Final Project: Forwarding MIPS CPU

CMPEN 331 Section 002

Chase McFarlane

cdm5688

12/14/20

ABSTRACT

The focus of this report is on a pipelined CPU design that implements the MIPS ISA. The MIPS architecture uses 32 registers (64 total if floating point registers are included) with 32-bit instructions. These instructions consist of 3 types, R-format, I-format, J-format. R-format instructions use data within 2 of the 32 registers to perform arithmetic, logical operations, etc. and the result of these are then written back into one of the 32-bit registers. I-type instructions have many uses from memory manipulation (loading and store data to memory), arithmetic with data from 1 register and with an immediate value, to branching for loops created in a higher-level language. J-Format instructions are mainly used for subroutines such as functions in Python.

INTRODUCTION

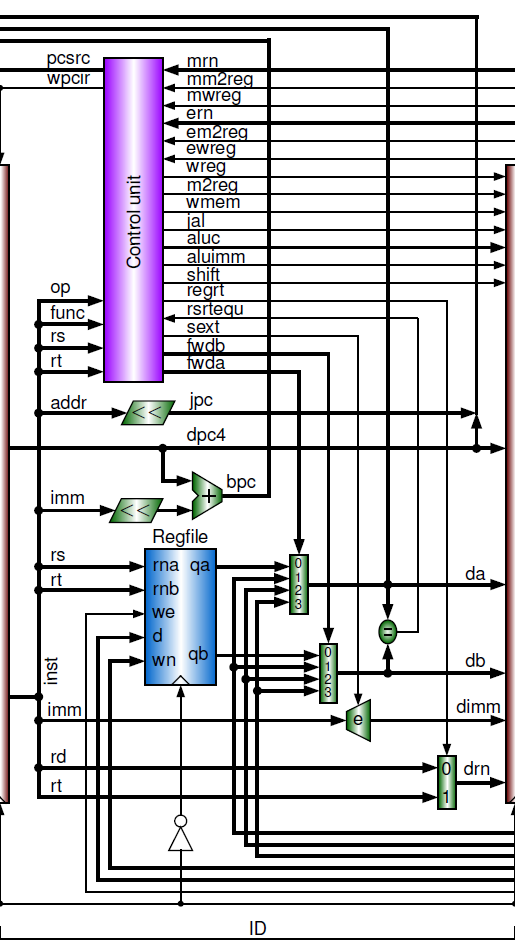
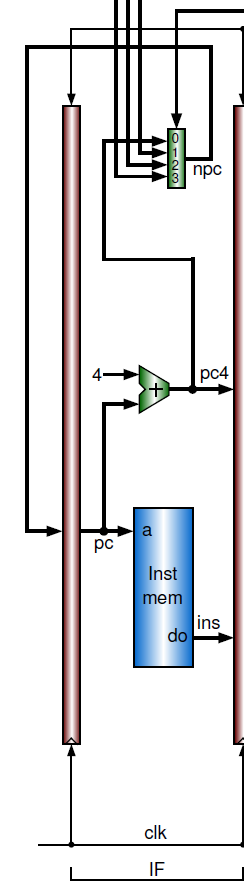
The advantages to a pipelined CPU vs a nonpipelined one are analogous to the advantages of Ford’s assembly line vs the competitor’s assembly approach at the time (1913). Ford was revolutionary and changed the manufacturing industry forever. Pipelined CPU did just about the same in the tech industry.

Instead of having many people create one car from beginning to end at one time, the assembly line allowed for many cars to be built simultaneously with workers specializing in one specific aspect of the build process. The same thought process was applied to CPUs where multiple instructions are being fetched, decoded, and executed instead of having to wait for every single instruction to be carried out before the next is fetched.

The assembly line was able to increase building efficiency which drove down manufacturing thus product costs. Pipelining also increases efficiency which allows the CPU to run faster thus increasing instruction execution throughput. This implementation was pivotal in the tech industry and has changed the game forever. All modern CPU are pipelined.

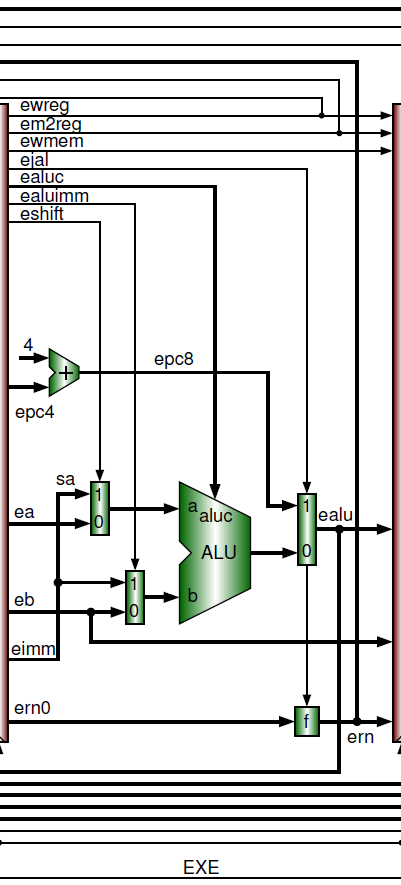
One of the first CPUs to utilize this advancement was implemented with MIPS ISA. In the MIPS architecture, pipelining is done in 5 stages: instruction fetch (IF), instruction decode (ID), execution (EX), memory access (MEM), and write back (WB).

The first stage, IF, a program counter (PC) – keeps track of the addresses of a program’s instructions – is incremented which is used for the input to the instruction memory where all the program’s instructions reside. The address of the PC is dependent on the instruction type (R-type, I-type, J-type) and specific instruction (add, j, jal, beq, etc.).

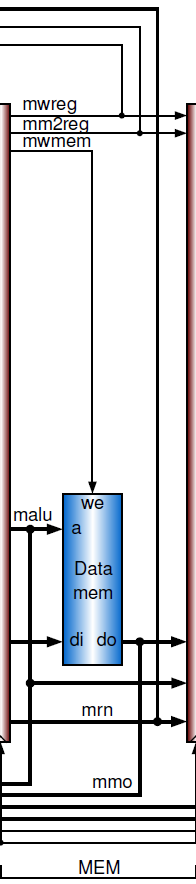
Next is the ID, which takes the instruction from the instruction memory and decodes the 1’s and 0’s into meaningful operations. The control unit identifies the instruction (add, sub, and, beq, bne, j, etc.) through the use of the op (6 right-most bits) and func (6 left-most bits). By analyzing these 12 bits, it will be able to determine what functionality the instruction has and will set the bits of certain signals. The control unit will also detect the need for forwarding and provide the remaining PC addressing signals.

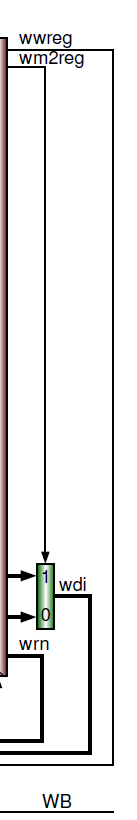
This is also the stage where the 32 registers reside. The registers hold values close to the CPU to increase the performance of the processor.

The rest of the decode determines what registers are being read from (if any) and what register is being written to (if any). From the signals created by the control unit, will dictate what register values will be used.



The EX stage performs the arithmetic of the instruction. The ALU or Arithmetic Logic Unit performs these operations whether that be to add, subtract, multiply, shift, etc. These results are used for register arithmetic write back, data memory addresses, etc.

The second to last stage is the data memory. This is the main memory of the system performing the program. Only two instructions will ever access the data memory which are store word (sw) or load word (lw). In an sw instruction, the value taken from the ALU will be written to a location into the memory. In an lw instruction, the ALU result is an address which will be used to location data inside the memory to be written back to a register for faster processing. Any other instruction uses the ALU result to be written directly back into a register without interacting with the data memory.

The final stage of the MIPS pipeline is the write back stage which simply takes the results from the memory stage and writes them back to a register in the ID stage. The data written back to registers is either the data located and extracted from the data memory or a raw ALU result from an R-Type instruction.

