컴퓨터 구조

프로젝트 1

이성원 교수님

컴퓨터정보공학부

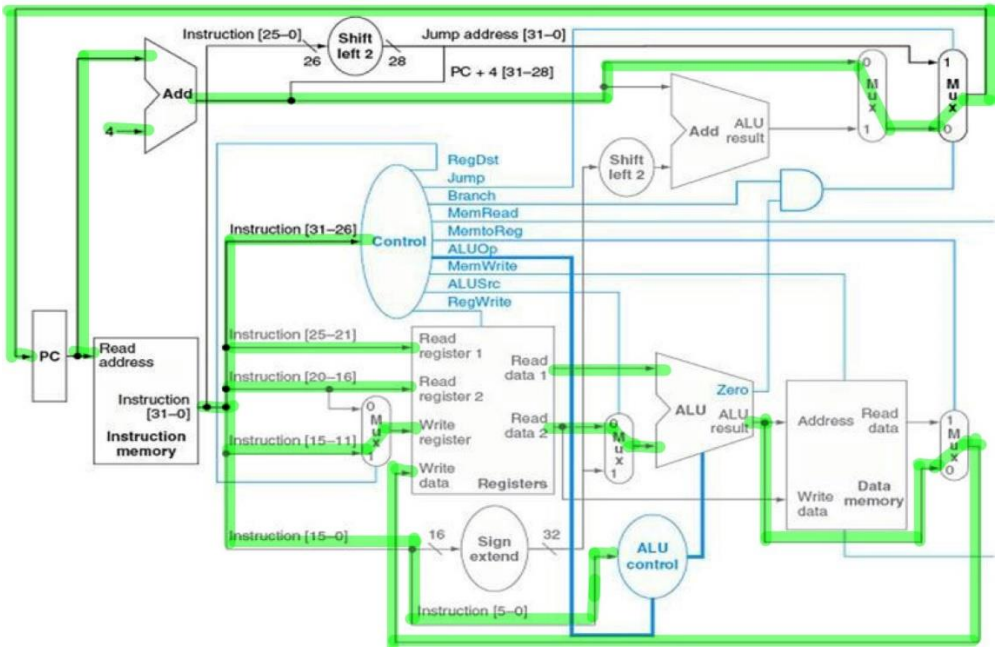
2018202076

이연걸

■ 문제의 해석 및 해결 방향

- sra

Instruction	Opcode/Function	Syntax	Operation
sra	000011	f $d, $t, sa	$d = $t >>> a



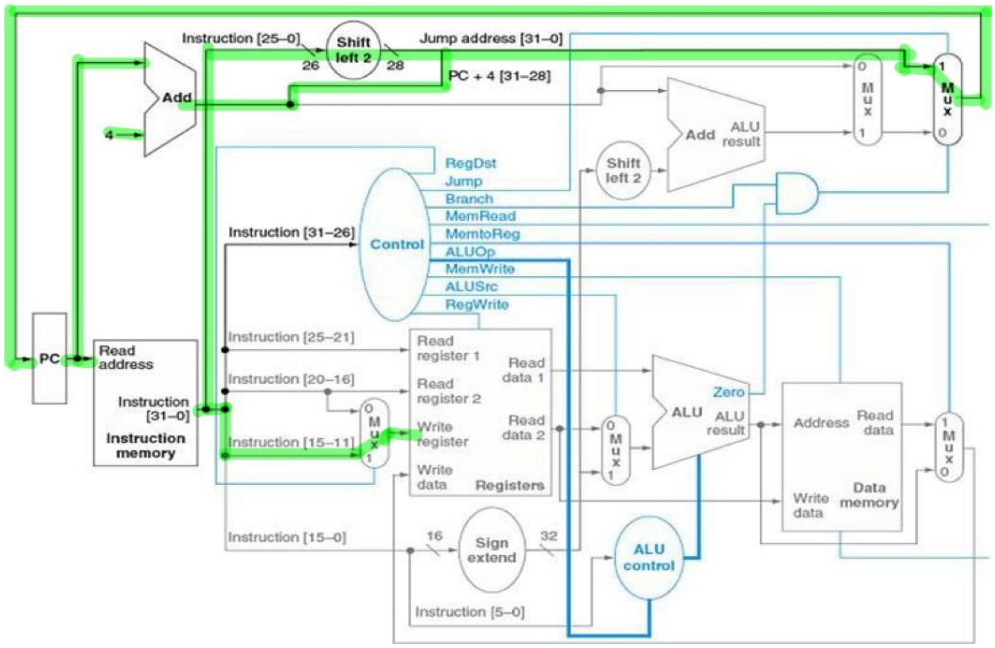
sra는 R type Instruction으로, $t를 a만큼 shift right한 값을 $d에 저장하는 명령어이다. shift right할 때 만약 $t의 MSB가 0이면 0을 채워주고 1이면 1을 채워준다. 이는 부호를 구분하기 위해 사용되고, 같은 기능을 하지만 부호를 구분하지 않는 srl 명령어가 존재한다.

sra 명령어가 Instruction memory에서 나오면 $t의 값과 a의 값이 register에서 나와서 ALU로 들어간다. ALU에서 shift 연산을 마치면 그 값이 레지스터($d)로 들어오면서 shift right를 수행한다. Adder를 통해 현재 PC의 값과 4가 더해져 다음 Instruction을 받아온다.

만약 sra가 I type instruction이 되어 상수를 이용한 연산을 할 수 있었다면 ALU에서 $t의 값과 sign extend한 immediate 값을 이용해 $t의 값을 조정한 후 $d에 넣어줄 수 있었을 것이다.

- jalr

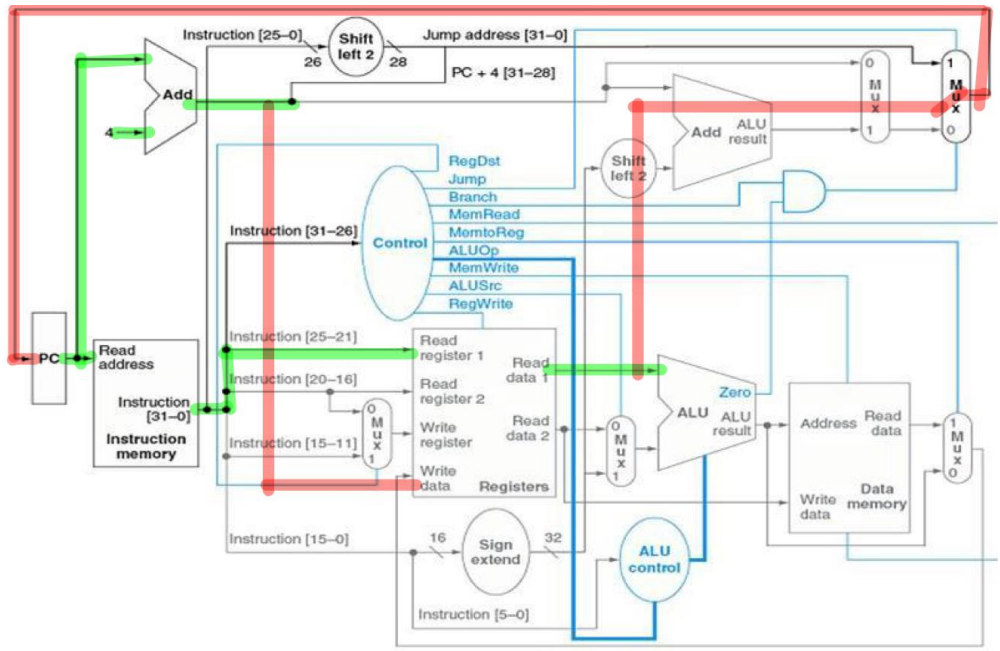
Instruction	Opcode/Function	Syntax	Operation
jalr	001001	f labelR	$31 = pc; pc = $s



jalr은 R type instruction으로 다음 명령어의 주소(PC + 4)를 $31에 저장한 뒤 $s로 점프한다.

