COMPE 475: Microprocessors

Single Cycle ARM Microprocessor - Report

Ana Maghradze

17th December 2021

Contents

1 Description of The Assignment

The assignment is to implement the single cycle microprocessor. As we have already implemented the smaller parts the assignment was divided into, now it's the final step to connect all the modules and make the code work.

2 Description of The Solution

The separate parts of the project are combined and connected to one another in Top module. Top module only takes the clock as input and all the processes are happening inside this module. At first, when PC counter is 0, instruction at address 0 is read and processed; PC counter is incremented every time the next instruction happens so it does not require any additional logic.

INSTRUCTION MEMORY

The instruction memory is for instruction processing. Here is an array of 16 32-bit registers for processing instructions to be done. PC counter tells instruction memory the address of the next instruction, then the corresponding instruction is taken. In my case, instructions are hard-coded as I could not read them from separate memory file (Quartus does not like it for some reason).

DECODER

The decoder module simply takes the entire 32-bit instruction, divides it into named parts and outputs them so that decoded parts can easily be used in other modules correspondingly.

REGISTER FILE

The register file module has the array of 16 32-bit registers, so the module handles writing data into registers and reading data from registers. The 16th register (regfile[15]) is for the PC counter in/from which we cannot write/read value other that PC counter.

ALU

The ALU takes A_in, B_in, and cmd inputs and based on the cmd type, does specific operation, provides corresponding result and generates negative, zero, carry and overflow flags. If cmd is shift operation, ALU already gets shifted value from shifter as input B_in.

SHIFTER

The shifter is a separate module for shift operations. The module, based on the shift type (sh) and 25th bit (immediate) value decides which type of shift will happen. In case of immediate value, only rotation happens, otherwise Rn is shifted by shamt_5 or Rs depending on the value of 4th instruction bit.

DATA MEMORY

The data memory is used for memory instructions (store, load). I have here an array of 32-bit registers. The data can be stored only if the writing is enabled. When load instruction happens, data is read from memory on the address provided as input.

CONTROLLER

The controller is for controlling data flow and managing some selectors for multiplexers or write enables. For example, when store instruction is happening, I disable writing in register file so the destination address for data memory comes from the ALU result (base + offset) and it does not require writing this address into register file. When load instruction is happening, writing in register file is allowed as we need to store the data loaded from memory into somewhere.

ZERO CONCATENATOR

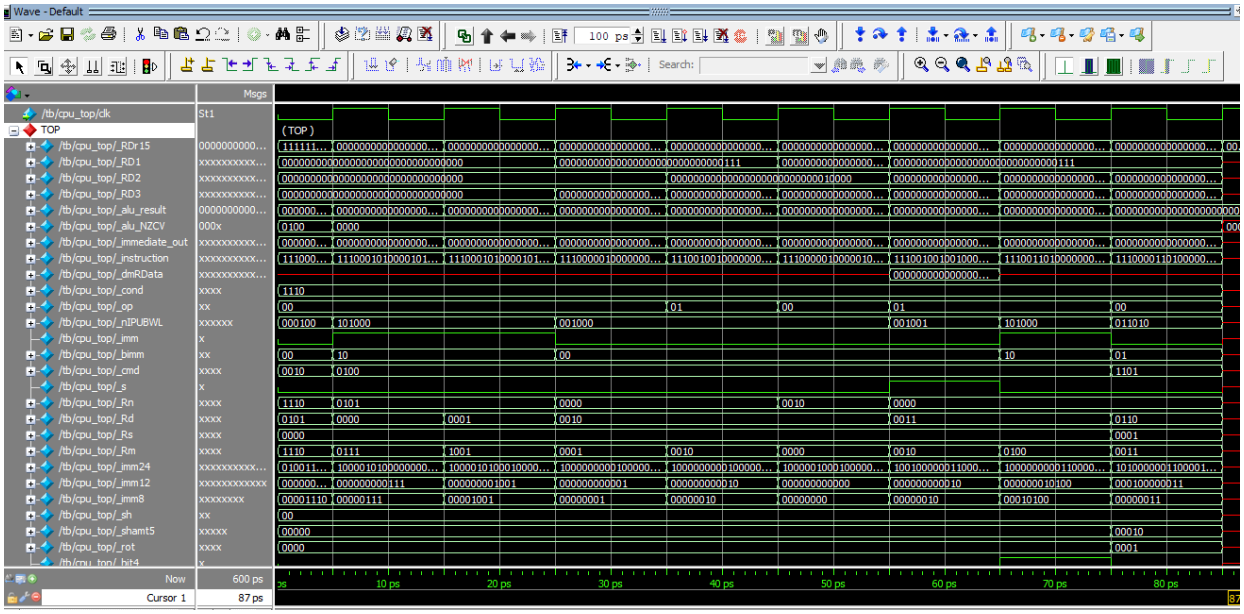
The zero concatenator is for extending immediate value i.e filling with zeros to get the the same value but with different size. Based on the type of operation (data processing, memory, branch) the immediate value is extended up to 32 bits.

