REPORT

전자공학도의 윤리 강령 (IEEE Code of Ethics)

(출처: http://www.ieee.org)

나는 전자공학도로서, 전자공학이 전 세계 인류의 삶에 끼치는 심대한 영향을 인식하여 우리의 직업, 동료와 사회에 대한 나의 의무를 짐에 있어 최고의 윤리적, 전문적 행위를 수행할 것을 다짐하면서, 다음에 동의한다.

1. **공중의 안전, 건강 복리에 대한 책임**: 공중의 안전, 건강, 복리에 부합하는 결정을 할 책임을 질 것이며, 공중 또는 환경을 위협할 수 있는 요인을 신속히 공개한다.
2. **지위 남용 배제**: 실존하거나 예기되는 이해 상충을 가능한 한 피하며, 실제로 이해가 상충할 때에는 이를 이해 관련 당사자에게 알린다. (이해 상충: conflicts of interest, 공적인 지위를 사적 이익에 남용할 가능성)
3. **정직성**: 청구 또는 견적을 함에 있어 입수 가능한 자료에 근거하여 정직하고 현실적으로 한다.
4. **뇌물 수수 금지**: 어떠한 형태의 뇌물도 거절한다.
5. **기술의 영향력 이해**: 기술과 기술의 적절한 응용 및 잠재적 영향에 대한 이해를 높인다.
6. **자기계발 및 책무성**: 기술적 능력을 유지, 증진하며, 훈련 또는 경험을 통하여 자격이 있는 경우이거나 관련 한계를 전부 밝힌 뒤에만 타인을 위한 기술 업무를 수행한다.
7. **엔지니어로서의 자세**: 기술상의 업무에 대한 솔직한 비평을 구하고, 수용하고, 제공하며, 오류를 인정하고 수정하며, 타인의 기여를 적절히 인정한다.
8. **차별 안하기**: 인종, 종교, 성별, 장애, 연령, 출신국 등의 요인에 관계없이 모든 사람을 공평하게 대한다.
9. **도덕성**: 허위 또는 악의적인 행위로 타인, 타인의 재산, 명예, 또는 취업에 해를 끼치지 않는다.
10. **동료애**: 동료와 협력자가 전문분야에서 발전하도록 도우며, 이 윤리 헌장을 준수하도록 지원한다.

위 IEEE 윤리헌장 정신에 입각하여 report를 작성하였음을 서약합니다.

학 부: 전자공학부
마감일: 2021. 05. 11.
과목명: 디지털시스템설계
교수명: 양회석 교수님
학 번: 201620837
성 명: 안민규

Homework 2. Implementation of A Simplified MIPS Datapath

1. Overview of design

이번 과제는 여러 모듈을 통해 간단한 MIPS를 구현하고 이의 Datapath를 이해하는 과제이다. 따라서 MIPS에 대한 이해가 필요했다. MIPS는 Instruction memory를 통해 명령어를 가져와 해독하고, ALU를 이용해 연산을 진행한다. 임시공간 Context에 저장된 피연산자들을 이용한다. 이러한 구조에서 이번 과제에서는 Instruction memory와 ALU를 구현하는 것이 중점이었다. 그러기 위해 기본적으로 구현되어야 하는 모듈들이 여러 가지 있었다. MUX, decoder, FA, ripple adder, ALU 등이 이에 속한다. 기본적인 이러한 모듈들은 전반적인 프로그램이 동작하는 것에 상관없이 기능해야하는 모듈들이므로 이를 먼저 설계하였다.

MUX의 경우 behavioral 구조로 구현을 하였다. 다른 모듈에서 mux를 불러오면 always 구문이 실행되게 구현하였다. MUX의 경우 SEL 신호에 맞추어 이에 맞는 값들을 저장하는 모듈이다. 따라서 en과 sel 신호를 가지고 제어를 하였고, 우선적으로 en이 0일 때 출력값은 0으로 default값을 지정하였다. en이 1일 때는 j라는 정수가 sel신호와 비교를 하며 sel신호와 같을 때 이를 출력값에 저장을 하는 식으로 진행하였다. M의 비트 수를 가지는 2^S개의 인풋이 입력이 되는데 SEL신호의 값에 따라 2^S개 중 하나를 선택해 이를 출력 Y에 저장하는 식으로 구성을 하였다. 이 때 array index와 vector index에 유의하며 대입하였다.

dec 또한 behavioral 구조로 구현을 하였다. dec의 경우 따로 sel 신호가 필요하지 않기 때문에 en신호에 맞추어 진행을 하였다. 기본적으로 0으로 default값을 지정하였고, en신호가 1이 되면 인풋의 값에 맞는 Y[i]에 1을 asserted하고 나머지는 0으로 선언하였다. dec의 경우 인풋이 S의 길이를 가지며 S가 3일 때 인풋이 3'b100이면 4이므로 Y[4]만 1이 되고 나머지 Y[i]는 모두 0이 된다.

FA의 경우 structural 구조로 구현하였다. 앞서 학습한 이론에 따라 FA를 AND게이트와 OR게이트를 이용해 구현하였다.

ripple adder는 이러한 FA가 연결된 구조로 첫 FA에서 나온 COUT이 다음 FA의 CIN으로 들어간다는 사실에 유의하며 구현하였다. structural 구조로 구현을 하였으므로, always 구문을 사용할 수 없어 일반적인 변수를 사용하는 looping statement는 사용할 수 없었다. 따라서 generate를 사용하여 I를 선언해주었고, 이를 가지고 반복문을 진행하였다. 하지만 0부터 M-1까지 M번을 진행하지 않고 M-1번을 진행하였고 마지막에 반복문을 나와 한 번 더 FA를 선언하여 덧셈을 진행하였다. 이는 cin[i+1]=cout[i] 때문이었는데, M-1까지 진행하게 되면 CIN의 비트 수보다 높아지므로 오류가 발생하기 때문이다.