DESIGN CODE

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company: Penn State University

// Engineer: Chase McFarlane

//

// Create Date: 10/13/2020 10:42:34 AM

// Design Name:

// Module Name:

// Project Name: MIPS CPU

// Target Devices:

// Tool Versions:

// Description: Fully Pipelined MIPS CPU

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

//IF Stage

module ProgramCounter (clk, wpcir, nextPC, outputPC);

input clk, wpcir;

input [31:0] nextPC;

output reg [31:0] outputPC;

always @(posedge clk) begin

if (!wpcir) begin

outputPC <= nextPC;

end

end

endmodule

module PCmux(pcsrc, pc4, bpc, forwardA, jpc, nextpc);

input [1:0] pcsrc;

input [31:0] pc4, bpc, forwardA, jpc;

output reg [31:0] nextpc;

always @(pcsrc, pc4, bpc, forwardA, jpc) begin

case(pcsrc)

2'b00: nextpc <= pc4; //next instruction

2'b01: nextpc <= bpc; //Next instruction for beq and bne

2'b10: nextpc <= forwardA; //target address in register for a jr instruction

2'b11: nextpc <= jpc; //jump target address of a j or jal instruction

endcase

end

endmodule

module adder (currPC, pc4);

input [31:0] currPC;

output reg [31:0] pc4;

initial begin

pc4 = 0; //Set PC = 0

end

always @(currPC) begin

pc4 <= currPC + 4; //Increment PC by 4

end

endmodule

module instruction\_memory (address, instructionset); // instruction memory, rom

input [31:0] address; // rom address

output [31:0] instructionset; // rom content = rom[a]

wire [31:0] rom [0:63]; // rom cells: 64 words \* 32 bits

// rom[word\_addr] = instruction // (pc) label instruction

assign rom[6'h00] = 32'h3c010000; // (00) main: lui $1, 0

assign rom[6'h01] = 32'h34240050; // (04) ori $4, $1, 80

//assign rom[6'h02] = 32'h00000000; // (08) call: jal sum

assign rom[6'h02] = 32'h0c00001b; // (08) call: jal sum

assign rom[6'h03] = 32'h20050004; // (0c) dslot1: addi $5, $0, 4

assign rom[6'h04] = 32'hac820000; // (10) return: sw $2, 0($4)

assign rom[6'h05] = 32'h8c890000; // (14) lw $9, 0($4)

assign rom[6'h06] = 32'h01244022; // (18) sub $8, $9, $4

assign rom[6'h07] = 32'h20050003; // (1c) addi $5, $0, 3

assign rom[6'h08] = 32'h20a5ffff; // (20) loop2: addi $5, $5, -1

assign rom[6'h09] = 32'h34a8ffff; // (24) ori $8, $5, 0xffff

assign rom[6'h0a] = 32'h39085555; // (28) xori $8, $8, 0x5555

assign rom[6'h0b] = 32'h2009ffff; // (2c) addi $9, $0, -1

assign rom[6'h0c] = 32'h312affff; // (30) andi $10,$9,0xffff

assign rom[6'h0d] = 32'h01493025; // (34) or $6, $10, $9

assign rom[6'h0e] = 32'h01494026; // (38) xor $8, $10, $9

assign rom[6'h0f] = 32'h01463824; // (3c) and $7, $10, $6

assign rom[6'h10] = 32'h10a00003; // (40) beq $5, $0, shift

assign rom[6'h11] = 32'h00000000; // (44) dslot2: nop

assign rom[6'h12] = 32'h08000008; // (48) j loop2

assign rom[6'h13] = 32'h00000000; // (4c) dslot3: nop

assign rom[6'h14] = 32'h2005ffff; // (50) shift: addi $5, $0, -1

assign rom[6'h15] = 32'h000543c0; // (54) sll $8, $5, 15

assign rom[6'h16] = 32'h00084400; // (58) sll $8, $8, 16

assign rom[6'h17] = 32'h00084403; // (5c) sra $8, $8, 16

assign rom[6'h18] = 32'h000843c2; // (60) srl $8, $8, 15

assign rom[6'h19] = 32'h08000019; // (64) finish: j finish

assign rom[6'h1a] = 32'h00000000; // (68) dslot4: nop

assign rom[6'h1b] = 32'h00004020; // (6c) sum: add $8, $0, $0

assign rom[6'h1c] = 32'h8c890000; // (70) loop: lw $9, 0($4)

assign rom[6'h1d] = 32'h01094020; // (74) stall: add $8, $8, $9

assign rom[6'h1e] = 32'h20a5ffff; // (78) addi $5, $5, -1

//assign rom[6'h1f] = 32'h00000000; // (7c) bne $5, $0, loop

assign rom[6'h1f] = 32'h14a0fffc; // (7c) bne $5, $0, loop

assign rom[6'h20] = 32'h20840004; // (80) dslot5: addi $4, $4, 4

assign rom[6'h21] = 32'h03e00008; // (84) jr $31

assign rom[6'h22] = 32'h00081000; // (88) dslot6: sll $2, $8, 0

assign instructionset = rom[address[7:2]]; // use 6-bit word address to read rom

endmodule

module InstrMem (address, instructionset);

input [31:0] address;

output reg [31:0] instructionset;

reg [31:0] IM[0:511];

initial begin

IM[100] = 32'b000000\_00001\_00010\_00011\_00000\_100000;

IM[104] = 32'b000000\_01001\_00011\_00100\_00000\_100010;

IM[108] = 32'b000000\_00011\_01001\_00101\_00000\_100101;

IM[112] = 32'b000000\_00011\_01001\_00110\_00000\_100110;

IM[116] = 32'b000000\_00011\_01001\_00111\_00000\_100100;

end

always @(address) begin

instructionset <= IM[address];

end

endmodule

module IFID(clk, wpcir, pc4, instructionset, dpc4, op, rs, rt, rd, shamt, func, imm, addr);

input clk, wpcir;

input [31:0] pc4, instructionset;

output reg [4:0] rs, rt, rd, shamt;

output reg [5:0] op, func;

output reg [15:0] imm;

output reg [25:0] addr;

output reg [31:0] dpc4;

always @(posedge clk) begin

if (!wpcir) begin

dpc4 <= pc4;

op <= instructionset[31:26];

rs <= instructionset[25:21];

rt <= instructionset[20:16];

rd <= instructionset[15:11];

shamt <= instructionset[10:6];

func <= instructionset[5:0];

imm <= instructionset[15:0];

addr <= instructionset[25:0];

end

else begin

op <= 6'b000000;

rs <= 5'b00000;

rt <= 5'b00000;

rd <= 5'b00000;

shamt <= 5'b00000;

func <= 6'b000000;

imm <= 16'h0000;

addr <= 26'h000000;

end

end

endmodule

module ControlUnit (pcsrc, wpcir, op, func, rs, rt, mrn, mm2reg, mwreg, ern, em2reg, ewreg, wreg, m2reg, wmem, jal, aluc, aluimm, shift, regrt, rsrtequ, sext, fwdb, fwda);

input [5:0] op, func;

input [4:0] rs, rt, mrn, ern;

input mm2reg, mwreg, em2reg, ewreg, rsrtequ;

output reg wpcir, wreg, m2reg, wmem, jal, aluimm, shift, regrt, sext;

output reg [3:0] aluc;

output reg [1:0] pcsrc, fwdb, fwda;

initial begin

pcsrc = 0;

wpcir = 0;

end

always @\* begin

//R-Type

if (op == 6'b000000) begin

m2reg <= 0;

wmem <= 0;

aluimm <= 0;

regrt <= 0;

jal <= 0;

if (func == 6'b000000) begin //sll

wreg <= 1;

aluc <= 4'b1111;

shift <= 1;

sext <= 1;

pcsrc <= 0;

end

else if (func == 6'b000010) begin //srl

wreg <= 1;

aluc <= 4'b1110;

shift <= 1;

sext <= 1;

pcsrc <= 0;

end

else if (func == 6'b000011) begin //sra

wreg <= 1;

aluc <= 4'b1101;

shift <= 1;

sext <= 1;

pcsrc <= 0;

end

else if (func == 6'b001000) begin //jr

wreg <= 0;

aluc <= 4'b1000;

shift <= 0;

sext <= 0;

pcsrc <= 2;

end

else if (func == 6'b100000) begin //add

wreg <= 1;

aluc <= 4'b0010;

shift <= 0;

sext <= 0;

pcsrc <= 0;

end

else if (func == 6'b100010) begin //sub

wreg <= 1;

aluc <= 4'b0110;

shift <= 0;

sext <= 0;

pcsrc <= 0;

end

else if (func == 6'b100100) begin //AND;

wreg <= 1;

aluc <= 4'b0000;

shift <= 0;

sext <= 0;

pcsrc <= 0;

end

else if (func == 6'b100101) begin //OR

wreg <= 1;

aluc <= 4'b0001;

shift <= 0;

sext <= 0;

pcsrc <= 0;

end

else if (func == 6'b100110) begin //XOR

wreg <= 1;

aluc <= 4'b0011;

shift <= 0;

sext <= 0;

pcsrc <= 0;

end

else if (func == 6'b101010) begin //slt

wreg <= 1;

aluc <= 4'b0111;

shift <= 0;

sext <= 0;

pcsrc <= 0;

end

end

else if (op == 6'b001000) begin //addi

wreg <= 1;

m2reg <= 0;

wmem <= 0;

aluimm <= 1;

regrt <= 1;

aluc <= 4'b0010;

shift <= 0;

sext <= 1;

jal <= 0;

pcsrc <= 0;

end

else if (op == 6'b001100) begin //andi

wreg <= 1;

m2reg <= 0;

wmem <= 0;

aluimm <= 1;

regrt <= 1;

aluc <= 4'b0000;

shift <= 0;

sext <= 0;

jal <= 0;

pcsrc <= 0;