MULTIPLEXERS

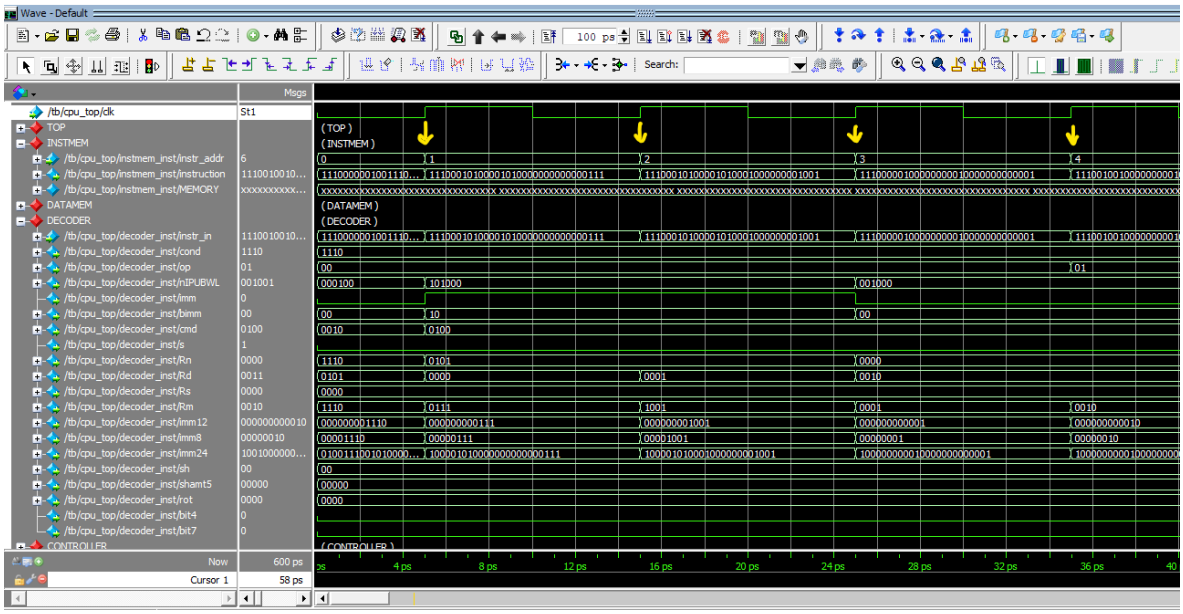
I had to use 3 multiplexers for ALU, regfile and data memory. For example, mux_2x1_alu_src2 is for deciding what will be the B_in input for ALU - extended immediate or shifted value.

