RISC Processor Design in Verilog HDL

# by

**Brandon Rubio**

**Isabela Bandrich**

**Rainier Young**

**Nikeem Dunkelly-Allen Group # 8**

Final Project Report

ECE 414: Computer Organization and Design

# December 2, 2022

©

Project directed by Dr. Onur Tigli

Asst. Professor of Electrical and Computer Engineering

## TABLE OF CONTENTS

Page

## 1. Abstract 1

| **2. Introduction** |  | 1 |
| --- | --- | --- |

**3. Design and Results** 1

3.1 Inputs and Outputs 3

3.2 Block Diagrams 4

3.2.1 Top Level Internals 4

3.2.2 CPU 4

3.2.3 CPU Internals 5

3.2.4 Datapath 6

3.2.5 Datapath Internals 6

3.2.6 Controller 7

3.2.7 Controller Internals 8

3.3 Simulation Results 9

3.3.1 Top Level 9

3.3.2 CPU 9

3.3.3 ALU 9

3.3.4 Register File 9

3.4 Power Dissipation 10

**4. Conclusion** 10

**5. References** 10

**6. Appendix** 11

## ABSTRACT

In this project, we designed a general purpose 16-bit RISC processor using Verilog-HDL. The processor was built from the ground up starting with basic combinational and sequential circuits. The architecture is capable of 16-bit instruction words, 16 internal general-purpose registers, 6 external address lines to a ROM and 6 external address lines to an external memory (RAM). Each module is designed, synthesized and tested at each implementation level. The modules were then connected together and integrated into the top-level simulation using the appropriate port mappings.

## INTRODUCTION

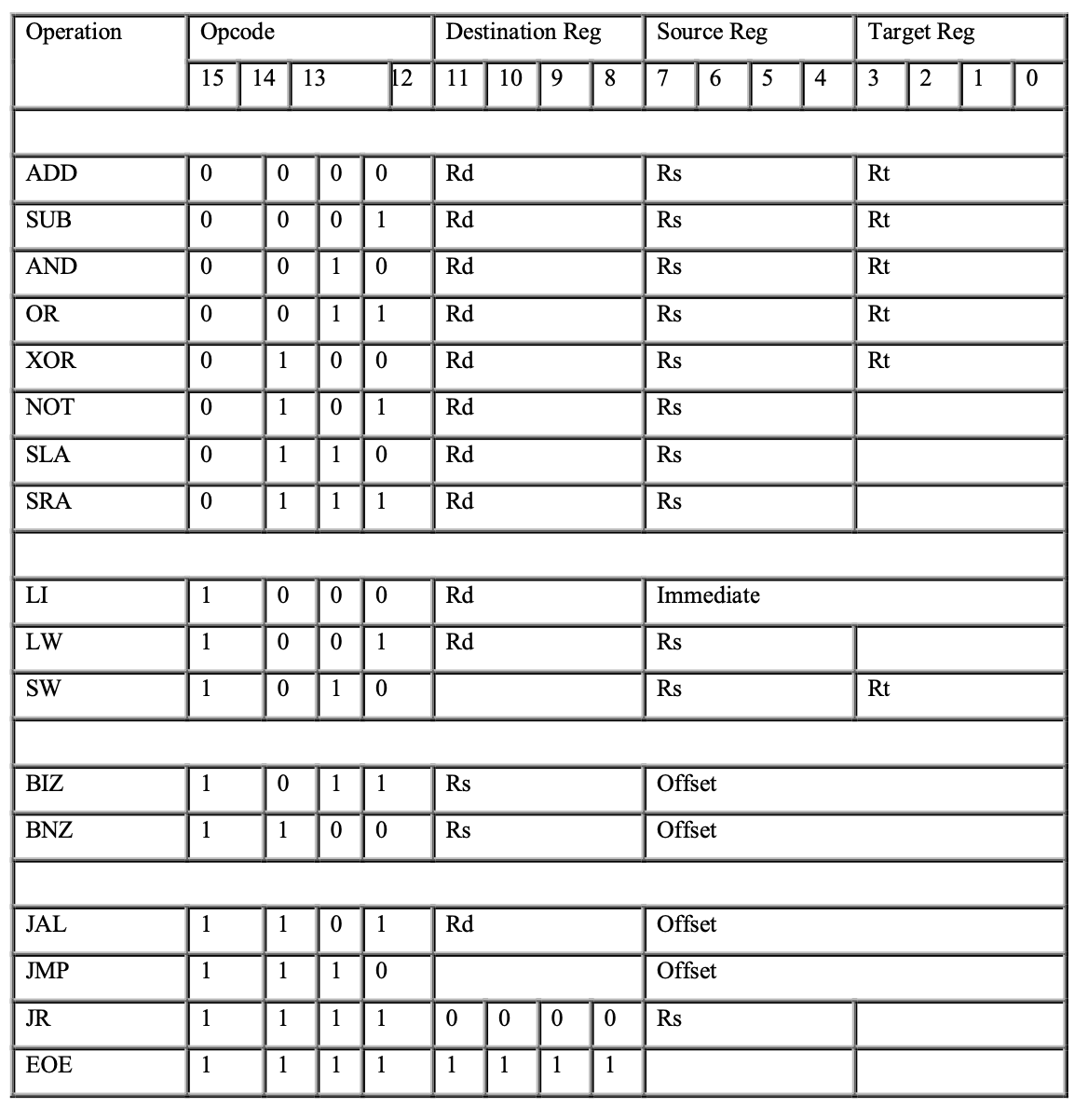
Reduced Instruction Set, or RISC, CPUs currently in use command the lion's share of the market. The majority of consumer products on the market use embedded systems, where it is the most frequently used. RISC CPUs are simple in design, have a low power requirement, and do not require many transistors like other processors which makes them compact. Because of the fundamental principles upon which it operates, they are sometimes referred to as load-store processors. The purpose of a RISC CPU is to boost speed while reducing system complexity. Any complicated operation can be divided into smaller chunks that, in most situations, can be calculated simultaneously. The word “Reduced Instruction Set” may be incorrectly interpreted to refer to “reduced number of instructions”. Although this is false, the phrase actually refers to a reduction in the number of cycles required to complete each instruction's task.

## DESIGN and RESULTS

(The code for each module described in the Design and Results section can be found in the Appendix section).

Our implementation of the 16-bit RISC architecture uses a hierarchical design that consists of two main parts: the control unit and the datapath unit. The purpose of the control unit is to interpret each instruction and deliver appropriate data to other modules in the system. In order to accomplish this task the control unit is separated into three modules: the program counter, instruction register, and control logic.