end

else if (op == 6'b001101) begin //ori

wreg <= 1;

m2reg <= 0;

wmem <= 0;

aluimm <= 1;

regrt <= 1;

aluc <= 4'b0001;

shift <= 0;

sext <= 0;

jal <= 0;

pcsrc <= 0;

end

else if (op == 6'b001110) begin //xori

wreg <= 1;

m2reg <= 0;

wmem <= 0;

aluimm <= 1;

regrt <= 1;

aluc <= 4'b0011;

shift <= 0;

sext <= 0;

jal <= 0;

pcsrc <= 0;

end

else if (op == 6'b001111) begin //lui

wreg <= 1;

m2reg <= 0;

wmem <= 0;

aluimm <= 1;

regrt <= 1;

aluc <= 4'b0100;

shift <= 0;

sext <= 0;

jal <= 0;

pcsrc <= 0;

end

else if (op == 6'b100011) begin //lw

wreg <= 1;

m2reg <= 1;

wmem <= 0;

aluc <= 4'b0010;

aluimm <= 1;

regrt <= 1;

shift <= 0;

sext <= 0;

jal <= 0;

pcsrc <= 0;

end

else if (op == 6'b101011) begin //sw

wreg <= 0;

m2reg <= 0;

wmem <= 1;

aluc <= 4'b0010;

aluimm <= 1;

regrt <= 1;

shift <= 0;

sext <= 0;

jal <= 0;

pcsrc <= 0;

end

else if (op == 6'b000100) begin //beq

wreg <= 0;

m2reg <= 0;

wmem <= 0;

aluc <= 4'b0110;

aluimm <= 0;

regrt <= 1;

shift <= 0;

sext <= 0;

jal <= 0;

case(rsrtequ)

1'b0: pcsrc <= 0;

1'b1: pcsrc <= 1;

endcase

end

else if (op == 6'b000101) begin //bne

wreg <= 0;

m2reg <= 0;

wmem <= 0;

aluc <= 4'b1110;

aluimm <= 0;

regrt <= 1;

shift <= 0;

sext <= 0;

jal <= 0;

case(rsrtequ)

1'b0: pcsrc <= 1;

1'b1: pcsrc <= 0;

endcase

end

//J-Type

else if (op == 6'b000010) begin //j

wreg <= 0;

m2reg <= 0;

wmem <= 0;

aluc <= 4'b1000;

aluimm <= 0;

regrt <= 1;

shift <= 0;

sext <= 0;

jal <= 0;

pcsrc <= 3;

end

else if (op == 6'b000011) begin //jal

wreg <= 1;

m2reg <= 0;

wmem <= 0;

aluc <= 4'b1000;

aluimm <= 0;

regrt <= 1;

shift <= 0;

sext <= 1;

jal <= 1;

pcsrc <= 3;

end

end

always @(rs, rt, mrn, mm2reg, mwreg, ern, em2reg, ewreg) begin

//EXE/MEM.RegisterRd == ID/EXE.RegisterRs

if ((ewreg) && (ern != 0) && rs == ern) begin

fwda = 01;

end

//MEM/WB.RegisterRd == ID/EXE.RegisterRs

else if ((mwreg) && (!mm2reg) && (mrn != 0) && rs == mrn) begin

fwda = 10;

end

//do

else if ((mwreg) && (em2reg) && (mrn != 0) && rs == mrn) begin

fwda = 11;

end

//Use qa

else begin

fwda = 00;

end

//EXE/MEM.RegisterRd == ID/EXE.RegisterRt

if ((ewreg) && (ern != 0) && rt == ern) begin

fwdb = 01;

end

//MEM/WB.RegisterRd == ID/EXE.RegisterRt

else if ((mwreg) && (!mm2reg) && (mrn != 0) && rt == mrn) begin

fwdb = 10;

end

//do

else if ((mwreg) && (em2reg) && (mrn != 0) && rt == mrn) begin

fwdb = 11;

end

else begin

fwdb = 00;

end

end

always @\* begin

if (ewreg && em2reg && (ern != 0) && (rs && (ern == rs) || rt && (ern == rt))) begin

wpcir <= 1;

end

else begin

wpcir <= 0;

end

end

endmodule

module JumpPC(dpc4, addr, jpc);

input [31:0] dpc4;

input [25:0] addr;

output reg [31:0] jpc;

always @(addr) begin

jpc <= {dpc4[31:28], addr, 2'b00};

end

endmodule

module BranchPCShift(imm, branchshift);

input [15:0] imm;

output reg [31:0] branchshift;

always @(imm) begin

branchshift <= {{14{imm[15]}}, imm, 2'b00};

end

endmodule

module BranchPC(dpc4, branchshift, bpc);

input [31:0] dpc4, branchshift;

output reg [31:0] bpc;

always @(branchshift, dpc4) begin

bpc <= dpc4 + branchshift;

end

endmodule

module Regfile (clk, we, rs, rt, wn, d, qa, qb);

input clk, we;

input [4:0] rs, rt, wn;

input [31:0] d;

output reg [31:0] qa, qb;

integer i;

reg [31:0] RF[0:31];

initial begin

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

RF[i] = 0;

end

end

always @(negedge clk, rs, rs) begin

qa <= RF[rs];

qb <= RF[rt];

end

always @(posedge clk, we, wn, d) begin

if (we) begin

RF[wn] <= d;

end

end

endmodule

module ForwardAMux(fwda, qa, ALUresultEXE, ALUresultMEM, do, ForwardAOutput);

input [1:0] fwda;

input [31:0] qa, ALUresultEXE, ALUresultMEM, do;

output reg [31:0] ForwardAOutput;

always @(fwda, qa, ALUresultEXE, ALUresultMEM, do) begin

case(fwda)

2'b00: ForwardAOutput <= qa;

2'b01: ForwardAOutput <= ALUresultEXE;

2'b10: ForwardAOutput <= ALUresultMEM;

2'b11: ForwardAOutput <= do;

endcase

end

endmodule

module ForwardBMux(fwdb, qb, ALUresultEXE, ALUresultMEM, do, ForwardBOutput);

input [1:0] fwdb;

input [31:0] qb, ALUresultEXE, ALUresultMEM, do;

output reg [31:0] ForwardBOutput;

always @(fwdb, qb, ALUresultEXE, ALUresultMEM, do) begin

case(fwdb)

2'b00: ForwardBOutput <= qb;

2'b01: ForwardBOutput <= ALUresultEXE;

2'b10: ForwardBOutput <= ALUresultMEM;

2'b11: ForwardBOutput <= do;

endcase

end

endmodule

module rsrtEqualCheck(ForwardA, ForwardB, rsrtequ);

input [31:0] ForwardA, ForwardB;

output reg rsrtequ;

always @(ForwardA, ForwardB) begin

if (ForwardA == ForwardB) begin

rsrtequ <= 1;

end

else begin

rsrtequ <= 0;

end

end

endmodule

module SignExt (sext, immediate, signextout);

input sext;

input [15:0] immediate;

output reg [31:0] signextout;

always @(immediate, sext) begin

if (sext) begin

signextout <= {{16{immediate[15]}}, immediate[15:0]};

end

else begin

signextout <= {16'b0, immediate[15:0]};

end

end

endmodule

module rsrtMux(regrt, rt, rd, IDmuxout);

input regrt;

input [4:0] rd, rt;

output reg [4:0] IDmuxout;

always @(rd, rt, regrt) begin

if (!regrt) begin

IDmuxout <= rd;

end

else begin

IDmuxout <= rt;

end

end

endmodule

module IDEXE(clk, wreg, m2reg, wmem, jal, aluc, aluimm, shift, dpc4, shamt, da, db, dimm, drn,

ewreg, em2reg, ewmem, ejal, ealuc, ealuimm, eshift, epc4, eshamt, ea, eb, eimm, ern0);

input clk, wreg, m2reg, wmem, jal, aluimm, shift;

input [3:0] aluc;

input [4:0] shamt, drn;

input [31:0] dpc4, da, db, dimm;

output reg ewreg, em2reg, ewmem, ejal, ealuimm, eshift;

output reg [3:0] ealuc;

output reg [4:0] eshamt, ern0;

output reg [31:0] epc4, ea, eb, eimm;

always @(posedge clk) begin

ewreg <= wreg;

em2reg <= m2reg;

ewmem <= wmem;

ealuc <= aluc;

ejal <= jal;

ealuimm <= aluimm;

eshift <= shift;

eshamt <= shamt;

epc4 <= dpc4;

ea <= da;

eb <= db;

eimm <= dimm;

ern0 <= drn;

end

endmodule

module epc4Adder(epc4, epc8);

input [31:0] epc4;

output reg [31:0] epc8;

always @(epc4) begin

epc8 <= epc4 + 4;

end

endmodule

module ALUmuxA(eshift, forwardA, shamt, ALUmuxoutA);

input eshift;

input [4:0] shamt;

input [31:0] forwardA;

output reg [31:0] ALUmuxoutA;

always @(forwardA, shamt, eshift) begin

if (eshift == 0) begin

ALUmuxoutA <= forwardA;

end

else begin

ALUmuxoutA <= shamt;

end

end

endmodule

module ALUmuxB(ealuimm, qb, signext, ALUmuxoutB);

input ealuimm;

input [31:0] qb, signext;

output reg [31:0] ALUmuxoutB;

always @(qb, signext, ealuimm) begin

if (ealuimm == 0) begin

ALUmuxoutB <= qb;

end

else begin

ALUmuxoutB <= signext;

end

end

endmodule

module ALU(aluc, a, b, r);

input [3:0] aluc;

input [31:0] a, b;

output reg [31:0] r;

always @(aluc, a, b) begin

case(aluc)