3 ModelSim Simulation

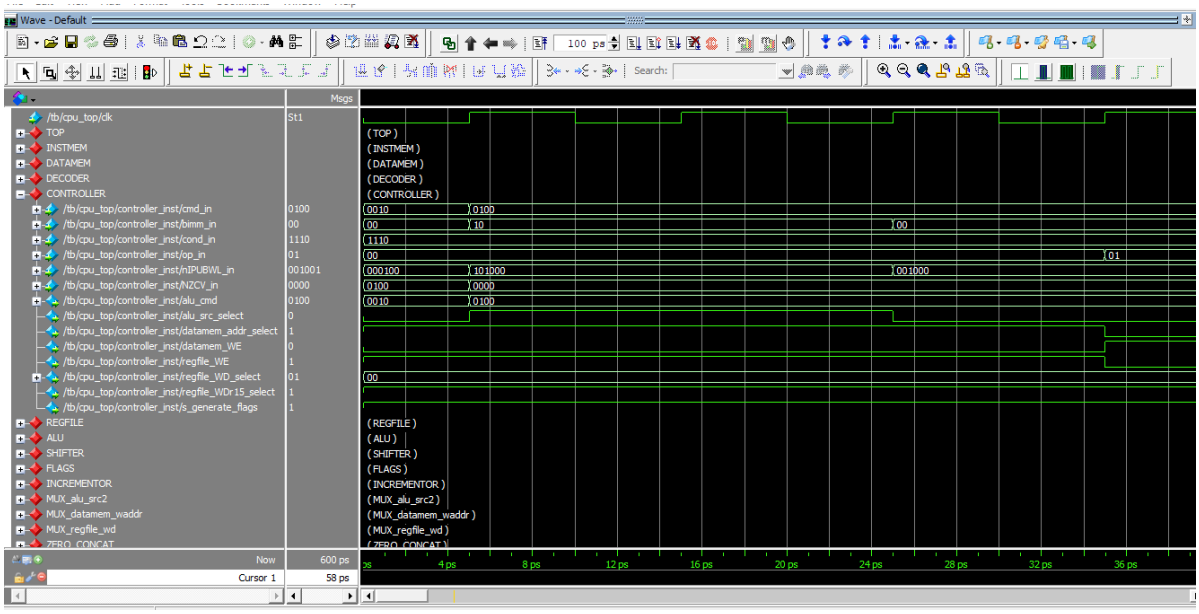
3.1 TOP



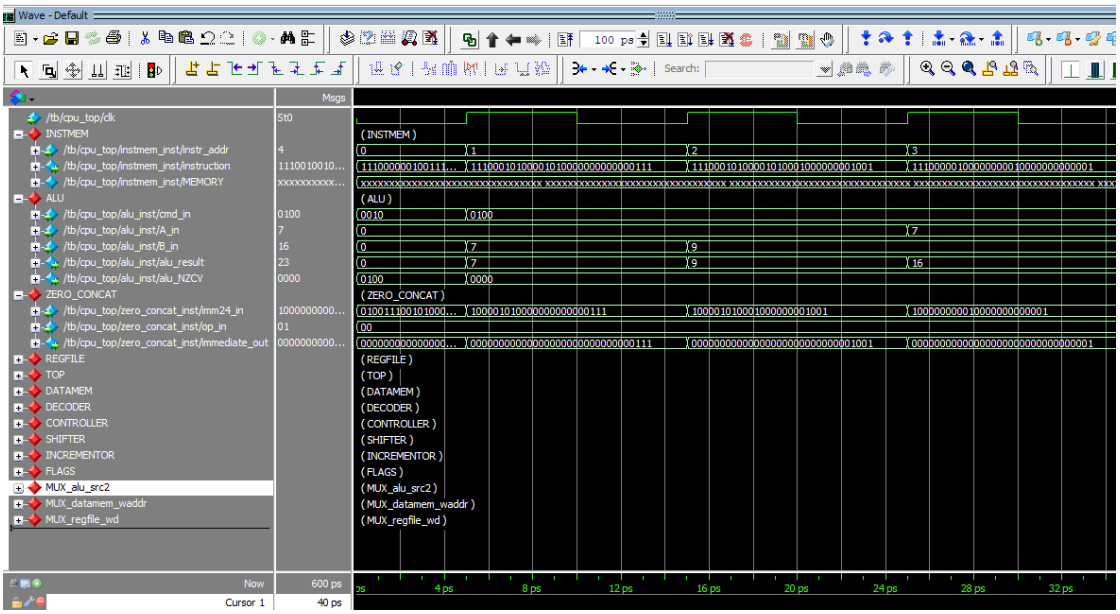
3.2 DECODER



3.3 CONTROLLER



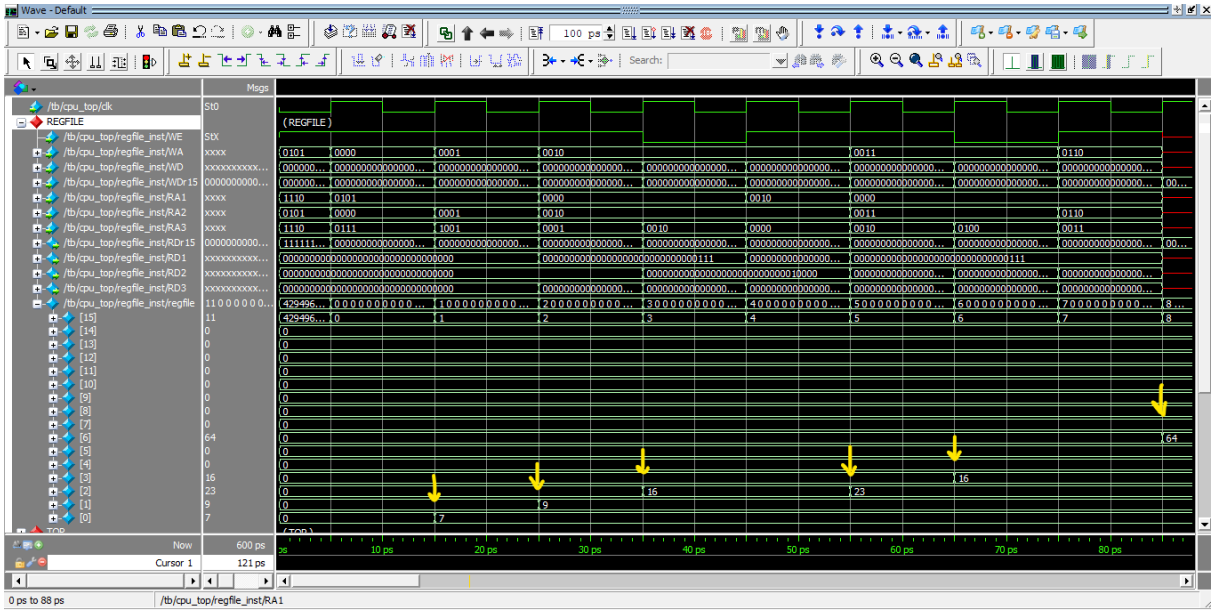
3.4 ALU



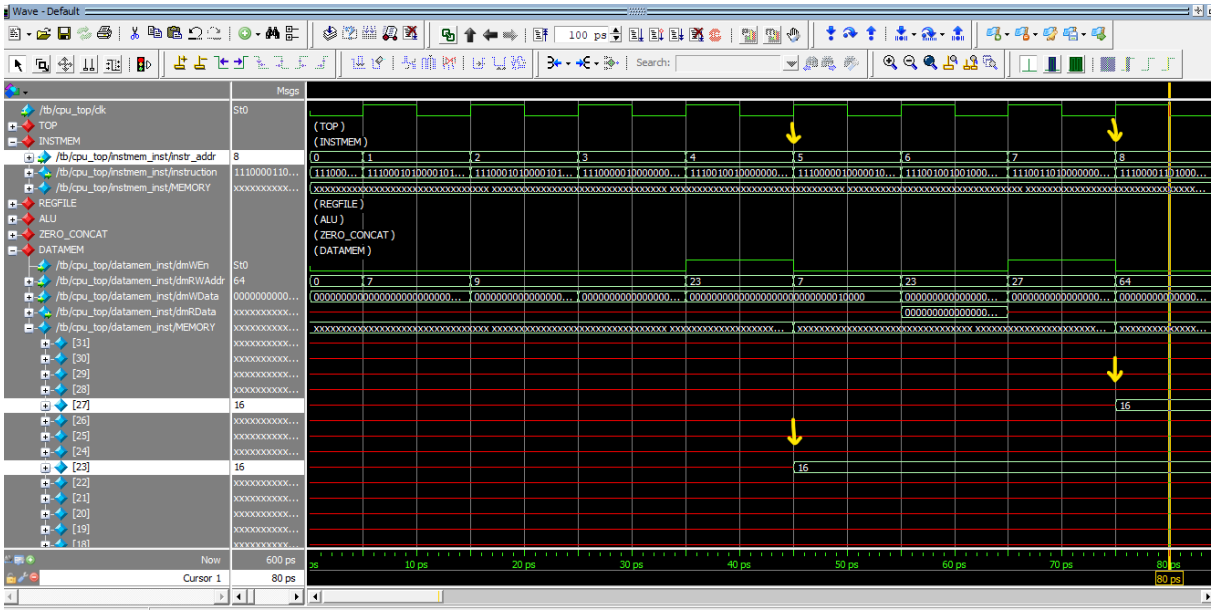
3.5 REGISTER FILE

These are the instructions that are going to be processed and on the simulation screenshots below, you can see how the data is written into register file and data memory.

```
SUB R5, R14, R14   // REG[5] = 0;
ADD R0, R5, #7     // REG[0] = 7;
ADD R1, R5, #9     // REG[1] = 9;
ADD R2, R0, R1     // REG[2] = 7 + 9 = 16
STR R2, [R0, R2]   // Mem[7+16=23] = 16
ADD R2, R2, R1     // REG[2] = 16 + 7 = 23
LDR R3, [R0, R2]   // REG[3] = Mem[23] = 16
STR R3, [R0, #20]  // Mem[27] = 16
LSL R6, R3, #2     // REG[6] = 64
```



3.6 DATA MEMORY



4 Appendix

4.1 Top

```
module top(
  input clk
);

// REGFILE
logic [31:0] _RDr15;
logic [31:0] _RD1;
logic [31:0] _RD2;
logic [31:0] _RD3;

// ALU
logic [31:0] _alu_result;
logic [3:0] _alu_NZCV;

// zero concatenator
logic [31:0] _immediate_out;

// instruction memory
logic [31:0] _instruction;

// data memory
logic [31:0] _dmRData;

// decoder
logic [3:0] _cond;
logic [1:0] _op;
logic [5:0] _nIPUBWL;
logic _imm;
logic [1:0] _bimm;
logic [3:0] _cmd;
logic _s;
logic [3:0] _Rn;
logic [3:0] _Rd;
logic [3:0] _Rs;
logic [3:0] _Rm;
logic [23:0] _imm24;
logic [11:0] _imm12;
logic [7:0] _imm8;
logic [1:0] _sh;
logic [4:0] _shamt5;
logic [3:0] _rot;
logic _bit4;
logic _bit7;

// controller
logic [3:0] _alu_cmd;
logic _alu_src_select;
logic _datamem_addr_select;
logic _datamem_WE;
logic _regfile_WE;
logic [1:0] _regfile_WD_select;
logic _regfile_WDr15_select;
logic _s_generate_flags;

// flags
logic [3:0] _flags;

// shifter
logic [31:0] _src2_shifted;

// incrementor
logic [31:0] _incremented_pc;

// mux alu src2
logic [31:0] _mux_out_alu_src2;
// mux regfile WD
logic [31:0] _mux_out_regfile_WD;
// mux datamem write address
logic [31:0] _mux_out_datamem_WAddr;
```

```
// *********** CONTROLLER ***********
controller controller_inst(
  .cmd_in(_cmd),
  .bimm_in(_bimm),
  .cond_in(_cond),
  .op_in(_op),
  .nIPUBWL_in(_nIPUBWL),
  .NZCV_in(_alu_NZCV),
  .s_generate_flags_in(_s_generate_flags_in),
  .alu_cmd(_alu_cmd),
  .alu_src_select(_alu_src_select),
  .datamem_addr_select(_datamem_addr_select),
  .datamem_WE(_datamem_WE),
  .regfile_WE(_regfile_WE),
  .regfile_WD_select(_regfile_WD_select),
  .regfile_WDr15_select(_regfile_WDr15_select),