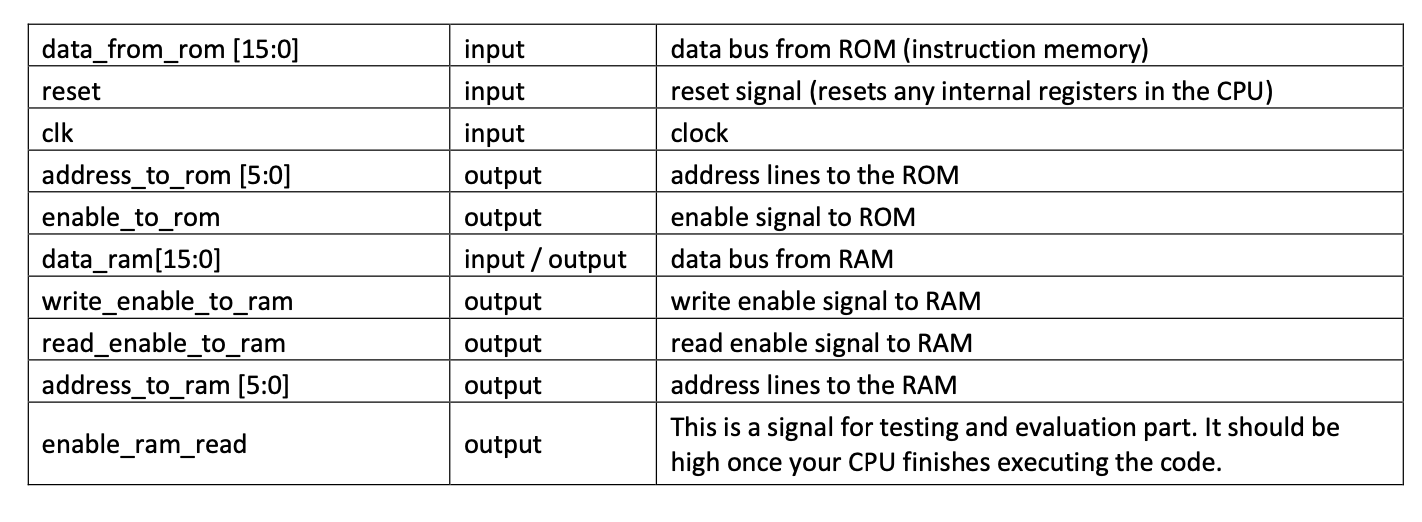
The program counter keeps track of the address location of each instruction being executed at the current time and holds the address of the next instruction to be executed. The instruction register stores a given instruction and determines whether or not to load that data. And finally, the control logic inputs the given instruction through the instruction decoder module and interprets that data according to the following table.



*We built our computer to do operations based on these three categories—arithmetic, logic, and branch operations—on the basis of the information in this table.*

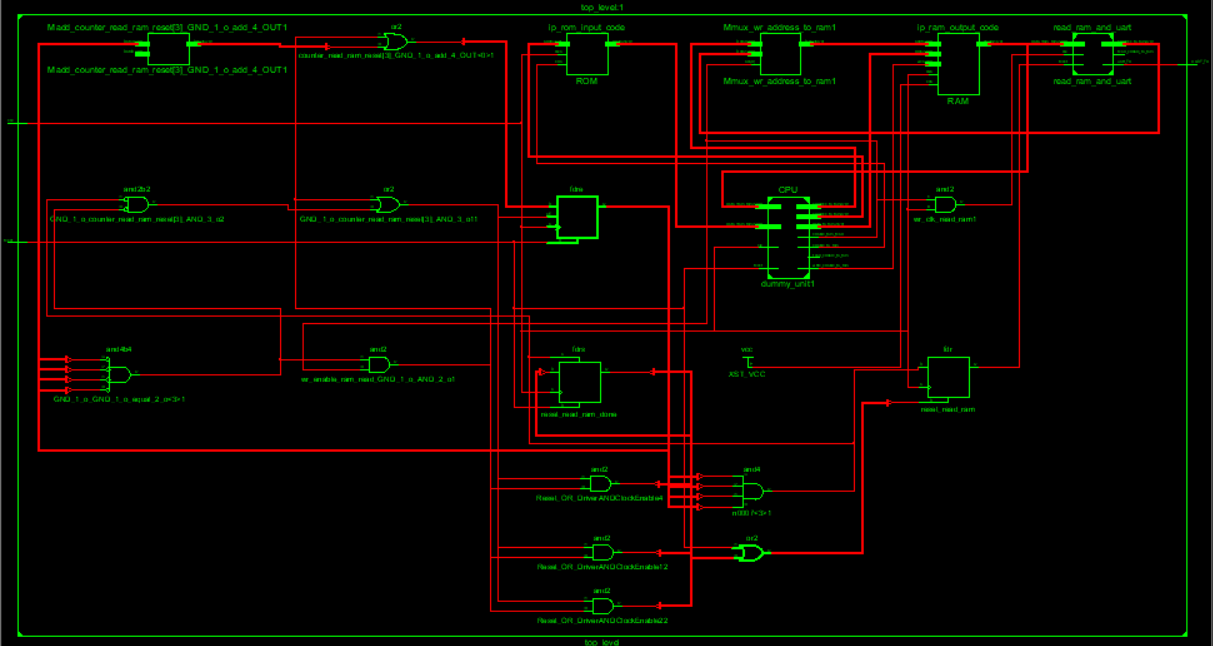
The purpose of the datapath is to carry out all data processing in the CPU. The datapath needs to allow data to flow between the buses of the ALU, CPU, and internal registers. When instructions containing select signals for “Address out”, “Data out”, Constant out, etc… are sent from the controller to the datapath, the datapath inputs data to MUX D or takes the desired output from A bus or MUX D accordingly. The datapath section of our CPU consists of the register file and the functional unit. The register file provides a storage area for 16 registers to interact with the CPU. Meanwhile, the Functional Unit consists of the mathematical operations (with the help of the ALU module) that manipulate the data within the registers according to the instructions and control logic provided by the control unit of the CPU. In our design wires are instantiated that connect all needed ports between the controller and the datapath within the CPU. Instructions are sent to the datapath and have already been decoded by the controller. The datapath places all needed values into the register file and functional unit where operations are completed. Finally the functional unit returns values to the datapath where they are ready for output to registers.

Inputs and Outputs:

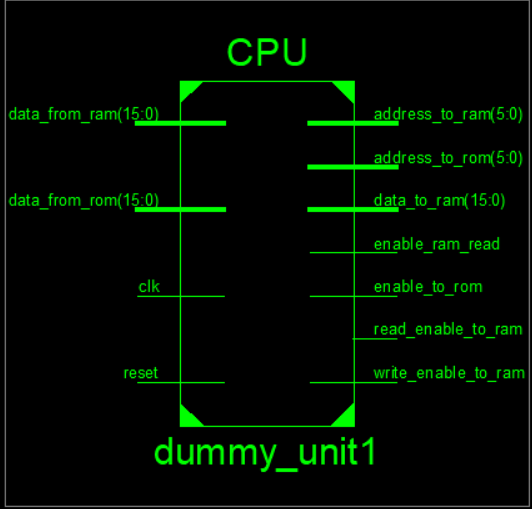


**Block Diagrams:**

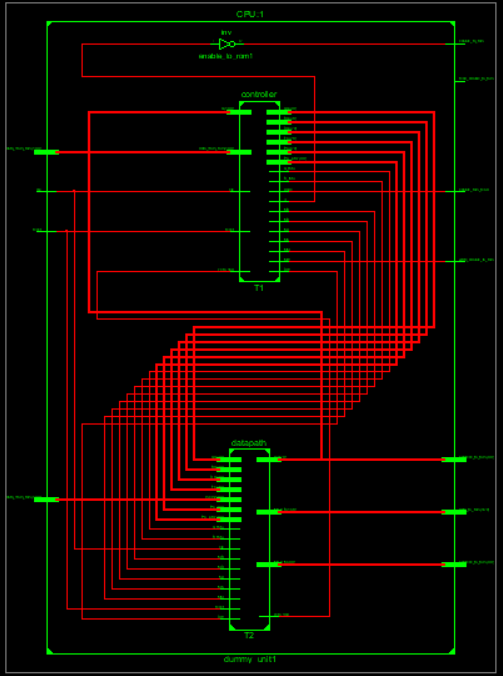
Top Level Internals:



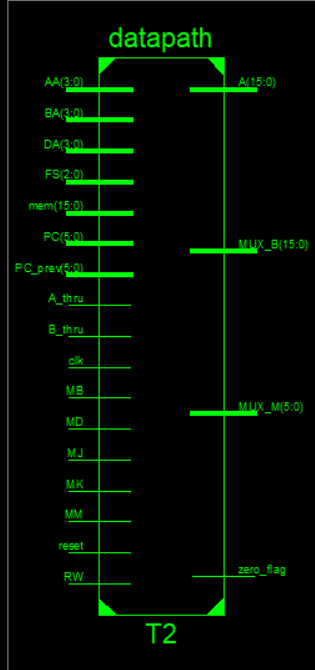
CPU:



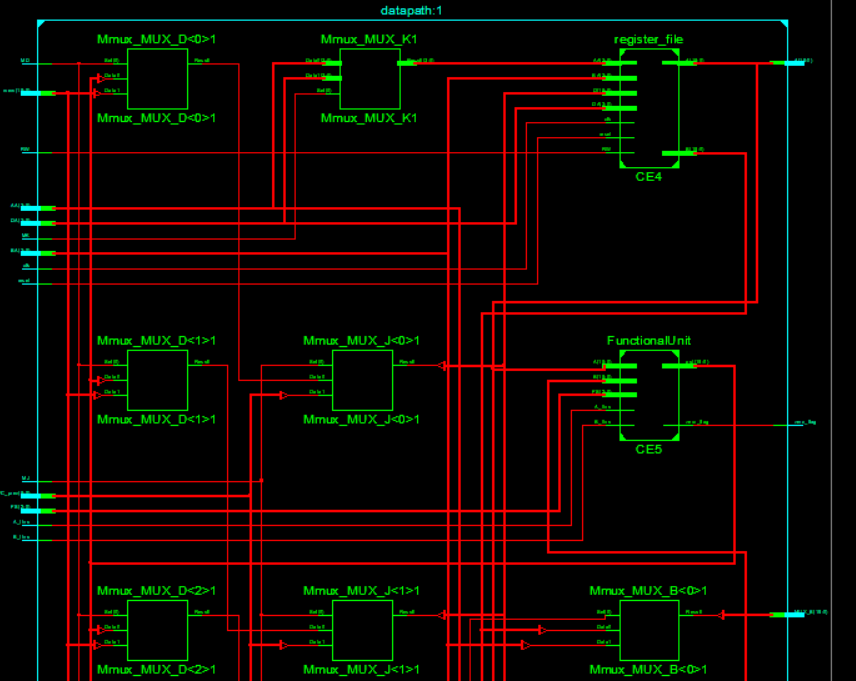
CPU Internals:



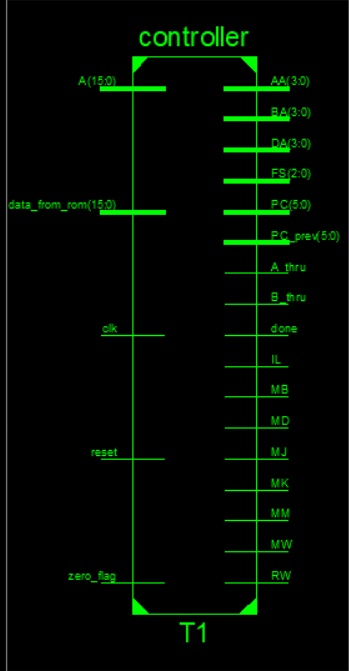
Datapath:



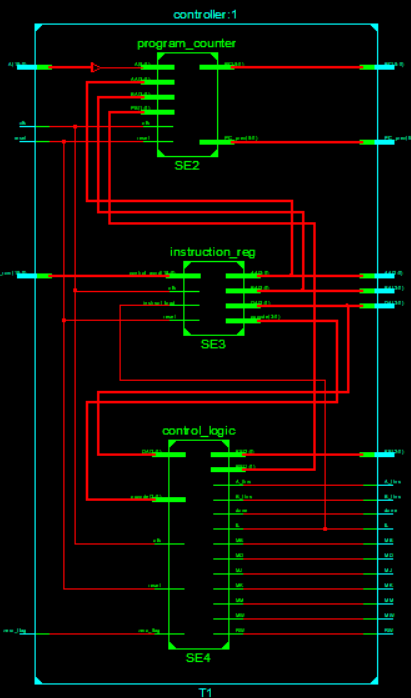
Datapath Internals:



Controller:



Controller Internals:

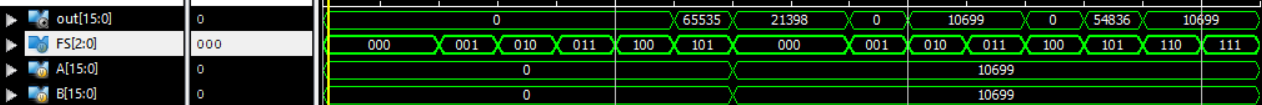


**Simulation Results:**

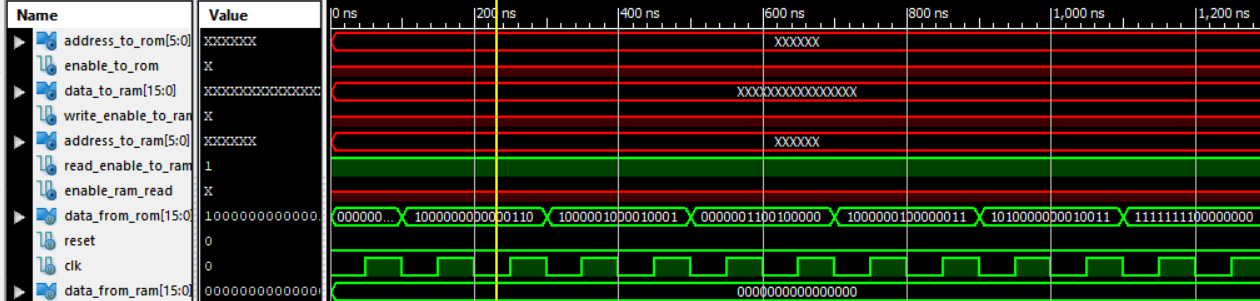
Top Level:



ALU:



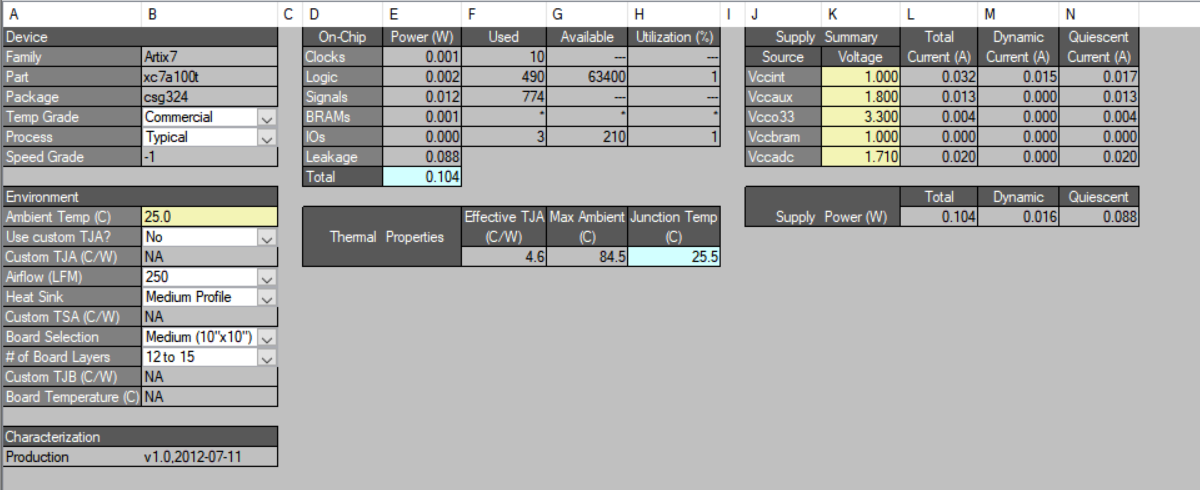
CPU:



Register File:



**Power Dissipation:**

****

## CONCLUSION

For our final project, we created a 16-bit RISC Processor CPU in Verilog-HDL. It featured 16-bit instruction words, 16 internal general-purpose registers, and six external address lines to a ROM and six external address lines to an external memory (RAM). This 16-bit processor's development has taught us a lot about computer architecture as a whole. In addition to learning about each component and how to use them in Verilog, we also gained knowledge of how each one interacts with the others and how the system should function as a whole. Given the size of the project, we put in a lot of coding and debugging time. The magnitude and complexity of this project allowed us to gain valuable insight into the difficulty of creating processors and computers even with FPGAs and synthesis tools. Being the final assignment for this class, we are confident in declaring; that this was a fantastic learning opportunity to dive deep and learn more about computer processors and design, and should prove immensely helpful in becoming an RTL engineer or in pursuing a career with a similar discipline.

## REFERENCES

All the references should follow the same format. A format you can use is as follows:

[1] O. Tigli, M. Zaghloul, “Fabrication and Characterization of a Surface Acoustic Wave Biosensor in CMOS technology for Cancer Biomarker Detection*,*” *IEEE Transactions on Biomedical Circuits and Systems*, Vol. 4, No.1, pp. 62-73, Feb. 2010.

[2] Nexys4™ FPGA Board Reference Manual, Nexys-4 rev. B; Revised November 19, 2013 by Diligent Beyond Theory

[3] Brown, S., Vranesic, Z., “Fundamentals of Digital Logic with VHDL Design,” McGraw Hill, 2000.

## APPENDIX

**ALU:**

module ALU(FS, A, B, out);

parameter nBit = 16;

input [2:0] FS;

input [nBit-1:0] A, B;

output reg [nBit-1:0] out;

wire [nBit-1:0] arith\_out;

add\_sub #(nBit) CE1(.A(A),

.B(B),

.cond(FS[0]),

.out(arith\_out));

always@(\*)

begin

if(FS[2:1] == 2'b00)

out <= arith\_out;

else if(FS[2] || FS[1])

begin

case(FS[1:0])

2'b10: out <= A & B;

2'b11: out <= A | B;

2'b00: out <= A ^ B;

2'b01: out <= ~A;

endcase

end

end

endmodule

**CPU:**

module CPU(data\_from\_rom, reset, clk, address\_to\_rom, enable\_to\_rom, data\_from\_ram, data\_to\_ram, write\_enable\_to\_ram, address\_to\_ram, read\_enable\_to\_ram, enable\_ram\_read);

parameter nBit = 16;

input [nBit-1:0] data\_from\_rom, data\_from\_ram;

input clk, reset;

output [nBit-1:0] data\_to\_ram;

output [5:0] address\_to\_rom, address\_to\_ram;

output enable\_to\_rom, write\_enable\_to\_ram, read\_enable\_to\_ram, enable\_ram\_read;

wire [nBit-1:0] A;

wire [5:0] PC, PC\_prev;

wire [3:0] DA, AA, BA;

wire [2:0] FS;

wire zero\_flag, IL, RW, MB, MD, MJ, MM, MW, MK, A\_thru, B\_thru, done;

controller #(nBit) T1(.A(A),

.data\_from\_rom(data\_from\_rom),

.zero\_flag(zero\_flag),

.clk(clk), .reset(reset),

.DA(DA),

.AA(AA),

.BA(BA),

.FS(FS),

.IL(IL),

.RW(RW),

.MB(MB),

.MD(MD),

.MJ(MJ),

.MM(MM),

.MW(MW),

.MK(MK),

.A\_thru(A\_thru),

.B\_thru(B\_thru),

.PC(PC),

.PC\_prev(PC\_prev),

.done(done));

datapath #(nBit) T2(.mem(data\_from\_ram),

.PC(PC),

.PC\_prev(PC\_prev),

.DA(DA),

.AA(AA),

.BA(BA),

.FS(FS),

.RW(RW),

.MB(MB),

.MD(MD),

.MJ(MJ),

.MM(MM),

.MK(MK),

.A\_thru(A\_thru),

.B\_thru(B\_thru),

.reset(reset),

.clk(clk),

.zero\_flag(zero\_flag),

.MUX\_B(data\_to\_ram),

.MUX\_M(address\_to\_rom),

.A(A));

assign enable\_to\_rom = ~IL;

assign write\_enable\_to\_ram = MW;

assign read\_enable\_to\_ram = 1'b1;

assign address\_to\_ram = A[5:0];

assign enable\_ram\_read = done;

endmodule

**FnctionalUnit:**

module FunctionalUnit(FS, A, B, A\_thru, B\_thru, zero\_flag, out);

parameter nBit = 16;

input [2:0] FS;

input [nBit-1:0] A, B;

input A\_thru, B\_thru;

output reg zero\_flag;

output reg [nBit-1:0] out;

wire [nBit-1:0] ALU\_out;

wire [nBit-1:0] shift\_out;

ALU #(nBit) CE2(.FS(FS),

.A(A),

.B(B),

.out(ALU\_out));

ShiftArray #(nBit) CE3(.In(A),

.sh\_cond(FS[0]),

.out(shift\_out));

always@(\*)

begin

if(B\_thru)

out <= B;

else if(A\_thru)

out <= A;

else

begin

if(FS[2] && FS[1])

out <= shift\_out;

else

out <= ALU\_out;

end

end

always@(\*)

begin

if(out == 0)

zero\_flag <= 1;

else

zero\_flag <= 0;

end

endmodule

**Shift Array:**

module ShiftArray(In, sh\_cond, out);

parameter nBit = 16;

input [nBit-1:0] In;

input sh\_cond;

output reg [nBit-1:0] out;

always@(\*)

begin

if(sh\_cond == 0)

out <= {In[nBit-2:0], 1'b0};

else if(sh\_cond == 1)

out <= {In[nBit-1], In[nBit-1:1]};

end

endmodule

**Adder/Subtractor:**

module add\_sub(A, B, cond, out);

parameter nBit = 16;

input [nBit-1:0] A, B;

input cond;

output [nBit-1:0] out;

reg [nBit-1:0] B\_actual;

wire [nBit-1:0] c;

always@(\*)

begin

if(cond)

B\_actual <= ~B + cond;

else

B\_actual <= B;

end

genvar i;

generate

for(i=0; i <= nBit-1; i=i+1)

begin: generate\_add\_sub

if(i==0)

half\_adder HA(A[0], B\_actual[0], c[0], out[0]);

else

full\_adder FA(A[i], B\_actual[i], c[i-1], c[i], out[i]); // why c[i-1]?