4'b0000: r = a & b; //and

4'b0001: r = a | b; //or

4'b0010: r = a + b; //add

4'b0011: r = a ^ b; //xor

4'b0100: r = b << 16; //lui

4'b0110: r = a - b; //sub

4'b0111: r = (a < b) ? 1 : 0; //slt

4'b1000: r = 0; //doesn't matter

4'b1101: r = a >>> b; //sra

4'b1110: r = a >> b; //srl

4'b1111: r = a << b; //sll

endcase

end

endmodule

module PostALUMux(ejal, epc8, ALUresult, ealu);

input ejal;

input [31:0] epc8, ALUresult;

output reg [31:0] ealu;

always @(ejal, epc8, ALUresult) begin

if (ejal) begin

ealu <= epc8;

end

else begin

ealu <= ALUresult;

end

end

endmodule

module f(ejal, ern0, ern);

input ejal;

input [4:0] ern0;

output reg [4:0] ern;

always @(ern0, ejal)

if (ejal) begin

ern <= 5'b11111;

end

else begin

ern <= ern0;

end

endmodule

module EXEMEM(clk, ewreg, em2reg, ewmem, ealu, eb, ern,

mwreg, mm2reg, mwmem, malu, di, mrn);

input clk, ewreg, em2reg, ewmem;

input [4:0] ern;

input [31:0] ealu, eb;

output reg mwreg, mm2reg, mwmem;

output reg [4:0] mrn;

output reg [31:0] malu, di;

always @(posedge clk) begin

mwreg <= ewreg;

mm2reg <= em2reg;

mwmem <= ewmem;

mrn <= ern;

malu <= ealu;

di <= eb;

end

endmodule

module data\_memory (clk, we, addr, datain, dataout); // data memory, ram

input clk, we; // clock

input [31:0] addr; // ram address

input [31:0] datain; // data in (to memory)

output [31:0] dataout; // data out (from memory)

reg [31:0] ram [0:31]; // ram cells: 32 words \* 32 bits

assign dataout = ram[addr[6:2]]; // use 5-bit word address

always @ (posedge clk) begin

if (we) ram[addr[6:2]] = datain; // write ram

end

integer i;

initial begin // ram initialization

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

ram[i] = 0;

// ram[word\_addr] = data // (byte\_addr) item in data array

ram[5'h14] = 32'h000000a3; // (50) data[0] 0 + a3 = a3

ram[5'h15] = 32'h00000027; // (54) data[1] a3 + 27 = ca

ram[5'h16] = 32'h00000079; // (58) data[2] ca + 79 = 143

ram[5'h17] = 32'h00000115; // (5c) data[3] 143 + 115 = 258

// ram[5'h18] should be 0x00000258, the sum stored by sw instruction

end

endmodule

module DataMem(we, a, di, do);

input we;

input [31:0] a, di;

output reg [31:0] do;

reg [31:0] DM[0:255];

initial begin

DM[0] = 32'hA00000AA;

DM[4] = 32'h10000011;

DM[8] = 32'h20000022;

DM[12] = 32'h30000033;

DM[16] = 32'h40000044;

DM[20] = 32'h50000055;

DM[24] = 32'h60000066;

DM[28] = 32'h70000077;

DM[32] = 32'h80000088;

DM[36] = 32'h90000099;

end

always @(a, di, we) begin

if (we == 0) begin

do <= DM[a];

end

else begin

DM[a] <= di;

end

end

endmodule

module MEMWB(clk, mwreg, mm2reg, malu, do, mrn,

wwreg, wm2reg, walu, wbdo, wrn);

input clk, mwreg, mm2reg;

input [4:0] mrn;

input [31:0] malu, do;

output reg wwreg, wm2reg;

output reg [4:0] wrn;

output reg [31:0] walu, wbdo;

always @(posedge clk) begin

wwreg <= mwreg;

wm2reg <= mm2reg;

wrn <= mrn;

walu <= malu;

wbdo <= do;

end

endmodule

//Write back stage

module WBmux(wm2reg, walu, wbdo, wdi);

input wm2reg;

input [31:0] walu, wbdo;

output reg [31:0] wdi;

always @(wm2reg, walu, wbdo) begin

if (wm2reg) begin

wdi <= wbdo;

end

else begin

wdi <= walu;

end

end

endmodule

TESTBENCH CODE

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 10/14/2020 02:05:46 PM

// Design Name:

// Module Name: testbench

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module testbench();

reg clk\_tb;

//Stage 1 wires

wire [31:0] pc4\_tb, npc\_tb, pc\_tb, instruction\_tb;

//Stage 2 wires

wire [5:0] op\_tb;

wire [4:0] rs\_tb, rt\_tb, rd\_tb, shamt\_tb;

wire [5:0] func\_tb;

wire [15:0] imm\_tb;

wire [25:0] addr\_tb;

wire wpcir\_tb, wreg\_tb, m2reg\_tb, wmem\_tb, jal\_tb, aluimm\_tb, shift\_tb, regrt\_tb, rsrtequ\_tb, sext\_tb;

wire [1:0] pcsrc\_tb, fwda\_tb, fwdb\_tb;

wire [3:0] aluc\_tb;

wire [31:0] jpc\_tb, dpc4, branchshift\_tb, bpc\_tb, qa\_tb, da\_tb, qb\_tb, db\_tb, dimm\_tb;

wire [4:0] drn\_tb;

//Stage 3 wires

wire ewreg\_tb, em2reg\_tb, ewmem\_tb, ejal\_tb, ealuimm\_tb, eshift\_tb;

wire [3:0] ealuc\_tb;

wire [4:0] eshamt\_tb;

wire [31:0] epc4\_tb, epc8\_tb, ea\_tb, eb\_tb, eimm\_tb, ALUinputA\_tb, ALUinputB\_tb, ALUresult\_tb, ealu\_tb;

wire [4:0] ern0\_tb, ern\_tb;

//Stage 4 wires

wire mwreg\_tb, mm2reg\_tb, mwmem\_tb;

wire [4:0] mrn\_tb;

wire [31:0] malu\_tb, di\_tb, do\_tb;

//Stage 5 wires

wire wwreg\_tb, wm2reg\_tb;

wire [4:0] wrn\_tb;

wire [31:0] walu\_tb, wdo\_tb, wdi\_tb;

//Instruction Fetch Modules

ProgramCounter PC(clk\_tb, wpcir\_tb, npc\_tb, pc\_tb);

adder incrementer(pc\_tb, pc4\_tb);

PCmux muxPC(pcsrc\_tb, pc4\_tb, bpc\_tb, da\_tb, jpc\_tb, npc\_tb);

instruction\_memory IM(pc\_tb, instruction\_tb);

//Decode Modules

IFID IF\_ID(clk\_tb, wpcir\_tb, pc4\_tb, instruction\_tb, dpc4, op\_tb, rs\_tb, rt\_tb, rd\_tb, shamt\_tb, func\_tb, imm\_tb, addr\_tb);

ControlUnit CU(pcsrc\_tb, wpcir\_tb, op\_tb, func\_tb, rs\_tb, rt\_tb, mrn\_tb, mm2reg\_tb, mwreg\_tb,

ern\_tb, em2reg\_tb, ewreg\_tb, wreg\_tb, m2reg\_tb, wmem\_tb, jal\_tb, aluc\_tb, aluimm\_tb, shift\_tb, regrt\_tb, rsrtequ\_tb, sext\_tb, fwdb\_tb, fwda\_tb);

JumpPC Jump(pc4\_tb,addr\_tb, jpc\_tb);

BranchPCShift BranchShift(imm\_tb, branchshift\_tb);

BranchPC BranchPC(dpc4, branchshift\_tb, bpc\_tb);

Regfile RF(clk\_tb, wwreg\_tb, rs\_tb, rt\_tb, wrn\_tb, wdi\_tb, qa\_tb, qb\_tb);

ForwardAMux ForwardA(fwda\_tb, qa\_tb, ealu\_tb, malu\_tb, do\_tb, da\_tb);

ForwardBMux ForwardB(fwdb\_tb, qb\_tb, ealu\_tb, malu\_tb, do\_tb, db\_tb);

rsrtEqualCheck rsrtEqual(da\_tb, db\_tb, rsrtequ\_tb);