  .s_generate_flags(_s_generate_flags)
);

// *********** REGISTE FILE ***********
register_file regfile_inst(
  .clk(clk),
  .WE(_regfile_WE),
  .WA(_Rd),
  .WD(_mux_out_regfile_WD),
  .WDr15(_incremented_pc),
  .RA1(_Rn),
  .RA2(_Rd),
  .RA3(_Rm),
  .RDr15(_RDr15),
  .RD1(_RD1),
  .RD2(_RD2),
  .RD3(_RD3)
);

// *********** SHIFTER ***********
shifter shifter_inst(
  .shifter_shamt5_in(_shamt5),
  .shifter_sh_in(_sh),
   .imm8extended(_immediate_out),
   .Rm_in(_RD3),
   .Rs_in(_RD2),
   .shifter_rot_in(_rot),
   .instrbit4(_bit4),
   .instrbit25(_imm),
  .src2_shifted(_src2_shifted)
);

// *********** ALU ***********
ALU alu_inst(
  .cmd_in(_alu_cmd),
  .A_in(_RD1),
  .B_in(_mux_out_alu_src2),
  .alu_result(_alu_result),
  .alu_NZCV(_alu_NZCV)
);

// *********** FLAGS ***********
flags flags_inst(
  .clk(clk),
  .s(_s),
  .flagsFromALU(_alu_NZCV),
  .flags(_flags)
);

// *********** INSTRUCTION MEMORY ***********
instrmem instmem_inst(
  .instr_addr(_incremented_pc),
  .instruction(_instruction)
);

// *********** DATA MEMORY ***********
```

```
datamem datamem_inst(
  .clk(clk),
  .dmWEn(_datamem_WE),
  .dmRWAddr(_mux_out_datamem_WAddr),
  .dmWData(_RD2),
  .dmRData(_dmRData)
);

// *********** DECODER ***********
decoder decoder_inst(
  .instr_in(_instruction),
  .cond(_cond),
  .op(_op),
  .nIPUBWL(_nIPUBWL),
  .imm(_imm),
  .bimm(_bimm),
  .cmd(_cmd),
  .s(_s),
  .Rn(_Rn),
  .Rd(_Rd),
  .Rs(_Rs),
  .Rm(_Rm),
  .imm12(_imm12),
  .imm8(_imm8),
  .imm24(_imm24),
  .sh(_sh),
  .shamt5(_shamt5),
  .rot(_rot),
  .bit4(_bit4),
  .bit7(_bit7)
);

// *********** ZERO CONCATENATOR ***********
zero_concatenator zero_concat_inst(
  .imm24_in(_imm24),
  .op_in(_op),
  .immediate_out(_immediate_out)
);

// *********** INCREMENTOR ***********
incrementor incrementor_inst(
  .in(_RDr15),
  .incremented(_incremented_pc)
);

// *********** MULTIPLEXERS ***********
mux_2x1 #(32) mux_2x1_alu_src2(
  .in1(_immediate_out),   // s=1
  .in2(_src2_shifted),    // s=0
  .s(_alu_src_select),
  .mux_out(_mux_out_alu_src2)
);

mux_2x1 #(32) mux_2x1_datamem_WAddr(
  .in1(_src2_shifted),    // s=1
  .in2(_alu_result),    // s=0
  .s(_datamem_addr_select),
  .mux_out(_mux_out_datamem_WAddr)
);

mux_3x1 #(32) mux_3x1_regfile_WD(
  .in1(_alu_result),      // s=0
  .in2(_dmRData),         // s=1
  .in3(_incremented_pc),    // s=2
  .s(_regfile_WD_select),
  .mux_out(_mux_out_regfile_WD)
);

endmodule
```