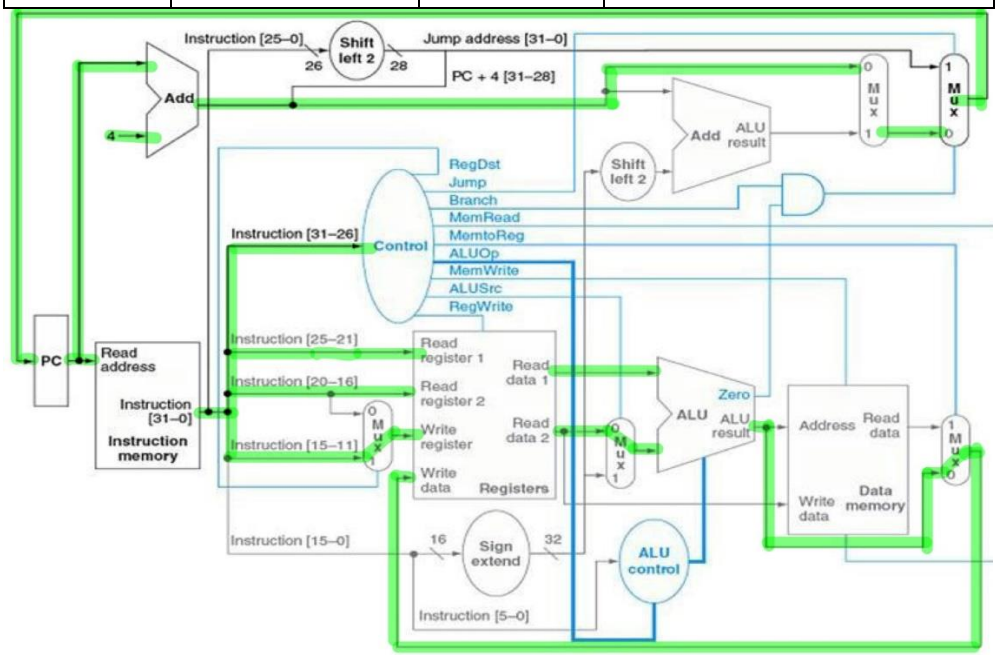
이상적인 회로의 기능은 jalr 명령어가 Instruction memory에서 나오면 Instruction 25-21에 있는 rs가 다시 PC로 들어가고 PC + 4의 값이 Register의 Write data에 들어가 작업이 종료되는 것이다.

다만 이 회로에는 문제점이 있다. PC + 4의 값을 $31에 저장해 주어야 하는데 PC + 4의 값이 Register로 전달될 수 있는 경로가 존재하지 않으며 점프해야 할 $s를 받을 수도 없다. 이상적으로 동작하게 회로를 수정하면 다음과 같다.



- subu

Instruction	Opcode/Function	Syntax	Operation
subu	100011	f $d. $s. $t	$d = $s - $t



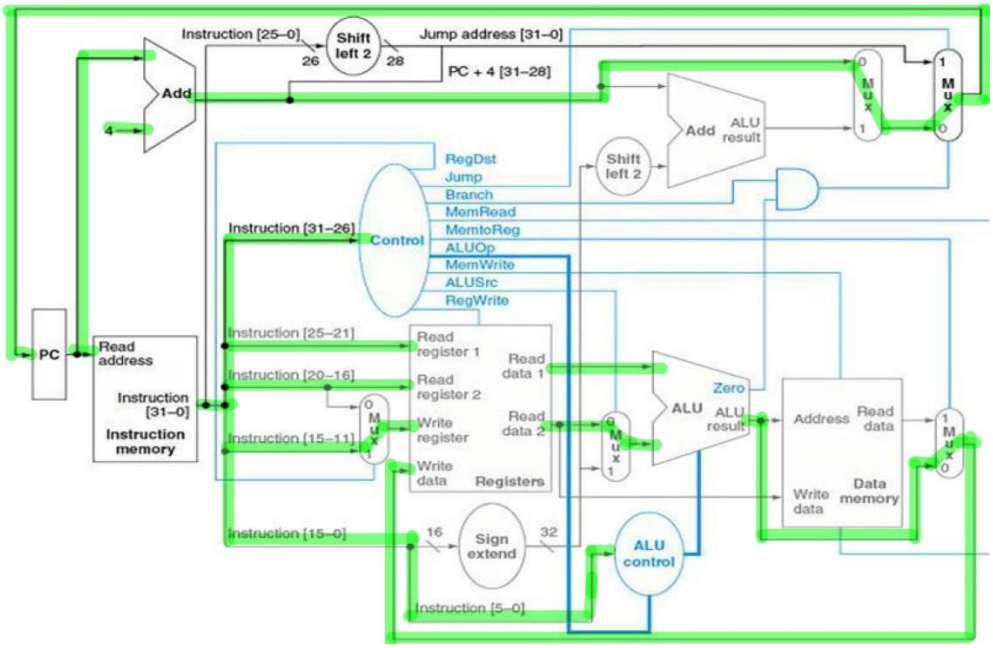
subu는 R type instruction으로$s와 $t 값의 부호를 없애고 뺄셈을 한 뒤 $d에 저장하는 명령어이다.

Instruction memory에서 나온 subu명령어는 $s, $t, $d를 Register file로보낸다. Register file을 통과한 $s, $t는 ALU에서 subu 연산을 진행하고 그 값은 Data memory로 가지 않고 다시 Register file로 향한다. 이곳으로 향해진 값이 $d에 저장되면서 명령은 종료된다.

sub가 아닌 subu 이므로 추가로 ALU control이 필요하다.

- xor

Instruction	Opcode/Function	Syntax	Operation
xor	100110	f $d, $s, $t	$d = $s ^ $t

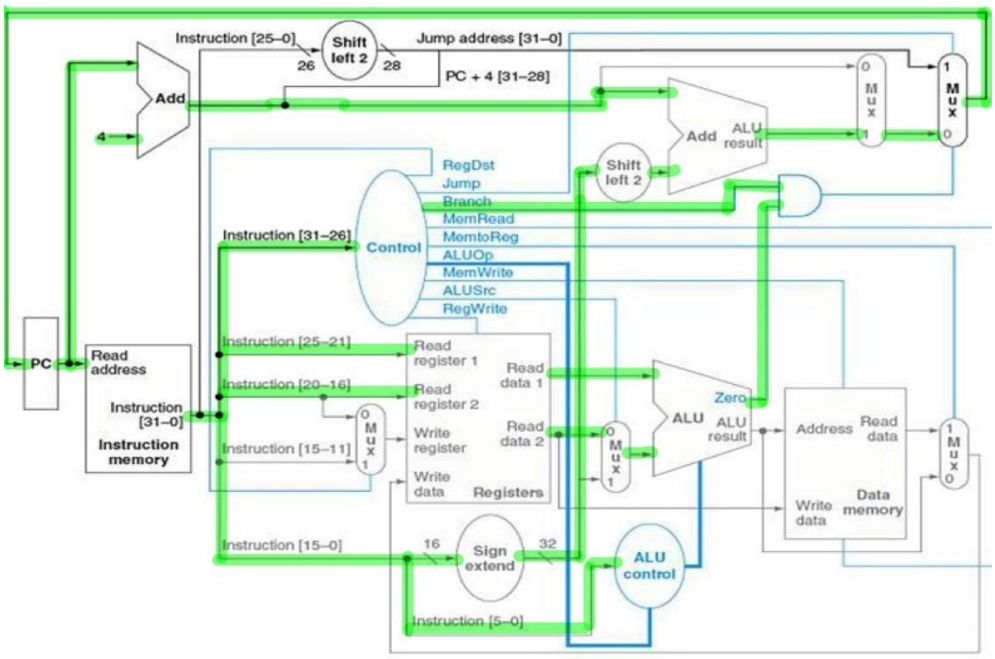


xor은 R type instruction으로 $s와 $t를 xor연산한 값을 $d에 저장한다.

Instruction memory를 통과한 명령어가 Control logic, reg1, reg2, write register에 값을 할당한다. branch나 jump 명령어가 아니므로 다음 PC의 값은 PC+4가 되고 reg 2개 즉, $s와 $t는 ALU에서 산술연산을 거친 이후에 결과값이 Register file의 write data로 들어가 $d에 할당된다. 그와 동시에 다음 PC값인 PC + 4를 PC에 넣어주며 명령어는 종료된다.