SignExt SignExtender(sext\_tb, imm\_tb, dimm\_tb);

rsrtMux rsrtMux(regrt\_tb, rt\_tb, rd\_tb, drn\_tb);

//Execution Modules

IDEXE ID\_EXE(clk\_tb, wreg\_tb, m2reg\_tb, wmem\_tb, jal\_tb, aluc\_tb, aluimm\_tb, shift\_tb, dpc4, shamt\_tb, da\_tb, db\_tb, dimm\_tb, drn\_tb,

ewreg\_tb, em2reg\_tb, ewmem\_tb, ejal\_tb, ealuc\_tb, ealuimm\_tb, eshift\_tb, epc4\_tb, eshamt\_tb, ea\_tb, eb\_tb, eimm\_tb, ern0\_tb);

epc4Adder epc4Adder(epc4\_tb, epc8\_tb);

ALUmuxA ALUmuxA(eshift\_tb, ea\_tb, eimm\_tb, ALUinputA\_tb);

ALUmuxB ALUmuxB(ealuimm\_tb, eb\_tb, eimm\_tb, ALUinputB\_tb);

ALU ALU(ealuc\_tb, ALUinputA\_tb, ALUinputB\_tb, ALUresult\_tb);

PostALUMux PostALUMux(ejal\_tb, epc8\_tb, ALUresult\_tb, ealu\_tb);

f u(ejal\_tb, ern0\_tb, ern\_tb);

//Memory Modules

EXEMEM EXE\_MEM(clk\_tb, ewreg\_tb, em2reg\_tb, ewmem\_tb, ealu\_tb, eb\_tb, ern\_tb,

mwreg\_tb, mm2reg\_tb, mwmem\_tb, malu\_tb, di\_tb, mrn\_tb);

data\_memory DM(clk\_tb, mwmem\_tb, malu\_tb, di\_tb, do\_tb);

//Write Back Module

MEMWB MEM\_WB(clk\_tb, mwreg\_tb, mm2reg\_tb, malu\_tb, do\_tb, mrn\_tb,

wwreg\_tb, wm2reg\_tb, walu\_tb, wdo\_tb, wrn\_tb);

WBmux muxWB(wm2reg\_tb, walu\_tb, wdo\_tb, wdi\_tb);

// wire [31:0] npc\_tb, pc\_tb, instruction\_tb, instrsetout\_tb, signextout\_tb, qa\_tb, qb\_tb, eqa\_tb, eqb\_tb, outsignext, exemuxout, ALUresult, memALUresult, di\_tb, do\_tb, wbALUresult, wbdo, wbmux\_output;

// wire [4:0] muxout, emux, memmuxout, wbmuxout;

// wire [3:0] aluc, ealuc;

// wire regwrite,memtoreg, memwrite, aluimm, regrt;

// wire eregwrite, ememtoreg, ememwrite, ealuimm;

// wire mwreg, mm2reg, mwmem;

// wire wwreg, wm2reg;

// ProgramCounter PC(clk\_tb, npc\_tb, pc\_tb);

// adder incrementer(pc\_tb, npc\_tb);

// InstrMem IM(pc\_tb, instruction\_tb);

// IFID IF\_ID(clk\_tb, instruction\_tb, instrsetout\_tb);

// ControlUnit CU(instrsetout\_tb[31:26], instrsetout\_tb[5:0], regwrite, memtoreg, memwrite, aluc, aluimm, regrt);

// mux muxID(regrt, instrsetout\_tb[15:11], instrsetout\_tb[20:16], muxout);

// Regfile RegisterFile(clk\_tb, wwreg, instrsetout\_tb[25:21], instrsetout\_tb[20:16], wbmuxout, wbmux\_output, qa\_tb, qb\_tb);

// SignExt signextenter(instrsetout\_tb[15:0], signextout\_tb);

// IDEXE ID\_EXE(clk\_tb, regwrite, memtoreg, memwrite, aluc, aluimm, muxout, qa\_tb, qb\_tb, signextout\_tb, eregwrite, ememtoreg, ememwrite, ealuc, ealuimm, emux, eqa\_tb, eqb\_tb, outsignext);

// ALUmux muxEXE(ealuimm, eqb\_tb, outsignext, exemuxout);

// ALU ALUEXE(ealuc, eqa\_tb, exemuxout, ALUresult);

// EXEMEM EXE\_MEM(clk\_tb, eregwrite, ememtoreg, ememwrite, emux, ALUresult, eqb\_tb, mwreg, mm2reg, mwmem, memmuxout,memALUresult, di\_tb);

// DataMem DM(mwmem, memALUresult, di\_tb, do\_tb);

// MEMWB MEM\_WB(clk\_tb, mwreg, mm2reg, memmuxout, memALUresult, do\_tb, wwreg, wm2reg, wbmuxout, wbALUresult, wbdo);

// WBmux muxWB(wm2reg, wbALUresult, wbdo, wbmux\_output);

initial begin

clk\_tb = 1'b1;

end

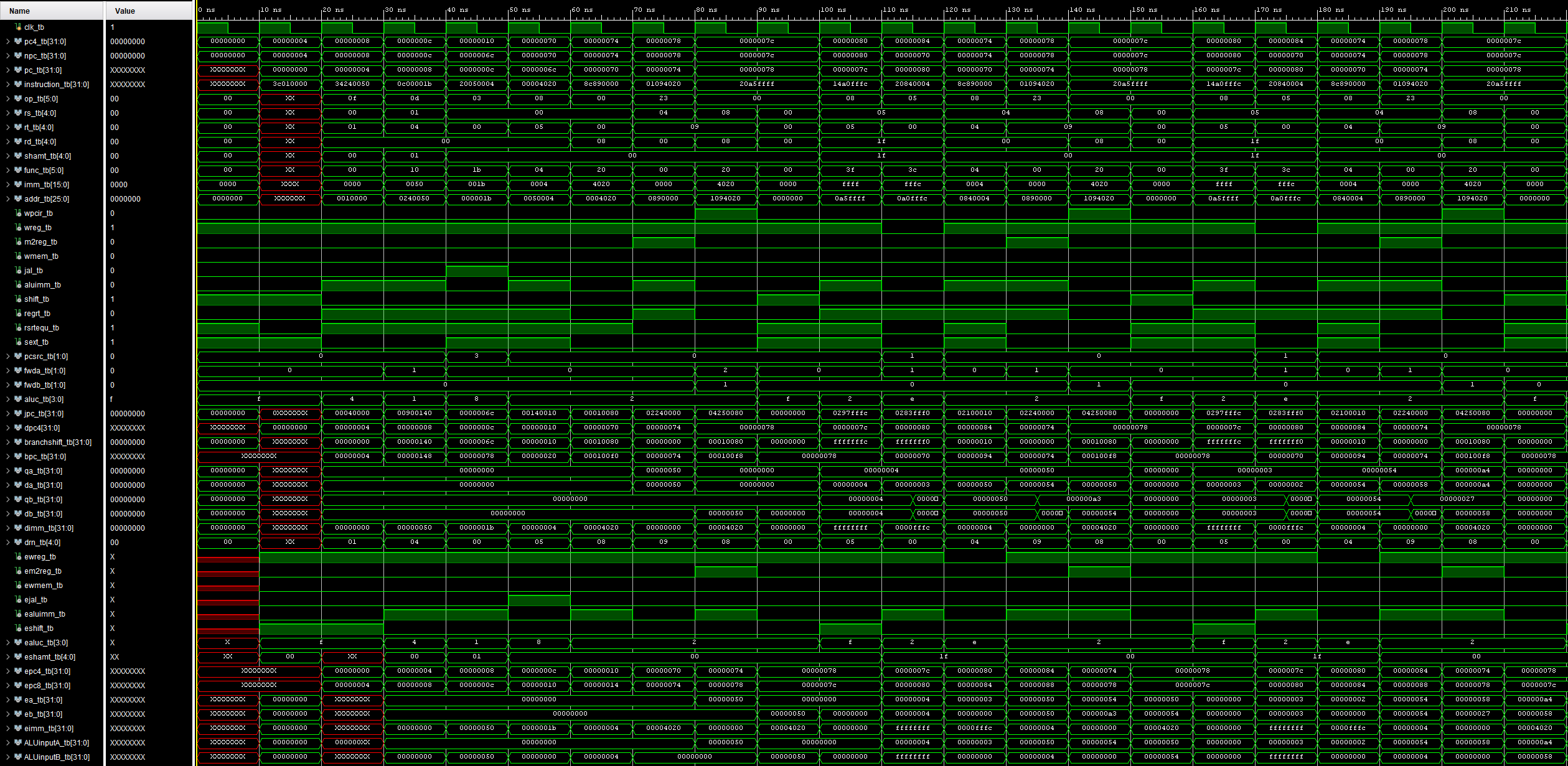
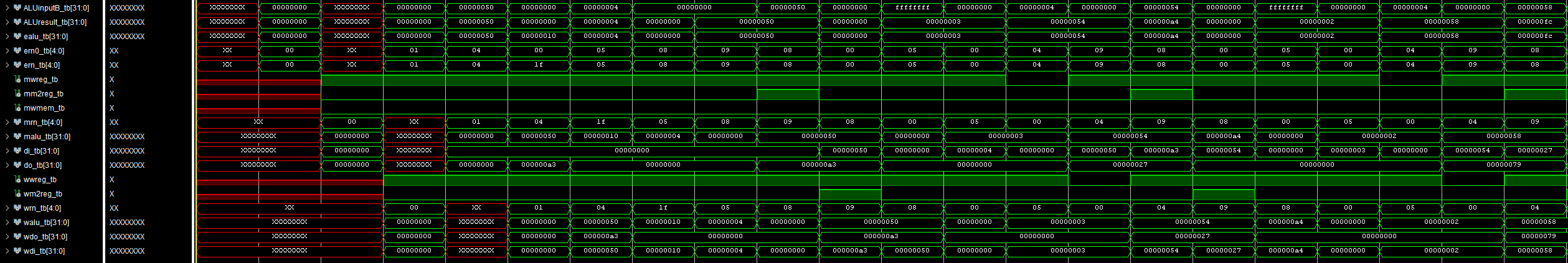
always @(clk\_tb) begin