end

endgenerate

endmodule

**Control Logic:**

module control\_logic(opcode, DA, zero\_flag, clk, reset, FS, PS, IL, MB, MD, MJ, RW, MM, MW, MK, A\_thru, B\_thru, done);

parameter nBit = 16;

parameter s0 = 3'b000,

s1 = 3'b001,

s2 = 3'b010,

s3 = 3'b011,

s4 = 3'b100,

s5 = 3'b101;

input [3:0] opcode, DA;

input zero\_flag, clk, reset;

output [2:0] FS;

output reg [1:0] PS;

output reg IL, MM, RW, done;

output MB, MD, MJ, MW, MK, A\_thru, B\_thru;

reg [2:0] state;

wire MM\_decode;

wire RW\_decode;

instruction\_decoder SE1(.opcode(opcode),

.DA(DA),

.FS(FS),

.RW(RW\_decode),

.MB(MB),

.MD(MD),

.MJ(MJ),

.MM(MM\_decode),

.MW(MW),

.MK(MK),

.B\_thru(B\_thru),

.A\_thru(A\_thru));

always@(posedge clk)

begin

if(reset)

begin

state <= s0;

end

else

begin

case(state)

s0: state <= s1;

s1:

begin

if(opcode[3] == 0 || opcode == 4'b1000 || opcode == 4'b1001 || opcode == 4'b1010 || (opcode == 4'b1011 && zero\_flag == 0) || (opcode == 4'b1100 && zero\_flag == 1))

state <= s2;

else if((opcode == 4'b1011 && zero\_flag == 1) || (opcode == 4'b1100 && zero\_flag == 0) || (opcode == 4'b1101) || (opcode == 4'b1110))

state <= s3;

else if(opcode == 4'b1111 && DA == 4'b0000)

state <= s4;

else if(opcode == 4'b1111 && DA == 4'b1111)

state <= s5;

end

s2: state <= s1;

s3: state <= s1;

s4: state <= s1;

s5: state <= s5;

endcase

end

end

always@(state)

begin

case(state)

s0:

begin

done <= 0;

PS <= 2'b00;

IL <= 0;

MM <= 1;

RW <= 0;

end

s1: // execution step

begin

PS <= 2'b00;

IL <= 1;

MM <= 0;

RW <= RW\_decode;

end

s2:

begin

PS <= 2'b01;

IL <= 0;

MM <= 1;

RW <= 0;

end

s3:

begin

PS <= 2'b10;

IL <= 0;

MM <= 1;

RW <= 0;

end

s4:

begin

PS <= 2'b11;

IL <= 0;

MM <= 1;

RW <= 0;

end

s5:

begin

done <= 1;

IL <= 0;

MM <= 1;

RW <= 0;

end

endcase

end

endmodule

**Controller:**

module controller(A, data\_from\_rom, zero\_flag, clk, reset, DA, AA, BA, FS, IL, RW, MB, MD, MJ, MM, MW, MK, A\_thru, B\_thru, PC, PC\_prev, done);

parameter nBit = 16;

input [nBit-1:0] A, data\_from\_rom;

input clk, reset, zero\_flag;

output [5:0] PC, PC\_prev;

output [3:0] DA, AA, BA;

output [2:0] FS;

output IL, RW, MB, MD, MJ, MM, MW, MK, A\_thru, B\_thru, done;

wire [3:0] opcode;

wire [1:0] PS;

program\_counter SE2(.A(A[5:0]),

.AA(AA),

.BA(BA),

.PS(PS),

.clk(clk),

.reset(reset),

.PC(PC),

.PC\_prev(PC\_prev));

instruction\_reg #(nBit) SE3(.control\_word(data\_from\_rom),

.instruct\_load(IL),

.reset(reset),

.clk(clk),

.opcode(opcode),

.DA(DA),

.AA(AA),

.BA(BA));

control\_logic #(nBit) SE4(.opcode(opcode),

.DA(DA),

.zero\_flag(zero\_flag),

.clk(clk),

.reset(reset),

.FS(FS),

.PS(PS),

.IL(IL),

.MB(MB),

.MD(MD),

.MJ(MJ),

.RW(RW),

.MM(MM),

.MW(MW),

.MK(MK),

.A\_thru(A\_thru),

.B\_thru(B\_thru),

.done(done));

endmodule

**Datapath:**

module datapath(mem, PC, PC\_prev, DA, AA, BA, FS, RW, MB, MD, MJ, MM, MK, A\_thru, B\_thru, reset, clk, zero\_flag, MUX\_B, MUX\_M, A);

parameter nBit = 16;

input [nBit-1:0] mem;

input [5:0] PC, PC\_prev;

input [3:0] DA, AA, BA;

input [2:0] FS;

input RW, MB, MD, MJ, MM, MK, A\_thru, B\_thru, reset, clk;

output zero\_flag;

output [nBit-1:0] A;

output reg [nBit-1:0] MUX\_B;

output reg [5:0] MUX\_M;

wire [nBit-1:0] D, B;

wire [3:0] MUX\_K;

reg [nBit-1:0] MUX\_D;

reg [nBit-1:0] MUX\_J;

assign MUX\_K = MK ? DA:AA;

register\_file #(nBit) CE4(.D(MUX\_J),

.DA(DA),

.AA(MUX\_K),

.BA(BA),

.RW(RW),

.reset(reset),

.clk(clk),

.A(A),

.B(B));

always@(\*)

begin

if(MB == 1)

MUX\_B <= {8'b00000000,{AA, BA}};

else if(MB == 0)

MUX\_B <= B;

end

always@(\*)

begin

if(MM == 1)

MUX\_M <= PC;

else if(MM == 0)

MUX\_M <= A[5:0];

end

FunctionalUnit #(nBit) CE5(.FS(FS),

.A(A),

.B(MUX\_B),

.A\_thru(A\_thru),

.B\_thru(B\_thru),

.zero\_flag(zero\_flag),

.out(D));

always@(\*)

begin

if(MD == 1)

MUX\_D <= mem;

else if(MD == 0)

MUX\_D <= D;

end

always@(\*)

begin

if(MJ == 1)

MUX\_J <= PC\_prev;

else if(MJ == 0)

MUX\_J <= MUX\_D;

end

endmodule

**Full Adder:**

module full\_adder(A, B, Cin, Cout, Sum);

input A, B, Cin;

output Cout, Sum;

wire S;

wire [1:0] C;

half\_adder HA\_1(.A(A), .B(B), .Cout(C[0]), .Sum(S));

half\_adder HA\_2(.A(S), .B(Cin), .Cout(C[1]), .Sum(Sum));

assign Cout = C[0] | C[1];

endmodule

module half\_adder(A, B, Cout, Sum);

input A, B;

output Cout, Sum;

assign Sum = A^B;

assign Cout = A&B;

endmodule

**Instruction Decoder:**

module instruction\_decoder(opcode, DA, FS, RW, MB, MD, MJ, MM, MW, MK, B\_thru, A\_thru);

input [3:0] opcode, DA;

output reg [2:0] FS;

output reg RW, MB, MD, MJ, MM, MW, MK, A\_thru, B\_thru;

always@(opcode)

begin

if(opcode[3] == 0)

begin

RW <= 1;

MB <= 0;

MD <= 0;

MJ <= 0;

MW <= 0;

MK <= 0;

A\_thru <= 0;

B\_thru <= 0;

case(opcode)

4'b0000: FS <= 3'b000;

4'b0001: FS <= 3'b001;

4'b0010: FS <= 3'b010;

4'b0011: FS <= 3'b011;

4'b0100: FS <= 3'b100;

4'b0101: FS <= 3'b101;

