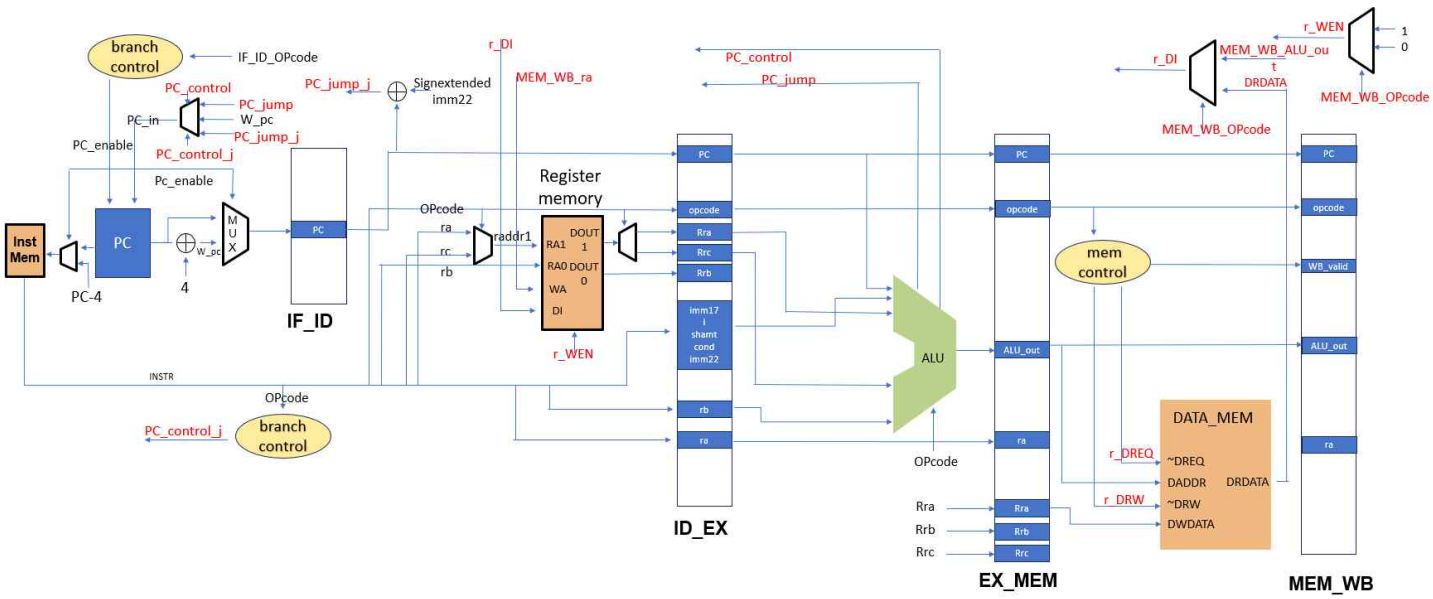
Project 2
RISC-Toy Design

Team 19
2018100720 윤동완
2019103999 유현영

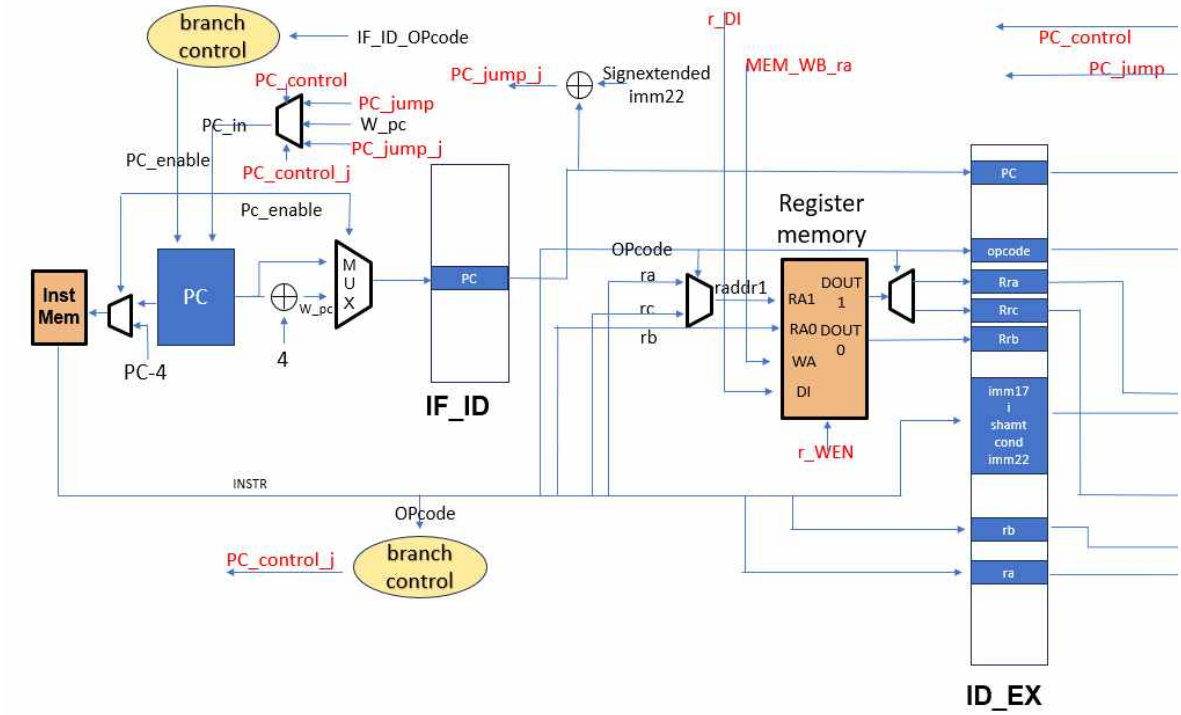
1. 아키텍쳐 설명

(1) 전체 아키텍쳐 및 데이터 패스



전체 데이터 패스는 위와 같다. Data hazard도 일부 구현하였지만, diagram에는 포함시키지 않았다. 각각 pipeline stage의 이름은 IF_ID, ID_EX, EX_MEM, MEM_WB이라고 정의하였고, 변수명 또한 IF_ID_PC와 같은 식으로 (stage+변수명)으로 정의하였다. 또한 register들은 파란 박스로 나타내었다. 각각의 자세한 설명은 후술하겠다.

(2) Instruction fetch && Decoding



-Decoding && Fetch

기본적으로 PC에서 Instruction memory에서 instruction을 읽어들이고, pc+4를 해주는 방식이다. 그러나 branch나 jump가 일어날 때는 pc가 멈춰야하기에 컨트롤을 통해 pc 증가를 멈춰준다. instruction memory엥서 받아온 INSTR을 Trucate를 통해 decoding을 하여, ra, rb, rc, opcode, imm17, imm22, I, cond로 decoding 해준다.

-Muxing