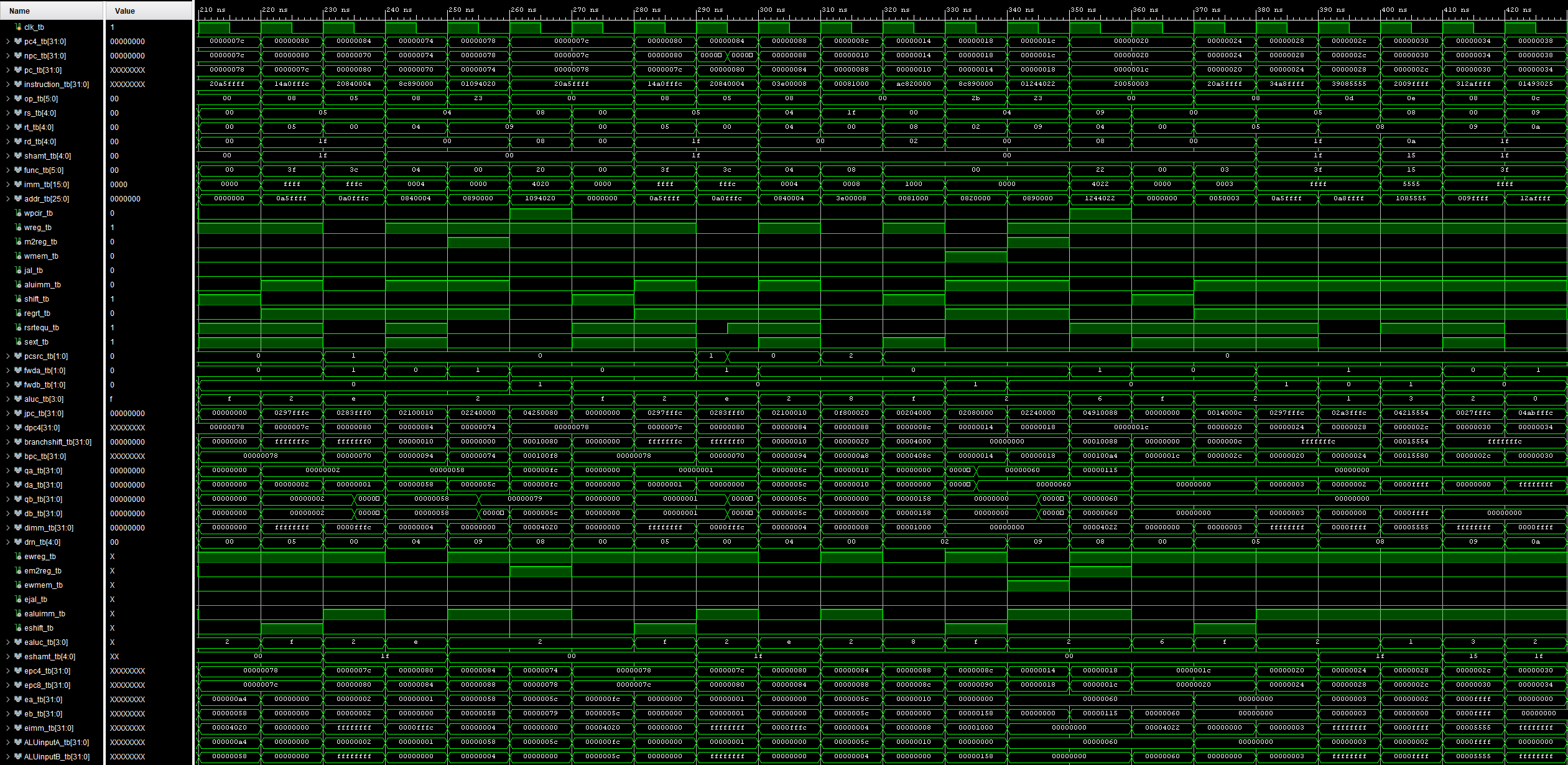
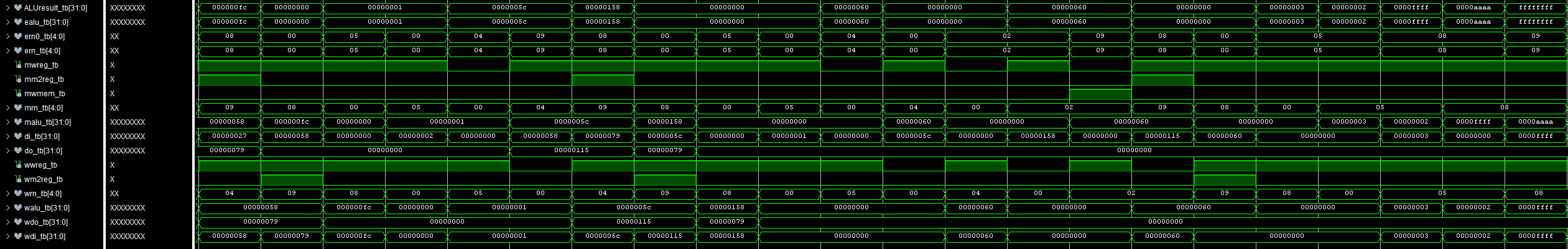
#5

