WILLIAM YOSHIDA & VISHAL KINNIMULKY

CMPEN 331  SECTION 002

FINAL PROJECT

The final project is the culmination of the previous three labs, labs 4-6. The final step of the project involves implementing pipeline stalls, which ensure data hazards are avoided given instructions that could induce them. For instance, if operands of an instruction are registers that are being written into due to a previous instruction, the processor might need to stall the instruction until the new value is written into the desired operand. In order to implement a stall, we first check if the new operand registers, rs and rt, of the new instruction are equivalent to the destination register of the previous two instructions. If they are equivalent, a stall occurs, which simply means we do not iterate the program counter and do not move on the next instruction. Simply speaking, we maintain the previous instruction and pc.

After the processor has been completed, we program the ZYBO board with our code. As inputs we have the clock, which replaces our implemented clock from previous labs, and the switch, which once flipped, initiates our program counter/our instructions. As output, we have the led, which lights up once we reach the end of our instructions.

------------------------------------------------------------------------------------------------------------------------------------------

Our processor works by having stages, which are our instruction fetch (IF), instruction decode (ID), execution (EXE), memory access (MEM), and write back (WB), as we see on Figure 1. Between each stage are registers that take in the outputs of the stage before it and delivers the values for the inputs for the stage after it. Essentially, what this is doing is storing the previous stage’s output values so it can feed it to the next stage. This is beneficial, since this is needed in order to have an implementation that uses several clock cycles per instruction. As one instruction is in the IF stage, another instruction is in the ID stage and another in the EXE stage. This allows five instructions to be running at the same time, which allows our program to run faster, taking advantage of parallelism. This is possible due to the values being stored in the registers so they can be used by the next stages.

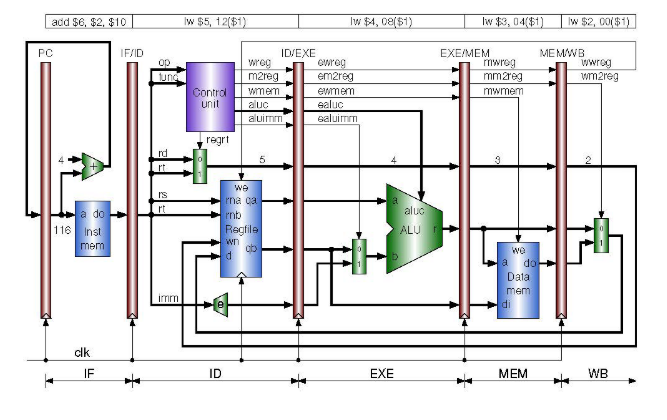


Figure 1: Pipeline stages

// design code

`timescale 1ns / 1ps

module top(clrn, clk, start, led);

input clrn;

input clk;

input start;

output led;

wire [31:0] pc;

wire [31:0] do;

wire [31:0] inst;

wire wreg;

wire m2reg;

wire wmem;

wire [3:0] aluc;

wire aluimm;

wire regrt;

wire [4:0] wr;

wire [31:0] qa;

wire [31:0] qb;

wire [31:0] extend;

wire ewreg;

wire em2reg;

wire ewmem;

wire [3:0] ealuc;

wire ealuimm;

wire [4:0] ewr;

wire [31:0] eqa;

wire [31:0] eqb;

wire [31:0] eextend;

wire [31:0] di;

wire mwmem;

wire [31:0] mdo;

wire [31:0] balu;

wire [31:0] ralu;

wire mwreg;

wire mm2reg;

wire [4:0] mwr;

wire [31:0] mr;

wire wwreg;

wire wm2reg;

wire [4:0] wwr;

wire [31:0] wbr;

wire [31:0] wdo;

wire [31:0] regd;

wire stall;

reg\_pc program\_counter (start, stall, clrn, clk, pc, led);

im instruction\_memory (pc, do);

reg\_if\_id if\_id (stall, clrn, clk, do, inst);

control cu (inst[31:26], inst[5:0], regrt, wreg, m2reg, wmem, aluc, aluimm);

