**Kustar.v**

`include "mem\_beh.v"

`include "regfile\_beh.v"

`include "components.v"

`include "controlunits.v"

`include "alu\_beh.v"

`include "mux2to1\_beh.v"

module kustar;

reg clock, clear;

wire [31:0] MuxtoPC, PCtoMux, MuxtoMem, MemtoInstReg, SigntoShift, ShifttoMux, MuxtoWriteData;

wire [31:0] RegtoA, RegtoB, AtoMux, BtoMux, AMuxtoALU, BMuxtoALU, ALUtoALUOut, ShifttoConcat, ConcattoMux;

wire Zero;

wire [31:0] ALUOuttoMux;

wire [31:0] MemdatatoMux;

wire [5:0] Op ;

wire [4:0] Rs, Rt;

wire [4:0] MuxtoWriteReg;

wire [15:0] AddrorConst ;

wire PCWriteCond, PCWrite, IorD, MemRead, MemWrite, MemtoReg, IRWrite, ALUSrcA, RegWrite, RegDst;

wire [1:0] PCSource, ALUOp, ALUSrcB;

wire [2:0] ALUcontrol;

begin

mcu Control\_Unit(clock, clear, Op, PCWriteCond, PCWrite, IorD, MemRead, MemWrite, MemtoReg, IRWrite, PCSource, ALUOp, ALUSrcB, ALUSrcA, RegWrite, RegDst);//Control Unit

single\_register PC (MuxtoPC, PCtoMux, clock, clear, (Zero&PCWriteCond)|PCWrite);

/\*

to do

\*/

acu ACU(AddrorConst[5:0], ALUOp, ALUcontrol);

mux2to1 mux\_IorD(PCtoMux,ALUOuttoMux, MuxtoMem, IorD);

mem Memory(MuxtoMem, BtoMux, MemtoInstReg, MemWrite, MemRead, clock );

single\_register IR(MemtoInstReg, {Op,Rs,Rt,AddrorConst}, clock, clear, IRWrite);//?

single\_register MR(MemtoInstReg, MemdatatoMux, clock, clear, 1'b1);//?

mux2to1 mux\_RegDst(Rt, AddrorConst[15:11],MuxtoWriteReg,RegDst);

mux2to1 mux\_MemtoReg(ALUOuttoMux,MemdatatoMux,MuxtoWriteData,MemtoReg);

registerfile Registers(Rs, RegtoA, Rt, RegtoB, MuxtoWriteReg, MuxtoWriteData, clock, clear, RegWrite);

single\_register AR(RegtoA, AtoMux, clock, clear, 1'b1);//?

single\_register BR(RegtoB, BtoMux, clock, clear, 1'b1);//?

signextd signextend(AddrorConst, SigntoShift);

shiftleft2 shiftleft2(SigntoShift, ShifttoMux);

mux2to1 mux\_ALUSrcA(PCtoMux, AtoMux, AMuxtoALU, ALUSrcA);

mux4to1 mux\_ALUSrcB(BtoMux, 4 , SigntoShift, ShifttoMux, BMuxtoALU, ALUSrcB);

shiftleft2 shiftleft2\_2({4'b0000,Rs, Rt, AddrorConst}, ShifttoConcat);

ALU ALU(AMuxtoALU, BMuxtoALU, ALUcontrol, ALUtoALUOut, Zero);

single\_register ALUR(ALUtoALUOut, ALUOuttoMux, clock, clear, 1'b1);

concatenate4to28 concat(ShifttoConcat, PCtoMux, ConcattoMux);

mux4to1 mux\_PCSource(ALUtoALUOut, ALUOuttoMux, ConcattoMux, z, MuxtoPC, PCSource);

end

initial

forever #50 clock = ~clock;

initial

begin

clock = 0;

clear = 1;

/\* we may not connect clear to register file and memory because we don't want our initial data get cleared\*/

#50 clear = 0;

end

initial

#10000 $stop;

Endmodule

**Controlunits.v**

module mcu(clk, clr, OP, PCWriteCond, PCWrite, IorD, MemRead, MemWrite, MemtoReg, IRWrite, PCSource, ALUOp, ALUSrcB, ALUSrcA, RegWrite, RegDst);

input clk, clr;

input [5:0] OP;

output PCWriteCond, PCWrite, IorD, MemRead, MemWrite, MemtoReg, IRWrite, ALUSrcA, RegWrite, RegDst;

reg PCWriteCond, PCWrite, IorD, MemRead, MemWrite, MemtoReg, IRWrite, ALUSrcA, RegWrite, RegDst;

output [1:0] PCSource, ALUOp, ALUSrcB;

reg [1:0] PCSource, ALUOp, ALUSrcB;