register memory에 mux를 통해 ra, rc중에 하나를 넣어준다. 왜냐하면 한 명령어 내에서 ra, rc가 동시에 필요한 경우가 존재하지 않고, 또한, reg memory의 read port가 2개이기 때문이다. 하나는 rb를 넣어준다. 또한 ra, rc에서 골라넣은 주소의 메모리값인 DOUT1은 demux를 통해 R[ra], R[rc]중에 하나로 사용한다.

-Branch control

Branch control은 pc값은 언제 멈추것인가를 결정한다. 우선적으로 br, j 같은 pc를 이동하는 명령어가 들어오면 우선적으로 한 사이클을 쉬어야 한다. Branch control unit에서 IF_ID_OPcode를 보고 pc를 이동하는 명령인지 판단하고, pc_stop이라는 신호를 켜준다. 내부적으로 pc_stop의 신호를 보고 PC_enable이라는 보내준다. PC_enable이라는 신호는 IF_ID_PC에 PC값 혹은 PC+4 값을 넘겨줄 것인지를 결정하고, Inst mem의 address에 PC를 넣을 것인지 PC-4를 넣을 것인지를 결정한다.

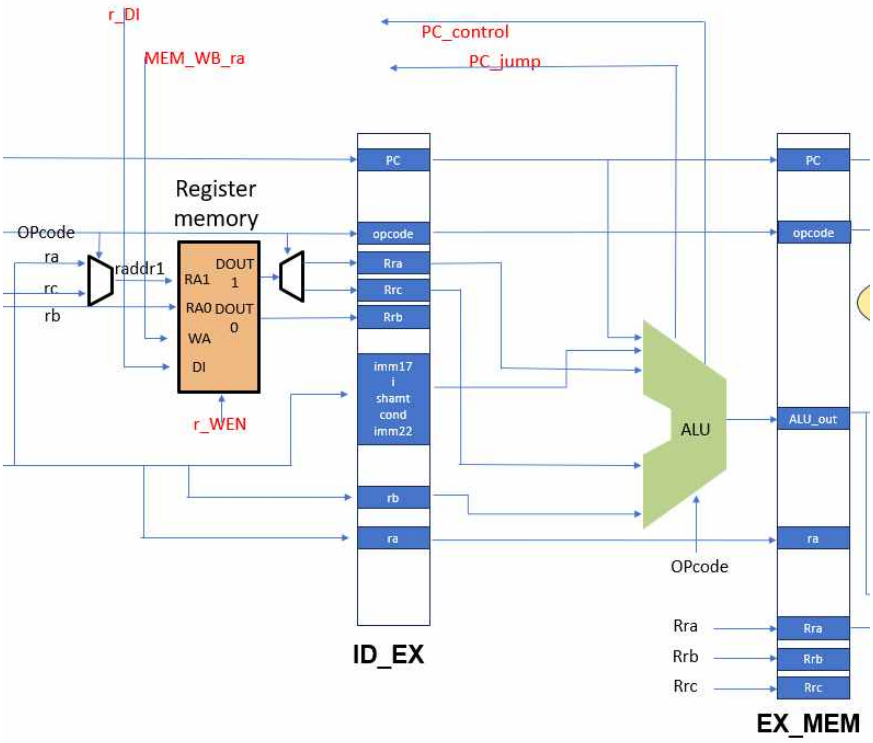
여기서 PC-4를 넘겨주는 이유는 간단하다. BR이 명령어로 들어온다고 가정하자. Inst mem에서 INSTR을 받아오는 동시에 PC는 PC+4로 넘어간다. 이 INSTR이 무슨 명령인지는 다음 cycle에 알 수 있기 때문에 이미 PC에는 저장해야하는 PC보다 4 큰 값이 저장된다. 명령어가 BR인 경우 PC+4값이 Inst mem에 들어가면 안되기에 명령어 Decoding을 통해 얻어진 pc_enable값을 통해 (PC+4)-4를 하여 제대로 된 PC값이 주소로 들어갈 수 있게 컨트롤 해주는 것이다.