4'b0110: FS <= 3'b110;

4'b0111: FS <= 3'b111;

endcase

end

else if(opcode == 4'b1000)

begin

RW <= 1;

MB <= 1;

MD <= 0;

MJ <= 0;

MW <= 0;

MK <= 0;

A\_thru <= 0;

B\_thru <= 1;

end

else if(opcode == 4'b1001 || opcode == 4'b1010)

begin

MM <= 0;

A\_thru <= 0;

B\_thru <= 0;

MJ <= 0;

MB <= 0;

MK <= 0;

if(opcode[1:0] == 2'b01)

begin

RW <= 1;

MD <= 1;

MW <= 0;

end

else if(opcode[1:0] == 2'b10)

begin

RW <= 0;

MD <= 0;

MW <= 1;

end

end

else if(opcode == 4'b1011 || opcode == 4'b1100)

begin

RW <= 0;

MB <= 0;

MD <= 0;

MJ <= 0;

MW <= 0;

MK <= 1;

A\_thru <= 1;

B\_thru <= 0;

end

else if(opcode == 4'b1101)

begin

RW <= 1;

MB <= 0;

MD <= 0;

MJ <= 1;

MW <= 0;

MK <= 0;

A\_thru <= 0;

B\_thru <= 0;

end

else if(opcode == 4'b1110)

begin

RW <= 0;

MB <= 0;

MD <= 0;

MJ <= 0;

MW <= 0;

MK <= 0;

A\_thru <= 0;

B\_thru <= 0;

end

else if(opcode == 4'b1111 && DA == 4'b0000)

begin

RW <= 0;

MB <= 0;

MD <= 0;

MJ <= 0;

MW <= 0;

MK <= 0;

A\_thru <= 0;

B\_thru <= 0;

end

else if(opcode == 4'b1111 && DA == 4'b1111)

begin

RW <= 0;

MB <= 0;

MD <= 0;

MJ <= 0;

MW <= 0;

MK <= 0;

A\_thru <= 0;

B\_thru <= 0;

end

end

endmodule

**Instruction Register:**

module instruction\_reg(control\_word, instruct\_load, reset, clk, opcode, DA, AA, BA);

parameter nBit = 16;

input [nBit-1:0] control\_word;

input instruct\_load;

input reset, clk;

output reg [3:0] opcode, DA, AA, BA;

always@(negedge clk)

begin

if(reset)

begin

opcode <= 0;

DA <= 0;

AA <= 0;

BA <= 0;

end

else

begin

if(instruct\_load)

begin

opcode <= control\_word[nBit-1:nBit-4];

DA <= control\_word[nBit-5:nBit-8];

AA <= control\_word[nBit-9:nBit-12];

BA <= control\_word[nBit-13:nBit-16];

end

else if(instruct\_load == 0)

begin

opcode <= opcode;

DA <= DA;

AA <= AA;

BA <= BA;

end

end

end

endmodule

**Program Counter:**

module program\_counter(A, AA, BA, PS, clk, reset, PC, PC\_prev);

input [5:0] A;

input [3:0] AA, BA;

input [1:0] PS;

input clk, reset;

output reg [5:0] PC;

output reg [5:0] PC\_prev;

always@(negedge clk)

begin

if(reset)

begin

PC <= 0;

PC\_prev <= 0;

end

else

begin

case(PS)

2'b00:

begin

PC <= PC;

PC\_prev <= PC\_prev;

end

2'b01:

begin

PC <= PC + 1'b1;

PC\_prev <= PC\_prev + 1'b1;

end

2'b10:

begin

PC <= PC + 1'b1 + ({AA[1:0], BA});

PC\_prev <= PC\_prev + 1'b1;

end

2'b11:

begin

PC <= A;

PC\_prev <= A;

end

endcase

end

end

endmodule

**Read from RAM:**

module read\_from\_ram(

input clk,

input reset,

input [15:0] data\_from\_ram,

input uart\_ready,

output reg [5:0] address\_to\_ram,

output reg read\_enable\_to\_ram,

output reg uart\_send,

output reg [7:0] uart\_data

);

//reg[3:0] mem\_counter ;

reg[2:0] byte\_counter ; // 2+2=4 bytes including new line code

reg stop ; //stop reading the ram

reg uart\_sec\_free ; // sending 32bit word to uart finished

reg read\_input\_from\_ram ;

//reg [7:0] byte1 ; //byte3, byte2 ;

reg [3:0] hex1, hex2, hex3 ; // store [11:0] of the 16 bit ram line

always @ (posedge clk) //address\_to\_ram

begin

if ( reset )

address\_to\_ram <= 0 ;

else if ( read\_enable\_to\_ram )

address\_to\_ram <= address\_to\_ram + 4'b0001 ;

else

address\_to\_ram <= address\_to\_ram ;

end

always @ (posedge clk) //stop reading ram (end reached.)

begin

if (reset)

stop <= 0 ;

else if ( ( &address\_to\_ram ) && ( read\_enable\_to\_ram ) )

stop <= 1 ;

else

stop <= stop ;

end

always @ (posedge clk) //read\_enable\_to\_ram ---should be single cycle

begin

if (reset)

read\_enable\_to\_ram <= 0 ;

else if ( ( ~stop ) && uart\_sec\_free && (~ read\_enable\_to\_ram) )

read\_enable\_to\_ram <= 1 ;

else

read\_enable\_to\_ram <= 0 ;

end

always @ (posedge clk) //byte\_counter [ 4 data bytes and 2 new line bytes ]

begin

if (reset)

byte\_counter <= 0 ;

else if ( read\_enable\_to\_ram )

byte\_counter <= 3'd6 ;

else if (uart\_send)

byte\_counter <= byte\_counter - 3'b001 ;

else

byte\_counter <= byte\_counter ;

end

always @ (posedge clk) //read\_input\_from\_ram

begin

if (reset)

read\_input\_from\_ram <= 0 ;

else

read\_input\_from\_ram <= read\_enable\_to\_ram ;

end

// always @ ( posedge clk ) //store data (second byte)

// begin

// if ( reset )

// begin

// // byte3 <= 0 ;

// // byte2 <= 0 ;

// byte1 <= 0 ;

// end

// else if ( read\_input\_from\_ram ) //(read\_enable\_to\_ram)

// begin

// // byte3 <= data\_from\_ram[23:16] ;

// // byte2 <= data\_from\_ram[15:8] ;

// byte1 <= data\_from\_ram[7:0] ;

// end

// else

// begin

// // byte3 <= byte3 ;

// // byte2 <= byte2 ;

// byte1 <= byte1 ;

// end

// end

always @ ( posedge clk ) //store data (second byte)

begin

if ( reset )

begin

hex3 <= 0 ;

hex2 <= 0 ;

hex1 <= 0 ;

end

else if ( read\_input\_from\_ram ) //(read\_enable\_to\_ram)

begin

hex3 <= data\_from\_ram[11:8] ;

hex2 <= data\_from\_ram[7:4] ;

hex1 <= data\_from\_ram[3:0] ;// byte1 <= data\_from\_ram[7:0] ;

end

else

begin

hex3 <= hex3 ;

hex2 <= hex2 ;

hex1 <= hex1 ;

end

end

always @ (posedge clk) //uart\_send

begin

if (reset)

uart\_send <= 0 ;

else if ( read\_input\_from\_ram ) //( read\_enable\_to\_ram )

uart\_send <= 1;

else if ( ( byte\_counter != 0) && ( uart\_ready ) && ( ~ uart\_send ) )

uart\_send <= 1;

else

uart\_send <= 0;

end

always @ ( posedge clk ) //uart\_data 8bit

begin

if ( reset )

uart\_data <= 8'd0 ;

else if ( read\_input\_from\_ram ) //(read\_enable\_to\_ram)

begin

case (data\_from\_ram[15:12])