mux1 destination\_reg (inst[15:11], inst[20:16], regrt, wr);

sign\_extend extension (inst[15:0], extend);

register\_file reg\_file (clrn, clk, wwreg, inst[25:21], inst[20:16], wwr, regd, qa, qb);

reg\_id\_exe id\_exe (clrn, clk, wreg, m2reg, wmem, aluc, aluimm, wr, qa, qb, extend, ewreg, em2reg, ewmem, ealuc, ealuimm, ewr, eqa, eqb, eextend);

mux2 alu\_mux (eqb, eextend, ealuimm, balu);

alu alu\_unit (eqa, balu, ealuc, ralu);

reg\_exe\_mem exe\_mem (clrn, clk, ewreg, em2reg, ewmem, ewr, ralu, eqb, mwreg, mm2reg, mwmem, mwr, mr, di);

data\_mem data\_memory (mr, di, mwmem, mdo);

reg\_mem\_wb mem\_wb (clrn, clk, mwreg, mm2reg, mwr, mr, mdo, wwreg, wm2reg, wwr, wbr, wdo);

mux3 wb\_mux (wbr, wdo, wm2reg, regd);

hdu hazard (inst[25:21], inst[20:16], ewr, mwr, stall);

endmodule

module reg\_pc (start, stall, clrn, clk, pc, led);

input start;

input stall;

input clrn;

input wire clk;

output reg [31:0] pc;

output reg led;

always @(start) begin

if (start == 1) begin

pc = 0;

led = 0;

end

end

always @(posedge clk) begin

if (stall == 1) begin

pc[31:0] = pc[31:0];

end

else begin

pc[31:0] = pc[31:0] + 32'd4;

end

if (pc > 00001124) begin

led = 0;

end

end

endmodule

module hdu (rs, rt, ewr, mwr, stall);

input [4:0] rs;

input [4:0] rt;

input [4:0] ewr;

input [4:0] mwr;

output reg stall;

initial begin

stall = 0;

end

always @(\*) begin

if (rs == ewr) begin

stall = 1;

end

else if (rt == ewr) begin

stall = 1;

end

else if (rs == mwr) begin

stall = 1;

end

else if (rt == mwr) begin

stall = 1;

end

else begin

stall = 0;

end

end

endmodule

module im (pc, do);

input [31:0] pc;

reg [31:0] im [500:0];

output reg [31:0] do;

initial begin

im[400] = 32'b00000000001000100001100000100000;

im[404] = 32'b00000001001000110010000000100010;

im[408] = 32'b00000000011010010010100000100101;

im[412] = 32'b00000000011010010011000000100110;

im[416] = 32'b00000000011010010011100000100100;

end

always @(pc) begin

do = im[pc];

end

endmodule

//

module reg\_if\_id (stall, clrn, clk, do, inst);

input stall, clrn, clk;

input [31:0] do;

output reg [31:0] inst;

always @(posedge clk) begin

if (stall == 1) begin

inst = inst;

end

else begin

inst = do;

end

end

endmodule

//

module control (op, func, regrt, wreg, m2reg, wmem, aluc, aluimm);

input [5:0] op;

input [5:0] func;

output reg regrt, wreg, m2reg, wmem, aluimm;

output reg [3:0] aluc;

always @(\*) begin

if (op == 6'b000000) begin

// to be implemented

if (func == 6'b100000) begin

regrt = 0;

wreg = 1;

m2reg = 0;

wmem = 0;

aluimm = 0;

aluc = 4'b0010;

end

else if (func == 6'b100010) begin

regrt = 0;

wreg = 1;

m2reg = 0;

wmem = 0;

aluimm = 0;

aluc = 4'b0110;

end

else if (func == 6'b100100) begin

regrt = 0;

wreg = 1;

m2reg = 0;

wmem = 0;

aluimm = 0;

aluc = 4'b0000;

end

else if (func == 6'b100101) begin

regrt = 0;

wreg = 1;

m2reg = 0;