-jump control

jump control은 그림이 복잡하여 넣지 않았지만 위와 같이 Instruction Fetech stage에서 IF_ID_opcode를 보고 j인 것을 확인하여 바로 jump를 하였다.

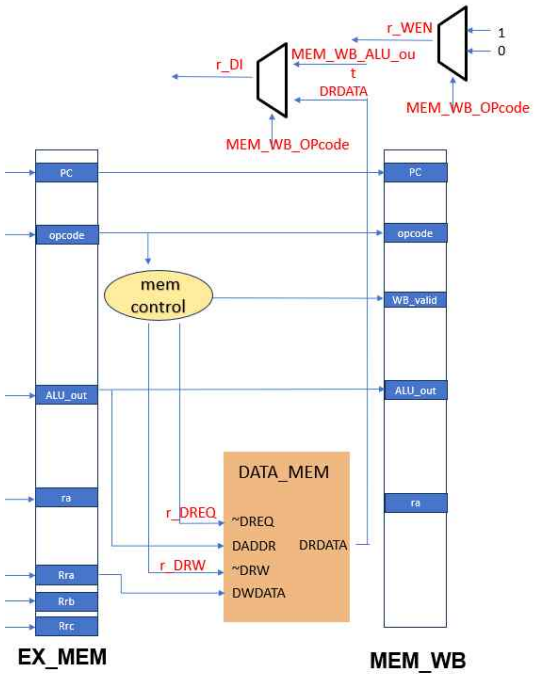
```
reg PC_control_j;
reg[31:0] PC_jump_j;
always@(*)begin
  if((IF_ID_opcode == 5'b10001) || (IF_ID_opcode == 5'b10010)) begin
      PC_control_j = 1;
      PC_jump_j = IF_ID_PC + {{10{IF_ID_imm22[21]}},IF_ID_imm22};
    end
    else begin
       PC_control_j = 0;
    end
end
```

(3) Excute



Excute는 ALU에서 case문을 통해 opcode에 따라 ALU_out을 내보낸다. 또한 위에 설명했듯 BR, BRL, J, JL의 명령어의 경우는 이동할 PC값을 ALU에서 계산해주어 PC_jump로 내보낸다. 이때 ALU에서 이동할 PC값을 보내주는 동시에 이동하라는 신호를 보내줘야하는데 이 신호가 PC_control이다.

(4) Memory && Write back



Data Memory stage의 경우는 OPcode가 st, str, ld, ldr인 경우 데이터 값을 저정하거나, 저장한 데이터 값을 특정한 register로 써야한다. 그러기 위에 OPcode를 보고 DATA_MEM 모듈의 enable과 read/write를 컨트롤 해주고, 또한 ld나 ldr의 경우는 나온 DRDATA값을 register memory에 써줘야한다. 여기서 다른 경우에는 ALU의 결과값인 MEM_WB_ALU_out을 register memory에 쓰게 되는데, 컨트롤을 위해 muxing을 해주어 register memory에 들어갈 데이터인 r_DI를 결정해준다. OPcode에 따라 register mem에 write하거나 read하는데, 여기서 이것을 MEM_WB_OPcode를 보고 r_WEN을 결정해준다.

