VLSI System Design (Graduate Level)
Fall 2010

HOMEWORK I (Part II)

` Due day: midnight Oct. 21 (Thursday), 2010

This homework is to let you familiar with tools and Verilog language. It includes A) basic exercises and B) the first part of a simplified multi-cycle **(SMILE)** CPU with only 13 instructions. Part B is to let the first-time CPU designer be familiar with the instruction set architecture, dataflow modeling and controller design. Some problems have reference code. They are for your reference. Most likely, you need to modify them to fit the problem specification.

General rules for deliverables

- This homework needs to be completed by INDIVIDUAL student.
- Compress all files described in the problem statements into one zip or rar.
- Submit the compress file to the course website before the due day. **Warning!** AVOID submit in the last minute. Late submission is not accepted.

Grading Notes

- **Important!** DO remember to include your Verilog code. NO code, NO grades. Also, if your code can not be recompiled by TA successfully using tools in SoC Lab, you receive NO credit.
- Write your report seriously and professionally. Incomplete description and information will reduce your chances to get more credits.
- If extra works (like synthesis, post-simulation or additional instructions) are done, please describe them in your final report clearly for bonus points.
- Please follow course policy.

Deliverables

1. All CPU Verilog codes including components, testbenches and machine code for each lab exercise. NOTE: Please DO NOT include source code in the report!
2. A homework report that includes
    a. A summary in the beginning to state what has been done (such as SMILE CPU, synthesis, post-synthesis simulation, additional branch instruction with verification)
    b. A block diagram for your completed SMILE CPU indicating all necessary components and I/O pins. Note please use MS Visio that is

available in computer center in the university.

c. Simulated waveforms with proper explanation

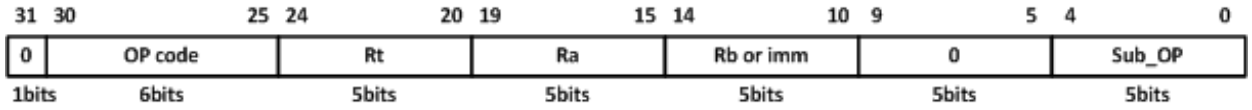
d. Learned lesson and Conclusion

3. Please write in MS word and follow the convention for the file name of your report: n26984795_蕭育書_hw1_report.doc

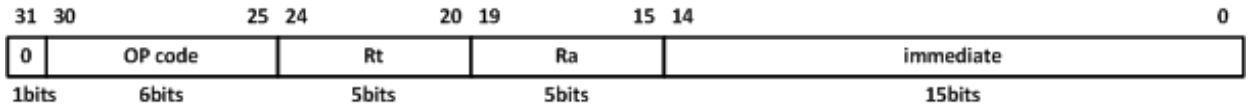
Exercise

We just use some of the instruction format in this homework.

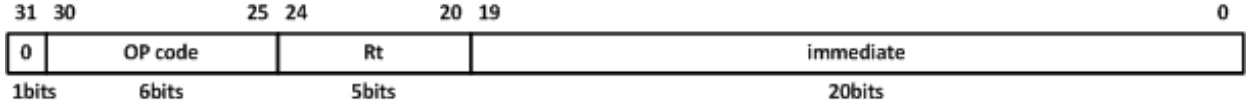
☞Data-processing (SE:sign-extened, ZE:zero-extened)



OP code	Mnemonics	DST	SRC1	SRC2	Sub OP	Description
100000	NOP	00000	00000	00000	01001	No operation
100000	ADD	$Rt	$Ra	$Rb	00000	Rt = Ra + Rb
100000	SUB	$Rt	$Ra	$Rb	00001	Rt = Ra - Rb
100000	AND	$Rt	$Ra	$Rb	00010	Rt = Ra & Rb
100000	OR	$Rt	$Ra	$Rb	00100	Rt = Ra | Rb
100000	XOR	$Rt	$Ra	$Rb	00011	Rt = Ra ^ Rb
100000	SRLI	$Rt	$Ra	imm	01001	Shift right : Rt = Ra >> imm
100000	SLLI	$Rt	$Ra	imm	01000	Shift left : Rt = Ra << imm
100000	ROTRI	$Rt	$Ra	imm	01011	Rotate right : Rt = Ra >> imm