- bne

Instruction	Opcode/Function	Syntax	Operation
bne	000101	o $s, $t, label	If ($s != $t) pc += i << 2

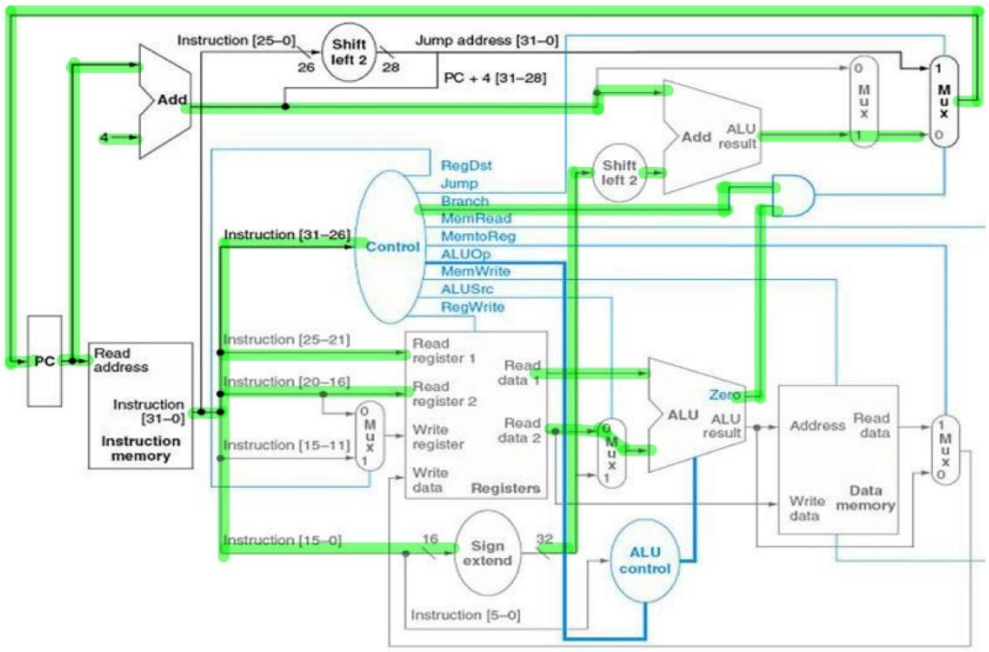


bne는 I type 명령어다. I type 명령어는 상수와의 연산을 다룰 수 있다. $s와 $t에 있는 값을 비교해 이 둘이 같지 않다면 imm으로 branch를 같다면 다음 PC 값은 PC+4가 된다. branch 명령어이므로 PC + 4 + branch 형태의 PC값이 필요하게 되므로 Data Path는 위와 같다.

Instruction memory를 통과하면서 Control logic, register file에 값이 들어가게 되고 register file에서 나온 $s, $t의 값이 ALU에서 연산 된다. 논리연산이 아닌 산술연산이 필요한 이유는 $s != $t를 알기 위해선 $s - $t != 0 이기 때문이다. 저 둘의 차가 0이 아니라면 저 둘은 같지 않고 따라서 branch가 진행된다. ALU의 Zero 값에 따라 branch가 결정되기 때문에 회로는 위와 같다.

- blez

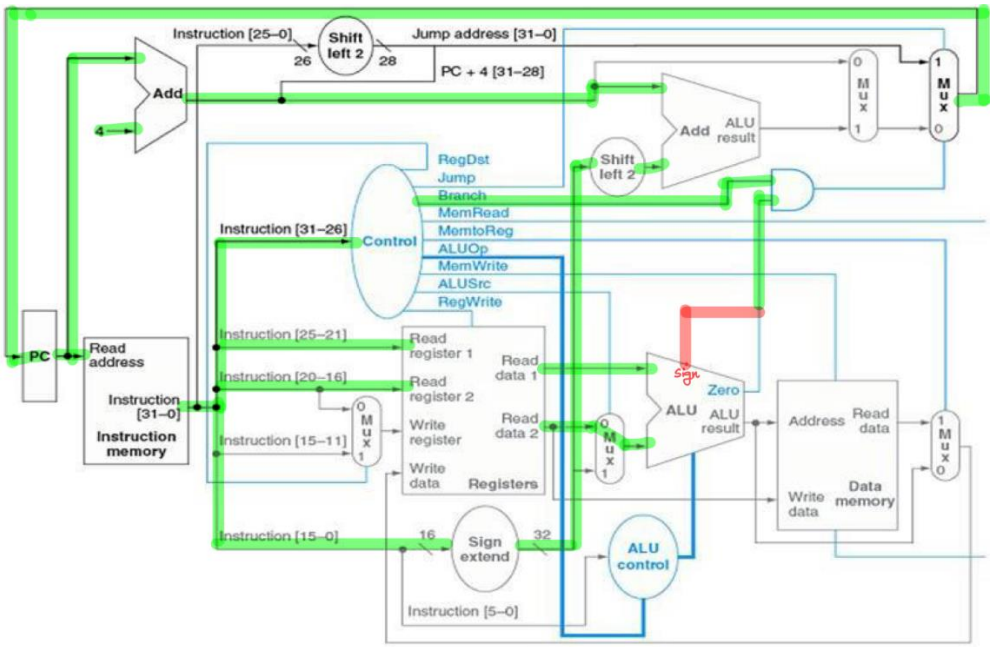
Instruction	Opcode/Function	Syntax	Operation
blez	000110	o $s, label	If ($s <= 0) pc += i << 2



blez는 I type명령어다. 조건에 따른 branch가 필요하다. $s의 값이 0보다 작거나 같으면 pc에 imme+PC+4의 값으로 branch된다. branch명령어는 imme를 branch에 더하기 전에 shift left 2가 필요한데 이는 표현할 수 있는 범위를 좀더 넓히고, 32bit명령어 이기 때문이다.

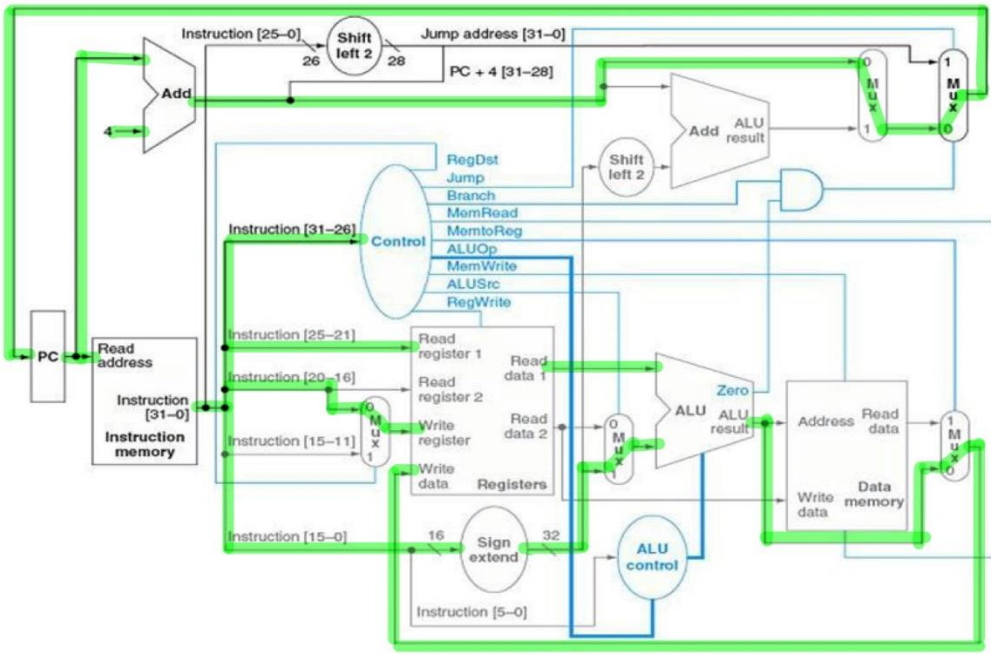
Instruction memory에서 나온 명령어가 Control logic, Register file으로 들어간다. Register file에서 나온 reg $s와 $0의 산술 연산을 통해 branch 여부를 알 수 있는데 $s - $0 <= 0이면 branch된다. 이것을 계산하기 위해서 ALU가 필요하며 따라서 ALU의 Zero에서 나온 데이터에 의해 branch된다.

이 회로에도 문제점이 존재하는데 Zero의 여부만 확인하기 때문에 부호의 확인이 어렵다는 것이다. 이를 위해 Sign bit flag가 필요하다. 이상적으로 동작하는 회로의 모습은 다음과 같다.