2. Data Harzard 방지

우선 데이터 해저드를 방지를 완벽히 구현하지는 않았다. BR, J, JL, BRL에 대해서는 구현을 하지 않았다. 또한 그 외에 경우에는 특수한 경우 작동을 하지 않는 경우도 존재하였다. 구현원리를 코드와 함께 설명하겠다.

```
621     ////////////////////////data hazard detection////////////////////////
622     always @(*) begin
623        //ADDI, ANDI, ORI
624        //R[rb] 사용
625        if (IF_ID_opcode == 5'b00000 || IF_ID_opcode == 5'b00001 || IF_ID_opcode == 5'b00010) begin
626           if ((ID_EX_ra == IF_ID_rb) && ID_EX_WB_valid)
627              data_hazard = 1;
628           else if ((EX_MEM_ra == IF_ID_rb) && EX_MEM_WB_valid)
629              data_hazard = 1;
630           else if ((MEM_WB_ra == IF_ID_rb) && MEM_WB_WB_valid)
631              data_hazard = 1;
632           //else if ((WB_ra == IF_ID_rb) && WB_WB_valid)
633           //   data_hazard = 1;
634           else
635              data_hazard = 0;
636
637        end
638        //ADD, SUB, AND, OR, XOR
639        //R[rb], R[rc] 사용
640        else if (IF_ID_opcode == 5'b00100 || IF_ID_opcode == 5'b00101 || IF_ID_opcode == 5'b01000 || IF_ID_opcode == 5'b01001 || IF_ID_opc
641           if ((ID_EX_ra == IF_ID_raddr1 || ID_EX_ra == IF_ID_rb) && ID_EX_WB_valid)
642              data_hazard = 1;
643           else if ((EX_MEM_ra == IF_ID_raddr1 || EX_MEM_ra == IF_ID_rb) && EX_MEM_WB_valid)
644              data_hazard = 1;
645           else if ((MEM_WB_ra == IF_ID_raddr1 || MEM_WB_ra == IF_ID_rb) && MEM_WB_WB_valid)
646              data_hazard = 1;
647           //else if ((WB_ra == IF_ID_raddr1 || WB_ra == IF_ID_rb) && WB_WB_valid)
648           //   data_hazard = 1;
649           else
650              data_hazard = 0;
651        end
652
```

IF_ID_OPcode를 보고, IF_ID_rb, ra, rc(우리가 data 꺼낼 register주소) 값이 전에 들어간 ra(데이터가 저장될 목적지)와 같으면 data_hazard 신호를 켜줘서 멈추는 식으로 구현하였다. 그러나 무조건 꺼낼 data 주소와 데이터가 저장될 목적지가 같다고 hazard 상태가 아닌데, instruction에 따라 그 주소에 저장이 안되는 경우가 존재하기 때문이다. 그 경우는 BR, J, ST, STR이다. 이런 경우에는 사실상 ra, rb, rc값에 상관없이 hazard condition이 아니다. 이 경우를 나타내기 위해 MEM_WB_valid와 같이 (pipline stage_WB_valid)로 신호를 뽑아 컨트롤 하였다.

```
always@(posedge CLK, negedge RSTN) begin
    if(!RSTN)
        PC <= 32'b0;
    else if(PC_enable)
        PC <= PC_in;
    else
        PC <= PC;
end

assign IADDR = PC_enable ? PC[29:0]: PC[29:0] -4;
//wire[31:0] w_PC = PC + 4;
wire[31:0] w_PC = PC +4;
assign PC_in = PC_control ? PC_jump : w_PC;


always@(*) begin
if(pedge_PC_stop || hazard)
  PC_enable = 0;
else
  PC_enable = 1;
end
```

또한 hazard(data_hazard와 같은 신호)를 보고 PC_enable을 키고, PC_enable이 주소값과 PC값을 컨트롤하여 STALL상태를 만들어준다. 그러나 가끔 Data hazard condition을 over detection하는 경우가 존재한다. 혹여나의 사태를 위해 data hazard 구현된 코드와 구현되지 않은 코드 둘 다를 첨부하겠다. 두 개 코드의 차이는 사실 단순하다.

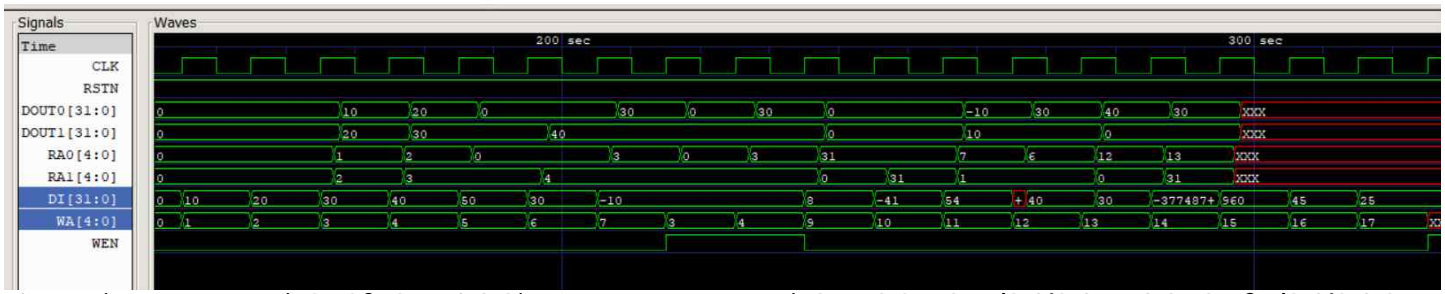
```
87  always@(*) begin
88  if(pedge_PC_stop || hazard)
89      PC_enable = 0;
90  else
91      PC_enable = 1;
92  end
```

```
87  always@(*) begin
88  if(pedge_PC_stop)
89      PC_enable = 0;
90  else
91      PC_enable = 1;
92  end
```

왼쪽은 hazard가 구현된 코드이다. 이 코드에서 88번째줄 if문 조건에서 hazard 변수를 제거하면, hazard를 고려하지 않은 risc processor가 된다.