OP code	Mnemonics	DST	SRC1	SRC2	Description
101000	ADDI	$Rt	$Ra	imm	Rt = Ra + SE(imm)
101100	ORI	$Rt	$Ra	imm	Rt = Ra | ZE(imm)
101011	XORI	$Rt	$Ra	imm	Rt = Ra ^ ZE(imm)



OP code	Mnemonics	DST	SRC1	Description
100010	MOVI	$Rt	imm	Rt = SE(immediate)

HOMEWORK I (Part II)

3. **(50 points) Write and verify an ALU which is from exercise 2 in part I combined with a register file as shown in Fig. 3. The 32-bit register file contains 32 registers. This register is shown in Fig. 4 consists of two Read ports and two data output ports (for source register and destination register) and one Write Address Port and one Write Data Port. It shall have the following functions:**
   - **Working at positive edge of clock**
   - **When reset, data values in all registers are reset to zero**
   - **When enable is high and write is asserted, the write_data is stored into the register as pointed by write_address**
   - **When enable is high and read is asserted, the data that are stored in registers as pointed by read_address1 & read_address2 are written to src1 & src2**
   - **It reference code is also attached for your reference.**

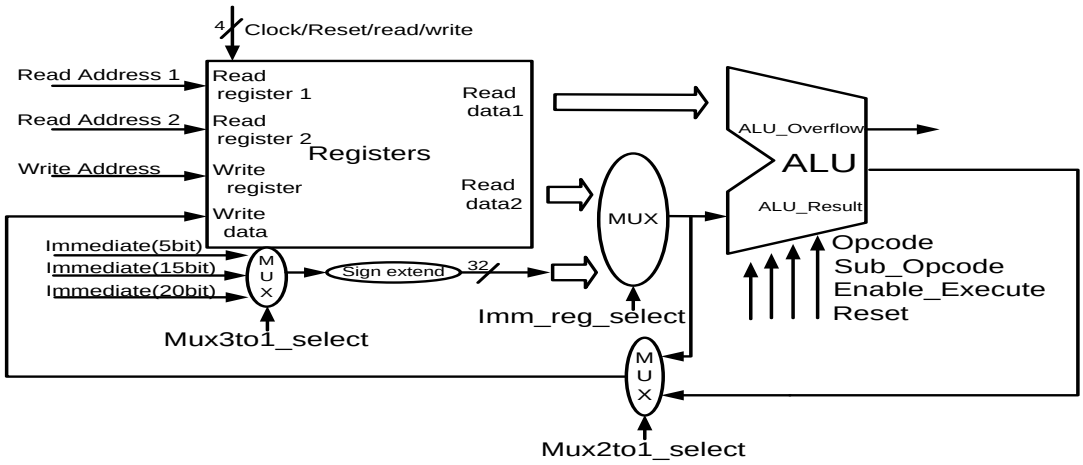


Fig. 3 ALU + Register File

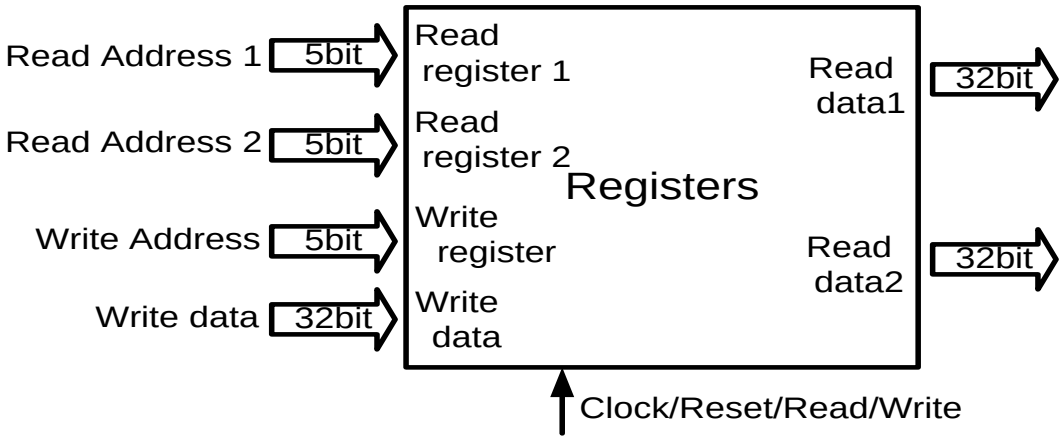


Fig. 4 Register File

Reference code for the register file

```
module regfile(read_data1,read_data2,read_address1,read_address2,
                write_address,write_data,clk,reset,read,write);

  parameter DateSize = 32;
  parameter AddrSize = 5;

  output reg [DateSize-1:0]read_data1;
  output reg [DateSize-1:0]read_data2;

  input [AddrSize-1:0]read_address1;
  input [AddrSize-1:0]read_address2;