ALU는 ripple adder를 가지고 structural하게 구현하였다. ALU의 경우 입력으로 들어오는 ALUCtrl 신호에 따라 덧셈과 뺄셈을 모두 할 수 있게 구현하는 것이 가장 큰 문제였다. 또한 뺄셈을 진행하고 두 값이 같을 때 zero라는 신호를 asserted 해주는 것이 관건인 모듈이었다. 따라서 zero 신호는 앞서 말한 두 가지 조건을 and로 묶어 두 개가 다 1일 때 1이라는 값이 되게끔 구현하였다. ALU에서 ripple adder를 가지고 뺄셈을 구현하기 위해서는 adder로 들어가는 인풋 중 한 개를 2의 보수화 시켜서 덧셈을 진행하여야 한다. 이러한 2의 보수를 구현하기 위해 인풋 한 개를 inverting 시키고 CIN에 0이 아닌 1을 넣어서 2의 보수를 구현하였다. 이렇게 ripple adder의 결과값은 ALUOut이라는 신호로 내보내 주었다.

이렇게 기본적인 모듈들을 구현해준 이후 이를 이용해 더 큰 모듈들을 구현하였다. 우선적으로 현재 수행할 명령어 주소를 가지고 있는 register인 Program counter(이하 PC)를 설계하였다. PC가 한 칸씩(4byte) 이동하는데 이는 Implementation 8-1로 PC에서 나오는 PC_OUT값에 4를 더하여 이를 다음 PC의 인풋값으로 지정하는 구조를 구현하였다. 하지만 이 값이 바로 PC_IN이 되는 것이 아니라 4-3 MUX를 거쳐 4-3MUX의 sel신호에 맞추어 결정된다. 주어진 여러 신호들을 가지고 현재 PC 값에 4를 더한 값을 4-3MUX의 한 인풋과 8-3 adder의 한 인풋으로 설정하였다. 이후에는 8-3adder의 다른 인

풋과 4-3MUX의 나머지 인풋을 설정을 해주어야했다. MUX의 나머지 인풋의 경우 8-3 adder의 결과값이므로 8-3 adder의 입력을 먼저 결정을 해주어야한다. 이때 먼저 들어간 PC값에 따라 32비트의 instruction 신호가 결정이 되는데 이 신호의 [15:0]부분을 signed extension한 후에 왼쪽으로 2bit만큼 이동한 값을 adder의 인풋으로 설정하였다. adder의 결과값을 mux의 한 인풋으로 설정하고 mux의 결과값을 PC_IN으로 설정하였다. 하지만 MUX의 경우 sel 신호가 중요하기 때문에 이를 위해 instruction 신호에 대해 분석하였다.

현재 PC값에 따라 결정되는 instruction 신호는 [31:26]은 Control을 담당하는 신호로, [25:21]와 [20:16]은 각각 Read register1, 2가 된다. 이 때 [20:16]와 [15:11]은 4-1MUX의 인풋이 된다. Control 신호는 6비트 길이의 instruction[31:26]에 따라 결정이 되게 구현하였는데, 이 부분이 Implementation 10이다. instruction[31:26]에 따라 MIPS의 행동이 달라져야 하므로 instruction[31:26]이 0이면 R-type을, 35 혹은 43일 때는 load,store를, 4이면 branch에 맞는 행동을 하게끔 구현하였다. 교안에 제시되어있는 진리표와 구조를 참고하여 이해를 할 수 있었다.

Control에서 나오는 RegDst신호를 4-1MUX의 sel 신호로 사용을 하였다. RegWrite신호는 Register에 들어가 Write Enable신호로 작동한다. ALUSrc는 4-2MUX의 sel신호로 사용하였다. ALUOp은 ALU control 모듈을 작동시키는 신호로, MemWrite는 data memory를 작동하는 신호이고 또한 4-4MUX의 enable신호로써 작동한다. MemtoReg는 4-4MUX의 sel신호로 사용을 하였고, Branch는 4-3MUX의 sel신호를 결정하기 위한 값으로 사용하였다.

4-1MUX의 sel신호를 결정하고 인풋을 결정하고난 후에 sel신호에 따라 register의 Write register로 들어간다. write register는 register 내부의 dec의 인풋 신호로 사용한다. read data를 구하기 위해 register 파일을 분석하였다. register 내부에서는 Write enable신호로 들어온 reg_write 신호에 따라 dec에서 pre_register_we의 결과값을 낸다. 이러한 결과값과 본래 들어온 write enable신호가 and게이트를 지난 것을 register 내부에서 데이터를 저장할 enable 신호로 사용한다. 또한 write_data를 register 내부에서 data_out로 저장한다. 그 후 이 데이터를 MUX를 이용해 sel신호로 들어오는 read register1,2에 따라 read data1,2로 내보낸다.

이렇게 결정된 read data1은 바로 8-2 main ALU의 인풋으로 들어가고 read data2는 4-2MUX의 인풋으로 들어간다. 4-2MUX의 경우 read data2와 앞서 signed extension한 inst[15:0]을 인풋으로 가지고 ALUSrc를 sel신호로 갖는다. ALUSrc에 따라 아웃풋을 8-2 main ALU의 나머지 인풋으로 결정하였다. 이렇게 인풋을 결정하고 ALU를 실행한다.