3. No hazard condition에서의 동작 확인 및 Testbench(BR, J아닌 경우)

PC	inst.hex	코드 설명	추가 설명
0	18000000	00011_00000_00000_00000_00000_00000_00 movi 0 0	
4	1840000A	00011_00001_00000_00000_00000_00010_10 movi 1 10	
8	18800014	00011_00010_00000_00000_00000_00101_00 movi 2 20	
12	18C0001E	00011_00011_00000_00000_00000_00111_10 movi 3 30	
16	19000028	00011_00100_00000_00000_00000_01010_00 movi 4 40	
20	19400032	00011_00101_00000_00000_00000_01100_10 movi 5 50	
24	21822000	00100_00110_00001_00010_00000_00000_00 add 6 1 2	reg[6]=30
28	29C43000	00101_00111_00010_00011_00000_00000_00 sub 7 2 3	reg[7]=-10
32	A8C00008	10101_00011_00000_00000_00000_00010_00 st 3 8	M[imm17]=(R[3]=30)
36	B1000004	10110_00100_00000_00000_00000_00001_00 str 4 8	M[36+4]=R[4]=40
40	42464000	01000_01001_00011_00100_00000_00000_00 and 9 3 4	reg[9] = 8
44	3A804000	00111_01010_00000_00100_00000_00000_00 not 10 4	reg[10] = -41
48	52C64000	01010_01011_00011_00100_00000_00000_00 xor 11 3 4	reg[11] = 54
52	9B3E0028	10011_01100_11111_00000_00000_01010_00 ld 12 40	reg[12]= M[40]=40
56	A37FFFD0	10100_01101_11111_11111_11111_10100_00 ldr 13 -48	reg[13]= M[8]= M[56-48]=30
60	738E1025	01110_01110_00111_00001_00000_01001_01 ror 14 7 1	reg[14] = {R7[10:0],R7[31:11]}
64	6BCC1005	01101_01111_00110_00001_00000_00001_01 shl 15 6 5	reg[15] = {R6[31:6],{00000}
68	04180005	00000_10000_01100_00000_00000_00001_01 addi 16 12 5	reg[16] = reg[12] + 5 = 45
72	045BFFFB	00000_10001_01101_11111_11111_11110_11 addi 17 13 (-5)	reg[17] = reg[13] - 5 = 25



각 REG(WA, decimal)에 대응되는 연산값(DI, signed decimal)이 들어가는지를 확인하였고 정상 작동을 확인하였다. 예시로, add 6 1 2는 1(10)과 2(20)이 더해진 (30)이 6에 저장되는걸 확인할 수 있다. and, not, xor, ror, shl같은 경우 binary로 결과를 확인하였다.

and : 00000_00000_00000_00000_00000_00111_10 & 00000_00000_00000_00000_00000_01010_00
= 00000_00000_00000_00000_00000_00010_00

not : ~00000_00000_00000_00000_00000_01010_00
=11111_11111_11111_11111_11111_10101_11

xor : 00000_00000_00000_00000_00000_00111_10 ^ 00000_00000_00000_00000_00000_01010_00
= 00000_00000_00000_00000_00000_01101_10

ror : I=1이므로, R[rc]=R[1]의 하위 5bit, 즉 10만큼 rotate right를 하게된다. 11111_11111_11111_11111_11111_11101_10을 10만큼 roate right하면, 11111_10110_11111_11111_11111_11111_11이 된다.