  input [AddrSize-1:0]write_address;
  input [DateSize-1:0]write_data;
  input clk,reset,read,write;

  reg [31:0]rw_reg[DateSize-1:0];
  integer i;

  always @ (posedge clk, posedge reset)begin
    if(reset)begin
      for(i=0;i<32;i=i+1)
        rw_reg[i]<=32'b0;
    end
    else begin
      if(read)begin
        read_data1<=rw_reg[read_address1];
        read_data2<=rw_reg[read_address2];
      end
      else if(write)begin
        rw_reg[write_address]<=write_data;
      end
      else begin
        read_data1<=32'b0;
        read_data2<=32'b0;
      end
    end
  end
endmodule
```

4. (60 points) Design a controller unit that is able to decode an input instruction as shown in Fig. 5 and control the datapath unit designed in Problem 4 to carry out the designated operation. A reference state diagram is attached in Fig. 6 for your reference. Develop a testbench that includes the instructions with the format as shown in Fig. 7 and verify the overall system (controller + ALU + register file).

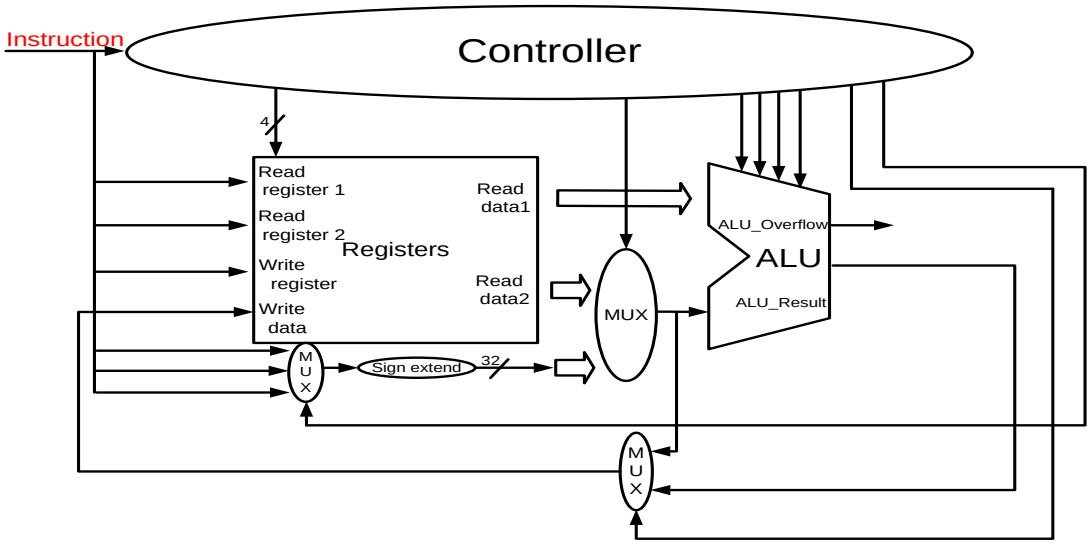


Fig. 5 Block diagram of a controller

The controller shall have the following features:

- All operations initiated at the positive edge trigger
- Control signals:
  - enable_fetch: To fetch and store the designated data from instruction memory into the register file and prepare them to be ready for execution
  - enable_execute: To order ALU to execute the operation
  - enable_writeback: To store the ALU result back to the register file or data memory
- 4 different operating states for the controller, i.e. stop operation, fetch, execute and write-back

Its state diagram of the controller is as shown in Fig. 6.