wmem = 0;

aluimm = 0;

aluc = 4'b0001;

end

else if (func == 6'b100110) begin

regrt = 0;

wreg = 1;

m2reg = 0;

wmem = 0;

aluimm = 0;

aluc = 4'b0011;

end

else if (func == 6'b000000) begin

end

else if (func == 6'b000010) begin

end

else if (func == 6'b000011) begin

end

else if (func == 6'b001000) begin

end

end

// other instructions to be implemented

// instruction of interest is lw

else if (op == 6'b100011) begin

regrt = 1;

wreg = 1;

m2reg = 1;

wmem = 0;

aluimm = 1;

aluc = 4'b0010;

end

end

endmodule

//

module mux1 (rd, rt, regrt, wr);

input [4:0] rd;

input [4:0] rt;

input regrt;

output reg [4:0] wr;

always@(\*) begin

if (regrt == 1) begin

wr <= rt;

end

else begin

wr <= rd;

end

end

endmodule

//

module sign\_extend (imm, extend);

input [15:0] imm;

output reg [31:0] extend;

always @(\*) begin

extend[31:0] = { {16{imm[15]}}, imm[15:0] };

end

endmodule

//

module register\_file (clrn, clk, wwreg, rs, rt, wwr, regd, qa, qb);

input clrn, clk;

input wwreg;

input [4:0] rs;

input [4:0] rt;

input [4:0] wwr;

input [31:0] regd;

reg [31:0] rf [31:0];

output reg [31:0] qa;

output reg [31:0] qb;

initial begin

rf[0] = 'h00000000;

rf[1] = 'hA00000AA;

rf[2] = 'h10000011;

rf[3] = 'h20000022;

rf[4] = 'h30000033;

rf[5] = 'h40000044;

rf[6] = 'h50000055;

rf[7] = 'h60000066;

rf[8] = 'h70000077;

rf[9] = 'h80000088;

rf[10] = 'h90000099;

end

always @(\*) begin

if (wwreg == 1) begin

rf[wwr] = regd;

end

qa = rf[rs];

qb = rf[rt];

end

endmodule

//

module reg\_id\_exe (clrn, clk, wreg, m2reg, wmem, aluc, aluimm, wr, qa, qb, extend,

ewreg, em2reg, ewmem, ealuc, ealuimm, ewr, eqa, eqb, eextend);

input clrn, clk;

input wreg, m2reg, wmem;

input [3:0] aluc;

input aluimm;

input [4:0] wr;

input [31:0] qa;

input [31:0] qb;

input [31:0] extend;

output reg ewreg, em2reg, ewmem;

output reg [3:0] ealuc;

output reg ealuimm;

output reg [4:0] ewr;

output reg [31:0] eqa;

output reg [31:0] eqb;

output reg [31:0] eextend;

always @(posedge clk) begin

ewreg = wreg;

em2reg = m2reg;

ewmem = wmem;

ealuc = aluc;

ealuimm = aluimm;

ewr = wr;

eqa = qa;

eqb = qb;

eextend = extend;

end

endmodule

//

module mux2 (eqb, eextend, ealuimm, balu);

input [31:0] eqb;

input [31:0] eextend;

input ealuimm;

output reg [31:0] balu;

always@(\*) begin

if (ealuimm == 1) begin

balu <= eextend;

end

else if (ealuimm == 0) begin

balu <= eqb;

end

end

endmodule

module alu (eqa, balu, ealuc, ralu);

input [31:0] eqa;

input [31:0] balu;

input [3:0] ealuc;

output reg [31:0] ralu;

always@(\*) begin

if (ealuc == 0) begin

ralu = eqa & balu;

end

else if (ealuc == 1) begin

ralu = eqa | balu;

end

else if (ealuc == 2) begin

//ralu = 2;

ralu = eqa + balu;

end

else if (ealuc == 3) begin

ralu = eqa ^ balu;

end

else if (ealuc == 6) begin

ralu = eqa - balu;