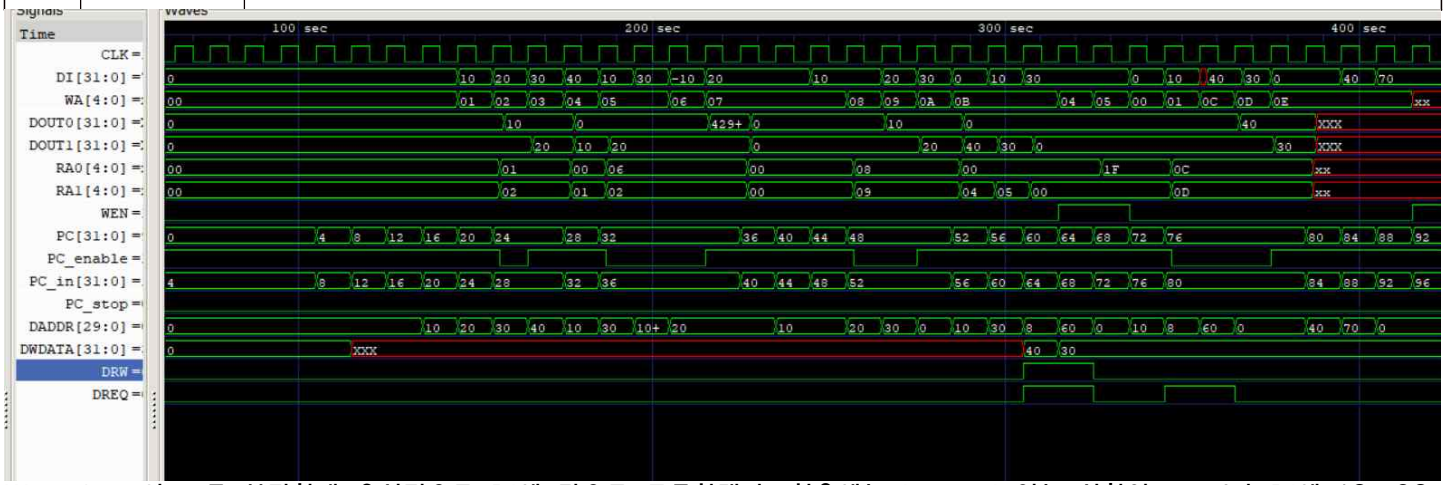
shl : I=0이므로, shamt, 즉 5(00101)만큼 shift left를 하게된다. 00000_00000_00000_00000_00000_00111_10을 5만큼 shift left하면, 00000_00000_00000_00000_00111_10000_00이 된다.

st : 이 instruction에서 M[imm17]= R[ra]이므로, M[8]의 memory에 R[3]값인 30을 저장한다.

str : 이 instruction에서 M[current PC + imm22]= R[ra]이고, current PC는 36이므로 M[36+4]의 memory에 R[4]값인 40을 저장한다.
ld : 이 instruction에서 R[12]의 레지스터에 M[imm17]=M[40]의 값, 즉 40을 load한다.
ldr : 이 instruction에서 R[13]= M[current PC + imm22]이고, current PC는 56, signExt(imm22) = −48이므로, R[13]에 M[8], 즉 30을 load한다.
최종적으로 마지막에서 addi instruction을 통해, 각 st, str, ld, ldr 동작이 잘되었는지 확인하였다.
ld에서 R[12]에 40이 load되었으므로, R[16]=40+5=45가 되고, ldr에서 R[13]에 30이 load되었으므로, R[17]=30−5=25가 된다.

4. Data hazard without JUMP

PC	inst.hex	코드 설명	추가 설명
0	18000000	00011_00000_00000_00000_00000_00000_00	movi 0 0
4	1840000A	00011_00001_00000_00000_00000_00010_10	movi 1 10
8	18800014	00011_00010_00000_00000_00000_00101_00	movi 2 20
12	18C0001E	00011_00011_00000_00000_00000_00111_10	movi 3 30
16	19000028	00011_00100_00000_00000_00000_01010_00	movi 4 40
20	21422000	00100_00101_00001_00010_00000_00000_00	add 5 1 2 (hazard 1) reg[5] = 30
24	29801000	00101_00110_00000_00001_00000_00000_00	sub 6 0 1 (no hazard) (−10)
28	21CC2000	00100_00111_00110_00010_00000_00000_00	add 7 6 2 (hazard 3) reg[7] = 10
32	1A00000A	00011_01000_00000_00000_00000_00010_10	movi 8 10
36	1A400014	00011_01001_00000_00000_00000_00101_00	movi 9 20
40	1A80001E	00011_01010_00000_00000_00000_00111_10	movi 10 30
44	22D09000	00100_01011_01000_01001_00000_00000_00	add 11 8 9 (hazard 2) reg[11] = 30
48	A9000008	10101_00100_00000_00000_00000_00010_00	st 4 8 (M[imm22]=R[4]) (M[8]=R[4]=40)
52	B1400008	10110_00101_00000_00000_00000_00010_00	str 5 8 (M[52 + 8]=R[5]=30)
56	18000000	00011_00000_00000_00000_00000_00000_00	movi 0 0
60	1840000A	00011_00001_00000_00000_00000_00010_10	movi 1 10
64	9B3E0008	10011_01100_11111_00000_00000_00010_00	ld 12 8 R[12]=M[8]=40
68	9B7E003C	10011_01101_11111_00000_00000_01111_00	ld 13 60 R[13]=M[60]=30
72	2398D000	00100_01110_01100_01101_00000_00000_00	add 14 12 13 (hazard 3) reg[14] = 70