- sltiu

Instruction	Opcode/Function	Syntax	Operation
sltiu	001011	o $d, $s, i	$t = ($s < ZE(i))



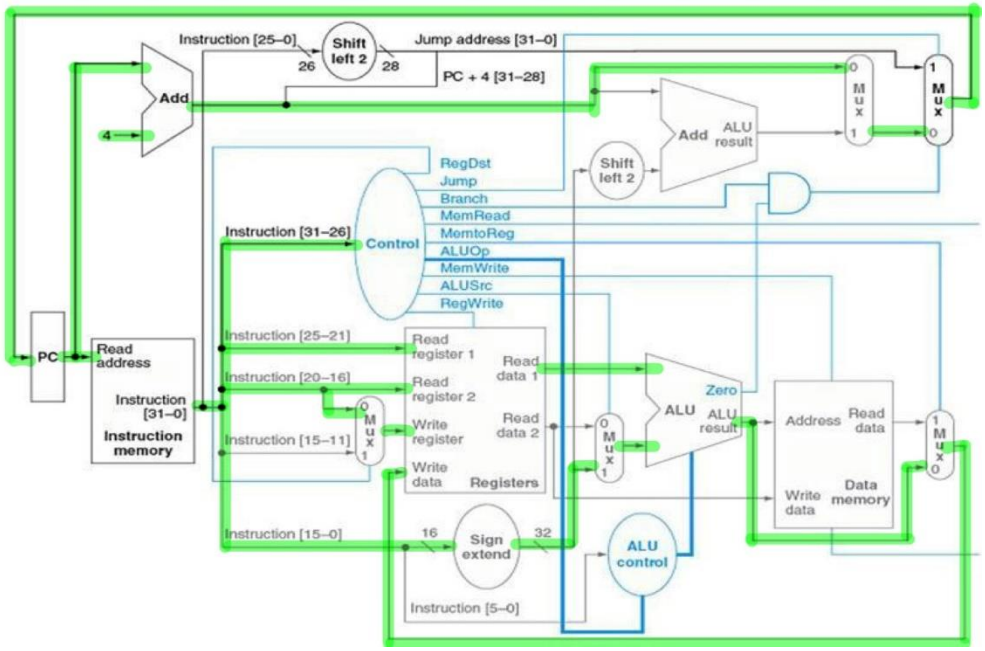
sltiu는 상수와의 연산이 들어있기 때문에 I type 명령어이다. $s의 imme보다 작다면 $t에는 1, imme보다 크거나 같다면 $t에는 0을 넣는다.

Instruction memory를 거친 I type 명령어는 Control logic, Register file에 들어가게 되고 Register file에서 나온 $s값과 Sign extend된 imme값이 ALU에서 만나 연산을 거친다. $s < imme을 증명하기 위해선 opcode의 set less than을 사용해

야한다. 따라서 ALU가 필요하며 조건이 참이라면 1, 아니라면 0을 $d에 할당한다. 그와 동시에 다음 PC값인 PC + 4를 PC에 넣어주며 명령어는 종료된다.

- andi

Instruction	Opcode/Function	Syntax	Operation
andi	001100	o $t, $s, i	$t = $s & ZE(i)

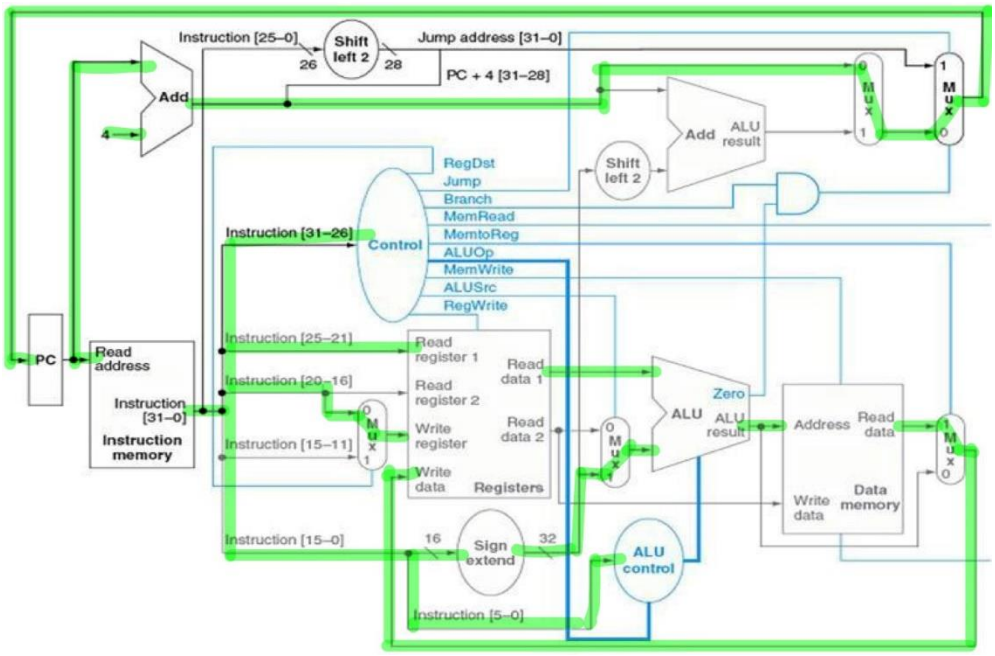


andi는 상수와의 연산으로 인해 I type 명령어다. $s의 값과 imme값을 and연산하고 결과 값을 $t에 저장한다.

Instruction memory에서 나온 I type 명령어가 Control logic Register file에 들어가고, $t에 값이 저장되므로 $t은 Write register에 들어간다. $s값과 Sign extend된 imme값이 ALU에서 AND 연산 되고 그 값이 다시 Write data로 들어가 $t에 할당된다. 그와 동시에 다음 PC값인 PC + 4를 PC에 넣어주며 명령어는 종료된다.

- lbu

Instruction	Opcode/Function	Syntax	Operation
lbu	100100	o $t, I ($s)	$t = ZE (MEM [$s + i]:1)

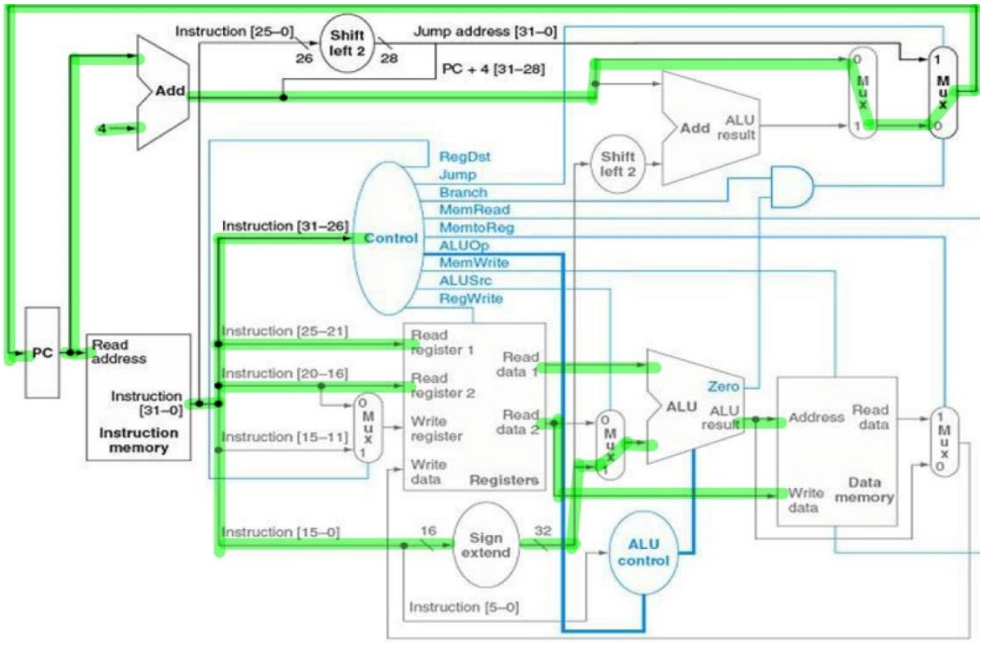


lbu는 상수와의 연산이 들어가므로 I type 명령어이다. $s + imme을 한 메모리 주소 값을 가져와 $t에 할당해 준다.

Instruction memory에서 나온 I type 명령어는 Control logic, Register file으로 들어가고 $t에 값이 할당되어야 하므로 $t는 Write register에 들어간다. Sign Extend에서 Control signal으로 zero extend imme값과 $s의 값이 ALU에서 합쳐져 Data memory로 들어간다. 이때 byte단위로 Data memory에서 조회된 값이 Write data에 들어가 $t에 할당된다. 그와 동시에 다음 PC값인 PC + 4를 PC에 넣어주며 명령어는 종료된다.