end

else if (ealuc == 4'b0111) begin

end

else if (ealuc == 4'b1100) begin

end

end

endmodule

//

module reg\_exe\_mem (clrn, clk, ewreg, em2reg, ewmem, ewr, ralu, eqb,

mwreg, mm2reg, mwmem, mwr, mr, di);

input clrn, clk;

input ewreg, em2reg, ewmem;

input [4:0] ewr;

input [31:0] ralu;

input [31:0] eqb;

output reg mwreg, mm2reg, mwmem;

output reg [4:0] mwr;

output reg [31:0] mr;

output reg [31:0] di;

always @(posedge clk) begin

mwreg = ewreg;

mm2reg = em2reg;

mwmem = ewmem;

mwr = ewr;

mr = ralu;

di = eqb;

end

endmodule

module data\_mem (mr, di, mwmem, mdo);

input [31:0] mr;

input [31:0] di;

input mwmem;

reg [31:0] dm [128:0];

output reg [31:0] mdo;

initial begin

dm[0] = 'h00000000;

dm[4] = 'hA00000AA;

dm[8] = 'h10000011;

dm[12] = 'h20000022;

dm[16] = 'h30000033;

dm[20] = 'h40000044;

dm[24] = 'h50000055;

dm[28] = 'h60000066;

dm[32] = 'h70000077;

dm[36] = 'h80000088;

dm[40] = 'h90000099;

end

always @(\*) begin

if (mwmem == 0) begin

mdo = dm[mr];

end

mdo = dm[mr];

if (mwmem == 1) begin

dm[di] = dm[mr];

end

end

endmodule

//

module reg\_mem\_wb (clrn, clk, mwreg, mm2reg, mwr, mr, mdo,

wwreg, wm2reg, wwr, wbr, wdo);

input clrn, clk;

input mwreg, mm2reg;

input [4:0] mwr;

input [31:0] mr;

input [31:0] mdo;

output reg wwreg, wm2reg;

output reg [4:0] wwr;

output reg [31:0] wbr;

output reg [31:0] wdo;

always @(posedge clk) begin

wwreg = mwreg;

wm2reg = mm2reg;

wwr = mwr;

wbr = mr;

wdo = mdo;

end

endmodule

module mux3 (wbr, wdo, wm2reg, regd);

input [31:0] wbr;

input [31:0] wdo;

input wm2reg;

output reg [31:0] regd;

always@(\*) begin

if (wm2reg == 1) begin

regd <= wdo;

end

else if (wm2reg == 0) begin

regd <= wbr;

end

end

endmodule

// testbench

`timescale 1ns / 1ps

module testbench();

wire clrn, clk;

top c

(

.clrn (clrn ),

.clk (clk )

);

tester t

(

.clrn (clrn ),

.clk (clk )

);

initial

begin

$monitor ($time,"clrn=%b, clk=%b", clrn, clk);

end

endmodule

// tester

`timescale 1ns / 1ps

module tester(clrn, clk);

output reg clrn;

output reg clk;

initial

begin

clk = 0;

clrn = 0;

end

always begin

#5 clk = ~clk;