//reg [3:0] current\_state;

//reg [3:0] next\_state;

integer current\_state;

integer next\_state;

always @(current\_state)

begin

case(current\_state)

-1:

begin

RegDst=1;

end

0:

begin

IorD=0;

MemRead=1;

MemWrite=0;

IRWrite=1;

ALUSrcA=0;

ALUSrcB=2'b01;

ALUOp=2'b00;

PCSource=2'b00;

PCWrite=1;

PCWriteCond=0;

RegWrite=0;

MemtoReg=0;

RegDst=0;

end

/\*

to do

\*/

1:

begin

MemRead=0;

ALUSrcA=0;

IRWrite=0;

ALUSrcB=2'b11;

ALUOp=2'b00;

PCWrite=0;

end

2:

begin

ALUSrcA=1;

ALUSrcB=2'b10;

ALUOp=2'b00;

end

3:

begin

MemRead=1;

IorD=1;

end

4:

begin

MemRead=0;

MemtoReg=1;

RegWrite=1;

RegDst=0;

end

5:

begin

MemWrite=1;

IorD=1;

end

6:

begin

ALUSrcA=1;

ALUSrcB=2'b00;

ALUOp=2'b10;

end

7:

begin

MemtoReg=0;

RegWrite=1;

RegDst=1;

end

8:

begin

ALUSrcA=1;

ALUSrcB=2'b00;

ALUOp=2'b01;

PCSource=2'b01;

PCWriteCond=1;

end

9:

begin

PCWrite=1;

PCSource=2'b10;

end

endcase

end

always @(posedge clk)

begin

//if (clr == 1'b0)

if (clr == 1'b1)

current\_state = -1;

else

current\_state = next\_state;

end

always @(current\_state or OP)

begin

case (current\_state)

-1: next\_state = 0;

0: next\_state = 1;

/\*

to do

\*/

1:

case (OP)

0: next\_state = 6;//rtype

35: next\_state = 2;//lw

43: next\_state = 2;//sw

4: next\_state = 8;//beq

2: next\_state = 9;//j

endcase

2:

case (OP)

35: next\_state = 3;//lw

43: next\_state = 5;//sw

endcase

3: next\_state = 4;

4: next\_state = 0;

5: next\_state = 0;

6: next\_state = 7;

7: next\_state = 0;

8: next\_state = 0;

9: next\_state = 0;

10: next\_state = 0;

default:

next\_state = -1;

endcase

end

endmodule

module acu(funct, ALUOp, ALUcontrol);

input [5:0] funct;

input [1:0] ALUOp;

output [2:0] ALUcontrol;

reg [2:0] ALUcontrol;

/\*

to do

\*/

always @(funct or ALUOp)

begin

case(ALUOp)

2'b00: ALUcontrol = 3'b010;//lw & sw

2'b01: ALUcontrol = 3'b110;//beq

2'b10:

case(funct)

6'b100000: ALUcontrol = 3'b010;//add

6'b100010: ALUcontrol = 3'b110;//sub

6'b100100: ALUcontrol = 3'b000;//and

6'b100101: ALUcontrol = 3'b001;//or

6'b101010: ALUcontrol = 3'b111;//slt

endcase

endcase

end

endmodule

**alu\_beh.v**

/\* 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

**Components.v**

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 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 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 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

/\* 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

**mem\_beh.v**

// a memory module

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

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

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

// currently the memory has 128 32-bit words

parameter mem\_capacity = 4096;

input clk;

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[0:mem\_capacity-1];

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, 0 , 30);

end

// only when a positive(rising) edge occurs,

always @(posedge clk) begin

if (MemWrite)

memory[addr>>2] = datain;

end

// only when a negatine edge occurs,

always @(negedge clk) begin

if (MemRead)

dataout = memory[addr>>2];

end

endmodule

**Mux2to1\_beh.v**

/\* 2 to 1 MUX, select from two 32-bit input\*/

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

**Regfile\_beh.v**

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] 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 read content of A and B only when a negative edge occurs

always @ (negedge clk )

begin

DATAA = register[ADDA];