ALU의 경우 control 신호에 따라 덧셈을 진행할 건지 뺄셈을 진행할 건지 결정이 되기 때문에 ALU control 모듈의 아웃풋이 ALUcontrol신호로 들어가게끔 설계하였다. ALU control 모듈은 ALUOp에 따라 값이 결정된다.

ALU control 모듈은 참고자료에 제시되어있는 것처럼 제시가 되도록 case 구문을 이용해 구현하였다.

이렇게 ALUOp에 따라 ALU control 값이 결정이 되고 이 값에 따라 ALU가 동작하게 구현하였다. ALU 내부에서는 ALU control이 010이면 덧셈을 110이면 뺄셈을 하게 된다. ALU가 진행되고 나오는 zero 신호가 Branch 신호와 and 게이트를 거쳐 나오는 값이 4-3MUX의 sel 신호로 사용하였다. 이를 위해 branch_and라는 새로운 wire를 선언하였다.

branch_and가 1이 된다는 것은 beq 동작을 하고 read data 2개가 값이 같다는 뜻이다. 이렇게 1이 되면 4-3MUX에서는 PC_out+4한 값이 아닌 8-3 adder를 진행한 결과값을 PC_in으로 연결한다. 8-3의 결과값은 PC+4에 (하위 16비트를 signed extension한 값)*4를 더하는 것이다. 예로 실제 결과사진에서 PC_out이 28일 때 +4를 하여 32가 되고 이에 하위 16비트 값이 -4이기 때문에 이에 4를 곱한 값인 -16을 더하여 다음 PC값이 16이 되는 것을 확인할 수 있다.

8-2 main ALU를 통해 나오는 결과값은 Data memory의 address로 들어가고 4-4MUX의 인풋으로 들어가게 된다. 또한 앞서 read data2는 data memory의 write data로 연결이 된다. memWrite 신호가 0이면 데이터를 read하고 1이면 데이터를 write한다. 따라서 memWrite 신호와 ALU를 거쳐

address로 들어오는 값에 따라 write data를 쓴다. 그 후 memWrite가 1이 되면 이렇게 저장되어 있는 값을 읽는 메커니즘으로 구성이 되어있다. R-type일때는 ALU 결과값이 data memory를 거치지않고 바로 4-4MUX를 지나 register의 write data로 들어간다. lw의 경우 address로 들어가고 이에 따라 read data가 4-4MUX를 지나 register의 write data로 들어간다. memWrite가 1이 되는 sw의 경우 data를 write하기 때문에 4-4MUX에서 나오는 아웃풋이 없다. 이로 인해 memWrite 신호를 4-4MUX의 enable신호로 이용하였다.

2. Verilog code

2-1) MUX (Implementation 1)

```
module Vr_HW2_MUX_S_sel_M_bits(EN, SEL, A, Y);
  parameter S = 4;
  parameter M = 1;

  input EN;
  input [S-1:0] SEL;
  input [M-1:0] A [0:2**S-1]; //32비트의 길이를 가지는 2^S개 입력
  output [M-1:0] Y;
  reg [M-1:0]temp; //Y에 값을 assign 하기 위해 reg 선언

  integer i, j; //반복문을 위한 정수 선언
  /* Implementation 1:
     You need to design a parameterizable active-high M-bit Multiplexer with S-bit SEL
     E.g., when S=4 and M=2, this MUX has 2^S inputs each of which is 2-bit. */
  always @ (*) begin
   if(EN==0) //EN이 0이면 temp에 기본값 32'b0값 대입
    for(i=M-1;i>=0;i=i-1)
     temp[i]=0;
   else //EN이 1이면
    for(j=0;j<2**S;j=j+1) //j는 0부터 2^S-1까지 반복
     if(j=={SEL}) begin //j가 SEL과 같다면
      for(i=0;i<M;i=i+1)
        temp[i]=A[j][i]; //temp에 값 저장 ex) sel=3이면 temp[i]=A[3][i]이 저장됨
     end
  end

  assign Y=temp; //Y에 temp값 저장

endmodule
```

2-2) decoder (Implementation 2)

```
module Vr_HW2_N_to_S_dec(A, EN, Y);
  parameter N=3, S=8;
  input [N-1:0] A;
```

```
  input EN;
  output reg[S-1:0] Y;

  integer i;
  /* Implementation 2:
     You need to design a parameterizable N-to-S binary decoder.
     E.g., when N=3 and S=8, it is a 3-to-8 binary decoder. */
  always @ (A, EN) begin //A, EN이 변할 때 always문 시작
   for (i=0;i<=S-1;i=i+1)
    Y[i]=0; /*default*/ //default값 저장
   if (EN==0) begin/*EN==0*/
    for (i=S-1;i>=0;i=i-1)
     Y[i]=0; //EN이 0일 때 Y값에 0을 저장
   end
   else begin/*EN==1*/
    for (i=0;i<=S-1;i=i+1) begin
     if (i=={A}) //i가 A의 값이랑 같으면
      Y[i]=1; //Y[i]에 1을 저장. 나머지는 0 ex)A가 3이면 Y=32'b00001000 (앞부분 생략)
    end
   end
  end
endmodule
```