end

endmodule

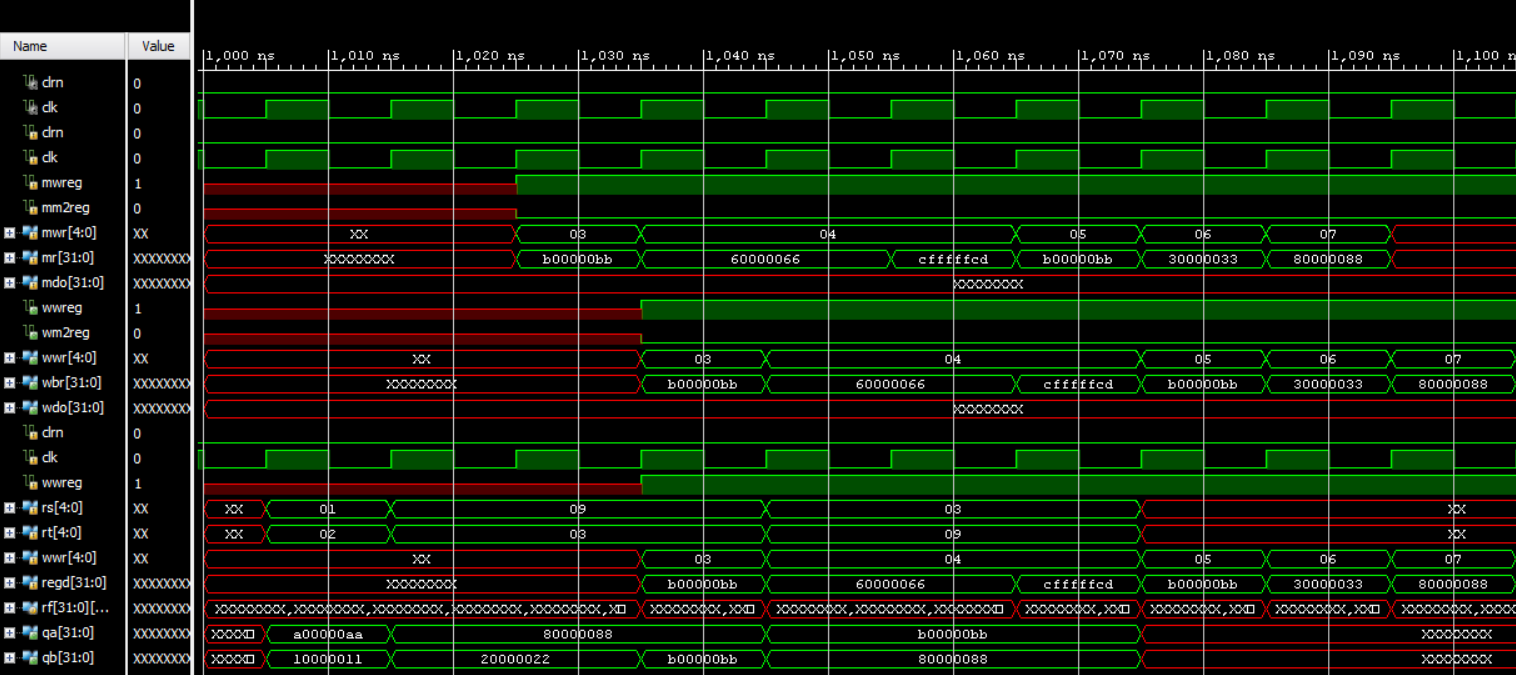


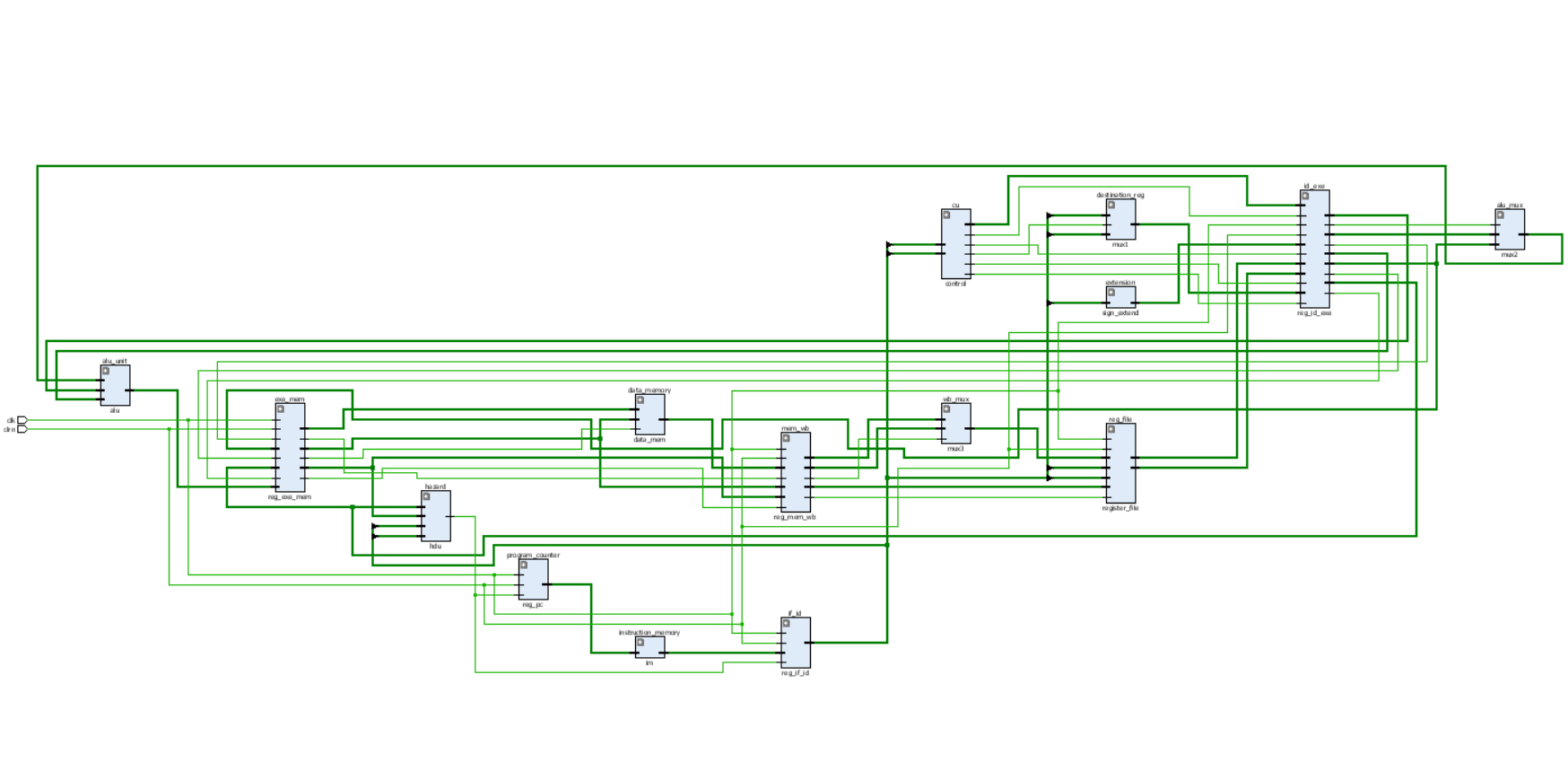