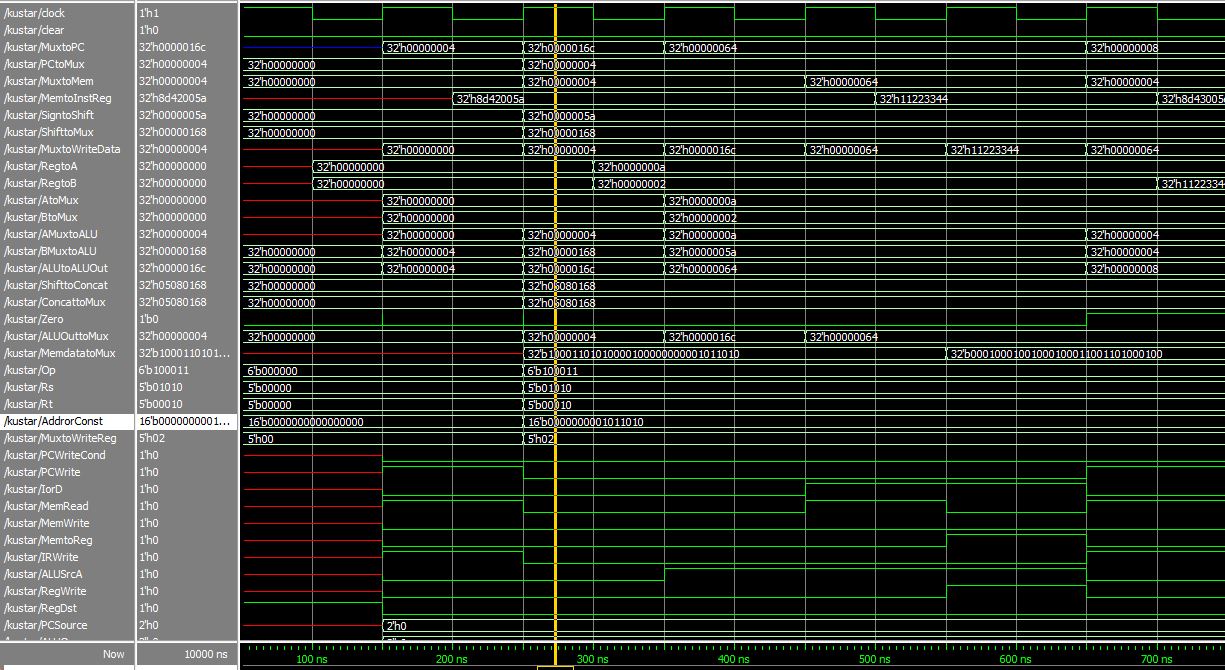
DATAB = register[ADDB];

end

endmodule

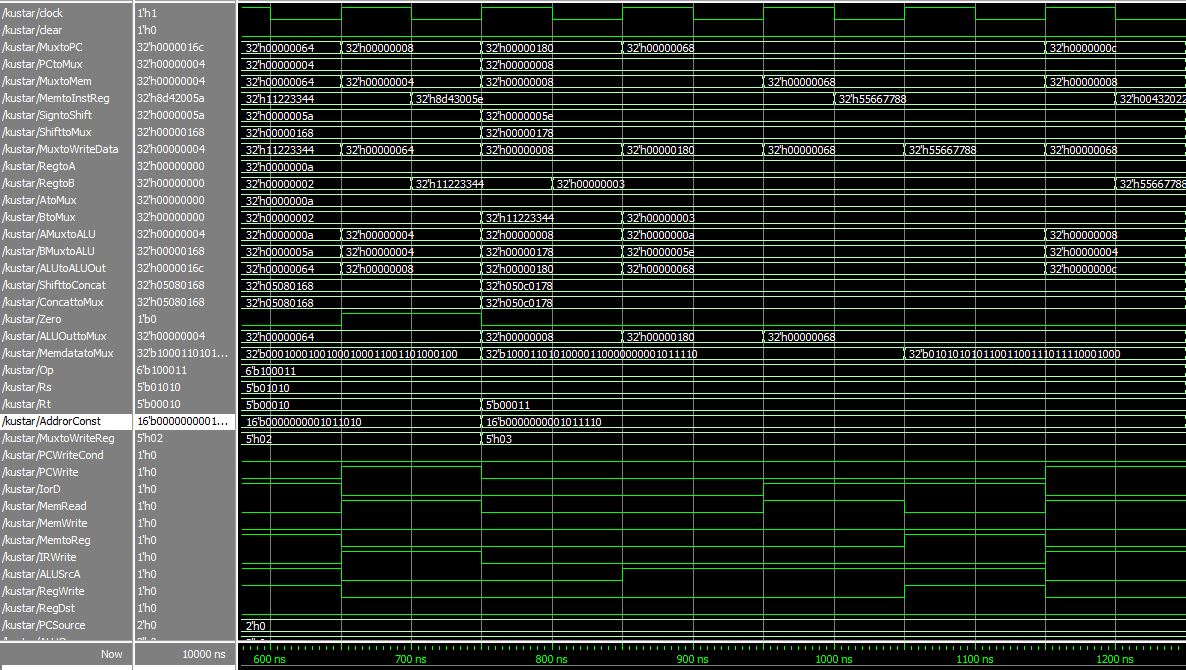
Control unit은 유한 상태 기계로 설계되었다. 클럭이 바뀔 때 마다 상태가 변하고 상태에 따라 control unit의 출력이 변한다. Op-code에 따라서 다른 상태를 오고 가며 명령어를 수행한다.