4.2 Decoder

```
module decoder(
```

```
  input logic [31:0] instr_in,
  output logic [3:0] cond,
  output logic [1:0] op,
  output logic [5:0] nIPUBWL,
  output logic imm,
  output logic [1:0] bimm, //branch immediate
  output logic [3:0] cmd,
  output logic s,
  output logic [3:0] Rn,
  output logic [3:0] Rd,
  output logic [3:0] Rs,
  output logic [3:0] Rm,
  output logic [11:0] imm12,
  output logic [7:0] imm8,
  output logic [23:0] imm24,
  output logic [1:0] sh,
  output logic [4:0] shamt5,
  output logic [3:0] rot,
  output logic bit4,
  output logic bit7
);


assign cond = instr_in[31:28];
assign op = instr_in[27:26];
assign nIPUBWL = instr_in[25:20];
assign imm = instr_in[25];
assign bimm = instr_in[25:24];
assign cmd = instr_in[24:21];
assign s = instr_in[20];
assign Rn = instr_in[19:16];
assign Rd = instr_in[15:12];
assign shamt5 = instr_in[11:7];
assign Rs = instr_in[11:8];
assign rot = instr_in[11:8];
assign bit7 = instr_in[7];
assign Rm = instr_in[3:0];
assign sh = instr_in[6:5];
assign bit4 = instr_in[4];
assign imm8 = instr_in[7:0];
assign imm12 = instr_in[11:0];
assign imm24 = instr_in[23:0];

endmodule
```

4.3 Controller

```
module controller(
  input logic [3:0] cmd_in,
  input logic [1:0] bimm_in,
  input logic [3:0] cond_in,
  input logic [1:0] op_in,
  input logic [5:0] nIPUBWL_in,
  input logic [3:0] NZCV_in,
  input logic s_generate_flags_in,
  output logic [3:0] alu_cmd,
  output logic alu_src_select,
  output logic datamem_addr_select,
  output logic datamem_WE,
  output logic regfile_WE,
  output logic [1:0] regfile_WD_select,
  output logic regfile_WDr15_select,
  output logic s_generate_flags
);

assign alu_cmd = ((op_in == 2'b01 && nIPUBWL_in[3]) || op_in == 2'b10) == 1
          ? 4'b0100
          : (op_in == 2'b01 && ~nIPUBWL_in[3]) == 1
          ? 4'b0010
          : cmd_in;

// nIPUBWL_in[0] = 0 for STORE, nIPUBWL_in[0] = 1 for LOAD
assign datamem_WE = (op_in == 2'b01 && nIPUBWL_in[0] == 0);
```

```
// datamem address becomes imm value
assign datamem_addr_select = regfile_WE ? 1 : 0;

// src2 is value from zero_concat (1), or value from shifter (0)
assign alu_src_select = nIPUBWL_in[5];

// enable if load instr, disable if store instr
assign regfile_WE = (op_in == 2'b01 && ~nIPUBWL_in[0]) ? 0 : 1;

assign regfile_WD_select = (op_in == 2'b01 && nIPUBWL_in[0]) ? 1 : 0;

assign regfile_WDr15_select = 1;
assign s_generate_flags = 1;

endmodule
```