Figure 2: Waveforms

Figure 3: Schematics

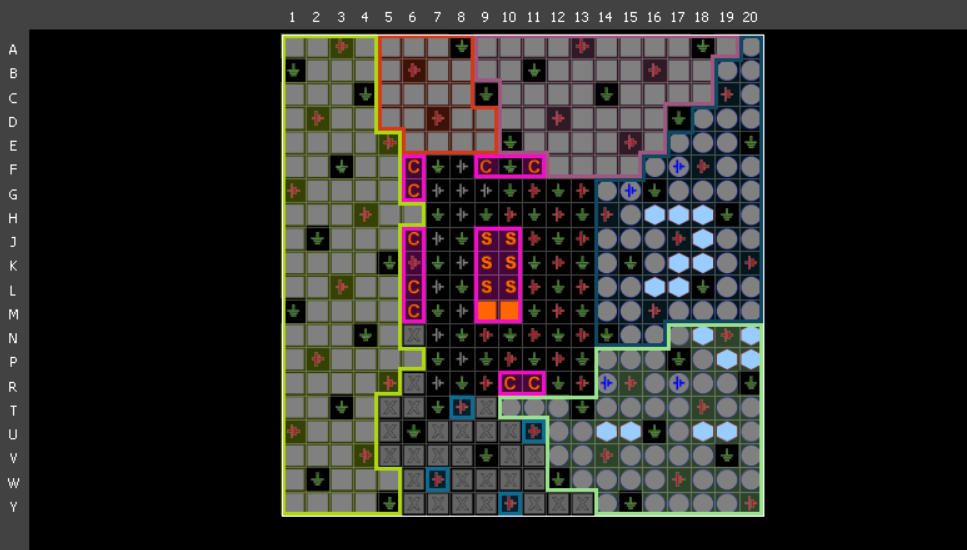


Figure 4: I/O Planning