첫번째 명령어 사이클



50ns에서 -1 상태에 들어갔다가 150ns에서 0 상태에 들어간다. 0 상태에선 PC에 있는 32비트를 Memory Register에 가기 위해 control unit을 조절한다. Memory Register는 MemRead가 1이라서 32비트에 해당하는 주소 값을 읽어서 Memdata 출력을 내보낸다. 출력 값은 Instruction Register로 가고 다음 클럭이 올 때까지 기다린다. 또 PC값은 ALU를 통해 4를 더한 후 다음 클럭이 들어올 때까지 PC의 입력으로 PC+4를 유지한다. 250ns에서 1 상태로 돌입한다. 1 상태에선 Beq 명령어를 위한 PC와 Const 값을 더한 것을 ALUOut 에 저장한다. 또 Instruction Register에서 비트를 나눠 Op, Rs, Rt, AddrorConst로 나눠서 Rs, Rt를 레지스터 파일의 입력으로, AddrorConst는 위의 Beq 명령을 위한 값으로 쓴다. 첫번째 명령어는 LW라서 세번째 상태는 2이다. 이 상태부터는 LW을 위한 상태만 존재한다. LW는 레지스터 파일의 첫 번째 입력에 해당하는 출력 값과 Const를 32비트로 확장한 값과 더한 값을 ALUOut에 저장한다. 세번째 상태는 3 상태인데, 2 상태에서 ALUOut에 저장된 값을 메모리 레지스터에서 데이터를 출력시킨 후 메모리 데이터 레지스터에 다음 클럭까지 입력시킨다. 다음 상태인 4 상태에선 그 입력 값을 레지스터 파일의 WriteData에, rt를 레지스터 파일의 Writereg에 입력하고 RegWrite 신호를 1로 주어서 LW 명령어를 수행한 후 다음 클럭에 0 상태에 가도록 한다.