- sb

Instruction	Opcode/Function	Syntax	Operation
sb	101000	o $t, I ($s)	MEM [$s + i]:1 = LB ($t)

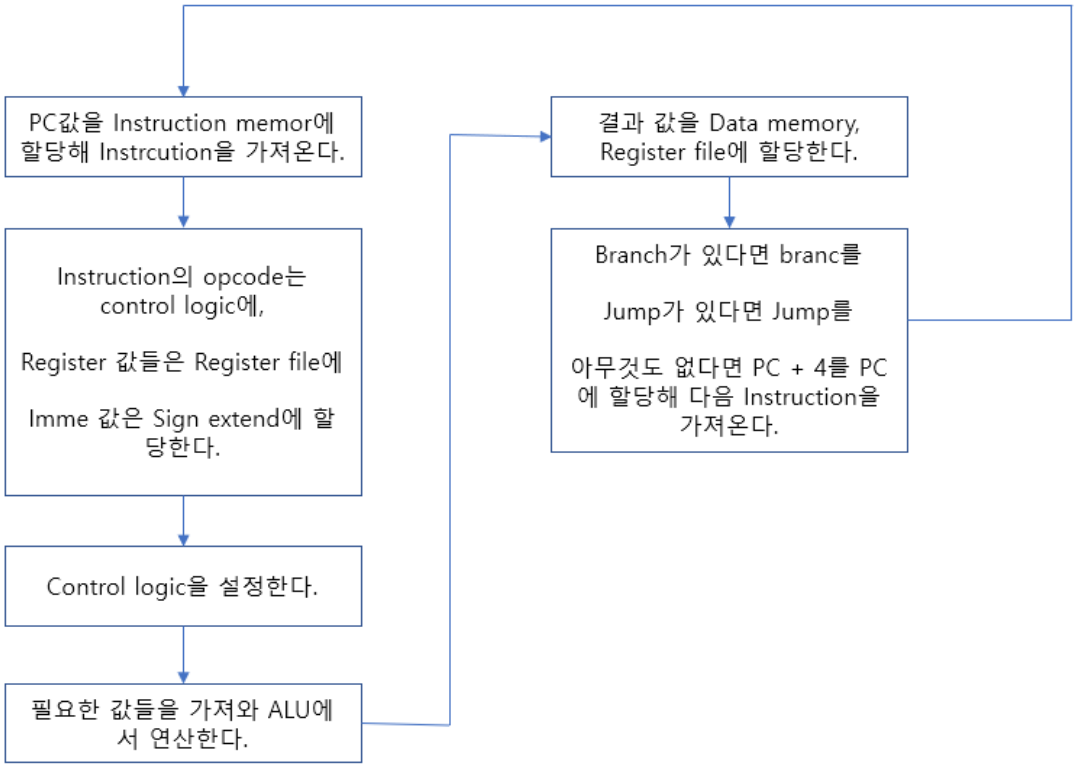


sb는 mermoy에 직접 접근하므로 I type명령어이다. store byte란 의미로 메모리 주소 $s + imme에 접근하여 $t의 값을 저장한다.

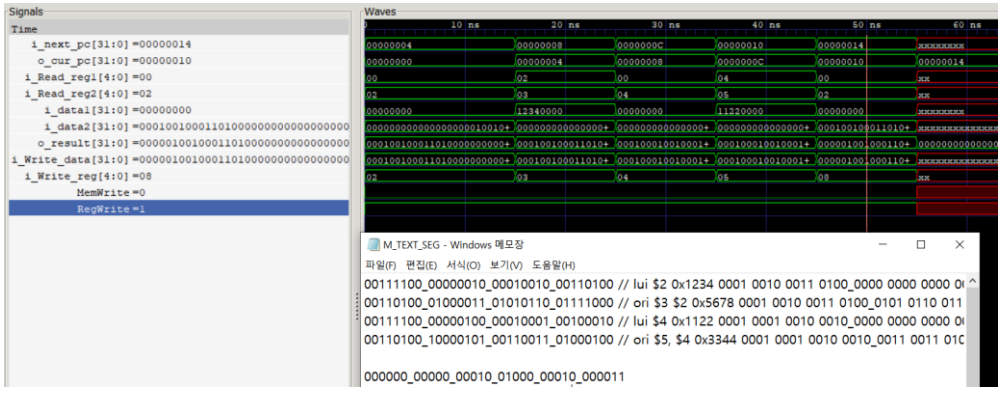
Instruction memory에서 나온 I type 명령어가 Control logic, Register file에 들어가고 Sign extend된 imme와 Register file에서나온 $s의 값이 ALU에서 합쳐진다. 그 값이 Data memory로 들어가 메모리 주소가 지정되고 지정된 주소에 Register file에서 나온 $t의 값이 저장된다. 그와 동시에 다음 PC값인 PC + 4를 PC에 넣어주며 명령어는 종료된다.

■ 설계 의도와 방법

Single Cycle CPU 블록도



- sra

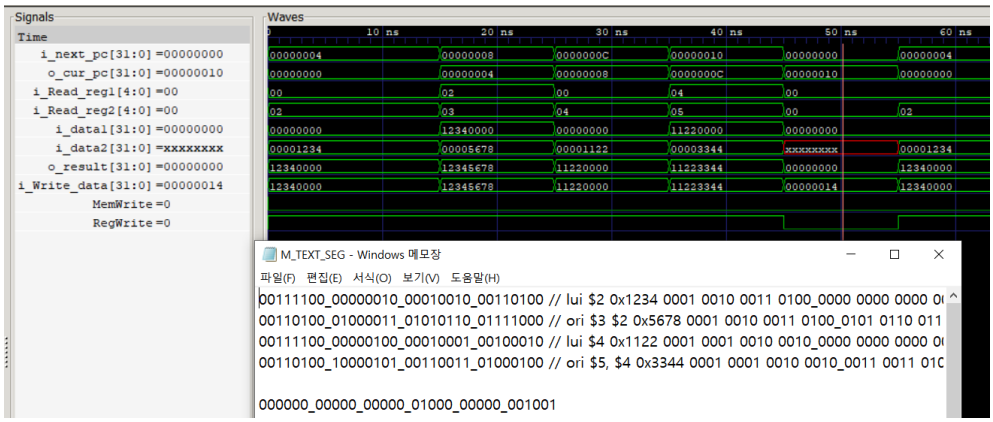


rt를 $2로 설정하고, rd를 $8로 설정한 뒤 shift amount에 2를 넣고 실행 한 모습이다. $2의 값이 오른쪽으로 2칸 밀려나고 그 자리에 0이 채워져 정상적으로 작동한 모습을 확인할 수 있다. reg에만 값이 할당되어 Write enable 또한 정상적으로 동작한 것을 알 수 있다.

Reg Dst	Reg Dat Sel	Reg Write	SEU Mode	ALU srcB	ALU Ctrl	ALU Op	Data Width	Mem Write	Mem To Reg	Branch	Jump
01	00	1	x	00	00	01110	000	0	0	000	00

sra는 값이 저장될 레지스터가 $d이며 ALU의 결과를 레지스터 파일에 저장하고 레지스터 파일에 값을 쓴다. 상수는 사용되지 않으며 B source는 다른 레지스터인 port B가 사용된다. ALU control은 사용하지 않고, OpCode는 01110, 연산 시 32bit word를 사용한다. 메모리에 값을 작성하지 않고, 메모리 값을 가져와 register에 작성하지 않는다. 또한branch, jump 명령이 아니다. 그러므로 제어신호를 위와 같이 설정했다.

- jalr

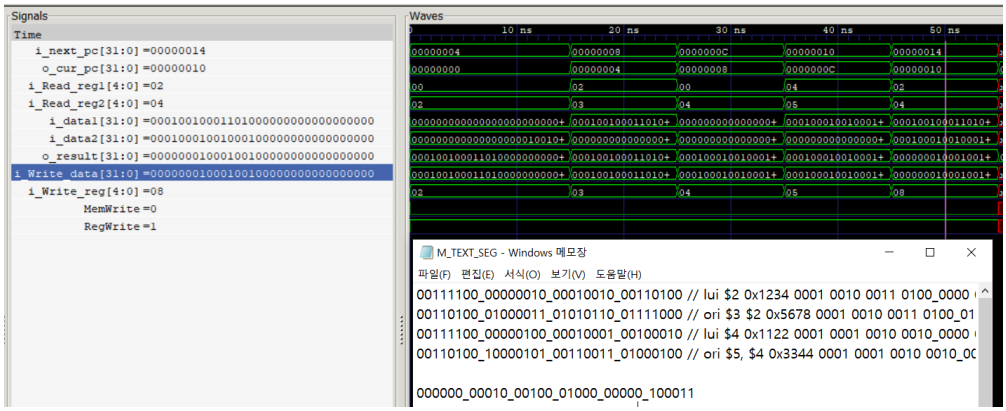


$s, $t를 0으로, $d를 $8로 설정한 모습이다. 다음 PC값이 $s이기 때문에 i_next_pc에 0이 들어가는 모습을 확인할 수 있고, register에 PC + 4를 넣어주므로 i_Write_reg에 10 + 4의 값이 들어가 1F가 된 것을 확인할 수 있다. 또한 reg, mem 어디에도 값이 할당되지 않아 Write enable가 정상적으로 동작한 것을 알 수 있다.