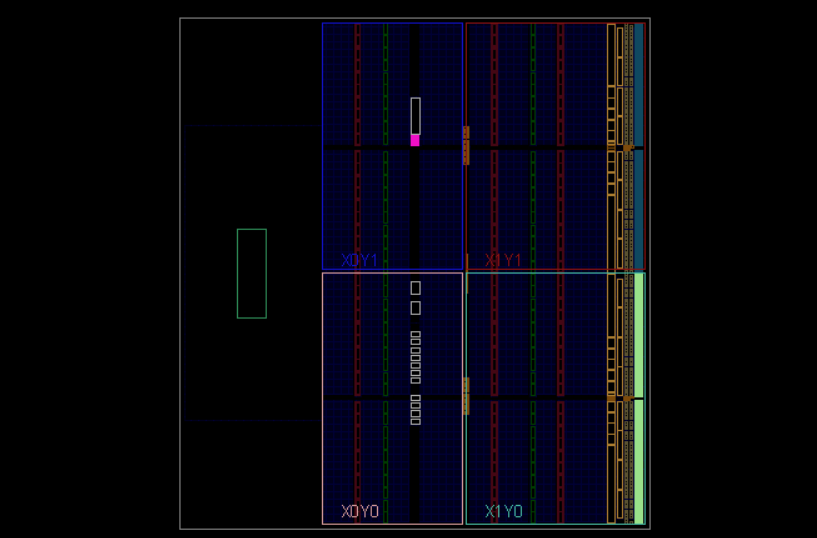


Figure 5: Floorplanning

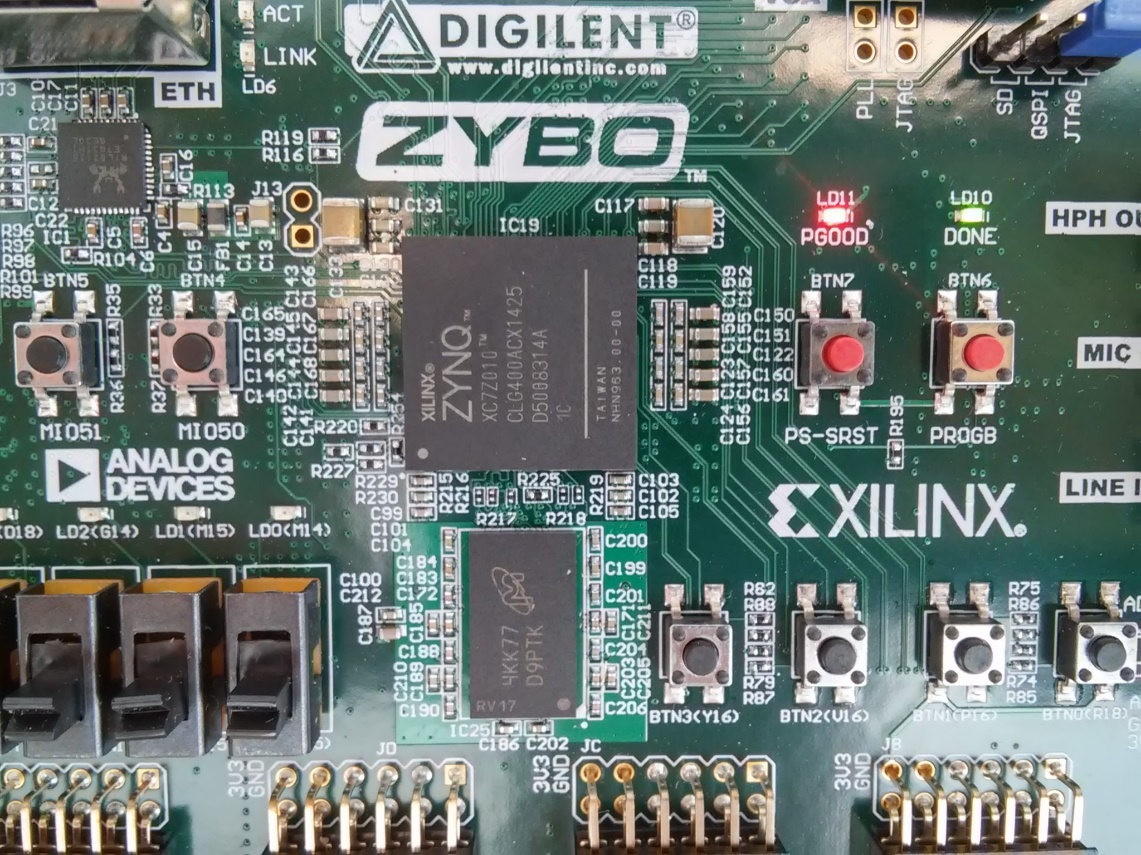