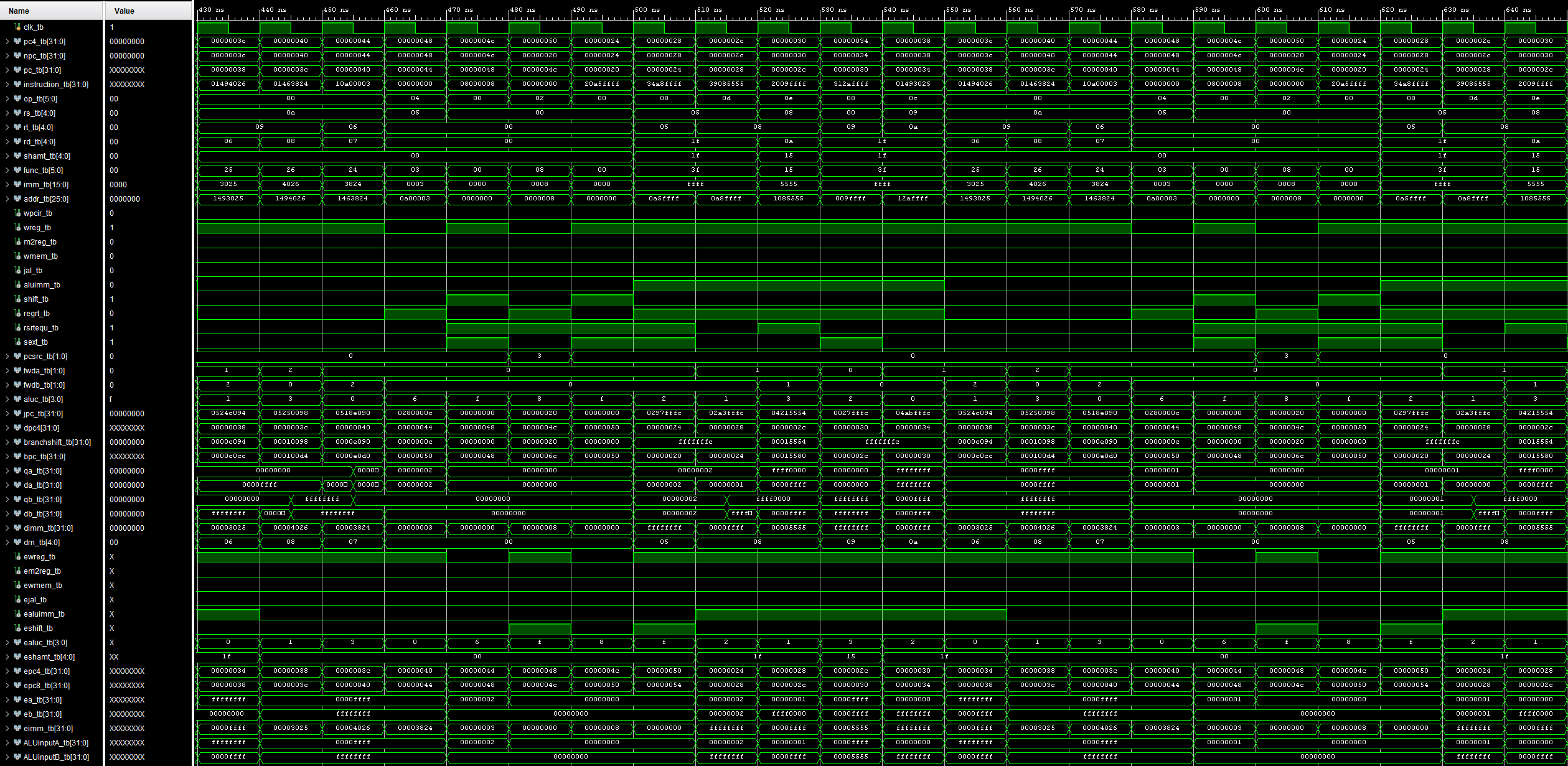
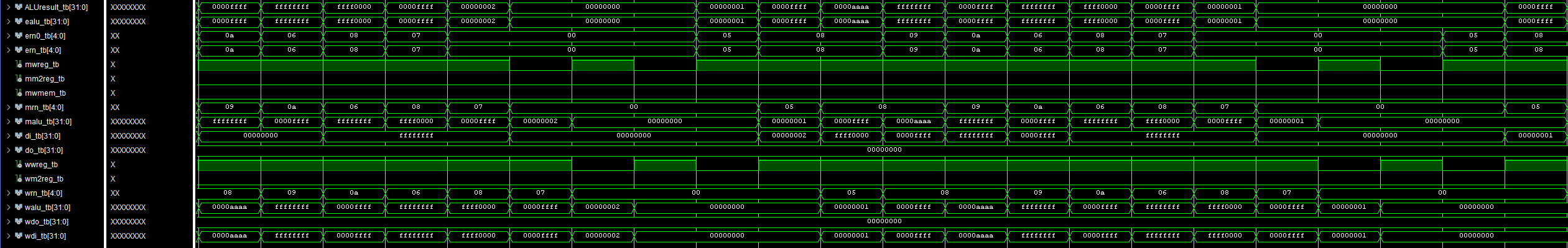
clk\_tb <= ~clk\_tb;

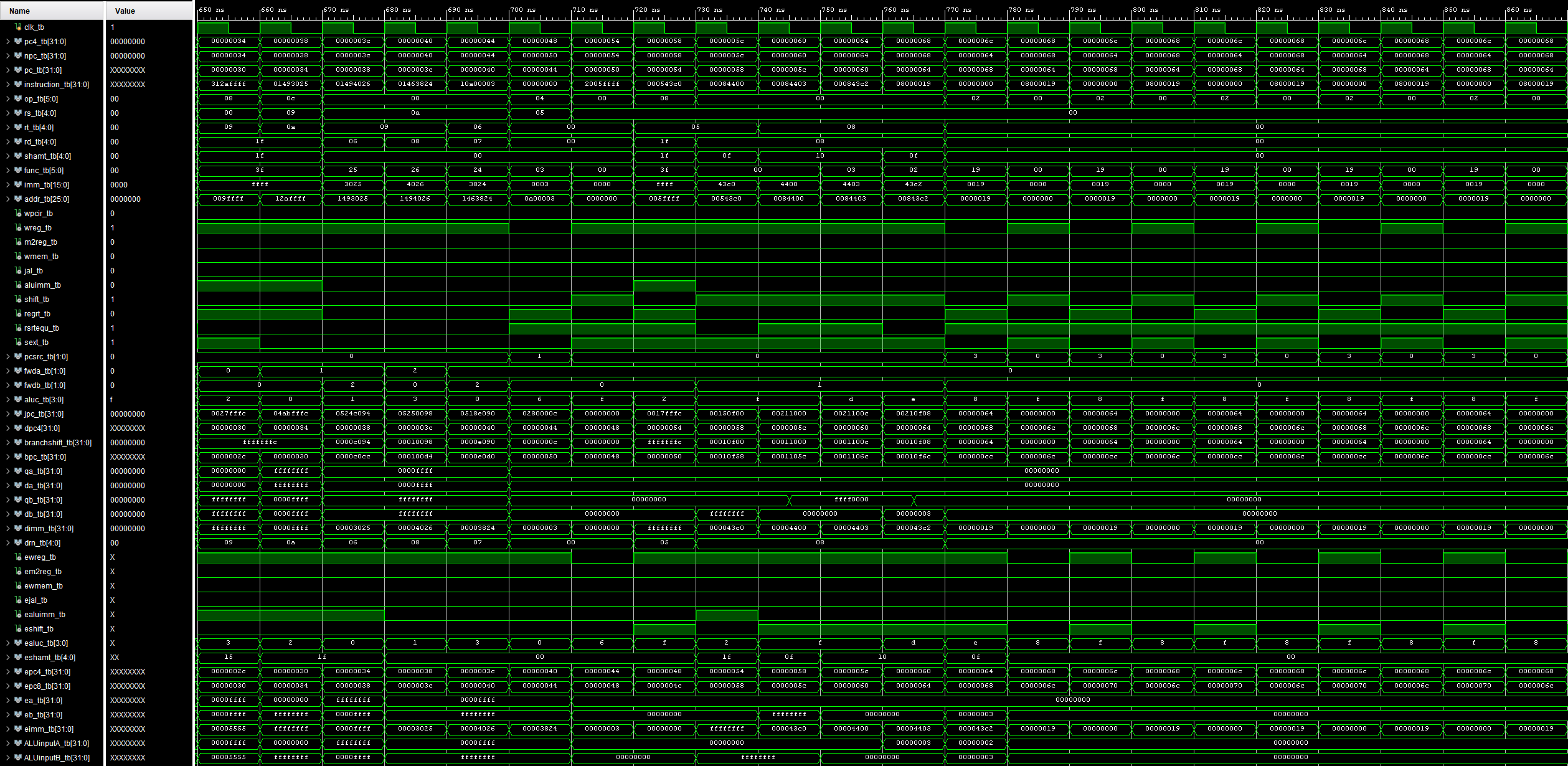
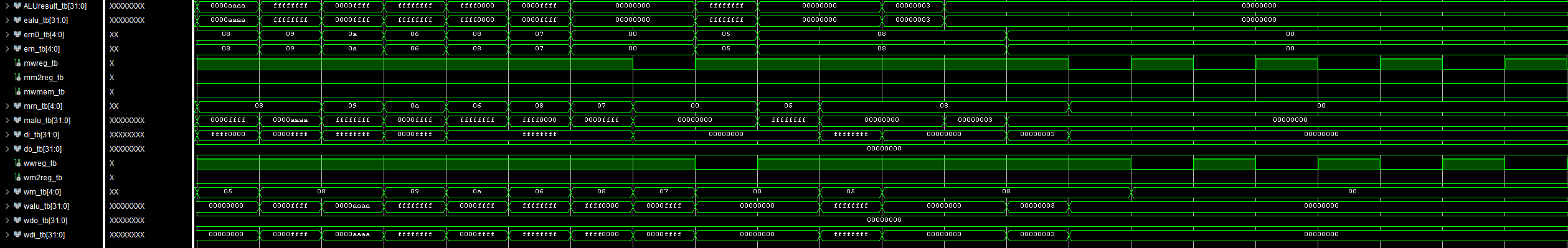
end