2-3) registerfile (Implementation 3)

```
module Vr_HW2_register_file(CLK, RST, RR1, RR2, WR, WD, WE, RD1, RD2);
  input CLK;
  input RST;
  input [4:0] RR1; //read register number1 -> mux의 sel 신호
  input [4:0] RR2; //read register number2 -> mux의 sel 신호
  input [4:0] WR; //write register number -> dec의 sel 신호
  input [31:0] WD; //write data
  input WE;
  output [31:0] RD1; //read data1
  output [31:0] RD2; //read data2



  wire [31:0] pre_register_we; //dec 통해 나오는 enable 신호
  wire [31:0] register_we; //register enable 신호
  reg [31:0] register_data_in[0:31]; //register에 들어갈 데이터
  wire [31:0] register_data_out[0:31]; //register에 들어올 데이터 저장값

  genvar i;
```

```
  /* register write enable signals: use binary decoder */
  /* Implementation 3-1: Decoder for write enable signals */
  Vr_HW2_N_to_S_dec # (.N(5), .S(32)) u_D1 (WR,WE,pre_register_we); //WE를 enable신호로 가지고
  for (i=0; i<=31;i=i+1)                                          //WR에 따라 pre_register_we값 변경
    and A1(register_we[i], pre_register_we[i], WE); //WE과 dec에서 나온 값의 and값 = registerenable
  /* register data_in */
  for(i=0;i<=31;i=i+1) assign register_data_in[i] = WD;

  /* Register instantiations: 32 x 32-bit Registers */
  for(i=0;i<=31;i=i+1)
    Vr_HW2_reg_N #(.N(32), .INIT_VAL(i)) u_regs (CLK, RST, register_we[i], register_data_in[i], regist
er_data_out[i]); //0부터 31까지 enable신호에 따라 값 저장
  /* The above line is equivalent to the following 32 lines:
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(0)) u_reg0  (CLK, RST, register_we[0], register_data_in[0], regis
ter_data_out[0]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(1)) u_reg1  (CLK, RST, register_we[1], register_data_in[1], regis
ter_data_out[1]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(2)) u_reg2  (CLK, RST, register_we[2], register_data_in[2], regis
ter_data_out[2]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(3)) u_reg3  (CLK, RST, register_we[3], register_data_in[3], regis
ter_data_out[3]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(4)) u_reg4  (CLK, RST, register_we[4], register_data_in[4], regis
ter_data_out[4]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(5)) u_reg5  (CLK, RST, register_we[5], register_data_in[5], regis
ter_data_out[5]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(6)) u_reg6  (CLK, RST, register_we[6], register_data_in[6], regis
ter_data_out[6]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(7)) u_reg7  (CLK, RST, register_we[7], register_data_in[7], regis
ter_data_out[7]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(8)) u_reg8  (CLK, RST, register_we[8], register_data_in[8], regis
ter_data_out[8]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(9)) u_reg9  (CLK, RST, register_we[9], register_data_in[9], regis
ter_data_out[9]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(10)) u_reg10 (CLK, RST, register_we[10], register_data_in[10], regi
ster_data_out[10]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(11)) u_reg11 (CLK, RST, register_we[11], register_data_in[11], regi
ster_data_out[11]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(12)) u_reg12 (CLK, RST, register_we[12], register_data_in[12], regi
ster_data_out[12]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(13)) u_reg13 (CLK, RST, register_we[13], register_data_in[13], regi
ster_data_out[13]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(14)) u_reg14 (CLK, RST, register_we[14], register_data_in[14], regi
ster_data_out[14]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(15)) u_reg15 (CLK, RST, register_we[15], register_data_in[15], regi
```

```
ster_data_out[15]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(16)) u_reg16 (CLK, RST, register_we[16], register_data_in[16], regi
ster_data_out[16]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(17)) u_reg17 (CLK, RST, register_we[17], register_data_in[17], regi
ster_data_out[17]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(18)) u_reg18 (CLK, RST, register_we[18], register_data_in[18], regi
ster_data_out[18]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(19)) u_reg19 (CLK, RST, register_we[19], register_data_in[19], regi
ster_data_out[19]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(20)) u_reg20 (CLK, RST, register_we[20], register_data_in[20], regi
ster_data_out[20]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(21)) u_reg21 (CLK, RST, register_we[21], register_data_in[21], regi
ster_data_out[21]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(22)) u_reg22 (CLK, RST, register_we[22], register_data_in[22], regi
ster_data_out[22]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(23)) u_reg23 (CLK, RST, register_we[23], register_data_in[23], regi
ster_data_out[23]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(24)) u_reg24 (CLK, RST, register_we[24], register_data_in[24], regi
ster_data_out[24]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(25)) u_reg25 (CLK, RST, register_we[25], register_data_in[25], regi
ster_data_out[25]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(26)) u_reg26 (CLK, RST, register_we[26], register_data_in[26], regi
ster_data_out[26]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(27)) u_reg27 (CLK, RST, register_we[27], register_data_in[27], regi
ster_data_out[27]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(28)) u_reg28 (CLK, RST, register_we[28], register_data_in[28], regi
ster_data_out[28]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(29)) u_reg29 (CLK, RST, register_we[29], register_data_in[29], regi
ster_data_out[29]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(30)) u_reg30 (CLK, RST, register_we[30], register_data_in[30], regi
ster_data_out[30]);
  Vr_HW2_reg_N #(.N(32), .INIT_VAL(31)) u_reg31 (CLK, RST, register_we[31], register_data_in[31], regi
ster_data_out[31]);*/



  /* MUX: data_out -> RD1/RD2 */
  /* Implementation 3-2: Two MUXes for two read register values */
  Vr_HW2_MUX_S_sel_M_bits #(.S(5), .M(32)) u_M1(1'b1,RR1,register_data_out,RD1);
  Vr_HW2_MUX_S_sel_M_bits #(.S(5), .M(32)) u_M2(1'b1,RR2,register_data_out,RD2);
endmodule //read register값을 sel신호로 받아 register_data_out을 read data에 씀
```