Figure 6: Program downloaded

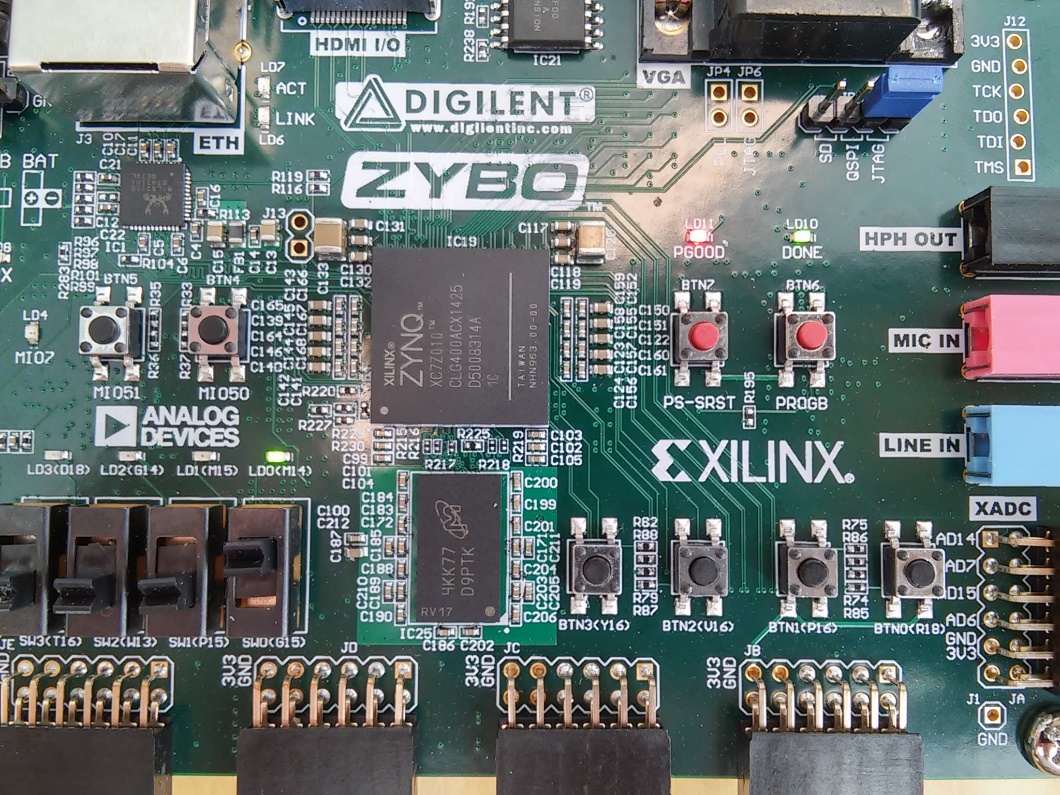


Figure 7: Instructions completed