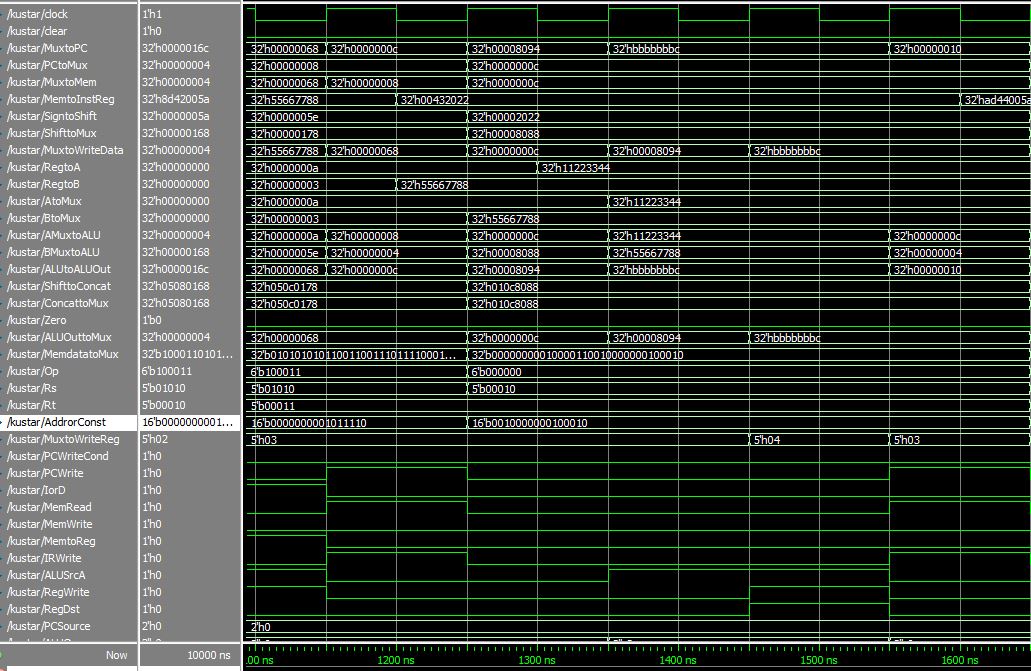
두번째 명령어 사이클



LW 명령어가 다시 나온다. 모든 사이클에서 첫번째(650ns)와 두번째 상태(750ns)는 0과 1이고 첫번째 사이클의 설명으로 대체한다. 세번째 상태 이후(850ns~)는 첫번째 사이클과 같은 LW 명령어이기 때문에 상태 전개 순서는 같다. 650ns에서 PC에서 4를 받고 메모리를 negedge에서 읽어서 700ns에서 8d43005e를 읽는 것을 확인할 수 있다. 또 MuxtoPC는 이미 00000008을 가리키면서 다음 클럭에 PCtoMux에 입력하기를 기다리고 있다. 750ns에서는 레지스터 파일에 입력과 출력이 제대로 나오는 것을 확인할 수 있다. 또 ALUOut에서 BEQ명령어이고 조건이 만족될 경우 이동될 주소값도 제대로 나오는 것을 확인할 수 있다. 그리고 850에서 그 연산값이 BEQ가 아니기 때문에 다시 68로 바뀌는 것을 확인할 수 있다. 950ns에선 그 주소값을 메모리의 Address입력에 넣고 1000ns에서 그 값(55667788)이 메모리 데이터 레지스터로 출력되는 것을 확인할 수 있다. 또 1050ns에서는 R3에 값 55667788을 저장하는 것을 확인할 수 있다.

LW 명령어는 총 5클럭동안 수행되는 것을 확인할 수 있다.

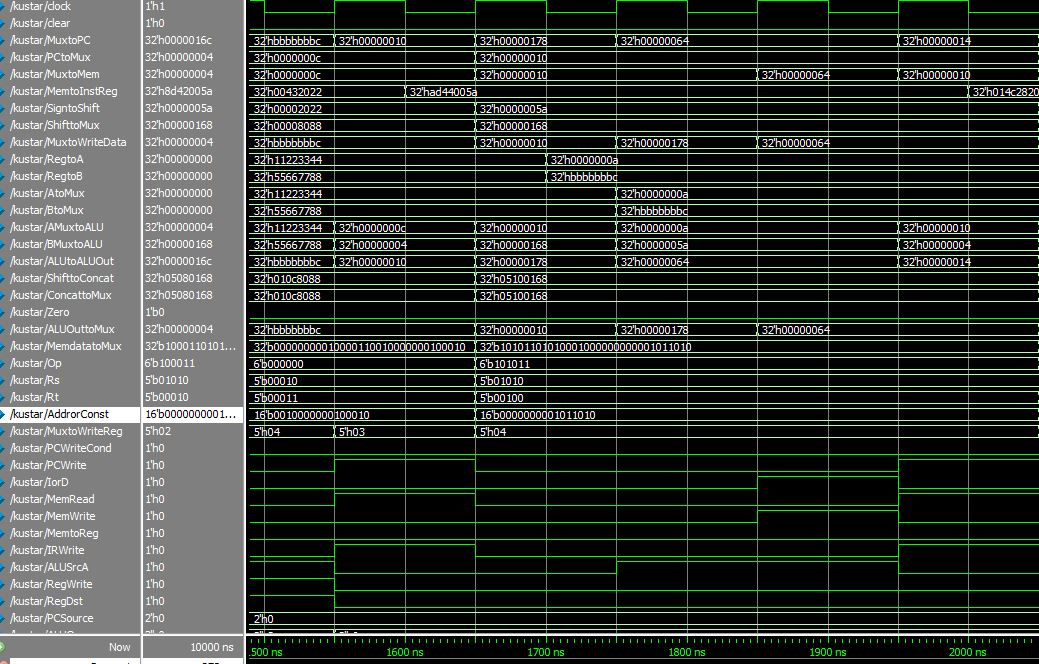
세번째 명령어 사이클



세번째 사이클은 R-type 명령어다. 앞 두 상태(1150~1250ns, 1250~1350ns)는 0, 1로 앞의 설명과 동일하다. 세번째 상태(1350~1450ns)는 6으로 A, B 레지스터를 ALU에 입력으로 넣고 ALUcontrol에 따라 정해진 연산값을 출력하도록 한다. 다음 상태(1450~1550ns)인 7에서 이 출력값을 레지스터 파일의 WriteData값으로 입력하고 AddrorConst 값을 레지스터 파일의 WriteReg 값으로 입력한다. 그리고 RegWrite를 1로 두어서 성공적으로 레지스터에 연산값을 저장한다. 그래프를 자세히 보면 앞의 LW을 통해 레지스터 (R2, R3)에 저장한 값 두개를 불러오는 것을 확인할 수 있다. 또 SUB 명령어라 뺄셈 연산을 위한 ALUcontrol을 입력함으로써 뺼셈 연산도 성공적으로 한 것을 확인할 수 있다. 1550ns에선 다음 명령어의 0 상태가 시작된다.