4.4 Register File

```
module register_file(
  input logic clk,
  input logic WE,
  input logic [3:0]  WA,
  input logic [31:0] WD,
  input logic [31:0] WDr15,
  input logic [3:0] RA1, RA2, RA3,
  output logic [31:0] RDr15,
  output logic [31:0] RD1, RD2, RD3
);

logic [31:0] regfile[15:0]; // 16 32-bit registers

assign RD1 = regfile[RA1];
assign RD2 = regfile[RA2];
assign RD3 = regfile[RA3];

assign RDr15 = regfile[15];

always @(posedge clk)
begin
  if(WE) regfile[WA] <= WD;
   regfile[15] <= WDr15;
end

initial
begin
  regfile[0] = 0;
  regfile[1] = 0;
  regfile[2] = 0;
  regfile[3] = 0;
  regfile[4] = 0;
  regfile[5] = 0;
  regfile[6] = 0;
  regfile[7] = 0;
  regfile[8] = 0;
  regfile[9] = 0;
  regfile[10] = 0;
  regfile[11] = 0;
  regfile[12] = 0;
  regfile[13] = 0;
  regfile[14] = 0;
  regfile[15] = -1;
end

endmodule
```

4.5 Instruction Memory

```
module instrmem(
  input logic [31:0] instr_addr,
   output logic [31:0] instruction
);
```

```
logic [31:0] MEMORY[15:0];

assign instruction = MEMORY[instr_addr];

initial
begin
  // SUB R5, R14, R14 --> REG[5] = 0;
  MEMORY[0][31:28]  = 4'b1110;    // cond
  MEMORY[0][27:26]  = 2'b00;      // op, data processing
  MEMORY[0][25]   = 0;        // imm
  MEMORY[0][24:21]  = 4'b0010;    // cmd, subtract
  MEMORY[0][20]   = 0;        // s
  MEMORY[0][19:16]  = 4'b1110;    // Rn
  MEMORY[0][15:12]  = 4'b0101;    // Rd
  MEMORY[0][11:8]   = 4'b0000;    // Rs
  MEMORY[0][7:0]  = 8'b00001110;  // Rm

  // ADD R0, R5, #7 --> REG[0] = 7;
  MEMORY[1][31:28]  = 4'b1110;    // cond
  MEMORY[1][27:26]  = 2'b00;      // op, data processing
  MEMORY[1][25]   = 1;        // imm
  MEMORY[1][24:21]  = 4'b0100;    // cmd, add
  MEMORY[1][20]   = 0;        // s
  MEMORY[1][19:16]  = 4'b0101;    // Rn
  MEMORY[1][15:12]  = 4'b0000;    // Rd
  MEMORY[1][11:8]   = 4'b0000;    // Rs
  MEMORY[1][7:0]  = 8'b00000111;  // imm8

  // ADD R1, R5, #9 --> REG[1] = 9;
  MEMORY[2][31:28]  = 4'b1110;    // cond
  MEMORY[2][27:26]  = 2'b00;      // op, data processing
  MEMORY[2][25]   = 1;        // imm
  MEMORY[2][24:21]  = 4'b0100;    // cmd, add
  MEMORY[2][20]   = 0;        // s
  MEMORY[2][19:16]  = 4'b0101;    // Rn
  MEMORY[2][15:12]  = 4'b0001;    // Rd
  MEMORY[2][11:8]   = 4'b0000;    // Rs
  MEMORY[2][7:0]  = 8'b00001001;  // imm8 or Rm ????

  // ADD R2, R0, R1 --> REG[2] = 7 + 9 = 16
  MEMORY[3][31:28]  = 4'b1110;    // cond
  MEMORY[3][27:26]  = 2'b00;      // op
  MEMORY[3][25]   = 0;        // imm
  MEMORY[3][24:21]  = 4'b0100;    // cmd, add
  MEMORY[3][20]   = 0;        // s
  MEMORY[3][19:16]  = 4'b0000;    // Rn
  MEMORY[3][15:12]  = 4'b0010;    // Rd
  MEMORY[3][11:8]   = 4'b0000;    // Rs
  MEMORY[3][7:0]  = 8'b00000001;  // imm8 or Rm ????

  // STR R2, [R0, R2] // Mem[7+16=23] = 16
  MEMORY[4][31:28]  = 4'b1110;    // cond
  MEMORY[4][27:26]  = 2'b01;      // op, memory operation
  MEMORY[4][25]   = 0;        // imm
  MEMORY[4][24:21]  = 4'b0100;    // cmd, add base and offset
  MEMORY[4][20]   = 0;        // store
  MEMORY[4][19:16]  = 4'b0000;    // Rn
  MEMORY[4][15:12]  = 4'b0010;    // Rd
  MEMORY[4][11:8]   = 4'b0000;    // Rs
  MEMORY[4][7:0]  = 8'b00000010;  // imm8 or Rm ????

  // ADD R2, R2, R1 --> REG[2] = 16 + 7 = 23
  MEMORY[5][31:28]  = 4'b1110;    // cond
  MEMORY[5][27:26]  = 2'b00;      // op
  MEMORY[5][25]   = 0;        // imm
  MEMORY[5][24:21]  = 4'b0100;    // cmd, add
  MEMORY[5][20]   = 0;        // s
  MEMORY[5][19:16]  = 4'b0010;    // Rn
  MEMORY[5][15:12]  = 4'b0010;    // Rd
  MEMORY[5][11:8]   = 4'b0000;    // Rs
  MEMORY[5][7:0]  = 8'b00000000;  // imm8 or 2nd src reg

  // LDR R3, [R0, R2] --> REG[3] = Mem[23] = 16
```

```
  MEMORY[6][31:28]  = 4'b1110;    // cond
  MEMORY[6][27:26]  = 2'b01;      // op, memory operation
  MEMORY[6][25]   = 0;        // imm
  MEMORY[6][24:21]  = 4'b0100;    // cmd, add base and offset
  MEMORY[6][20]   = 1;        // load
  MEMORY[6][19:16]  = 4'b0000;    // Rn
  MEMORY[6][15:12]  = 4'b0011;    // Rd
  MEMORY[6][11:8]   = 4'b0000;    // Rs
  MEMORY[6][7:0]  = 8'b00000010;  // imm8 or 2nd src reg

  // STR R3, [R0, #20] --> Mem[27] = 16
  MEMORY[7][31:28]  = 4'b1110;    // cond
  MEMORY[7][27:26]  = 2'b01;      // op, memory operation
  MEMORY[7][25]   = 1;        // imm
  MEMORY[7][24:21]  = 4'b0100;    // cmd, add base and offset
  MEMORY[7][20]   = 0;        // store
  MEMORY[7][19:16]  = 4'b0000;    // Rn
  MEMORY[7][15:12]  = 4'b0011;    // Rd
  MEMORY[7][11:8]   = 4'b0000;    // Rs
  MEMORY[7][7:0]  = 8'b00010100;  // imm8 or 2nd src reg

  // LSL R6, R3, #2 --> reg[6] = 64
  MEMORY[8][31:28]  = 4'b1110;    // cond
  MEMORY[8][27:26]  = 2'b00;      // op, data processing
  MEMORY[8][25]   = 0;        // imm
  MEMORY[8][24:21]  = 4'b1101;    // cmd, shift
  MEMORY[8][20]   = 0;        // s
  MEMORY[8][19:16]  = 4'b0000;    // Rn, ignored
  MEMORY[8][15:12]  = 4'b0110;    // Rd
  MEMORY[8][11:7]   = 5'b00010;   // shamt-5, shift amount
  MEMORY[8][6:5]  = 2'b00;      // sh, LSL
  MEMORY[8][4]    = 0;        // bit4, shift by shamt-5 times
  MEMORY[8][3:0]  = 4'b0011;    // Rm
end

endmodule
```

