EXTmode- Don't care

ALUsrcA- Don't care

ALUsrcB- Don't care

ALUop- Don't care.

ALUctrl- Don't care

Branch-Don't Care

PCsrc- Don't care

PCwrite-PC에 write하지 않는다.

StateSel-다음 State로 가기위해 Current State+1

Load-WB에서는 Memory로부터 받아온 값을 레지스터($rt)에 저장해야 하므로 다음과 같이 정의될 것이다.

lord-Don't care

MemRead- Don't care

MemWrite-Memory에 Write하지 않는다.

DatWidth- Don't care

IRwrite-Instruction Register에 Write할 수 없다.

RegDst-$rt에 저장된다.

RegDatSel-MDR의 값을 Register file에 적는다.

RegWrite-Register file에 write한다.

EXTmode- Unsigned하기 때문에 Zero Extension을 한다.

ALUsrcA- Don't care

ALUsrcB- Don't care

ALUop- Don't care.

ALUctrl- Don't care

Branch-Don't Care

PCsrc- Don't care

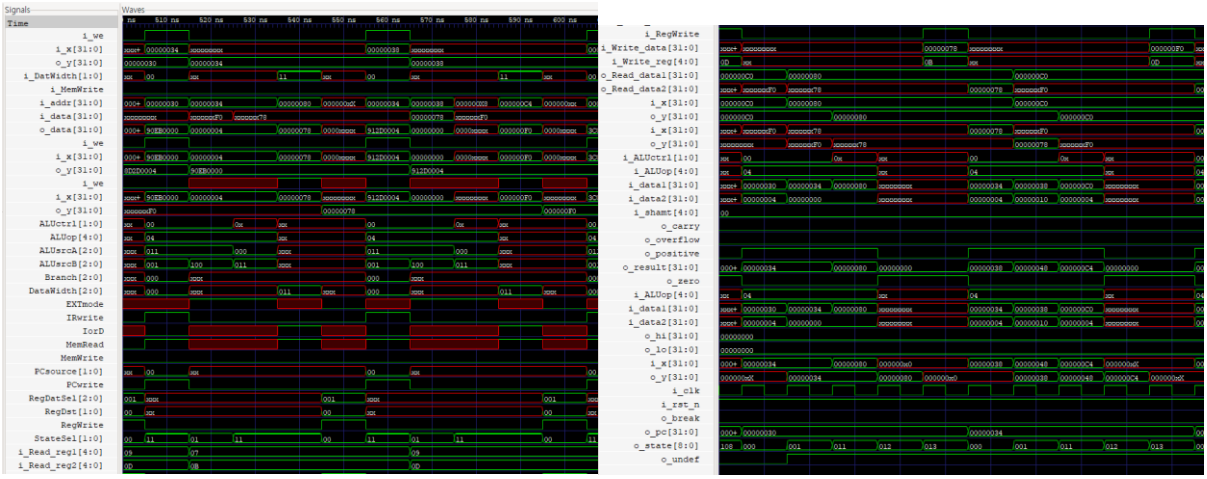
PCwrite-PC에 write하지 않는다.

StateSel-명령어의 마지막 stage이므로 IF로 돌아가기 위해 Next State =0

LBU를 테스트하기 위해 다음과 같은 테스트를 진행하였다.

```
00111100_00000010_00010010_00110100
00110100_01000011_01010110_01111000
00111100_00000100_10011010_10111100
00110100_10000101_11011110_11110000

00111100_00000110_00000000_00000000
00110100_11000111_00000000_10000000
00111100_00001000_00000000_00000000
00110101_00001001_00000000_11000000

101000_00111_00011_0000000000000000
101000_01001_00101_0000000000000100
100100_00111_01011_00000000_00000000
100100_01001_01101_00000000_00000100

00000000_00000000_00000000_00001101        //break;
```

해당 명령어는 Memory의 지정된 위치에 $rs에 있는 값을 8bit 값으로 저장하는 역할을 한다. 이를 위해 Register3=0x12345678 Register4=0x9ABCDEF0을 주고 해당 값들을 MEM($s+i):1에 sb를 통해 저장하게 해주었다. 이때 값은 little endian이기 때문에 하위 8bit만을 남긴 0xxxxxxx78, 0xxxxxxxF0이 저장된다. 것이다. LBU는 해당 위치에 값을 읽어 Unsigned의 형태, 즉 zero extend를 진행한 결과인 0x00000078과 0x000000F0의 값을 레지스터(0x1011,0x1101)에 저장할 것이라 예측했다.



위는 LBU로 출력하는 명령어를 추가하여 출력한 것이다.