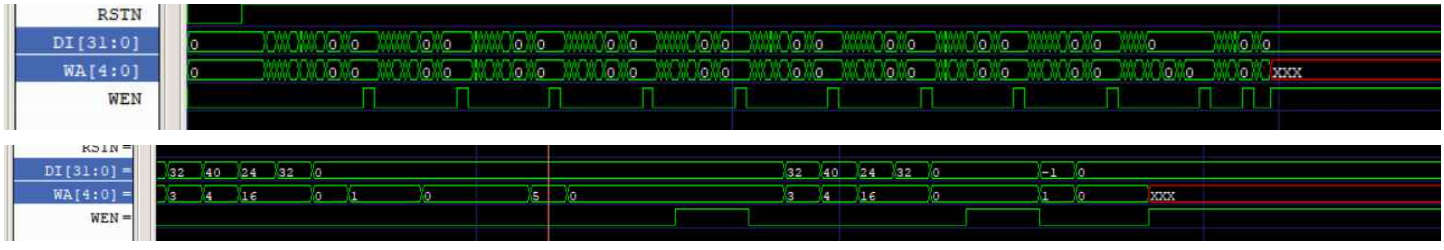
Waveform이 조금 복잡한데 우선적으로 DI에 값으로 주목하겠다. 처음에는 hazard 없는 상황인 movi라 DI에 10, 20, 30, 40이 들어오고 동시에 주소값인 WA는 넣을 reg값으로 변하는 것을 볼 수 있다. hazard 발생 상황인 add 5 1 2을 보면 처음에는 이상한 값인 10이 출력되는데 1cycle 후에 30이 들어온다. 그 와중에 주소값이 변하지 않기 때문에 쓰레기값이 DI에 들어오더라도 register memory에 최종으로 저장되는 것은 정상적인 값이 저장되는 것을 알 수 있다. 그 다음에는

-10이 들어오고, add 7 6 2이 실행되면 hazard 상황으로 3번 stall되다가 정상적인 값인 10으로 변하고, stall이 풀린다. 또한 계속 10, 20, 30이 들어오는 것을 볼 수 있고, add 11 8 9인 경우, 또 hazard 발생한는 것을 확인할 수 있다. stall이 될 때 WA가 변하지 않는 것을 확인할 수 있고, 또한 ST될 때 Data memory의 input값인 DWDATA 또한 파형으로 확인하고, 그때 Write enable인 DRW와 chip selection인 DREQ가 켜지는 것을 확인할 수 있다. 또한 뒤에 LD instruction의 경우는 DRW는 켜지지 않고, DREQ만 켜져서 read하는 경우이고 그때 memory에서 값을 읽어 DI값에 40, 30이 나오는 것을 확인할 수 있다.

5. Jump, branch 구현 및 Testbench

PC	inst.hex	코드 설명	추가 설명
0	18000000	00011_00000_00000_00000_00000_00000_00 movi 0 0	
4	1840000A	00011_00001_00000_00000_00000_00010_10 movi 1 10	
8	1940000A	00011_00101_00000_00000_00000_00010_10 movi 5 10	
12	18800028	00011_00010_00000_00000_00000_01010_00 movi 2 40	
16	18C00020	00011_00011_00000_00000_00000_01000_00 movi 3 32	
20	19000028	00011_00100_00000_00000_00000_01010_00 movi 4 40	
24	19000028	00011_00100_00000_00000_00000_01010_00 movi 4 40	
28	8404502B	10000_10000_00010_00101_00000_01010_11 BRL 16 2 5	cond = 3, if(R[5]!=0) PC=R[rb=2]=40
32	18000000	00011_00000_00000_00000_00000_00000_00 movi 0 0	
36	88000024	10001_00000_00000_00000_00000_01001_00 j current PC + 36	PC=current PC + 40 = 36+40 = 76
40	004BFFFF	00000_00001_00101_11111_11111_11111_11 addi 1 5 (-1)	R[1] = R[5] -1
44	18000000	00011_00000_00000_00000_00000_00000_00 movi 0 0	
48	18000000	00011_00000_00000_00000_00000_00000_00 movi 0 0	
52	18000000	00011_00000_00000_00000_00000_00000_00 movi 0 0	R[5] = R[1]
56	01420000	00000_00101_00001_00000_00000_00000_00 addi 5 1 0	
60	18000000	00011_00000_00000_00000_00000_00000_00 movi 0 0	
64	18000000	00011_00000_00000_00000_00000_00000_00 movi 0 0	
68	18000000	00011_00000_00000_00000_00000_00000_00 movi 0 0	
72	883FFFCC	10001_00000_11111_11111_11111_10011_00 j current PC + (-52)	PC=current PC +(-52)
76	18000000	00011_00000_00000_00000_00000_00000_00 movi 0 0	

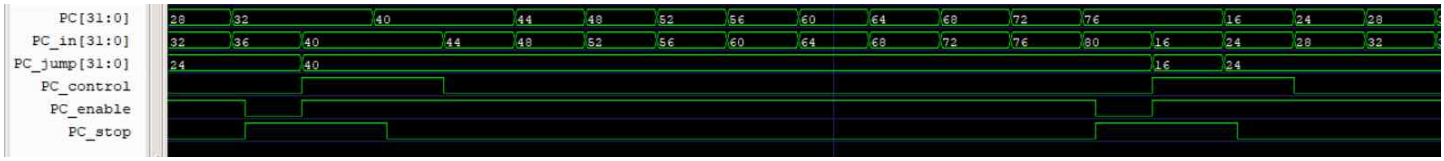
위 instruction은 data hazard가 없는 조건에서, R[5]가 10에서 0이 될 때까지 -1을 반복하다 R[5]가 0이 되면 종료하게 구성한 instruction이다. 아래 DI, WA signal을 보면 addi 1 5 (-1) 동작을 11번 반복하고, R[5]=0이 된 후 반복이 종료된 것을 확인할 수 있다.