R-TYPE 명령어는 총 4클럭동안 수행되는 것을 확인할 수 있다.

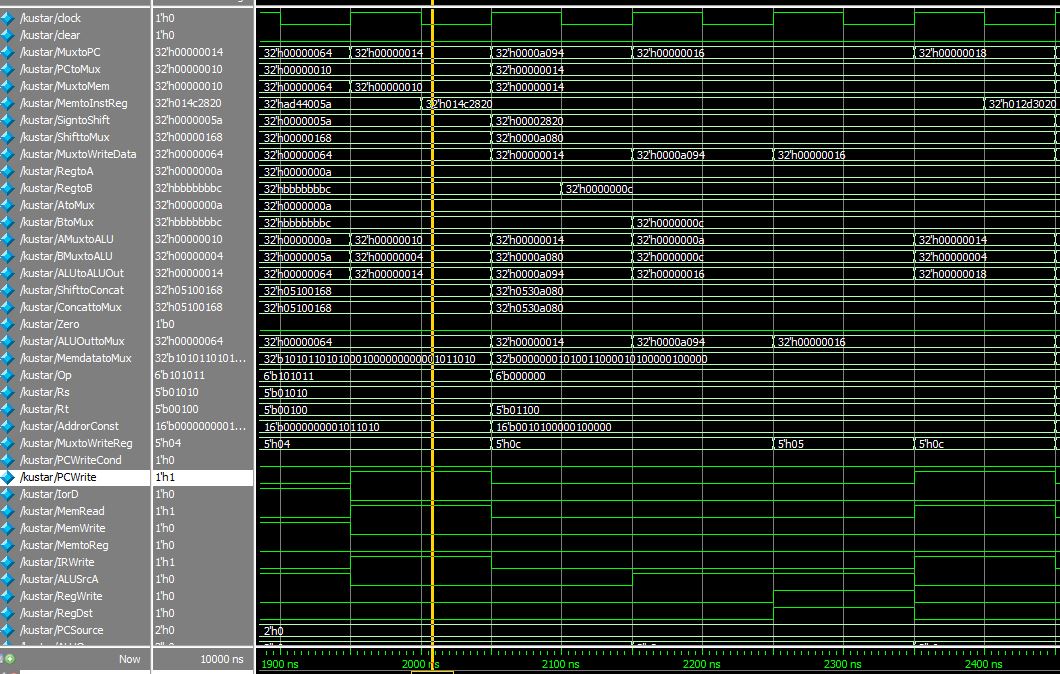
네번째 명령어 사이클



네번째 사이클은 SW 명령어다. (1550~1750ns 생략) 세번째 상태(1750ns)는 3상태이고 LW와 같은 매커니즘으로 작용한다. 네번째 상태(1850ns)는 5 상태이다. 5 상태에선 메모리에 3상태에서 ALUOut에 저장한 값을 Address 입력에, B 레지스터 출력을 Data 입력에 넣고 값을 저장하기 위해 Memwrite를 1 신호를 넣는다. 1950에서는 다음 명령의 0 상태가 시작된다.

