**/\* A alu module\*/**

module ALU(inputA, inputB, ALUop, result, zero);

input [31:0] inputA, inputB;

input [2:0] ALUop;

output [31:0] result;

reg [31:0] result;

output zero;

reg zero;

/\*whenever input or ALUop changes\*/

always @(inputA or inputB or ALUop)

begin

/\*it supports AND, OR, ADD, SLT with a zero output\*/

/\*

to do

\*/

casex(ALUop)

3'bx00:result = inputA & inputB;

3'bx01:result = inputA | inputB;

3'b010:result = inputA + inputB;

3'b110:result = inputA - inputB;

3'bx11:result = (inputA < inputB);

endcase

if (inputA == inputB)

zero = 1;

else

zero = 0;

end

endmodule

module ALUTestbench;

reg [31:0] inputA, inputB;

reg [2:0] ALUop;

wire [31:0] result;

wire zero;

ALU UUT(inputA, inputB, ALUop, result, zero);

initial begin

#20 inputA = 5;

inputB = 6;

ALUop = 3'b000;

#40 inputA = 5;

inputB = 6;

ALUop = 3'b001;

#40 inputA = 5;

inputB = 6;

ALUop = 3'b010;

#40 inputA = 5;

inputB = 6;

ALUop = 3'b110;

#40 inputA = 5;

inputB = 6;

ALUop = 3'b011;

#40 inputA = 9;

inputB = 3;

ALUop = 3'b011;

#40 inputA = 6;

inputB = 6;

ALUop = 3'b001;

#40 inputA = -8;

inputB = 6;

ALUop = 3'b001;

end

initial

#340 $stop;

Endmodule

**// a memory module**

// note that addr is a byte address. To simplify our implementation

// we use addr>>2 to index the memory array.

module mem(addr,datain,dataout, MemWrite, MemRead);

// currently the memory has 128 32-bit words

parameter mem\_capacity = 4096;

input [31:0] addr, datain;

output [31:0] dataout;

reg [31:0] dataout;

// controls when writing and when readiing

input MemWrite, MemRead;

// memory cells are defined here

// reg [31:0] memory [mem\_capacity-1:0];

reg [31:0] memory [mem\_capacity-1:0];

integer i;

// reset it at the beginning

initial begin

for (i=0; i<mem\_capacity-1; i=i+1)

memory[i] = 0;

// read initial data into the memory

// pls refer to http://www.asic-world.com/verilog/memory\_fsm1.html

$readmemh("mem.dat", memory);

end

// whenever the any of sigals (addr or datain or memwrite or memread) changes

always @(addr or datain or MemWrite or MemRead)

begin

if (MemRead)

dataout = memory[addr>>2];

if (MemWrite)

memory[addr>>2] = datain;

end

endmodule

module MemTestbench;

reg [31:0] addr, datain;

reg MemWrite, MemRead;

wire [31:0] dataout;

mem UUT(addr, datain, dataout, MemWrite, MemRead);

initial begin

addr = 0;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_0000\_0100;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_0001\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_0010\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_0011\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_0100\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_0101\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_0110\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_0111\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_1000\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_1001\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_1010\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_1011\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_0000\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_0001\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_0010\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_0011\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_0100\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_0101\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_0110\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_0111\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_1000\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_1001\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0001\_0000\_0000\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 5;

datain = 6;

MemWrite = 1;

MemRead = 0;

#20 addr = 1;

datain = 10;

MemWrite = 1;

MemRead = 0;

#20 addr = 8;

datain = 11;

MemWrite = 1;

MemRead = 0;

#20 addr = 5;

datain = 10;

MemWrite = 0;

MemRead = 1;

#20 addr = 1;

datain = 13;

MemWrite = 0;

MemRead = 1;

#20 addr = 8;

datain = 13;

MemWrite = 0;

MemRead = 1;

end

initial

#600 $stop;

endmodule

**// a memory module**

// note that addr is a byte address. To simplify our implementation

// we use addr>>2 to index the memory array.

module mem(addr,datain,dataout, MemWrite, MemRead);

// currently the memory has 128 32-bit words

parameter mem\_capacity = 4096;

input [31:0] addr, datain;

output [31:0] dataout;

reg [31:0] dataout;

// controls when writing and when readiing

input MemWrite, MemRead;

