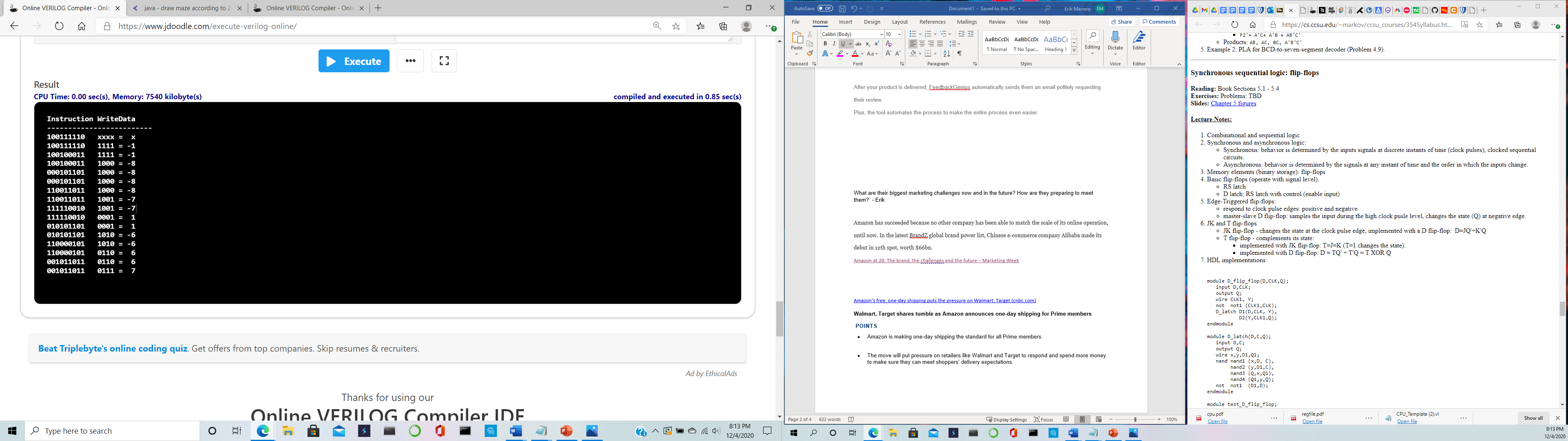
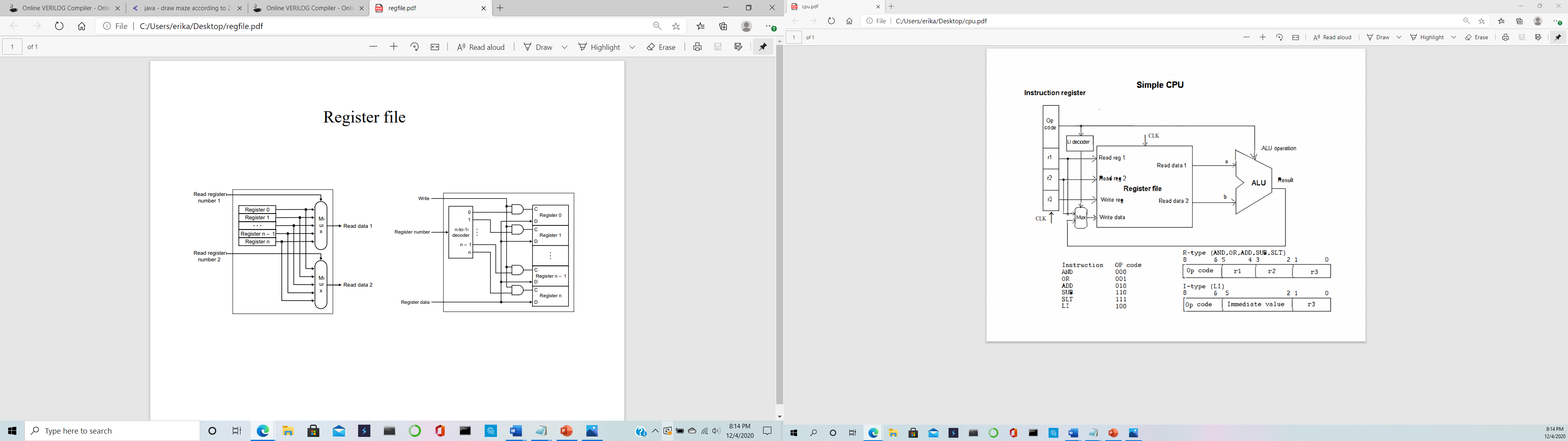
Erik Marrero





module halfAdder (S,C,x,y);

input x,y;

output S,C;

and (C,x,y);

xor (S,x,y);

endmodule

module fullAdder(x,y,z,C,S);

input x,y,z;

output S,C;

wire S1,D1,D2;