Reg Dst	Reg Dat Sel	Reg Write	SEU Mode	ALU srcB	ALU Ctrl	ALU Op	Data Width	Mem Write	Mem To Reg	Branch	Jump
10	11	x	x	xx	10	01000	000	0	0	000	00

jalr은 값이 저장될 레지스터가 $31이며 PC +4의 결과를 레지스터 파일에 저장하고 레지스터 파일에 값을 쓴다. 상수는 사용되지 않으며 B source는 ALU연산이 필요하지 않기 때문에 상관하지 않는다. ALU control은 normal input을 사용하고, OpCode는 01000, 연산 시 32bit word를 사용한다. 메모리에 값을 작성하지 않고, 메모리 값을 가져와 register에 작성하지 않는다. 또한 branch, jump 명령이 아니다. 그러므로 제어신호를 위와 같이 설정했다.

- subu

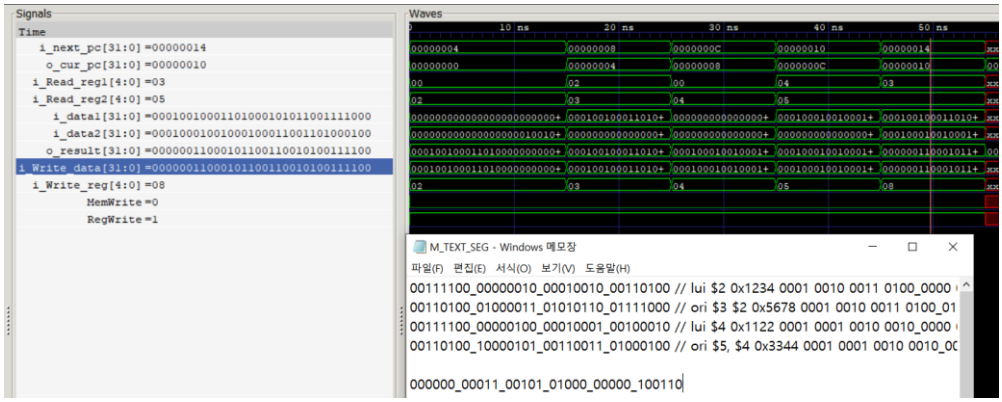


$s에 $2, $t에 $4, $d에 $8을 넣고 테스트 벤치를 실행하였다. $2의 값이 0x1234이고 $4의 값이 0x1122이고 i_Write_Data에 이 둘을 마이너스 연산 한 결과가 표시되고 값이 저장되는 레지스터가 08임을 확인할 수 있다. 또한 reg에만 값이 할당되어 Write enable이 정상적으로 동작한 것을 알 수 있다. 또한 reg, mem 어디에도 값이 할당되지 않아 Write enable가 정상적으로 동작한 것을 알 수 있다.

Reg Dst	Reg Dat Sel	Reg Write	SEU Mode	ALU srcB	ALU Ctrl	ALU Op	Data Width	Mem Write	Mem To Reg	Branch	Jump
01	00	1	x	00	00	00111	xxx	0	x	000	00

subu는 값이 저장될 레지스터가 $d이며 ALU의 결과를 레지스터 파일에 저장하고 레지스터 파일에 값을 쓴다. 상수는 사용되지 않으며 B source는 다른 레지스터인 port B가 사용된다. ALU control은 사용하지 않고, OpCode는 00111, 연산 시 32bit word를 사용한다. 메모리에 값을 작성하지 않고, 메모리 값을 가져와 register에 작성하지 않는다. 또한 branch, jump 명령이 아니다. 그러므로 제어신호를 위와 같이 설정했다.

- xor

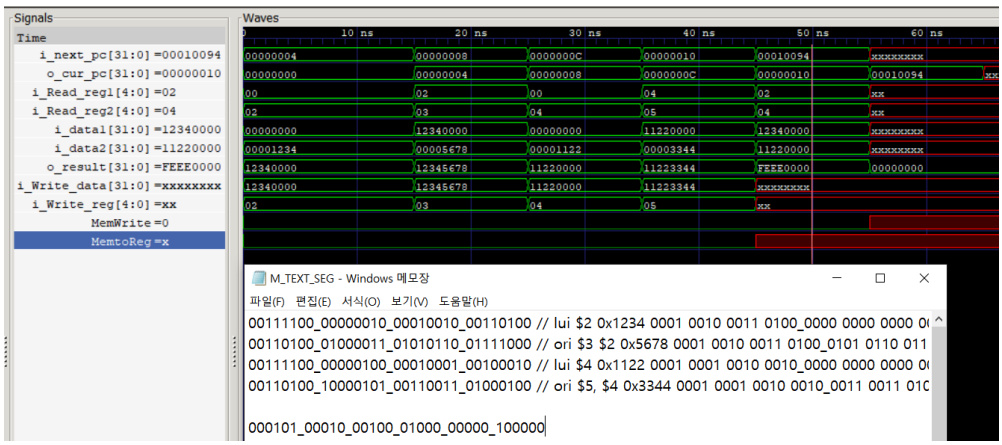


$s에 $3을, $t에 $5를, $d에 $8을 넣고 xor명령어로 연산을 진행했다. 두개의 레지스터의 값이 xor연산되고 i_Write_data에 기록된 것을 확인할 수 있고 $8에 그 값이 들어간 것을 확인할 수 있다. 또한 reg에만 값이 할당되어 Write enable이 정상적으로 동작한 것을 알 수 있다.

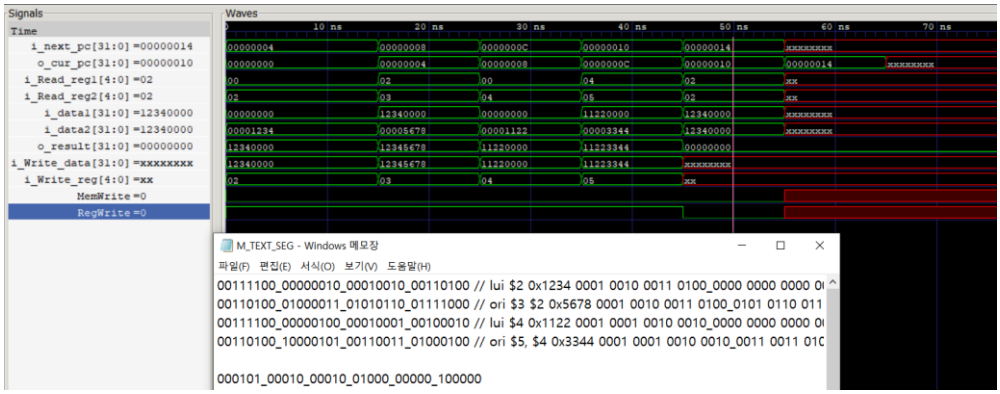
Reg Dst	Reg Dat Sel	Reg Write	SEU Mode	ALU srcB	ALU Ctrl	ALU Op	Data Width	Mem Write	Mem To Reg	Branch	Jump
01	00	1	x	00	00	00011	xxx	0	0	000	00

xor는 값이 저장될 레지스터가 $d이며 ALU의 결과를 레지스터 파일에 저장하고 레지스터 파일에 값을 쓴다. 상수는 사용되지 않으며 B source는 다른 레지스터인 port B가 사용된다. ALU control은 사용하지 않고, OpCode는 00011, 연산 시 32bit word를 사용한다. 메모리에 값을 작성하지 않고, 메모리 값을 가져와 register에 작성하지 않는다. 또한 branch, jump 명령이 아니다. 그러므로 제어신호를 위와 같이 설정했다.