총 5clock으로 작동하는 것을 확인할 수 있으며 Memory access에서 i_data1,2를 통해 레지스터 값과 immediate value가 더해져 memory access(o_result)에서 쓰일 주소 값을 만들고 있음을 확인할 수 있다. 이후 Load에서 해당 값을 바탕으로 Memory에 접근하여 O_Read_data2(레지스터와 규격이 맞지 않아 x가 포함되어 있다.)의 값으로 받음을 확인할 수 있다. 이후 Load-WB에서 EXTmode가 0이고, 이를 바탕으로 i_Write_data에는 zero extension된 O_Read_data2의 값이, i_Write_reg에는 저장이 되는 Register가 기재되어 있음을 확인할 수 있었다. 예측한대로 해당 값들은 0x00000078, 0x000000F0로 각각 0x1011, 0x1101에 저장되었다.

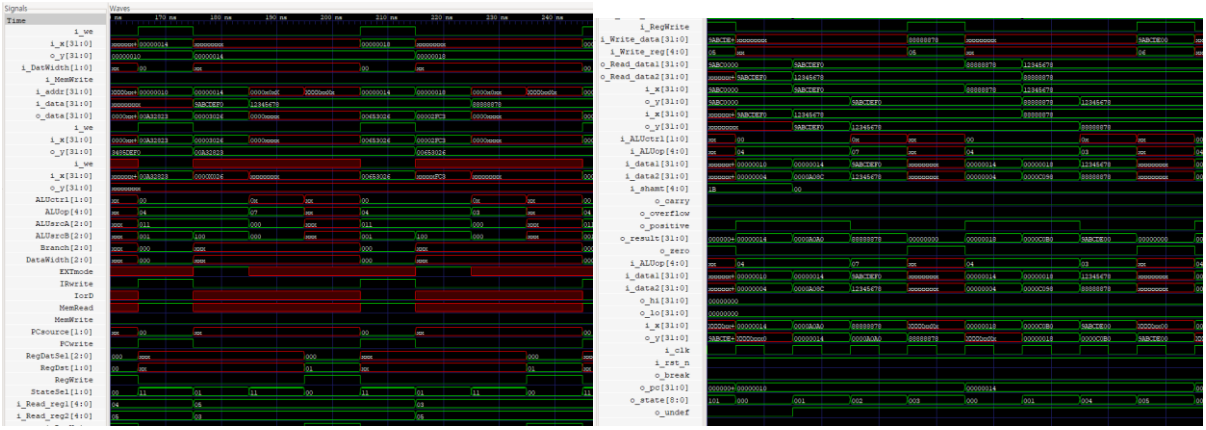
-all

M_TEXT_SEG - Windows 메모장

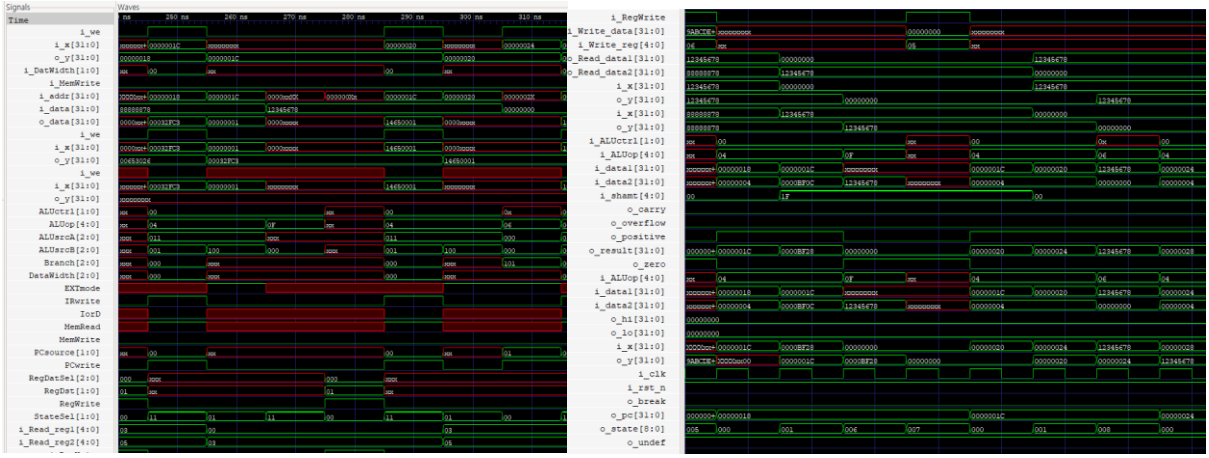
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
00111100_00000010_00010010_00110100
00110100_01000011_01010110_01111000
00111100_00000100_10011010_10111100
00110100_10000101_11011110_11110000