4.6 Data Memory

```
module datamem(
  input logic clk,
  input logic dmWEn,
  input logic [31:0] dmRWAddr,
  input logic [31:0] dmWData,
  output logic [31:0] dmRData
);

logic [31:0] MEMORY[31:0];

assign dmRData = MEMORY[dmRWAddr];

always @(posedge clk)
begin
  if(dmWEn) MEMORY[dmRWAddr] <= dmWData;
end

endmodule
```

4.7 ALU

```
module ALU(
  input logic [3:0] cmd_in,
  input logic [31:0] A_in, B_in,
  output logic [31:0] alu_result,
  output logic [3:0] alu_NZCV
);

reg [32:0] result;
wire [3:0] NZCV;
wire [3:0] rot;

// cmd
```

```
localparam cmd_bitwise_and        = 4'b0000;
localparam cmd_bitwise_xor        = 4'b0001;
localparam cmd_subtract           = 4'b0010;
localparam cmd_reverse_subtract     = 4'b0011;
localparam cmd_add              = 4'b0100;
localparam cmd_add_with_carry       = 4'b0101;
localparam cmd_subtract_with_carry    = 4'b0110;
localparam cmd_reverse_sub_with_carry   = 4'b0111;
localparam cmd_test               = 4'b1000;
localparam cmd_test_equivalence       = 4'b1001;
localparam cmd_compare            = 4'b1010;
localparam cmd_compare_negative     = 4'b1011;
localparam cmd_bitwise_or           = 4'b1100;
localparam cmd_shift            = 4'b1101;
localparam cmd_bitwise_clear        = 4'b1110;
localparam cmd_bitwise_not        = 4'b1111;

//assignments
assign alu_NZCV = NZCV[3:0];
assign alu_result = result[31:0];

assign NZCV[3] = result[31]; // negative
assign NZCV[2] = result[31:0] == 0; // zero
assign NZCV[1] = result[32]; // carry
assign NZCV[0] = (A_in[31] == 1 && B_in[31] == 1 && result[31] == 0) || (A_in[31] == 0 && B_in[31] == 0 &&
     result[31] == 1); // overflow

always @(*)
begin
  case(cmd_in)
    cmd_bitwise_not: result[31:0] = ~A_in;
    cmd_bitwise_or: result[31:0] = A_in | B_in;
    cmd_bitwise_and: result[31:0] = A_in & B_in;
    cmd_bitwise_xor: result[31:0] = A_in ^ B_in;
    cmd_bitwise_clear: result[31:0] = A_in & ~B_in;
    /// ...
    cmd_add:
    begin
      result[31:0] = A_in + B_in;
      result[32] = 0;
    end
    cmd_subtract:
    begin
      result[31:0] = A_in - B_in;
      result[32] = 0;
    end
    cmd_reverse_subtract:
    begin
      result[31:0] = B_in - A_in;
      result[32] = 0;
    end
    cmd_add_with_carry: result[32:0] = A_in + B_in;
    cmd_subtract_with_carry: result[32:0] = A_in - B_in;
    cmd_reverse_sub_with_carry: result[32:0] = B_in - A_in;
    cmd_test: result[32:0] = A_in & B_in;
    cmd_test_equivalence: result[32:0] = A_in ^ B_in;
    cmd_compare: result[32:0] = A_in - B_in;
    cmd_compare_negative: result[32:0] = A_in + B_in;
    /// ...
    cmd_shift: result = B_in; // B_in value is already shifted by cpu_shifter
  endcase
end

endmodule
```