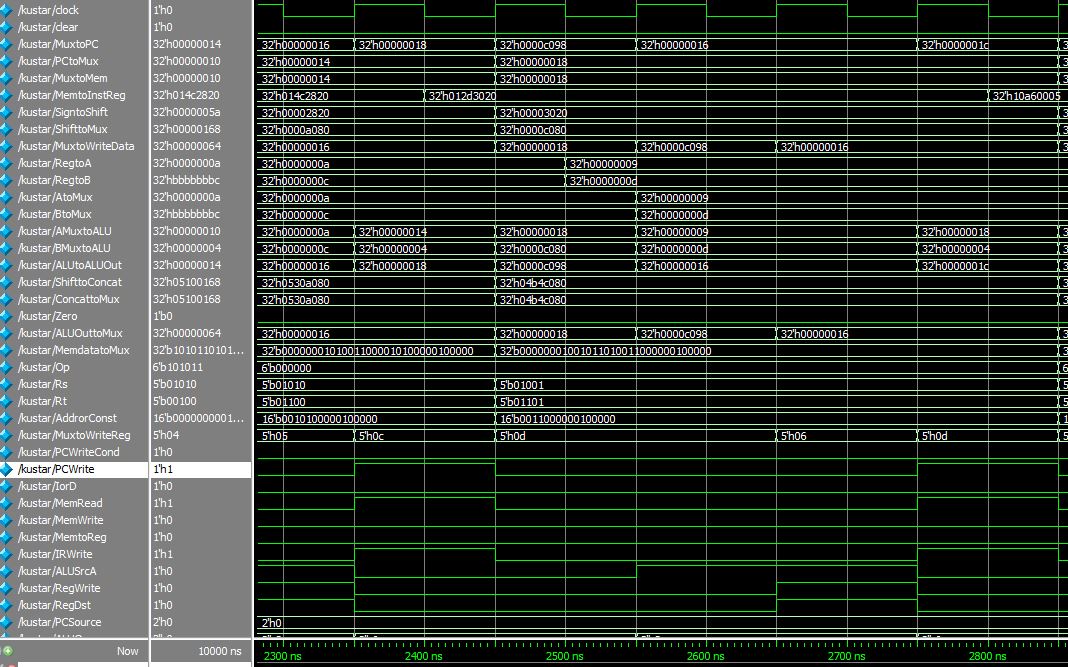
SW 명령어는 총 4클럭동안 수행되는 것을 확인할 수 있다.

다번째 명령어 사이클



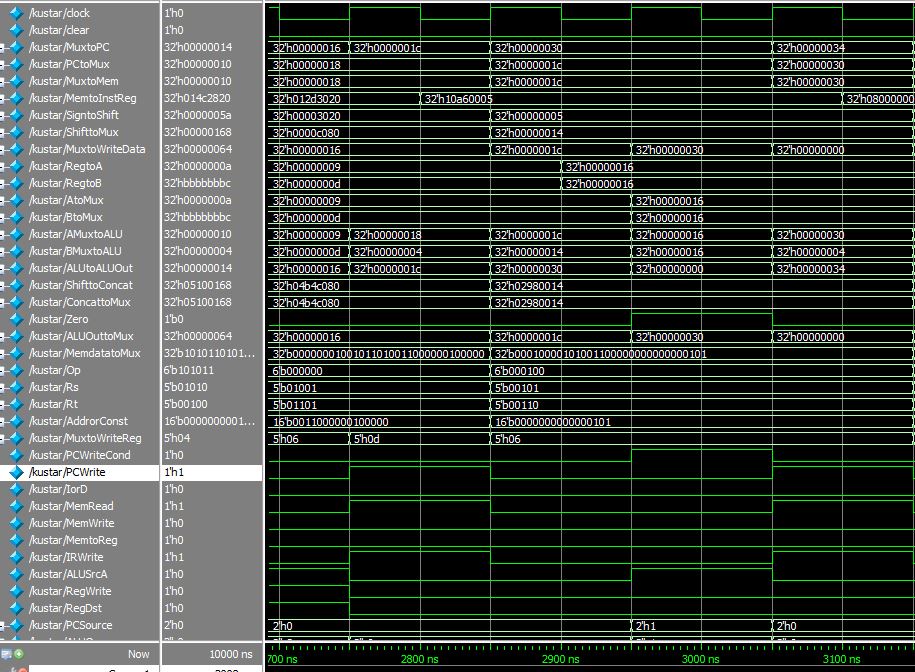
다섯번째 명령어도 R-TYPE이다. 따라서 세번째 명령어와 동일한 방식으로 작동한다. 1950ns에서 PC에서 10을 받고 메모리를 negedge에서 읽어서 2000ns에서 014c2820을 읽는 것을 확인할 수 있다. 또 MuxtoPC는 이미 14를 가리키며 다음 명령을 기다리며 다음 클럭에 PC에 입력하기를 기다리고 있다. 2100ns에서는 레지스터 파일에 입력한 대로 출력이 제대로 나오는 것을 확인할 수 있다. 2150ns에서는 6상태가 시작되며 A(a)와 B(c)를 ALU에서 계산해서 ALUtoALUOut에 제대로 출력하는 것(16)을 확인할 수 있다. 2250ns에선 muxtowritereg에 쓰려는 레지스터 주소(5)와 muxtowritedata(16)이 제대로 입력되는 것을 확인할 수 있다. 2350ns에선 다음 명령의 0 상태가 시작된다.

여섯번째 명령어 사이클



ADD 명령어가 또 나왔다. 위의 명령어와 같은 방식이다. 9와 d를 더한 후 6에 저장하는 과정이 위의 ADD와 동일하다.