BRL 16 2 5는 if(R[5]!=0) PC=R[rb=2]=40이므로, R[5]가 0이 아니면, R[1] = R[5] -1 동작으로 jump한다.
j (current PC + 36)는 branch 동작에서 Jump 동작을 하지 않는 경우인 (R[5]=0)일 때, PC = 76인 마지막

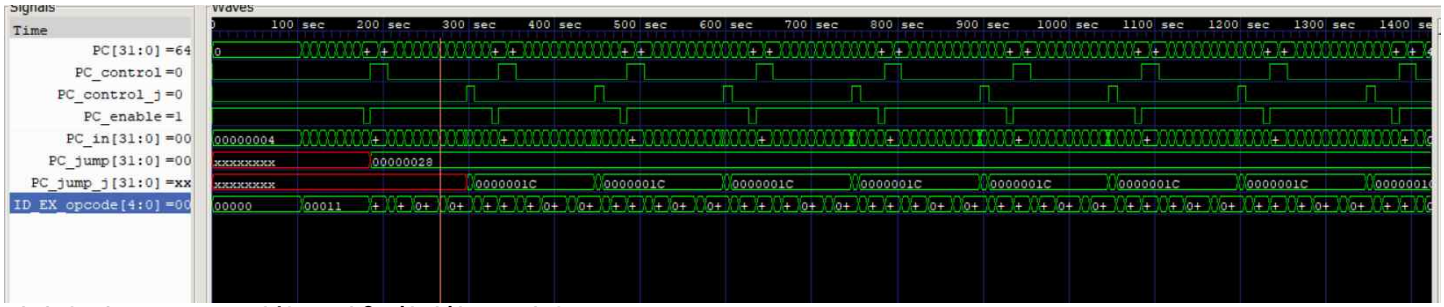
instruction으로 Jump한다.

j (current PC + (−52))는 R[1] = R[5] −1과 R[5] = R[1] 연산 후, 다시 Branch로 돌아가 확인하기 위해, PC = current PC −52로 (movi 4 40)로 Jump한다.



branch와 jump의 한 cycle에 대한 결과이다. PC=32일 때, 즉 current PC = 28인 Branch instruction에서 PC = 40으로 jump한다. PC+4씩 커지며 각 연산을 하다, PC=76일 때, 즉 current PC = 72인 Jump instruction PC = current PC −52 = 20, 즉 PC=24의 movi로 Jump하게 된다.

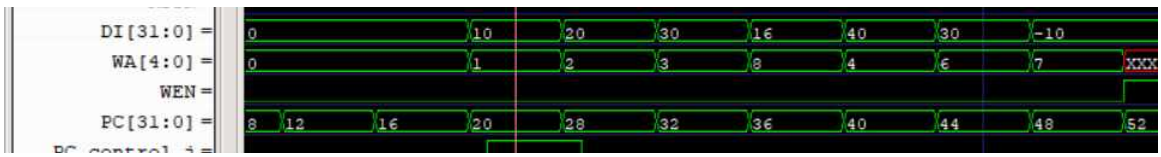
또한 위와 같이 반복문을 만들었을 때 계속 작동을 하는 것을 알 수 있다.



이렇게 반복문으로 동작하는 것을 확인할 수 있다.

6. JL(jump and link)

PC	inst.hex	코드 설명	추가 설명
0	18000000	00011_00000_00000_00000_00000_00000_00 movi 0 0	
4	1840000A	00011_00001_00000_00000_00000_00010_10 movi 1 10	
8	18800014	00011_00010_00000_00000_00000_00101_00 movi 2 20`	
12	18C0001E	00011_00011_00000_00000_00000_00111_10 movi 3 30	
16	92000008	10010_01000_00000_00000_00000_00010_00 jl 8 8	R[8] = current PC, PC=currentPC +8
20	19000028	00011_00100_00000_00000_00000_01010_00 movi 4 40	
24	19400032	00011_00101_00000_00000_00000_01100_10 movi 5 50	
28	21822000	00100_00110_00001_00010_00000_00000_00 add 6 1 2	
32	29C43000	00101_00111_00010_00011_00000_00000_00 sub 7 2 3	



JL 명령에서의 동작을 확인하였다. R[8]에 current PC인 16이 저장되는걸 확인하였다. 그리고 PC값을 확인하였을 때, PC =20에서 PC=28(20+8)로 Jump하는 것을 확인하였다.