4.8 Shifter

```
module shifter(
  input logic [4:0] shifter_shamt5_in,
   input logic [1:0] shifter_sh_in,
   input logic [31:0] imm8extended,
   input logic [31:0] Rm_in,
   input logic [31:0] Rs_in,
```

```
   input logic [3:0] shifter_rot_in,
   input logic instrbit4,
   input logic instrbit25,
  output logic [31:0] src2_shifted
);

localparam LSL = 2'b00;
localparam LSR = 2'b01;
localparam ASR = 2'b10;
localparam ROR = 2'b11;

always @(*)
begin
  // if immediate bit set to 1
  if (instrbit25)
   begin
    // shift imm8extended by rot*2 to the right, store result in src2_shifted
    src2_shifted = {imm8extended, imm8extended} >> shifter_rot_in * 2;
   end
  else
  begin
    if (instrbit4)
      begin
      // shift Rm_in by Rs_in, with shifter_sh_in type of shift, store result in src2_shifted
      case (shifter_sh_in)
        LSL: src2_shifted <= Rm_in << Rs_in;
            LSR: src2_shifted <= Rm_in >> Rs_in;
        ASR: src2_shifted <= Rm_in >>> Rs_in;
        ROR: src2_shifted <= {Rm_in, Rm_in} >> Rs_in;
        default : src2_shifted = Rm_in;
      endcase
    end
      else
    begin
      // shift Rm_in by shifter_shamt5_in, with shifter_sh_in type of shift, store result in src2_shifted
      case (shifter_sh_in)
        LSL: src2_shifted <= Rm_in << shifter_shamt5_in;
            LSR: src2_shifted <= Rm_in >> shifter_shamt5_in;
        ASR: src2_shifted <= Rm_in >>> shifter_shamt5_in;
        ROR: src2_shifted <= {Rm_in, Rm_in} >> shifter_shamt5_in;
        default: src2_shifted <= Rm_in;
      endcase
    end
  end
end

endmodule
```

4.9 Zero Concatenator

```
module zero_concatenator(
  input logic [23:0] imm24_in,
  input logic [1:0] op_in,
  output logic [31:0] immediate_out
);

always @(*)
begin
  case(op_in)
    2'b00: immediate_out = { 24'b0, imm24_in[7:0] }; // data-processing, 8-bit imm
    2'b01: immediate_out = { 20'b0, imm24_in[11:0] }; // memory, 12-bit imm
    2'b10: immediate_out = { { 6 {imm24_in[23:0] } }, imm24_in[23:0], 2'b00 }; // branch, 24-bit signed imm
    default: immediate_out = 32'bx;
  endcase
end

endmodule
```

4.10 Incrementor

```
module incrementor(
  input logic [31:0] in,
```

```
  output logic [31:0] incremented
);

assign incremented = in + 1;

endmodule
```

4.11 Flags

```
module flags(
  input logic clk,
  input logic s,
  input logic [3:0] flagsFromALU,
  output logic [3:0] flags
);

always @(posedge clk)
begin
  if(s) flags <= flagsFromALU;
end

endmodule
```

4.12 Multiplexers

```
module mux_2x1  #(parameter BITS = 32)(
  input logic [BITS-1:0] in1, in2,
  input logic s,
  output logic [BITS-1:0] mux_out
);

assign mux_out = s ? in1 : in2;

endmodule


module mux_3x1  #(parameter BITS = 32)(
  input logic [BITS-1:0] in1, in2, in3,
  input logic [1:0] s,
  output logic [BITS-1:0] mux_out
);

assign mux_out = s == 2'b00 ? in1 : s == 2'b01 ? in2 : in3;

endmodule
```

4.13 Testbench

```
module tb;

reg clk = 0;

always #5 clk <= ~clk;

top cpu_top(clk);

endmodule
```