2-4) FA (Implementation 5)

```
module Vr_HW2_FA(A, B, CIN, S, COUT);
  input A, B, CIN;
```

```
  output S, COUT;

  /* Implementation 5: complete the design of a full adder */
  wire L1,L2,L3,L4,L5,L6,L7,L8,L9,L10;
  not U1(L1,A);
  not U2(L2,B);
  not U3(L3,CIN);
  and U4(L4,A,L2,L3);
  and U5(L5,L1,B,L3);
  and U6(L6,L1,L2,CIN);
  and U7(L7,A,B,CIN);
  or U8(S,L4,L5,L6,L7); //S=ABCIN+A'BCIN+AB'CIN+ABCIN'

  and U9(L8,A,B);
  and U10(L9,A,CIN);
  and U11(L10,B,CIN);
  or U12(COUT,L8,L9,L10); //COUT=AB+ACIN+BCIN
endmodule
```

2-5) ripple adder (Implementation 6)

```
module Vr_HW2_ripple_adder_M_bits(A, B, CIN, S, COUT);
  parameter M=32;
  input [M-1:0] A, B;
  input CIN; /* CIN_0 for LSB */
  output [M-1:0] S;
  output COUT; /* COUT_M-1 for MSB */

  genvar i;
  wire [M-1:0] sig_cin, sig_s, sig_cout;

  /* Implementation 6: complete the design of an M-bit parameterizable ripple adder
     using full adders (structural design) */
  assign sig_cin[0]=CIN; //sig_cin[0]에 CIN 연결
  assign COUT=sig_cout[M-1]; //COUT에 sig_cout[M-1] 연결
  assign S=sig_s; //출력값 S에 sig_s 연결

  for(i=0;i<M-1;i=i+1) begin
    assign sig_cin[i+1]=sig_cout[i]; //cout값이 다음 cin값으로 들어가게끔 연결
    Vr_HW2_FA U1 (A[i],B[i], sig_cin[i], sig_s[i], sig_cout[i]); //M-2까지 진행
  end
  Vr_HW2_FA U2 (A[M-1],B[M-1],sig_cin[M-1],sig_s[M-1],sig_cout[M-1]); //마지막 덧셈 진행
endmodule
```

2-6) ALU (Implementation 7)

```
module Vr_HW2_ALU (A, B, ALUCtrl, ALUOut, Zero);
```

```
  input [31:0] A;
  input [31:0] B;
  input [2:0] ALUCtrl;
  output [31:0] ALUOut;
  output Zero;

  wire [31:0] sig_a;
  wire [31:0] sig_b;
  wire [31:0] sig_sum;
  wire sig_cin;
  wire sig_cout;

  /* Implementation 7: ALU design using only 1 ripple adder instance */
  /* ALU Control - 010: add, 110: sub */
  assign sig_a=A; //sig_a에 인풋 A 연결
  assign ALUOut=sig_sum; //ALUOUT에 ripple adder 진행하여 나오는 sum값 연결
  assign Zero=(ALUCtrl==3'b110)&(sig_sum==0); //ALUCtrl이 3'b110이고 sum이 0 일 때 zero
  assign sig_b=(ALUCtrl==3'b110) ? ~B:B; //ALUCtrl이 110일 때 B를 inverting 시켜서 연결 아니라면
B
  assign sig_cin=(ALUCtrl==3'b110) ? 1'b1:1'b0; //ALUCtrl이 110일 때 cin은 1'b1(앞서 inverting한 B
에 1을 더해줘야 정확한 보수값이 됨)

  Vr_HW2_ripple_adder_M_bits #(.M(32)) u_adder (sig_a,sig_b,sig_cin,sig_sum,sig_cout); //ripple adder
진행
endmodule
```

2-7) Control (Implementation 10)

```
module Vr_HW2_control (op, RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, Re
gWrite);
  input [5:0] op;
  output RegDst, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite;
  output [1:0] ALUOp;

  wire sig_r_format, sig_lw, sig_sw, sig_beq;

  /* Implementation 10: fill in your implementation here */
  wire L0,L1,L2,L3,L4,L5;
  not n1 (L5,op[5]);
  not n2 (L4,op[4]);
  not n3 (L3,op[3]);
  not n4 (L2,op[2]);
  not n5 (L1,op[1]);
  not n6 (L0,op[0]);
```

```
  and a1 (sig_r_format,L0,L1,L2,L3,L4,L5); //모든 op의 not의 and
  and a2 (sig_lw,op[0],op[1],L2,L3,L4,op[5]); //op0,1,5 시그널 제외 나머지 not and
  and a3 (sig_sw,op[0],op[1],L2,op[3],L4,op[5]); //op0,1,3,5 시그널 제외 나머지 not and
  and a4 (sig_beq,L0,L1,op[2],L3,L4,L5); //op2 시그널 제외 나머지 not and
  or a5 (ALUSrc, sig_lw, sig_sw);
  or a6 (RegWrite, sig_r_format, sig_lw);
  assign RegDst=sig_r_format;
  assign MemRead=sig_lw;
  assign MemtoReg=sig_lw;
  assign MemWrite=sig_sw;
  assign Branch=sig_beq;
  assign ALUOp[1]=sig_r_format;
  assign ALUOp[0]=sig_beq;
endmodule
```