// memory cells are defined here

// reg [31:0] memory [mem\_capacity-1:0];

reg [31:0] memory [mem\_capacity-1:0];

integer i;

// reset it at the beginning

initial begin

for (i=0; i<mem\_capacity-1; i=i+1)

memory[i] = 0;

// read initial data into the memory

// pls refer to http://www.asic-world.com/verilog/memory\_fsm1.html

$readmemh("mem.dat", memory);

end

// whenever the any of sigals (addr or datain or memwrite or memread) changes

always @(addr or datain or MemWrite or MemRead)

begin

if (MemRead)

dataout = memory[addr>>2];

if (MemWrite)

memory[addr>>2] = datain;

end

endmodule

module MemTestbench;

reg [31:0] addr, datain;

reg MemWrite, MemRead;

wire [31:0] dataout;

mem UUT(addr, datain, dataout, MemWrite, MemRead);

initial begin

addr = 0;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_0000\_0100;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_0001\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_0010\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_0011\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_0100\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_0101\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_0110\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_0111\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_1000\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_1001\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_1010\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_0000\_1011\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_0000\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_0001\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_0010\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_0011\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_0100\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_0101\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_0110\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_0111\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_1000\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0000\_1000\_1001\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 32'b0000\_0000\_0000\_0000\_0001\_0000\_0000\_0000;

MemWrite = 0;

MemRead = 1;

#20 addr = 5;

datain = 6;

MemWrite = 1;

MemRead = 0;

#20 addr = 1;

datain = 10;

MemWrite = 1;

MemRead = 0;

#20 addr = 8;

datain = 11;

MemWrite = 1;

MemRead = 0;

#20 addr = 5;

datain = 10;

MemWrite = 0;

MemRead = 1;

#20 addr = 1;

datain = 13;

MemWrite = 0;

MemRead = 1;

#20 addr = 8;

datain = 13;

MemWrite = 0;

MemRead = 1;

end

initial

#600 $stop;

Endmodule

**//mux2to1 module**

module mux2to1(datain0,datain1, dataout, select);

input [31:0] datain0, datain1;

input select;

output [31:0] dataout;

reg [31:0] dataout;

/\*whenever datain0 or datain1 or select signals is changed\*/

always @(datain0 or datain1 or select)

begin

if (select == 0)

dataout = datain0;

else

dataout = datain1;

end

endmodule

module muxTestbench;

reg [31:0] datain0,datain1;

reg select;

wire [31:0] dataout;

mux2to1 UUT(datain0, datain1, dataout,select);

initial begin

#20 datain0 = 8;

datain1 = 13;

select = 0;

#20 datain0 = 8;

datain1 = 13;

select = 1;

#20 datain0 = 20;

datain1 = 21;

select = 1;

#20 datain0 = 55;

datain1 = 66;

select = 1;

#20 datain0 = 55;

datain1 = 66;

select = 0;

end

initial

#120 $stop;

Endmodule

**// A register file module**. 32 registers, each of it has 32 bit

// Supports 2 reading ports, 1 writing port

// ADDA: reading address of A, ADDB: reading addr of B

// ADDC: writing address of C

// clk: clock signal

// clr: clear signal

// WE: Write Enable

**//registerfile module**

module registerfile(ADDA, DATAA, ADDB, DATAB, ADDC, DATAC, clk, clr, WE);

input [4:0] ADDA, ADDB, ADDC;

input [31:0] DATAC;

input clk, clr, WE;

output [31:0] DATAA, DATAB;

reg [31:0] register [31:0];

integer i;

//clear all the registers in the register file

initial begin

for (i=0; i<32; i=i+1)

register[i] = 0;

$readmemh("reg.dat", register);

end

//only when a positive(rising) edge occurs

always @(posedge clk or posedge clr)

begin

// clear signal will reset all register as well

if (clr)

for (i=0; i<32; i=i+1)

register[i] = 0;

else

// only when WE is 1, we write the register file

if (WE == 1)

begin

register[ADDC] = DATAC;

register[0] = 0;

end

end

// we always reading content of A and B

assign DATAA = register[ADDA];

assign DATAB = register[ADDB];

endmodule

module RegFileTestbench;

reg [31:0] data;

reg [4:0] addrA, addrB, addrC;

wire [31:0] outA;

wire [31:0] outB;

reg clks, clr, WE;

integer ctr;

registerfile UUT(addrA, outA, addrB, outB, addrC, data, clks, clr, WE);

initial begin

clks = 0;