000000_00101_00011_00101_00000_100011      //sub r5-r3              result:r5=0x88888878
000000_00011_00101_00110_00000100110       //xor r3,r5              result:r6=0x9ABCDE00
000000_00000_00011_00101_11111_000011      //sra  r3 >>31 =>r5  result:r5=0x00000000
000101_00011_00101_00000000_00000001       //bne r3!=r5             result:next pc=0x00000024

xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
000110_00101_00000_00000000_00000001       //blez r5<=0         result:next pc=0x0000002C
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
001100_00011_00101_01000010_01111000       //andi r3,imme       result:e5=0x00004278

101000_00011_00101_0000000000001000        //sb r5 in memory[r3]
100100_00011_00101_0000000000001000        //lbu r5 from memory[r3]         result:r5=0x00000078
001011_00011_01001_00000000_11111111       //sltiu if r3<0x00FF put 1 to r9  result:r9=0x00000000
000000_01001_00000_00000000_00001001       //jalr //save pc and jump to r9  result:next pc=0x00000000

00000000_00000000_00000000_00001101        //break;
```

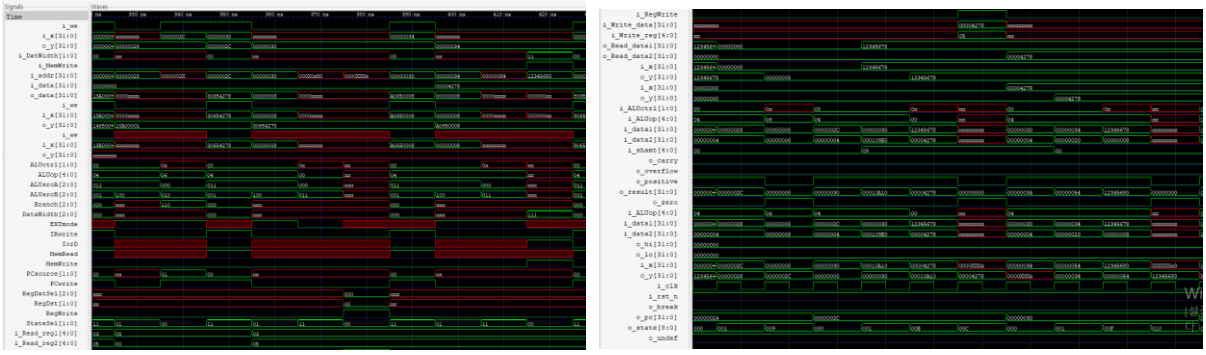
모든 명령어가 한번 이상씩 사용되는 다음과 같은 테스트 벤치를 사용해서 다양한 명령어들이 동시에 사용되었을 때 문제가 없는지 확인했다.



다음은 SUBU와 XOR를 사용한 결과이다. 위에서 예상한 대로 R-WB의 값들을 통해 각각 0x88888878(i_Write_data) 값이 register5(i_Write_reg)에, 0x9ABCDE00(i_Write data) 값이 register6 (i_Write_reg)에 대입됨을 확인할 수 있다.



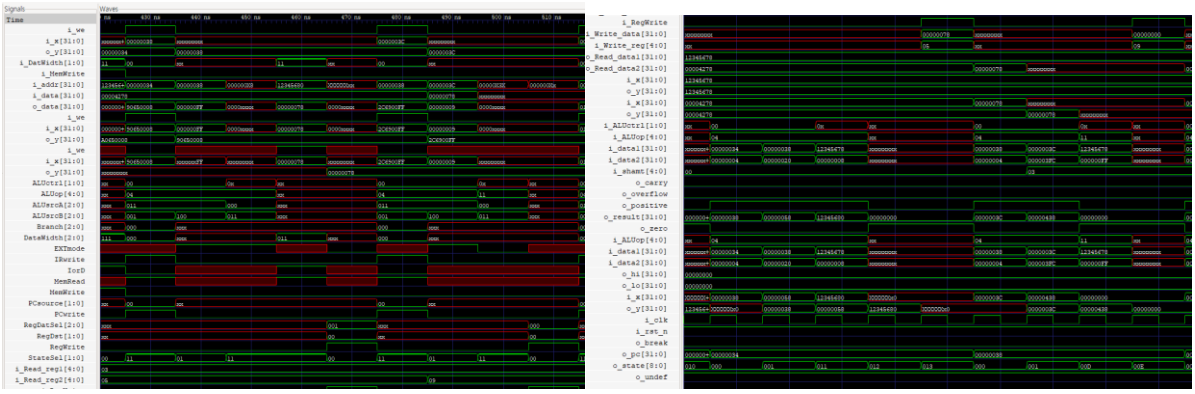