2-8) ALU Control (Implementation 11)

```
module Vr_HW2_ALU_control(funct, ALUOp, ALUCtrl);

  input [5:0] funct;
  input [1:0] ALUOp;
  output reg [2:0] ALUCtrl;

  /* Implementation 11: fill in your implmentation here */
  always @ (ALUOp, funct) begin //ALUOp, funct가 변하면 always문 시작
   case(ALUOp)
    2'b00:ALUCtrl=3'b010; //ALUOp가 2'b00일 때 ALUCtrl=3'b010
    2'b01:ALUCtrl=3'b110; //ALUOp가 2'b01일 때 ALUCtrl=3'b110
    2'b10:if(funct==6'b100000) ALUCtrl=3'b010; //ALUOp가 2'b10이고, funct가 6'b100000일 때 ALUC
trl=3'b010
          else if(funct==6'b100010) ALUCtrl=3'b110; //ALUOp가 2'b10이고, funct가 6'b100010일 때
ALUCtrl=3'b110
   endcase
  end
endmodule
```

2-9) HW2 (Implementation 4, 8, 9)

```
module Vr_HW2(CLK, RST);

  input CLK, RST;


  wire [31:0] sig_pc_in, sig_pc_out; /* input/output of PC */
  wire [31:0] sig_pc_plus_4; //8-1 adder 결과값
  wire sig_pc_plus_4_cout; //8-1 adder cout값
```

```
  wire [31:0] sig_branch_addr; //8-3 adder 결과값
  wire sig_branch_cout; //8-3 adder cout값
  wire [31:0] sig_extended_offset; //signed extended한 결과값 (9)

  wire branch_and; /*andgate output*/
  wire [31:0] inst; /* instruction */

  /* for register file */
  wire [4:0] sig_rr1, sig_rr2, sig_wr;
  wire [4:0] sig_wr_mux_in [0:1]; //4-1 mux의 input
  wire [31:0] sig_rf_write_data; //register의 write data에 쓰이는 데이터
  //wire sig_rf_we;
  wire [31:0] sig_rf_rd1, sig_rf_rd2; //read data 2개

  wire [31:0] sig_alu_a, sig_alu_b, sig_alu_out; //8-2 ALU의 input과 output

  wire sig_alu_zero; //8-2 ALU의 결과값이 zero일 때 보내는 신호

  wire [31:0] sig_data_mem_out; //data memory의 output

  /* control signals */
  wire sig_reg_dst, sig_branch, sig_mem_read, sig_mem_to_reg, sig_mem_write, sig_alu_src, sig_reg_wr
ite; //inst[31:26]에 의해 결정되는 control 신호들
  wire [1:0] sig_alu_op; //ALU Contorl을 조작하는 operation 값
  wire [2:0] sig_alu_ctrl; //ALU Control 모듈에서 나오는 output

  wire [31:0] sig_pc_mux_in [0:1]; //4-3 mux의 input
  wire [31:0] sig_alu_b_mux_in [0:1]; //4-2 mux의 input
  wire [31:0] sig_rf_wd_mux_in [0:1]; //4-4 mux의 input

  /* program counter */
  Vr_HW2_reg_N #(.N(32)) u_pc (CLK, RST, 1'b1, sig_pc_in, sig_pc_out);  //CLK신호가 올 때 마다 pc_i
n 값을 pc_out에 저장

  /* PC + 4 */
  /* Implementation 8-1: PC+4 (instantiate the ripple adder implemented in Vr_HW2_ripple_adder_
M_bits.v */
  Vr_HW2_ripple_adder_M_bits #(.M(32)) u_ad1 (sig_pc_out, 32'b0100, 1'b0, sig_pc_plus_4, sig_pc_plu
s_4_cout); //pc_out값에 4를 더해줌 (8-1)



  /* instruction memory */
  Vr_HW2_inst_mem u_inst_m (sig_pc_out, inst); //instruction memory에 들어온 pc_out에 따라 inst93
1:0] 값 결정
```

```
  /* Implementation 4-1: top-level MUX 1 (given as an example; you don't need to modify) */
  assign sig_wr_mux_in[0] = sig_rr2;
  assign sig_wr_mux_in[1] = inst[15:11];
  Vr_HW2_MUX_S_sel_M_bits #(.S(1),.M(5)) u_mux_wr (1'b1, sig_reg_dst, sig_wr_mux_in, sig_wr); /* inp
ut MUX to WR */

  /* register file */
  assign sig_rr1 = inst[25:21];
  assign sig_rr2 = inst[20:16]; //read register 값 설정
  Vr_HW2_register_file u_rf (CLK, RST, sig_rr1,sig_rr2,sig_wr, sig_rf_write_data, sig_reg_write, sig_rf_rd1,
sig_rf_rd2); //설정한 read register값에 따라 register에 데이터를 쓰거나, read data를 내보냄




  /* control */
  Vr_HW2_control u_control (inst[31:26], sig_reg_dst, sig_branch, sig_mem_read, sig_mem_to_reg, sig_
alu_op, sig_mem_write, sig_alu_src, sig_reg_write ); //control 신호 결정
  Vr_HW2_ALU_control u_alu_control (inst[5:0],sig_alu_op, sig_alu_ctrl);



  /* Implementation 4-2: top-level MUX 2 */
  assign sig_alu_b_mux_in[0]=sig_rf_rd2;
  assign sig_alu_b_mux_in[1]=sig_extended_offset; //4-2 mux의 input
  Vr_HW2_MUX_S_sel_M_bits #(.S(1), .M(32)) u_mux_2(1'b1,sig_alu_src,sig_alu_b_mux_in,sig_alu_b); //m
ux를 진행하여 ALU에 어떤 값을 내보낼지 결정

  /* main ALU */
  /* Implementation 8-2: main ALU (instantiate the ALU implemented in Vr_HW2_ALU.v) */
  assign sig_alu_a=sig_rf_rd1; //ALU의 a값에는 read data1
  Vr_HW2_ALU u_alu_main (sig_alu_a,sig_alu_b,sig_alu_ctrl,sig_alu_out,sig_alu_zero); //main ALU 결과값
과 zero신호를 asserted 혹은 deasserted한다.

  /* data memory */
  Vr_HW2_data_mem u_data_m (sig_alu_out, sig_mem_write, sig_rf_rd2, sig_data_mem_out);



  /* Implementation 4-4: top-level MUX 4 */
  assign sig_rf_wd_mux_in[0]=sig_alu_out;
  assign sig_rf_wd_mux_in[1]=sig_data_mem_out; //4-4 mux의 input
  Vr_HW2_MUX_S_sel_M_bits #(.S(1),.M(32)) u_mux_4 (~sig_mem_write, sig_mem_to_reg,sig_rf_wd_mux
_in,sig_rf_write_data); //MemtoReg 신호를 sel 신호로 하여 어떤 값을 register의 write data로 쓸지 결
```