- bne



rs에는 $2, rt에는 $4, rd에는 $8을 각각 넣어주었다. bne 명령어를 사용해 테스트 벤치를 작성하였다. $3와 $5에는 할당된 값이 다르므로 next_pc 값이 10 + 4가 아닌 내가 할당해준 값으로 바뀌어 들어가는 것을 확인할 수 있다. 또한 reg, mem 어디에도 값이 할당되지 않아 Write enable가 정상적으로 동작한 것을 알 수 있다.

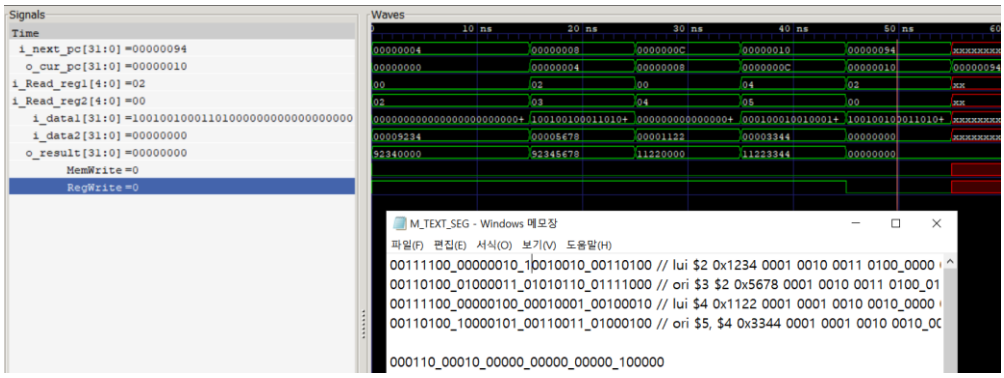


이번엔 $s와 $t에 같은 값을 할당해 branch가 되지 않는 모습이다.

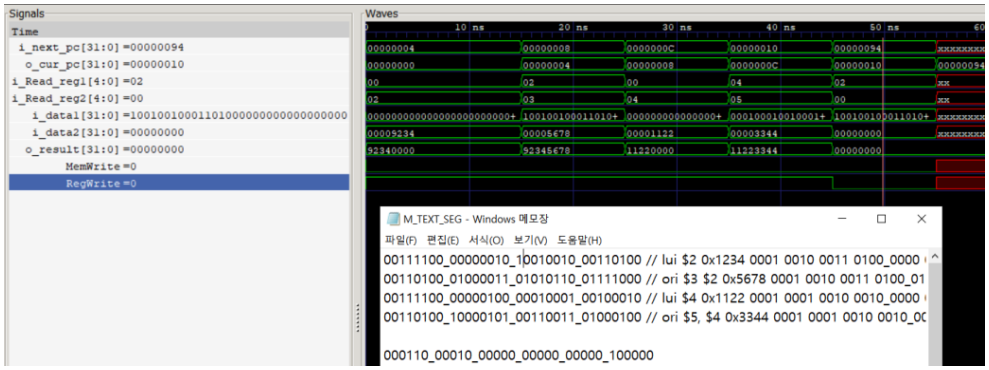
Reg Dst	Reg Dat Sel	Reg Write	SEU Mode	ALU srcB	ALU Ctrl	ALU Op	Data Width	Mem Write	Mem To Reg	Branch	Jump
xx	xx	0	1	00	10	00110	xxx	0	x	101	00

bne는 값이 저장될 레지스터가 없으며 ALU의 결과를 레지스터 파일에 저장하지 않고 레지스터 파일에 값을 쓰지 않는다. 상수는 Sign extend되어 사용되고, B source는 다른 레지스터인 port B가 사용된다. ALU control은 normal input으로 사용되고, OpCode는 00110, 연산 시 32bit word를 사용한다. 메모리에 값을 작성하지 않고, 메모리 값을 가져와 register에 작성하지 않는다. 또한 not zero일 때 branch한다. 그러므로 제어신호를 위와 같이 설정했다.

- blez



rs, rt 에 각각 $2, 0을 할당하고 address값에 100000을 넣어주었다. $2는 부호비트를 1로 바꾸어 미리 음수로 만들어 두었다. 값을 비교할 때 $2는 음수이므로 0보다 작다. 따라서 branch가 발생하고 next_pc값을 보면 정상적으로 branch되는 것을 확인할 수 있다. 또한 reg, mem 어디에도 값이 할당되지 않아 Write enable가 정상적으로 동작한 것을 알 수 있다.

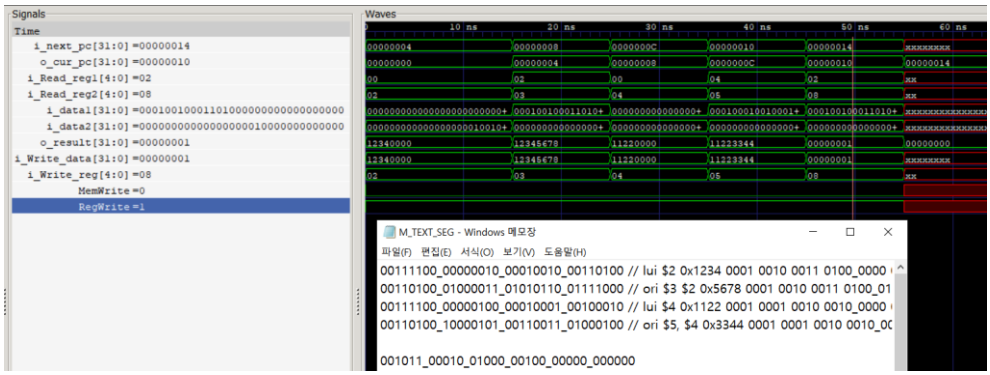


이것은 $2의 값이 0보다 크므로 branch되지 않는 모습이다.

Reg Dst	Reg Dat Sel	Reg Write	SEU Mode	ALU srcB	ALU Ctrl	ALU Op	Data Width	Mem Write	Mem To Reg	Branch	Jump
xx	xx	0	x	10	10	10000	000	0	x	110	00

blez는 값이 저장될 레지스터가 없으며 ALU의 결과를 레지스터 파일에 저장하지 않고 레지스터 파일에 값을 쓰지 않는다. imme는 주소로만 사용되고 B source는 zero가 사용된다. ALU control은 normal input만을 사용하고, OpCode는 10000, 연산 시 32bit word를 사용한다. 메모리에 값을 작성하지 않고, 메모리 값을 가져와 register에 작성하지 않는다. 또한 branch명령이다. 그러므로 제어신호를 위와 같이 설정했다.

- sltiu

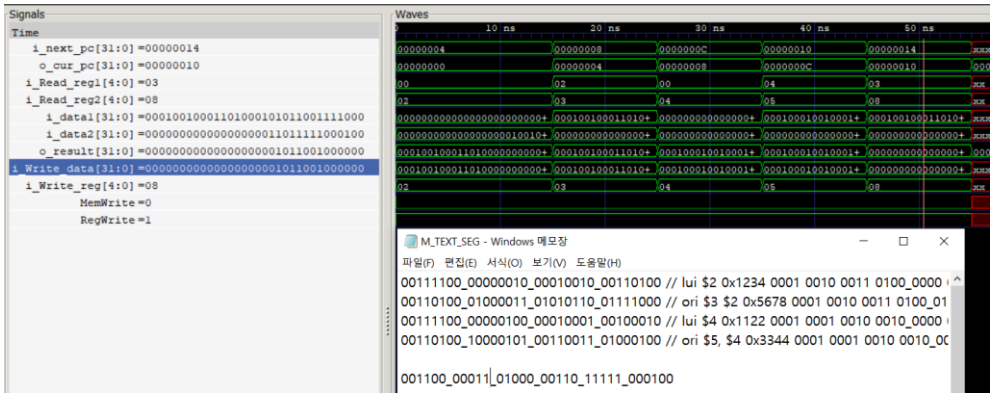


rs, rt 에 각각 $2, $8를 할당한다. 여기서 $2는 할당된 imme값인 0010 0000 0000 0000보다 작기 때문에 $8에 1이 저장되어야 한다. i_Write_data에 1이 저장된 것을 볼 수 있고, 값이 저장될 레지스터는 $8인 것 또한 확인할 수 있다. 또한 reg에만 값이 할당되어 Write enable이정상적으로 동작한 것을 알 수 있다.