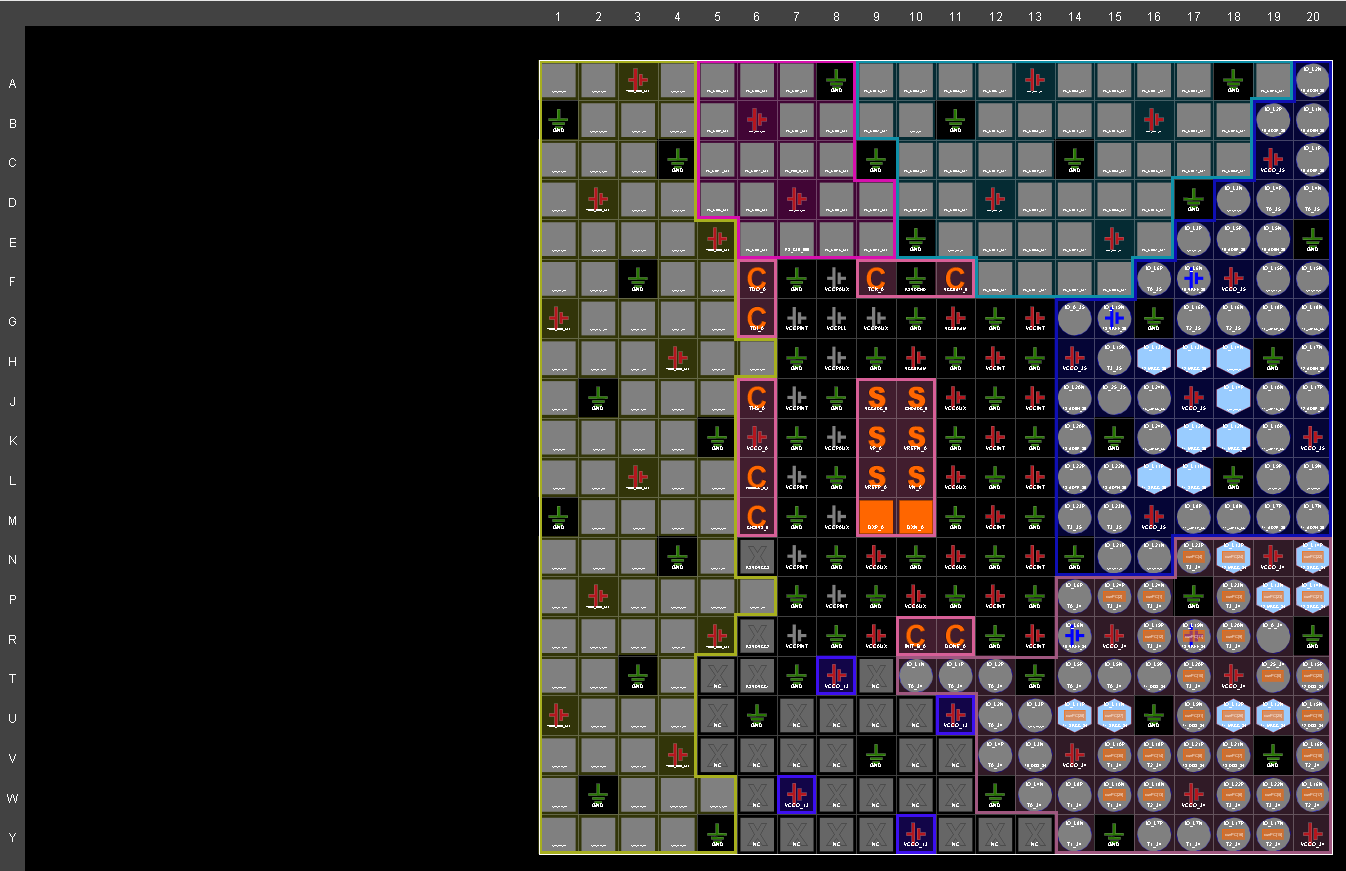
endmodule

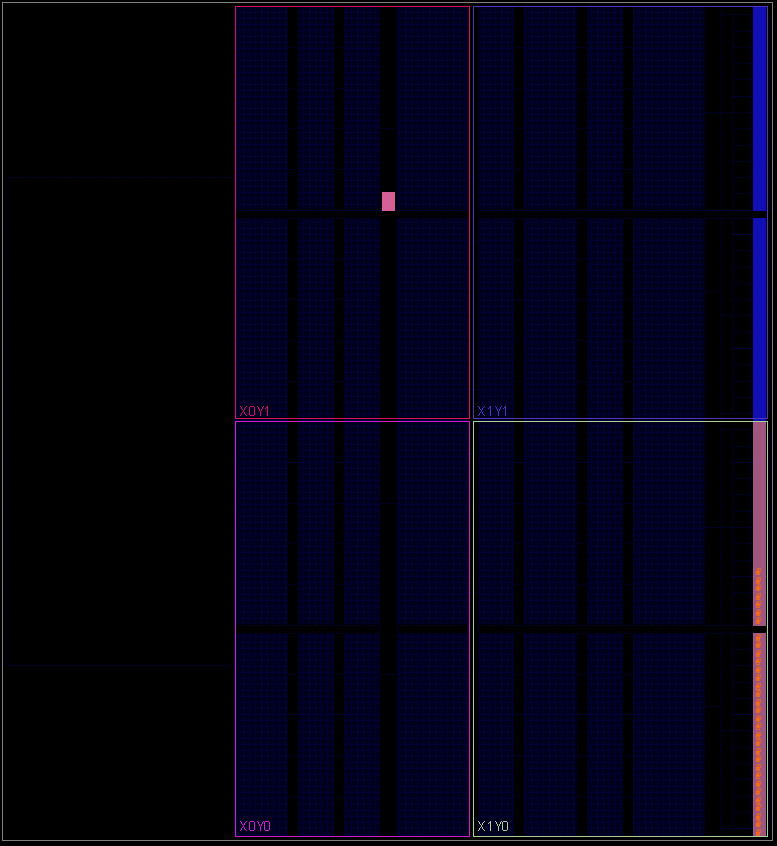
WAVEFORM

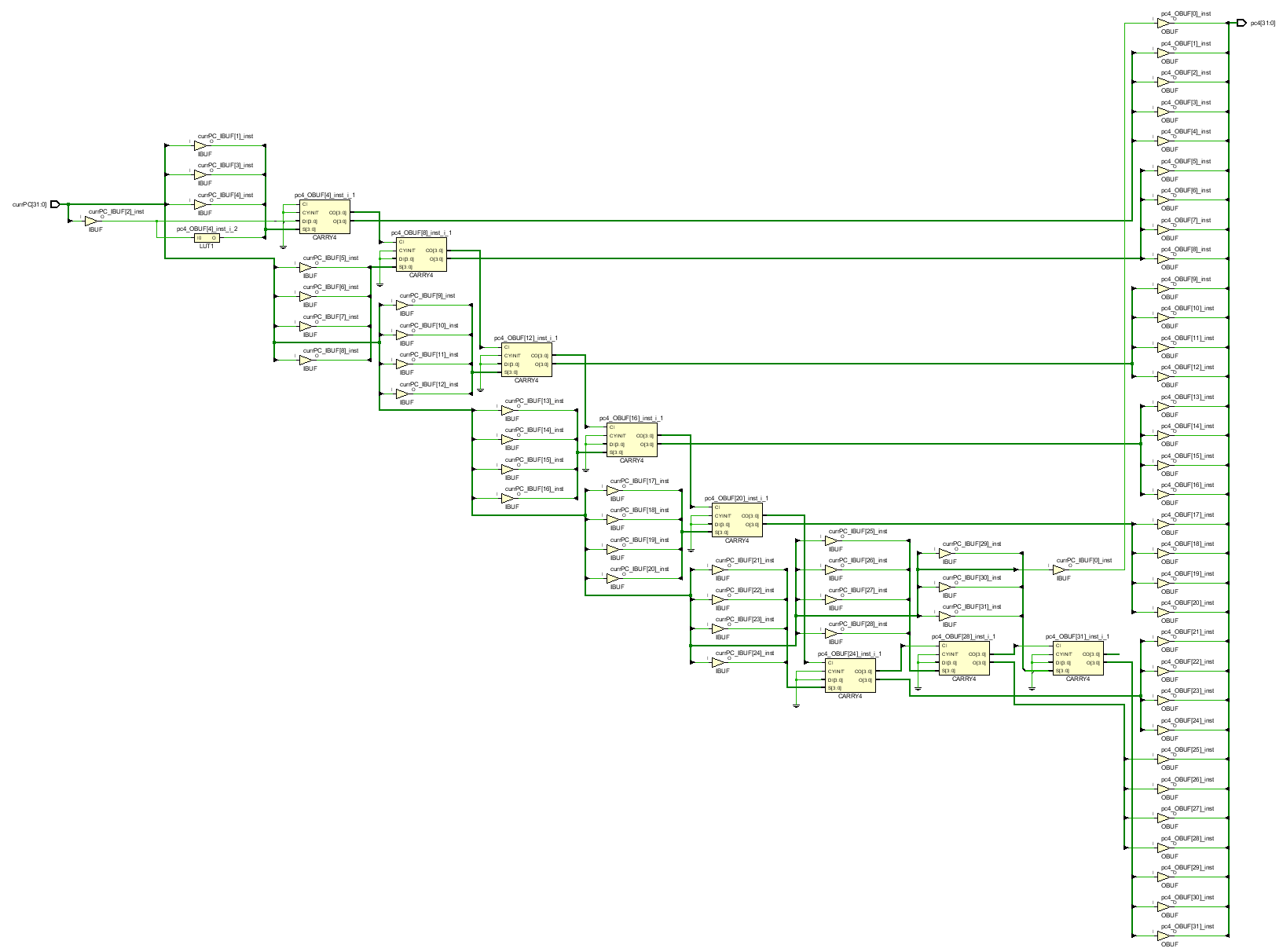
WAVEFORM (CONT)

WAVEFORM (CONT)

WAVEFORM (CONT)

I/O PLANNING

FLOOR PLANNING

SCHEMATIC