정 mem_write 신호가 asserted 되면 write_data를 쓰지 않으므로 이를 enable 신호로 받음.

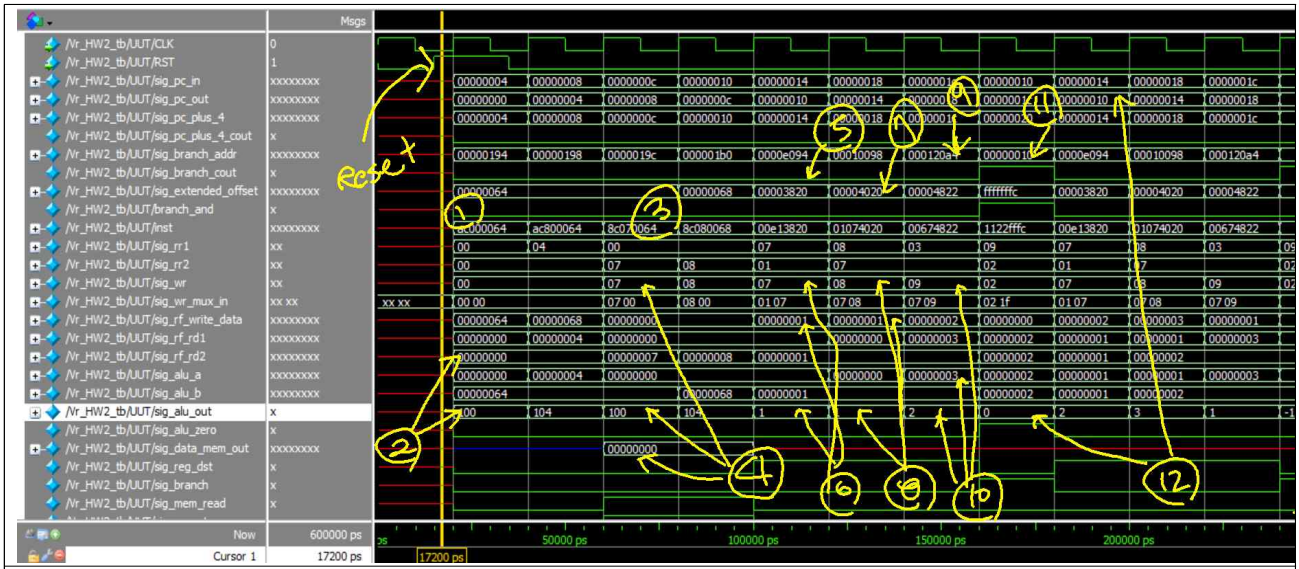
```
  /* PC MUX */
  /* Implementation 4-3: top-level MUX 3 */
  assign sig_pc_mux_in[0] = sig_pc_plus_4;
  assign sig_pc_mux_in[1] = sig_branch_addr; //4-3 mux의 input
  and u_and (branch_and, sig_alu_zero, sig_branch); //4-3 mux의 selection signal
  Vr_HW2_MUX_S_sel_M_bits #(.S(1), .M(32)) u_mux_pc (1'b1,branch_and,sig_pc_mux_in,sig_pc_in); //pc
_mux의 경우 계속해서 pc_in을 업데이트해줘야하므로 enable 신호는 항상 1

  /* Implementation 9: sign extension and shift (input for the calculation of branch address) */
  assign sig_extended_offset=$signed(inst[15:0]); //signed_extended 앞자리 남은 비트수는 MSB에 맞
추어 확장

  /* branch address */
  /* Implementation 8-3: adder for branch address (instantiate the ripple adder implemented in Vr
_HW2_ripple_adder_M_bits.v */
  Vr_HW2_ripple_adder_M_bits #(.M(32)) u_add2 (sig_pc_plus_4, sig_extended_offset<<<2, sig_pc_plus
_4_cout, sig_branch_addr, sig_branch_cout); //8-3 adder로 pc_out에 4를 더한 값과 inst[15:0]을 signe
d extended하고 left로 2 shift한 값을 add하여 이의 결과를 4-3 mux의 인풋으로 내보냄


endmodule
```

3. Screenshots of the simulation results



reset : RST 신호가 들어와서 리셋한 후 프로그램 시작

① sw $r0, 100. mem[100]에 0을 저장. pc_out이 0이므로 inst값 ac000064

② Address:100, value : 0 (alu_a는 0이고 alu_b는 64(hexa)이므로 100(decimal)를 출력)

③ lw $r7, 100. mem[100]을 $r7에 불러옴. pc_out이 8이므로 inst값 8C070064

④ 따라서 주소는 100, 값은 0, register 값은 7이다.

⑤ pc_out이 16이므로 inst값 00E13820이고 이는 i($r7) = i($r7) + ($r1)를 수행한다.