#10 clr = 0;

WE = 0;

for (ctr=0; ctr<31; ctr = ctr+1)

begin

#25 addrA <= ctr;

addrB = ctr+1;

end

addrA = 2;

addrB = 3;

addrC = 3;

WE = 1;

for (ctr=0; ctr<10; ctr = ctr+1)

begin

#25 data <= ctr;

end

WE = 0;

end

initial

forever #20 clks = ~clks;

initial

#1100 $stop;

Endmodule

**//component module**

module single\_register(datain, dataout, clk, clr, WE);

input [31:0] datain;

output [31:0] dataout;

input clk, clr, WE;

reg [31:0] register;

integer i;

/\*

to do

\*/

always @(posedge clk or posedge clr)

begin

// clear signal will reset all register as well

if (clr)

for (i=0; i<32; i=i+1)

register[i] = 0;

else

// only when WE is 1, we write the register file

if (WE == 1)

begin

register = datain;

end

end

// we always read register

assign dataout = register;

endmodule

module single\_register\_Testbench;

reg [31:0] datain;

wire [31:0] dataout;

reg clks, clr, WE;

integer ctr;

/\*

to do

\*/

single\_register A(datain, dataout, clks,clr,WE);

initial begin

clks = 0;

#10

clr = 0;

WE = 0;

for (ctr=0; ctr<10; ctr = ctr+1)

begin

#25

datain = ctr;

end

WE = 1;

#10

for (ctr=0; ctr<10; ctr = ctr+1)

begin

#40

datain = ctr;

end

end

initial

forever #20 clks = ~clks;

initial

#800 $stop;

endmodule

module mux4to1(datain0, datain1, datain2, datain3, dataout, select);

input [31:0] datain0, datain1, datain2, datain3;

input [1:0] select;

output [31:0] dataout;

reg [31:0] dataout;

/\*

to do

\*/

always @(datain0 or datain1 or datain2 or datain3 or select)

begin

if(select == 0)

dataout = datain0;

else if(select == 1)

dataout = datain1;

else if(select == 2)

dataout = datain2;

else

dataout = datain3;

end

endmodule

module mux4to1\_Testbench;

reg [31:0] datain0,datain1,datain2,datain3;

reg [1:0] select;

wire [31:0] dataout;

integer ctr;

/\*

to do

\*/

mux4to1 UUT(datain0, datain1, datain2, datain3, dataout,select);

initial begin

#20

select = 0;

datain0 = 8;

datain1 = 13;

datain2 = 12;

datain3 = 14;

for(ctr = 0 ; ctr < 4 ; ctr= ctr+1)

begin

#5 select=ctr;

end

#20 datain0 = 55;

datain1 = 66;

datain2 = 21;

datain3 = 44;

for(ctr = 0 ; ctr < 4 ; ctr = ctr+1)

begin

#5 select=ctr;

end

end

initial

#100 $stop;

endmodule

module signextd(datain, dataout);

input [15:0] datain;

output [31:0] dataout;

reg [31:0] dataout;

/\*

to do

\*/

always @(datain)

begin

if(datain[15] == 0)

assign dataout = datain;

else if(datain[15] == 1)

assign dataout = (2\*\*32-1) - (2\*\*16-1) + datain;

end

endmodule

module signextd\_Testbench;

reg [15:0] datain;

wire [31:0] dataout;

/\*

to do

\*/

signextd A(datain, dataout);

initial begin

#10

datain = 2^13;

#10

datain = 16'b1010101010101010;

end

initial

#30 $stop;

endmodule

module shiftleft2(datain, dataout);

input [31:0] datain;

output [31:0] dataout;

reg [31:0] dataout;

/\*

to do

\*/

always @(datain)

begin

dataout = datain << 2;

end

endmodule

module shiftleft2\_Testbench;

reg [31:0] datain;

wire [31:0] dataout;

/\*

to do

\*/

shiftleft2 A(datain, dataout);

initial begin

#10 datain = 100000;

#10 datain = 2\*\*32-1;

#10 datain = 2\*\*30-1;