Reg Dst	Reg Dat Sel	Reg Write	SEU Mode	ALU srcB	ALU Ctrl	ALU Op	Data Width	Mem Write	Mem To Reg	Branch	Jump
00	00	1	1	01	00	10001	000	0	0	000	00

sltiu는 값이 저장될 레지스터가 $t이며 ALU의 결과를 레지스터 파일에 저장하고 레지스터 파일에 값을 쓴다. 상수가 sign extend되어 사용되고 B source는 sign extend된 상수 값이 사용된다. ALU control은 사용하지 않고, OpCode는 10001, 연산 시 32bit word를 사용한다. 메모리에 값을 작성하지 않고, 메모리 값을 가져와 register에 작성하지 않는다. 또한 branch, jump 명령이 아니다. 그러므로 제어신호를 위와 같이 설정했다.

- andi



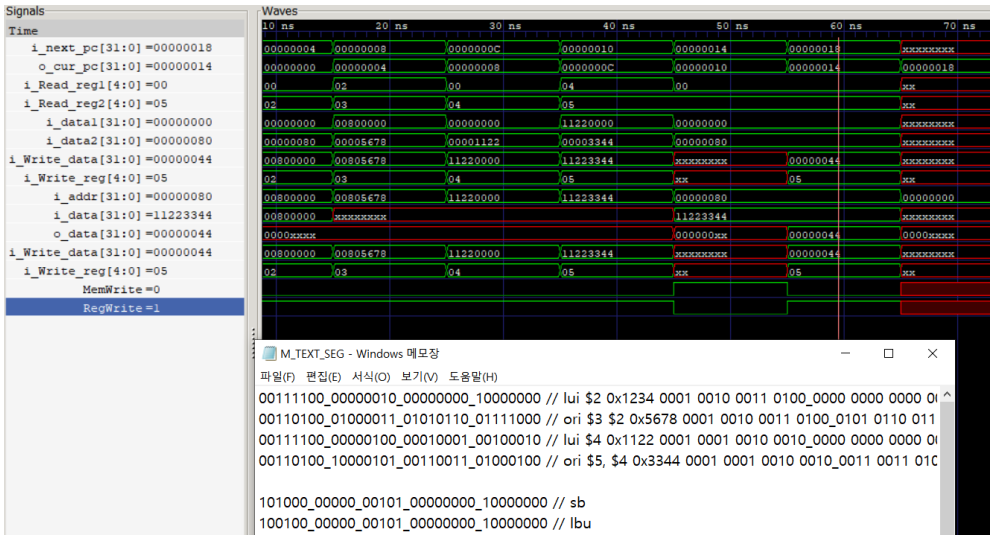
rs, rt에 각각 $3, $8을 할당하고 andi연산을 진행했다. and연산된 값이 i_Write_data에 저장되는 것을 확인할 수 있고, 값이 저장되는 레지스터 또한 내가 설정한 $8인 것을 볼 수 있다. 또한 reg에만 값이 할당되어 Write enable이정상적으로 동작한 것을 알 수 있다.

Reg Dst	Reg Dat Sel	Reg Write	SEU Mode	ALU srcB	ALU Ctrl	ALU Op	Data Width	Mem Write	Mem To Reg	Branch	Jump
00	00	1	0	01	10	00000	000	0	0	000	00

sra는 값이 저장될 레지스터가 $t이며 ALU의 결과를 레지스터 파일에 저장하고

레지스터 파일에 값을 쓴다. 상수는 zero extend되어 사용되고 B source는 zero extend된 상수 값이 사용된다. ALU control은 normal input이며, OpCode는 00000, 연산 시 32bit word를 사용한다. 메모리에 값을 작성하지 않고, 메모리 값을 가져와 register에 작성하지 않는다. 또한 branch, jump 명령이 아니다. 그러므로 제어신호를 위와 같이 설정했다.

- lbu



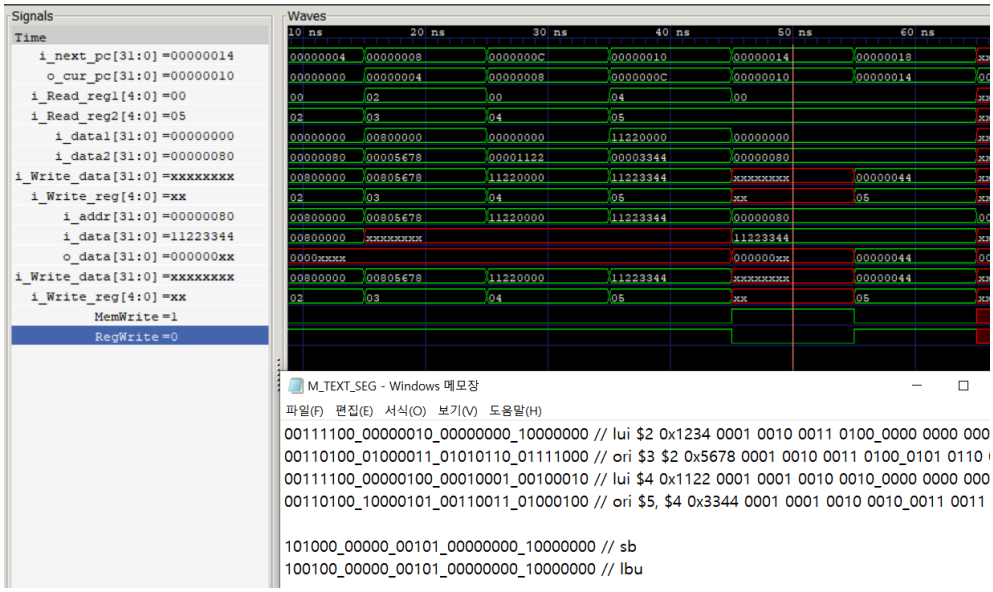
rs, rt에 각각 0, $5를 할당하고 imme 값으로 1000000을 넣어주었다. 그리고 lbu 명령을 실행한다. 이것은 0 + 10000000 메모리 주소 있는 값을 가져와 $5값에 할당한다는 의미인데, 이전에 sb로 동일한 주소에 $5의 값을 sign extend Byte 단위로 할당한 적이 있다. 따라서 그 값을 그대로 가져온다. i_write_data에 정상적으로 값이 들어오는 것이 보이고, i_write_reg는 내가 지정한 $5 레지스터가 사용되었음을 알 수 있다. 또한 reg에만 값이 할당되어 Write enable이정상적으로 동작한 것을 알 수 있다.

Reg Dst	Reg Dat Sel	Reg Write	SEU Mode	ALU srcB	ALU Ctrl	ALU Op	Data Width	Mem Write	Mem To Reg	Branch	Jump
00	00	1	1	01	00	00100	011	0	1	000	00

lbu는 값이 저장될 레지스터가 $t이며 ALU의 결과를 레지스터 파일에 저장하고 레지스터 파일에 값을 쓴다. 상수는 sign extend되어 사용되고 B source는 sign extend된 상수 값이 사용된다. ALU control은 사용하지 않고, OpCode는 00100, 연산 시 byte단위가 사용된다. 메모리에 값을 작성하지 않고, 메모리 값을 가져와 register에 작성하며 branch, jump 명령이 아니다. 그러므로 제어신호를 위와

같이 설정했다.

- sb



rs, rt에 각각 0, $5를 넣어주었다. rs와 imme값인 10000000를 합친 메모리 주소에 $5의 값을 할당해 주어야 하는데 Data memory에 저장되는 값인 i_Write_reg에 $5의 값을 확인할 수 있고, i_MemWrite를 보면 Data memory에 값을 기록하는 것을 확인할 수 있다. 또한 mem에만 값이 할당되어 Write enable이정상적으로 동작한 것을 알 수 있다.

Reg Dst	Reg Dat Sel	Reg Write	SEU Mode	ALU srcB	ALU Ctrl	ALU Op	Data Width	Mem Write	Mem To Reg	Branch	Jump
xx	xx	0	1	01	0x	00100	111	1	x	000	00

sb는 값이 저장될 레지스터가 없으며 ALU의 결과를 레지스터 파일에 저장하지 않고 레지스터 파일에 값을 쓰지 않는다. 상수는 sign extend되어 사용되고 B source는 sign extend된 상수 값이 사용된다. ALU control은 normal input을 사용하고, OpCode는 00100, 연산 시 sign extend byte단위가 사용된다. 메모리에 값을 작성하고 않고, 메모리 값을 가져와 register에 작성하지 않는다. 또한 branch, jump 명령이 아니다. 그러므로 제어신호를 위와 같이 설정했다.