⑥ r7은 0이고 r1은 1이므로 더한 값이 sig_out값은 1이다. 이 값이 r7에 쓰여진다. 쓰여지는 data가 1인지 sig_rf_write_data가 1인 것을 보고 확인할 수 있다.

⑦ pc_out이 20이 되면 inst가 01074020가 되고 r8에 r7을 더하여 r8에 쓰는 작업을 수행한다.

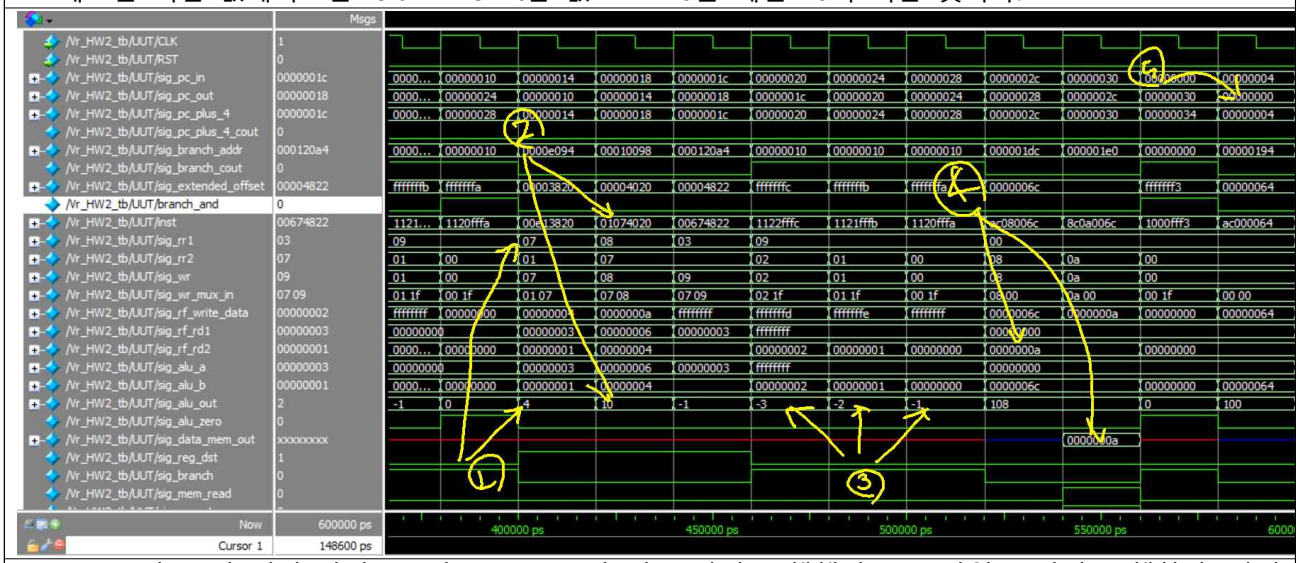
⑧ r7은 1이고 r8은 0이므로 더한 값은 1인 것을 확인할 수 있다. 이 데이터가 r8에 쓰여진다. register는 wr로 확인이 가능하고 08인 것을 보아 r8에 쓰여지는 것을 알 수 있다. 앞서 마찬가지로 write_data가 1인 것을 확인할 수 있다.

⑨ pc_out이 24가 되면 inst는 00674822가 된다. 이는 r3-r7를 하여 결과값을 r9에 쓴다.

⑩ r3에는 3이 들어가 있고, r7에는 1이 들어가 있으므로 alu_out값이 2가 되고 이가 r9에 쓰인다. 앞의 결과들과 마찬가지로 wr와 write_data를 확인하여 올바른 register에 올바른 data가 들어가는 것을 확인할 수 있다.

⑪ pc_out이 28이 되면 inst는 1122FFFC가 된다. 이는 beq r9,r2를 수행한다.

⑫ r9과 r2 모두에 2가 들어있기 때문에 alu_out 값은 0이 되고, sig_zero가 1이 된 것을 확인할 수 있다. 이후에는 pc_out이 28에서 16으로 간다. 이는 beq가 잘 진행됨을 알 수 있다. 들어온 pc값 28에 4를 더한 값에서 4를 left로 2 shift한 값 4*4=16을 빼면 16이 되는 것이다.



① pc_out이 16이 되어 다시 inst가 00E13820이 되고 앞서 수행했던 add 작업을 다시 수행한다. 앞서서 계속 수행을 하여 r7에 3이 들어가있는 상태이다. r7은 3이고 r1은 1이므로 더한 값은 4가 되어 r7에 쓰여진다. (r7:0->1(0+1)->2(1+1)->3(2+1)->4(3+1))

② 앞선 결과와 마찬가지로 현재 r8은 6이고 r7이 4이므로 더한 값인 10이 r8에 쓰이는 것을 알 수 있다. (r8:0->1(1+0)->3(2+1)->6(3+3)->10(6+4))

③ pc_out이 28,32,36일 때 beq 동작을 한다. 같지 않으므로 zero값은 0으로 유지되고 따라서 register에 값을 쓰지 않는다.

④ pc_out이 40이 되면 inst는 AC08006C가 되어 mem[108]에 r8의 데이터를 입력하고, pc_out이 44가 되어 inst가 8C0A006C가 된다. 이때 data_mem_out은 10이 되는데 이는 pc_out이 20일 때 마지막으로 진행된 값이 출력된 것이다. 이 값은 pc_out이 40일 때 mem[108]번지에 입력이 되어 있으므로 alu_out은 108이 나타나고 10이 잘 저장되어있다는 것을 나타낸다.

⑤ 작업이 끝난 후 다시 시작점으로 돌아가 반복해서 작동한다.