0 : uart\_data <= 8'h30 ;

1 : uart\_data <= 8'h31 ;

2 : uart\_data <= 8'h32 ;

3 : uart\_data <= 8'h33 ;

4 : uart\_data <= 8'h34 ;

5 : uart\_data <= 8'h35 ;

6 : uart\_data <= 8'h36 ;

7 : uart\_data <= 8'h37 ;

8 : uart\_data <= 8'h38 ;

9 : uart\_data <= 8'h39 ;

10 : uart\_data <= 8'h41 ;

11 : uart\_data <= 8'h42 ;

12 : uart\_data <= 8'h43 ;

13 : uart\_data <= 8'h44 ;

14 : uart\_data <= 8'h45 ;

15 : uart\_data <= 8'h46 ;

endcase

end

else if ( uart\_ready && ( byte\_counter != 0 ) && ( ~ uart\_send ) )

begin

case (byte\_counter)

7 : uart\_data <= 8'hFF ;

6 : uart\_data <= 8'hFF ;

5 :

begin

case (hex3)

0 : uart\_data <= 8'h30 ;

1 : uart\_data <= 8'h31 ;

2 : uart\_data <= 8'h32 ;

3 : uart\_data <= 8'h33 ;

4 : uart\_data <= 8'h34 ;

5 : uart\_data <= 8'h35 ;

6 : uart\_data <= 8'h36 ;

7 : uart\_data <= 8'h37 ;

8 : uart\_data <= 8'h38 ;

9 : uart\_data <= 8'h39 ;

10 : uart\_data <= 8'h41 ;

11 : uart\_data <= 8'h42 ;

12 : uart\_data <= 8'h43 ;

13 : uart\_data <= 8'h44 ;

14 : uart\_data <= 8'h45 ;

15 : uart\_data <= 8'h46 ;

endcase

end

4 :

begin

case (hex2)

0 : uart\_data <= 8'h30 ;

1 : uart\_data <= 8'h31 ;

2 : uart\_data <= 8'h32 ;

3 : uart\_data <= 8'h33 ;

4 : uart\_data <= 8'h34 ;

5 : uart\_data <= 8'h35 ;

6 : uart\_data <= 8'h36 ;

7 : uart\_data <= 8'h37 ;

8 : uart\_data <= 8'h38 ;

9 : uart\_data <= 8'h39 ;

10 : uart\_data <= 8'h41 ;

11 : uart\_data <= 8'h42 ;

12 : uart\_data <= 8'h43 ;

13 : uart\_data <= 8'h44 ;

14 : uart\_data <= 8'h45 ;

15 : uart\_data <= 8'h46 ;

endcase

end

3 :

begin

case (hex1)

0 : uart\_data <= 8'h30 ;

1 : uart\_data <= 8'h31 ;

2 : uart\_data <= 8'h32 ;

3 : uart\_data <= 8'h33 ;

4 : uart\_data <= 8'h34 ;

5 : uart\_data <= 8'h35 ;

6 : uart\_data <= 8'h36 ;

7 : uart\_data <= 8'h37 ;

8 : uart\_data <= 8'h38 ;

9 : uart\_data <= 8'h39 ;

10 : uart\_data <= 8'h41 ;

11 : uart\_data <= 8'h42 ;

12 : uart\_data <= 8'h43 ;

13 : uart\_data <= 8'h44 ;

14 : uart\_data <= 8'h45 ;

15 : uart\_data <= 8'h46 ;

endcase

end

2 : uart\_data <= 8'h0d ; // new line

1 : uart\_data <= 8'h0a ; // new line

0 : uart\_data <= 8'hFF;

endcase

end

end

/\*

always @ ( posedge clk ) //uart\_data 8bit

begin

if ( reset )

uart\_data <= 8'd0 ;

else if ( read\_input\_from\_ram ) //(read\_enable\_to\_ram)

uart\_data <= data\_from\_ram[15:8] ;

else if ( uart\_ready && ( byte\_counter != 0 ) && ( ~ uart\_send ) )

begin

case (byte\_counter)

6 : uart\_data <= 8'hFF; // should never happen

5 : uart\_data <= 8'hFF; // should never happen

4 : uart\_data <= 8'hFF; // should never happen

3 : uart\_data <= byte1 ;

2 : uart\_data <= 8'h0d ; // new line

1 : uart\_data <= 8'h0a ; // new line

0 : uart\_data <= 8'hFF;

endcase

end

end

\*/

always @ ( posedge clk ) //uart\_sec\_free

begin

if (reset)

uart\_sec\_free <= 1 ;

else if ( ( byte\_counter == 0 ) && uart\_ready && ( ~read\_enable\_to\_ram ) )

uart\_sec\_free <= 1 ;

else

uart\_sec\_free <= 0 ;

end

endmodule

**Read RAM and UART:**

module read\_ram\_and\_uart(

input clk,

input reset,

input [15:0] data\_from\_ram,

output read\_enable\_to\_ram,

output [5:0] address\_to\_ram,

output uart\_TX

);

wire wr\_uart\_send, wr\_uart\_ready ;

wire[7:0] wr\_data\_byte ;

//for ram

//wire[31:0] wr\_data\_from\_ram ;

//wire[3:0] wr\_address\_to\_ram ;

// wire wr\_read\_enable\_to\_ram ;

/\* // this ROM should be replaced by RAM later

ip\_rom\_in rom\_input (

.clka(clk), // input clka

.ena(wr\_read\_enable\_to\_ram), // input ena

.addra(wr\_address\_to\_ram), // input [3 : 0] addra

.douta(wr\_data\_from\_ram) // output [31 : 0] douta

);

\*/

read\_from\_ram read\_RAM(

.clk(clk),

.reset(reset),

.data\_from\_ram(data\_from\_ram), //32bit

.uart\_ready(wr\_uart\_ready),

.address\_to\_ram(address\_to\_ram), //4bit

.read\_enable\_to\_ram(read\_enable\_to\_ram),

.uart\_send(wr\_uart\_send),

.uart\_data(wr\_data\_byte) //8bit

);

UART\_TX\_CTRL UART\_cont(

.SEND(wr\_uart\_send),

.DATA(wr\_data\_byte), // 8bit

.CLK(clk),

.READY(wr\_uart\_ready),

.UART\_TX(uart\_TX)

) ;

endmodule

**Register File:**

module register\_file(D, DA, AA, BA, RW, clk, reset, A, B);

parameter nBit = 16;

input [15:0] D;

input [3:0] DA, AA, BA;

input RW, reset, clk;

output reg [15:0] A, B;

reg [15:0] reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7, reg8, reg9, reg10, reg11, reg12, reg13, reg14, reg15;

always @(negedge clk)

begin

if(reset)

begin

reg0 <= 0;

reg1 <= 0;

reg2 <= 0;

reg3 <= 0;

reg4 <= 0;

reg5 <= 0;

reg6 <= 0;

reg7 <= 0;

reg8 <= 0;

reg9 <= 0;

reg10 <= 0;

reg11 <= 0;

reg12 <= 0;

reg13 <= 0;

reg14 <= 0;

reg15 <= 0;

end

else

begin

if(RW)

begin

case(DA)

4'b0000: reg0 <= D;