#10 datain = (2\*\*32-1) - (2\*\*31-1);

end

initial #50 $stop;

endmodule

/\* concatenate pcin[31-28] with datain[27-0] to form a jump address\*/

module concatenate4to28(datain, pcin, pcout);

input [31:0] datain, pcin;

output [31:0] pcout;

reg [31:0] pcout;

always @(datain, pcin)

begin

pcout = ( pcin[31:28] \* (2\*\*28) ) + datain[27:0];

end

endmodule

module concatenate4to28\_Testbench;

reg [31:0] datain, pcin;

wire [31:0] pcout;

/\*

to do

\*/

concatenate4to28 A(datain, pcin, pcout);

initial begin

#10 pcin = (2\*\*32 - 1);

datain = (2\*\*25-1);

#10 pcin = (2\*\*31 - 1);

datain = (2\*\*23-1);

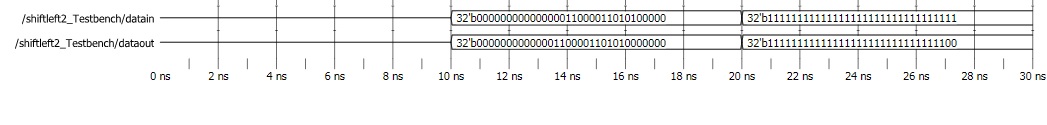
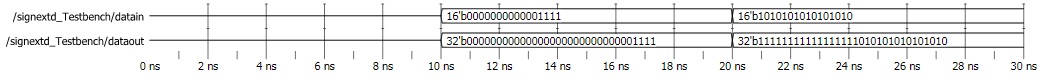
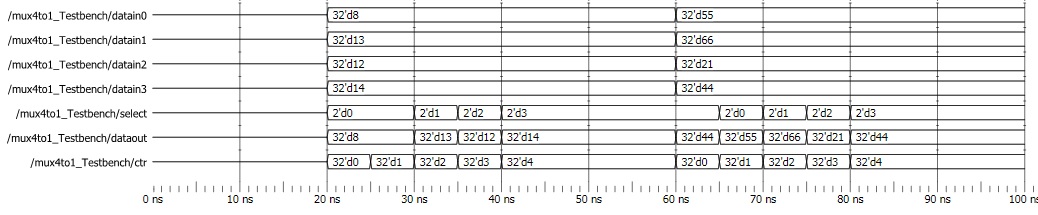
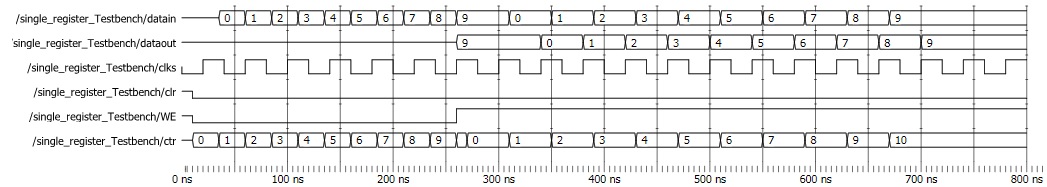
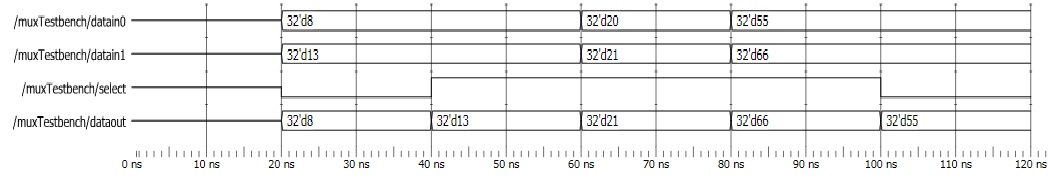
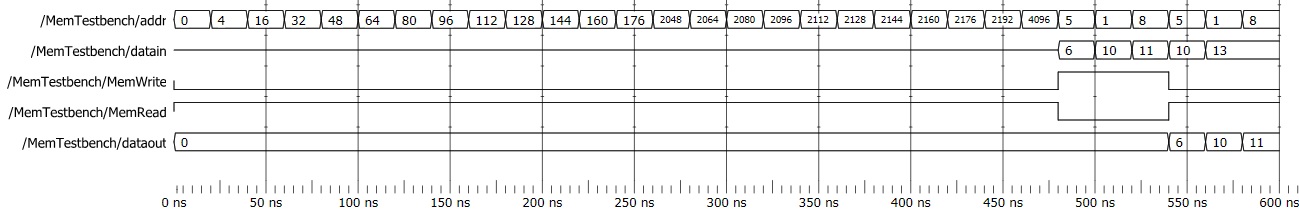
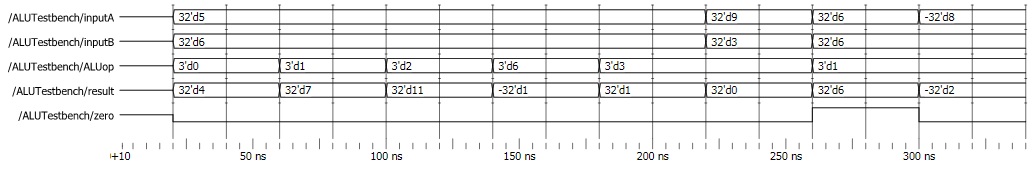
#10 pcin = (2\*\*22 - 1);