halfAdder HA1 (S1,D1,x,y),

HA2 (S,D2,S1,z);

or g1(C,D2,D1);

endmodule

module one\_bit\_ALU(a, b, less, binvert, carry\_in, op, carry\_out, result);

input a, b, less, binvert, carry\_in;

input [1:0] op;

output carry\_out, result;

wire w0, w1, w2, w3;

inverter i1(binvert, b, w0);

and g0(w1, a, w0);

or g1(w2, a, w0);

fullAdder fulladder(a, w0, carry\_in, carry\_out, w3);

mux multi(w1, w2, w3, less, op, result);

endmodule

module one\_bit\_ALU\_set(a, b, less, binvert, carry\_in, op, overflow, set, result);

input a, b, less, binvert, carry\_in;

input [1:0] op;

output set, overflow, result;

wire w0, w1, w2, w3;

inverter i1(binvert, b, w0);

and g0(w1, a, w0);

or g1(w2, a, w0);

fullAdder fulladder(a, w0, carry\_in, w3, set);

mux multi(w1, w2, set, less, op, result);

xor g2(overflow, carry\_in, w3);

//xor g2(w4, carry\_in, w3);

endmodule

module mux(i0, i1, i2, i3, select, out);

input[1:0] select;

input i0, i1, i2, i3;

output out;

wire q, q1, q2, q3, q4, NOTselect0, NOTselect1;

wire[1:0] select;

wire[3:0] d;

not n1(NOTselect0, select[0]);

not n2(NOTselect1, select[1]);

and a1(q1, NOTselect0, NOTselect1, i0);

and a2(q2, select[0], NOTselect1, i1);

and a3(q3, NOTselect0, select[1], i2);

and a4(q4, select[0], select[1], i3);

or o1(out, q1, q2, q3, q4);

endmodule

module inverter(binvert, b0, out);

output out;

input b0, binvert;

wire b1, w0, w1, w2, w3;

not (w0, binvert);

not (b1,b0);

and (w2, binvert, b1);

and (w1, w0, b0);

or (out, w1, w2);

endmodule

module four\_bit\_ALU(a, b, op, zero, overflow, result);

input [3:0] a;

input [3:0] b;

input [2:0] op;

output overflow;

output zero;

output [3:0] result;

wire w1, w2, w3, w4, w5, set;

one\_bit\_ALU alu0 (a[0], b[0], set, op[2], op[2], op[1:0], w1, result[0]),

alu1(a[1], b[1], 1'b0, op[2], w1, op[1:0], w2, result[1]),

alu2 (a[2], b[2], 1'b0, op[2], w2, op[1:0], w3, result[2]);

one\_bit\_ALU\_set alu3(a[3], b[3], 1'b0, op[2], w3, op[1:0], overflow, set, result[3]);

or g1(w5, result[0], result[1], result[2], result[3]);

not g2(zero, w5);

//and g3(overflow,w4,op[1]);

endmodule

// LI decoder. 0 if 100, 1 otherwise

module LI\_decoder(opCode[2:0],out);

input [2:0] opCode;

output out;

wire w0,w1;

// out = a'+b+c

not (w0,opCode[2]); // negate first bit of opcode

or (out,w0,opCode[1],opCode[0]);

endmodule

//D flip-flop

module D\_flip\_flop(D,CLK,Q);

input D,CLK;

output Q;

wire CLK1, Y;

not not1 (CLK1,CLK);

D\_latch D1(D,CLK, Y),

D2(Y,CLK1,Q);

endmodule

//part for D flip-Flop

module D\_latch(D,C,Q);

input D,C;

output Q;

wire x,y,D1,Q1;

nand nand1 (x,D, C),

nand2 (y,D1,C),

nand3 (Q,x,Q1),

nand4 (Q1,y,Q);

not not1 (D1,D);

endmodule

//Multiplexer, quadruple 2-1

module multiplexer(result,IR,Li,out);

input [3:0] result; //from ALU

input [3:0] IR; // 4bits from IR

input Li; // from LI decoder, determines what bits get loaded

output [3:0] out; //4 bit output to "write data"

wire w0,w1,w2,w3,w4,w5,w6,w7,w8;

not n1(w0,Li);

and a1(w1,result[3],Li),

a2(w2,IR[3],w0),

a3(w3,result[2],Li),

a4(w4,IR[2],w0),

a5(w5,result[1],Li),

a6(w6,IR[1],w0),

a7(w7,result[0],Li),

a8(w8,IR[0],w0);

or o1(out[3],w1,w2), //begining of 4bit output

o2(out[2],w3,w4),

o3(out[1],w5,w6),

o4(out[0],w7,w8);

endmodule

// Instruction register. Must be completed.

module instr\_reg (Instruction,IR,CLK);

input [8:0] Instruction;

input CLK;

output [8:0] IR;

// add D flip-flops here

D\_flip\_flop f0(Instruction[0],CLK,IR[0]),

f1(Instruction[1],CLK,IR[1]),

f2(Instruction[2],CLK,IR[2]),

f3(Instruction[3],CLK,IR[3]),

f4(Instruction[4],CLK,IR[4]),

f5(Instruction[5],CLK,IR[5]),

f6(Instruction[6],CLK,IR[6]),

f7(Instruction[7],CLK,IR[7]),

f8(Instruction[8],CLK,IR[8]);

endmodule

/\* 4-bit register file implemented in behavioral modeling (don't change it).