4'b0001: reg1 <= D;

4'b0010: reg2 <= D;

4'b0011: reg3 <= D;

4'b0100: reg4 <= D;

4'b0101: reg5 <= D;

4'b0110: reg6 <= D;

4'b0111: reg7 <= D;

4'b1000: reg8 <= D;

4'b1001: reg9 <= D;

4'b1010: reg10 <= D;

4'b1011: reg11 <= D;

4'b1100: reg12 <= D;

4'b1101: reg13 <= D;

4'b1110: reg14 <= D;

4'b1111: reg15 <= D;

endcase

end

else if(RW == 0)

begin

reg0 <= reg0;

reg1 <= reg1;

reg2 <= reg2;

reg3 <= reg3;

reg4 <= reg4;

reg5 <= reg5;

reg6 <= reg6;

reg7 <= reg7;

reg8 <= reg8;

reg9 <= reg9;

reg10 <= reg10;

reg11 <= reg11;

reg12 <= reg12;

reg13 <= reg13;

reg14 <= reg14;

reg15 <= reg15;

end

end

end

always @(\*)

begin

case(AA)

4'b0000 : A <= reg0;

4'b0001 : A <= reg1;

4'b0010 : A <= reg2;

4'b0011 : A <= reg3;

4'b0100 : A <= reg4;

4'b0101 : A <= reg5;

4'b0110 : A <= reg6;

4'b0111 : A <= reg7;

4'b1000 : A <= reg8;

4'b1001 : A <= reg9;

4'b1010 : A <= reg10;

4'b1011 : A <= reg11;

4'b1100 : A <= reg12;

4'b1101 : A <= reg13;

4'b1110 : A <= reg14;

4'b1111 : A <= reg15;

endcase

case(BA)

4'b0000 : B <= reg0;

4'b0001 : B <= reg1;

4'b0010 : B <= reg2;

4'b0011 : B <= reg3;

4'b0100 : B <= reg4;

4'b0101 : B <= reg5;

4'b0110 : B <= reg6;

4'b0111 : B <= reg7;

4'b1000 : B <= reg8;

4'b1001 : B <= reg9;

4'b1010 : B <= reg10;

4'b1011 : B <= reg11;

4'b1100 : B <= reg12;

4'b1101 : B <= reg13;

4'b1110 : B <= reg14;

4'b1111 : B <= reg15;

endcase

end

endmodule

**Top Level:**

module top\_level(

input reset,

input clk,

output UART\_TX

);

//address lines

wire[5:0] wr\_address\_to\_rom ;

wire [5:0] wr\_read\_address\_to\_ram\_from\_readout, wr\_address\_to\_ram\_from\_cpu, wr\_address\_to\_ram ;

//enable lines

wire wr\_enable\_to\_rom, wr\_enable\_ram; //ROM and RAM enable lines

wire wr\_enable\_ram\_read; //RAM enable line from UART

wire wr\_write\_to\_ram\_from\_cpu, wr\_read\_enable\_to\_ram\_from\_cpu; // RAM enable lines from CPU

//data lines

wire [15:0] wr\_data\_out\_from\_rom, wr\_data\_to\_ram, wr\_data\_from\_ram;

//clock lines and uart control

wire wr\_clk\_read\_ram, wr\_read\_ram, wr\_reset\_read\_ram ;

//---------------- clk for the CPU (uut)

wire clk\_cpu; //enable clk\_for\_CPU

assign clk\_cpu = clk ; //& enable\_cpu ;

// controls for read\_ram\_and\_uart

assign wr\_clk\_read\_ram = clk && wr\_enable\_ram\_read ; //wr\_enable\_ram\_read && clk ;

assign wr\_reset\_read\_ram = reset\_read\_ram ;

reg [3:0] counter\_read\_ram\_reset ;

reg reset\_read\_ram, reset\_read\_ram\_done ;

//-------------reset signal for read\_ram\_module------------------------

always @ ( posedge clk ) //counter\_read\_ram\_reset -- to start reading ram

begin

if (reset)

counter\_read\_ram\_reset <= 4'b0 ;

else if ( wr\_enable\_ram\_read && ( counter\_read\_ram\_reset == 0 ) )

counter\_read\_ram\_reset <= 4'b1 ;

else if ( ( counter\_read\_ram\_reset != 0) && ( counter\_read\_ram\_reset != 4'b1111 ) )

counter\_read\_ram\_reset <= counter\_read\_ram\_reset + 4'b1 ;

else

counter\_read\_ram\_reset <= counter\_read\_ram\_reset ;

end

always @ (posedge clk ) //reset\_read\_ram

begin

if (reset)

reset\_read\_ram <= 0 ;

else if ( ( counter\_read\_ram\_reset == 4'b1111 ) && ( reset\_read\_ram\_done != 1) )

reset\_read\_ram <= 1 ;

else

reset\_read\_ram <= 0 ;

end

always @ ( posedge clk )

begin

if (reset)

reset\_read\_ram\_done <= 0 ;

else if ( counter\_read\_ram\_reset == 4'b1111 )

reset\_read\_ram\_done <= 1 ;

else

reset\_read\_ram\_done <= reset\_read\_ram\_done ;

end

//-------------------------------------------------------------------

ip\_rom\_input\_code ROM (

.clka(clk), // input clka

.ena(wr\_enable\_to\_rom), // input ena

.addra(wr\_address\_to\_rom), // input [5 : 0] addra

.douta(wr\_data\_out\_from\_rom) // output [15 : 0] douta

);

CPU dummy\_unit1( //reads ROM, reads and writes to RAM

.data\_from\_rom(wr\_data\_out\_from\_rom),

.reset(reset),

.clk(clk\_cpu),

.address\_to\_rom(wr\_address\_to\_rom),

.enable\_to\_rom(wr\_enable\_to\_rom),

.data\_from\_ram(wr\_data\_from\_ram),

.data\_to\_ram(wr\_data\_to\_ram),

.write\_enable\_to\_ram(wr\_write\_to\_ram\_from\_cpu),

.address\_to\_ram(wr\_address\_to\_ram\_from\_cpu),

.read\_enable\_to\_ram(wr\_read\_enable\_to\_ram\_from\_cpu),

.enable\_ram\_read(wr\_enable\_ram\_read) //enable ram read and uart module

);

read\_ram\_and\_uart read\_ram\_and\_uart(

.clk(wr\_clk\_read\_ram),

.reset(wr\_reset\_read\_ram),

.data\_from\_ram(wr\_data\_from\_ram),

.read\_enable\_to\_ram(wr\_read\_ram),

.address\_to\_ram(wr\_read\_address\_to\_ram\_from\_readout),

.uart\_TX(UART\_TX)

);

assign wr\_address\_to\_ram = (wr\_enable\_ram\_read) ? wr\_read\_address\_to\_ram\_from\_readout : wr\_address\_to\_ram\_from\_cpu;//address ram

assign wr\_enable\_ram = wr\_read\_enable\_to\_ram\_from\_cpu || wr\_write\_to\_ram\_from\_cpu || wr\_read\_ram ;

ip\_ram\_output\_code RAM (

.clka(clk), // input clka

.ena(wr\_enable\_ram), // input ena

.wea(wr\_write\_to\_ram\_from\_cpu), // input wea

.addra(wr\_address\_to\_ram), // input [5 : 0] addra

.dina(wr\_data\_to\_ram), // input [15 : 0] dina

.douta(wr\_data\_from\_ram) // output [15 : 0] douta

);

endmodule