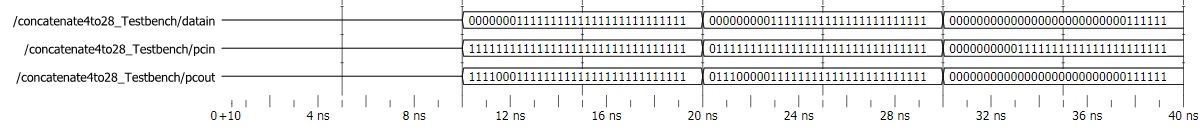
datain = (2\*\*6-1);

end

initial #40 $stop;

endmodule





Mem 모듈은 Memwrite와 Memread라는 입력에 따라 두가지 역할을 수행한다. 데이터 메모리에서 데이터를 불러올 때는 Memread가 1로 들어온다. 반대로 쓸 때는 Memwrite가 1로 들어온다. Memread가 1로 있을 때 dataout에 addr 입력을 4로 나눈 값의 주소를 memory에서 읽어서 출력한다. Memwrite가 1로 있을 땐 dataout으로는 출력이 연결되지 않고 datain의 값을 addr 입력을 4로 나눈 값의 주소의 memory에 값을 입력한다.

0~480(ns)에선 값을 mem.dat에서 제대로 읽어오지 못한 것을 그래프에서 확인할 수 있다. 처음에 설정한 0의 값을 읽어오는 것을 확인할 수 있었고 코드상의 문제가 없어서 당황스러웠다. 해결하려고 했지만 상대 주소말고 절대 주소로 바꿔봐도 해결되지 않았다. 아마 $readmemh에서 제대로 값을 읽지 못한 것 같다. 480~540에선 Memwrite를 1로 설정해서 값을 쓴다. 주소와 입력 데이터를 받아서 그 주소에 해당하는 메모리 값에 데이터를 입력한다. 입력된 값을 확인하기 위해 540~600(ns)에서 Memread를 1로 설정하고 addr와 dataout을 출력하는데 datain은 출력값에 영향을 미치지 않는 것을 알 수 있다.

Registerfile 모듈은 입력값은 3개의 주소값(2개는 레지스터를 읽기 위한 주소, 하나는 레지스터에 쓰기 위한 주소다)과 쓸 데이터값 그리고 clk, clr, WE가 있다. Clk은 클럭을 뜻하고 clr은 레지스터를 초기화를 할 때 쓰이며 WE는 레지스터에 데이터를 쓰고 싶을 때 1로 설정한 후 쓰기 위한 입력이다.

10~785(ns)에선 WE는 0이다. 즉 레지스터에 있는 데이터를 outA와 outB로 출력한다. 하지만 mem 모듈에서의 문제와 마찬가지로 $readmemh가 제대로 동작하지 않는다. 따라서 addrA와 addrB를 다양하게 입력해도 outA와 outB는 처음에 기본값으로 설정한 0을 출력한다. 785~1035(ns)에선 WE=1로 설정되있고 addrC=3으로 설정되있다. 이 구간에서 주목할 점은 posedge에서 쓰기를 실행하고 읽기는 clk에 관계없이 항상 진행하기 때문에 posedge마다 3에 새로운 값을 계속 집어넣으면서 동시에 addrB도 3을 가리키기 때문에 outB는 쓰이자 마자 값을 출력한다. 하지만 1035~1050(ns)에선 WE=0이기 때문에 새로운 값이 쓰이지 않고 따라서 outB값도 마지막 입력값인 8을 계속 출력한다.