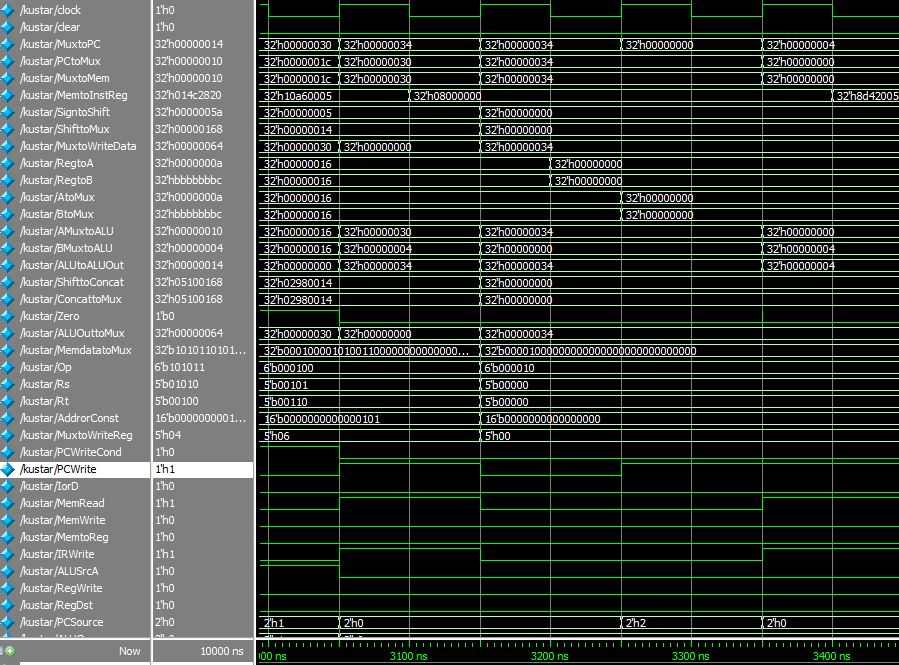
일곱번째 명령어 사이클



BEQ 명령어가 나왔다. R5와 R6를 비교해서 같을 경우 현재 PC에 5값을 더하는지 확인하자. 1 상태(2850ns)에서 계산한 조건이 만족될 경우 이동할 주소 값(30)이 제대로 계산된 것을 확인할 수 있다. (2950ns)에서는 zero가 1이 출력이 되면서 PCWrite가 1로 변하면서 30으로 PCWrite가 점프하는 것을 확인할 수 있다. 중요한 것은 (PC+4)+(5\*4)로 점프한 것인데 +4 한 후에 20만큼 더 간다는 것을 확인할 수 있다. 조건이 만족해서 3050ns에서 30에서 PC가 나온 것을 확인할 수 있다.

BEQ 명령어는 3클럭동안 실행되는 것을 확인할 수 있다.

여덟번째 명령어 사이클



Jump 명령어가 마지막으로 나왔다. 이 명령어는 BEQ의 조건이 만족했기 때문에 실행된 명령어다. 3050ns 에서 상태 0이 시작된다. 08000000을 읽어서 Jump 명령어라는 것을 상태 1(3150ns)에서 확인한다. 상태 1에서 상태 9로 가도록 하고 상태 9(3250ns)에서는 PC의 상위 4비트와 명령어 비트의 하위 26비트를 2번 왼쪽으로 비트 이동한 값을 Concat한 값을 PC에 넣는 것을 확인할 수 있다. 0(x6)과 0(x28)을 concat 하기 때문에 0이 PC로 들어가고 다음 명령은 PC가 0을 가리키기 때문에 처음에 실행했던 명령이 나온다. 즉 무한 루프가 만들어지고 10000ns에서 루프가 3번 반복되는 것을 확인할 수 있었다.

Jump 명령어는 총 3클럭동안 수행되는 것을 확인할 수 있다.

이번 과제는 우리가 봐야하는 값만 입력되고 출력되는 것이 아니라 쓸모 없는 값도 웨이브에 같이 나와서 분석하기가 까다로웠다. Control signal에 따라 어떤 어떤 값을 읽고 어떤 값을 쓰며 어떤 계산을 할 지 정하는 것을 카운터 형식으로 Behavioral Model로 구현하는 것과 clock의 negedge, posedge를 구분하며 분석하는 것들, 그리고 명령에 따라 카운터가 다른 방식으로 작동하는 것을 한 웨이브로 분석을 하자니 까다로운 일이 많았다.