다음은 SRA와 BNE의 결과이다. SRA의 결과로 들어왔던 3clock에서 r3(i_Read_reg2)의 값이 shamt(32)만큼 shift되어 o_result값이 되었다.(0x00000000) 이후 이 값이 r5(i_Write_reg)에 0x00000000(i_Write_data) 값으로 저장되는 것을 확인할 수 있다. 또한 r3와 r5 값을 받아 BNE의 3clock에서 비교하고(2-3clock의 i_Read_reg1,2) 이 값들을 ALU에 넣고(3clock i_data1,2) 이를 통해 구해진 값을 이용하여(3clock의 o_result) 둘이 같지 않음을 확인, ID에서 계산된 Branch(2clock의 o_result)로 PC(0x00000024)를 이동한다.



Blez의 경우 r5 값을 받아(2-3clock의 i_Read_reg1) Blez의 3clock에서 ALU를 이용해 0과 비교하고(3clock의i_data1,2) r5가 0보다 작음을 확인, ID에서 계산된 Branch(2clock의 o_result)로 PC(0x00000024)를 이동한다.

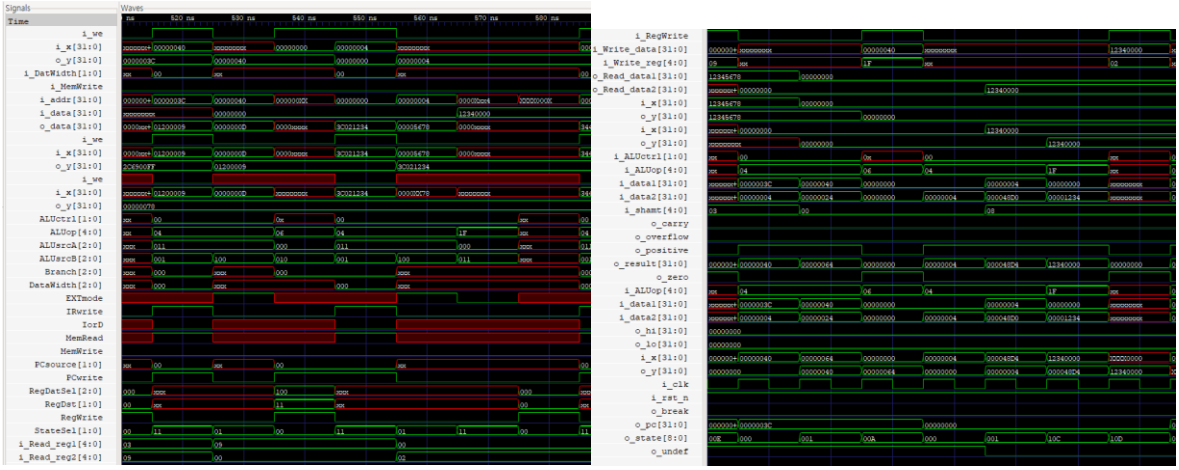
ANDI의 경우, r3(2,3clock의 i_Read_reg1)의 값과 immediate value를 받고(3clock의 i_data1,2) 그 결과를(3clock의 o_result=0x4278) r5(4clock의 i_write_reg)에 저장해주었다.

Sb의 경우 r3, r5(i_Read_reg1,2)의 값(0x12345678, 0x00004278)을 받고 r3와 immediate value(3clock i_data1,2)를 더한(3clock o_result=0x12345680) 주소값이 가리키는 memory에 r5의 하위 8bit(0xxxxxxx78)를 넣는다.



LBU는 r3(i_Read_reg1)의 값(0x12345678)을 받고 r3와 immediate value(3clock i_data1,2)를 더한(3clock o_result=0x12345680) 주소값이 가리키는 memory에서 값을 받아와(4clock의 o_data) r5(5clock의 i_Write_ref)에 해당 값을(5 clock의 i_Write_data=0x00000078) 넣는다.

SLTIU는 r3(i_Read_reg1)의 값(0x12345678)을 받고 r3와 immediate value(3clock i_data1,2)를 unsigned SLT하여 나온 결과(3clock의 o_result=0x00000000)를 확인하여 r9(4clock의 i_Write_reg)에 결과인 0x00000000(4clock의 i_Write_data이자 3clock의 o_result)을 넣어주었다.



이후 JALR에서 $31(3clock의 i_Write_reg)에 pc값(1clock i_data1+idata2)을 저장하고(i_Write_data), r9(2,3clock의 i_Read_reg1)의 값(3clock의 i_data1)을 받고 이를 0(3clock의 i_data2)과 더하여 해당 값으로 이동하는 것을 확인할 수 있다. 이후 0x00000000으로 돌아가 무한반복이 되는 것을 확인할 수 있었다.