. Mux2to1 모듈은 select 입력과 두개의 입력값을 받아서 select를 통해 두개의 입력값중 하나를 고르는 회로이다.

그래프는 select가 변할때마다 입력값 두개중에 하나를 선택하는 것을 확인할 수 있다. 0일 경우는 datain1을 선택하고 1일 경우 datain2을 선택한다.

ALU는 두개의 입력과 ALUop라는 선택값을 입력으로 받고 연산 결과값과 zero를 출력한다. Zero는 뺄셈일 경우에 쓰이지만 ALUop와는 상관 없이 항상 출력한다. Casex를 통해 result를 ALUop에 따라 다른 값을 출력하도록 설계했다. Don’t Care를 casex를 통해 보기 쉽게 설계할 수 있었다. Zero 출력은 A와 B가 같다면 무조건 출력하도록 설정했다. 위의 설계는 inputA나 inputB나 ALUop의 값이 바뀔 때 작동하는 회로이다. 이러한 것은 always@(조건문)을 통해 구현했다. 위의 그래프를 보면 ALUop가 바뀔 때, 아니면 두 입력값중 하나가 바뀔 때마다 실행되는 것을 볼 수 있다.

20~220 (ns)에선 두 입력은 일정하고 ALUop를 계속 바꾸고 있다. 다양한 연산의 결과가 Result로 출력된다. 220~240(ns)에선 값을 바꾸고 ALUop를 일정하게 유지하고 Result 값을 새로 출력하는 것을 확인할 수 있다. 260~300(ns)에선 OR 연산자를 선택했지만 두 수가 같기 때문에 Zero 출력이 1을 가리키는 것을 확인할 수 있다. 또 300(ns)부터는 다시 zero 값이 0으로 돌아온 것을 확인할 수 있다.

Single\_register는 위의 registerfile와 마찬가지로 WE 값을 통해 쓰기를 통제한다.

테스트 벤치를 보면 260ns 까지는 입력된 값이 없기 때문에 출력을 못하다가 이후에는 WE가 1로 바뀌면서 쓰기를 시작한다. posedge마다 레지스터에 새로운 값(datain)을 넣고 dataout은 datain이 바뀔때마다 (posedge마다) 출력한다.

Mux4to1도 mux2to1과 같은 방식으로 작동하지만 2개의 입력에서 4개의 입력으로 늘어난 것과 그에 따라 select입력도 2개의 비트로 입력된다는 점이 달라졌다.

60ns를 기준으로 4개의 입력값을 바꿨고 select는 0,1,2,3을 한번씩 가리킨다. Select가 가리키는 입력값이 dataout으로 출력되는 것을 확인할 수 있다.

Signextd 모듈은 16비트를 32비트로 확장시켜주는 모듈이다. 이 모듈은 16비트의 부호가 중요하다. 왜냐하면 추가할 16비트는 모두 같은 값인데 16비트의 부호비트의 값으로 확장하기 때문이다. 따라서 테스트 벤치에선 음수의 경우와 양수의 경우 하나씩 넣었다. 그래프에서 확인할 수 있듯이 10~22(ns) 에선 양수 값을 확장했고 그 이후는 음수값을 확장했고 양수의 경우 16개의 0이 추가된 것을 음수의 경우 16개의 1이 추가된 것을 확인할 수 있다.

Shiftleft2 모듈은 비트를 왼쪽으로 2번 이동해주는 모듈이다. 곱하기 4를 하는 효과가 있다. 10~20(ns)에선 양수값을 20~30(ns)에선 음수값을 입력해봤고 값이 이상이 없는 것을 확인할 수 있었다. 30~40(ns)에서는 부호비트가 있을 경우 양수가 음수로 변하는 오버플로가 발생하는 것을 확인할 수 있었다. 40~50(ns)에서도 음수가 양수로 변하는 오버플로를 확인할 수 있었다.

Concatenate는 두 비트 datain과 pcin을 입력 받아서 pcin의 상위 4비트와 datain의 하위 28비트를 합성하는 모듈이다. 테스트 벤치는 무난하게 세가지를 구성해봤는데 pcin의 상위 4비트와 datain의 하위 28비트가 적절하게 잘 조합되서 출력되는 것을 확인할 수 있었다.