Dont't use behavioral modeling for the other CPU modules. They

must be implemented at gate level (no reg data, assign, and always).

\*/

module regfile (ReadReg1,ReadReg2,WriteReg,WriteData,ReadData1,ReadData2,CLK);

input [1:0] ReadReg1,ReadReg2,WriteReg;

input [3:0] WriteData;

input CLK;

output [3:0] ReadData1,ReadData2;

reg [3:0] Regs[0:3];

assign ReadData1 = Regs[ReadReg1];

assign ReadData2 = Regs[ReadReg2];

initial Regs[0] = 0;

always @(negedge CLK)

Regs[WriteReg] <= WriteData;

endmodule

// CPU

module cpu (Instruction, WriteData, CLK);

input [8:0] Instruction; // 9bits for mips instructions

input CLK; // clock

output [3:0] WriteData; // 4bit output of cpu

output [3:0] aluToMux; //4bit connection from alu result to mux

wire [3:0] A,B; // connects register file to alu

wire [8:0] IR; // instruction register

// Declare more wires as needed

wire decToMux; //connects li decoder to the mux

/\* The instruction register is instantiated below. You have to provide

the module definition (the module header and ports are already provided

below). The OP code and other fiels of this register that have to be

passed to other modules must be represented by their respective

indices (see the register file instance below).

\*/

instr\_reg instr(Instruction,IR,CLK);

// Define a module for the quadruple 2x1 mux and instatiate it here.

multiplexer multplxr(aluToMux,IR[5:2],decToMux,WriteData);

// Define a module for the LI decoder and instantiate it here.

LI\_decoder decoder(IR[8:6],decToMux);

// register file (the module definition is incuded in this file)

regfile regs(IR[5:4],IR[3:2],IR[1:0],WriteData,A,B,CLK);

// Define a module for the ALU and instantiate it here.

four\_bit\_ALU alu(A, B, IR[8:6], zero, overflow, aluToMux);

endmodule

// Test module. Add more instructions as provided in the test program.

module test\_cpu;

reg [8:0] Instruction;

reg CLK;

wire signed [3:0] WriteData;

cpu cpu1 (Instruction, WriteData, CLK);

initial

begin

// LI $2, 15 # load decimal 15 in $2

Instruction = 9'b100111110;

#1 CLK=1;

#1 CLK=0; // negative edge - execute instruction

/\* Add machine code for the test program instructions here.

Use the format shown above. Pay attention to the register

order - the destination register is first in the assembly code

and last (LSB) in the machine code.

After each instruction a negative edge must be generated.

\*/

//LI $3, 8 # load decimal 8 (unsigned binary) into $3

#1 Instruction = 9'b100100011;

#1 CLK=1;

#1 CLK=0; // negative edge - execute instruction

//AND $1, $2, $3 # $1 = $2 AND $3 (two's complement -8)

#1 Instruction = 9'b000101101;

#1 CLK=1;

#1 CLK=0; // negative edge - execute instruction

//SUB $3, $1, $2 # $3 = $1 - $2 = -8 - (-1) = -7

#1 Instruction = 9'b110011011;

#1 CLK=1;

#1 CLK=0; // negative edge - execute instruction

//SLT $2, $3, $0 # $2 = 1 ($3 < 0)

#1 Instruction = 9'b111110010;

#1 CLK=1;

#1 CLK=0; // negative edge - execute instruction

//ADD $1, $2, $3 # $1 = $2 + $3 = 1 + (-7) = -6

#1 Instruction = 9'b010101101;

#1 CLK=1;

#1 CLK=0; // negative edge - execute instruction

//SUB $1, $0, $1 # $1 = $0 - $1 = 0 - (-6) = 6

#1 Instruction = 9'b110000101;

#1 CLK=1;

#1 CLK=0; // negative edge - execute instruction

//OR $3, $1, $2 # $3 = $1 OR $2 = 7

#1 Instruction = 9'b001011011;

#1 CLK=1;

#1 CLK=0; // negative edge - execute instruction

end

initial

begin

$display("\Instruction WriteData\n-------------------------");

$monitor("%b %b = %d",Instruction, WriteData, WriteData);

end

endmodule