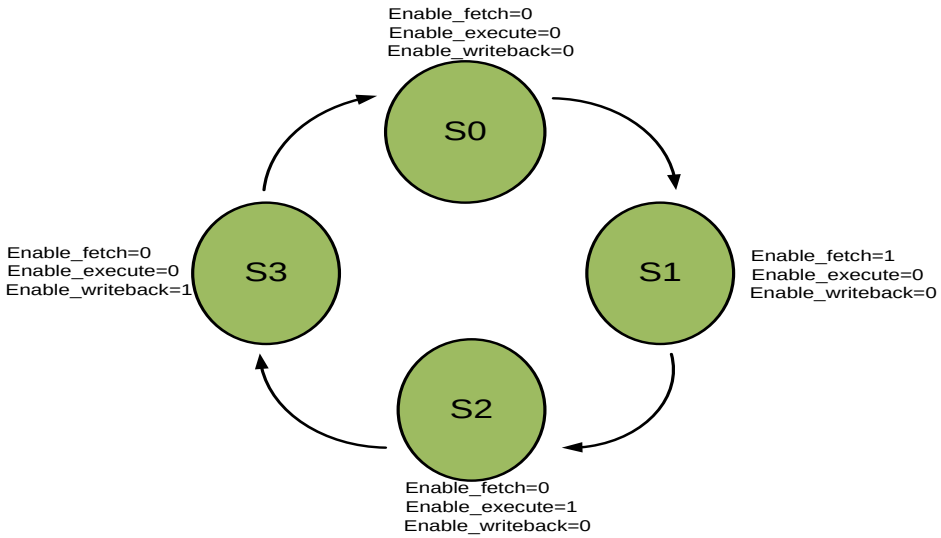


Fig. 6 State Diagram of the controller

```
0_101000_00000_00000_0000_0000_0001_101  // ADDI  R0 = R0 + 5'b01101
0_101000_00001_00001_0000_0000_0001_100  // ADDI  R1 = R1 + 5'b01100
0_100010_00010_0000_0000_0000_0001_0000  // MOVI  R2 = 5'b01000
0_100000_00011_00000_00001_00000_00000   // ADD   R3 = R0 + R1
0_100000_00100_00000_00001_00000_00001   // SUB   R4 = R0 – R1
0_100000_00101_00011_00100_00000_00010   // AND   R5 = R3 & R4
0_100000_00110_00011_00100_00000_00100   // OR    R6 = R3 | R4
0_100000_00111_00011_00100_00000_00011   // XOR   R7 = R3 ^ R4
0_100000_01000_00000_00100_00000_01000   // SLLI  R8 = R0 << 5'b00100
0_100010_01001_00001_01000_00000_01011   // ROTRI R9 = R1 >> 5'b01000
0_101100_00000_00000_0000_0000_0011_111  // ORI   R0 = R0 | 5'b11111
0_101011_00001_00001_0000_0000_0010_101  // XORI  R1 = R1 + 5'b10101
```

Fig. 7 Input instruction in binary format for verifying Problem 5

HOMEWORK I (Part II)

Reference code of the controller

```
// concroller
`define OPCODE  ir[30:25]
`define SUBOPCODE ir[4:0]
`define SRLI 5'b01001
`define SLLI 5'b01000
`define ROTRI 5'b01011

module controller(enable_execute, enable_fetch, enable_writeback, opcode, sub_opcode,
                  mux3to1_select, mux2to1_select, imm_reg_select, clock, reset, PC, ir);
  input clock;
  input reset;
  input [31:0] PC;
  input [31:0] ir;

  output reg enable_execute;
  output reg enable_fetch;
  output reg enable_writeback;
  output [5:0] opcode;
  output [4:0] sub_opcode;
  output reg mux3to1_select;
  output reg mux2to1_select;
  output reg imm_reg_select;

  wire [5:0] opcode = ir[30:25];
  wire [4:0] sub_opcode = ir[4:0];
  reg [1:0] current_state;
  reg [1:0] next_state;
  reg [31:0] present_instruction;

  parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

  always @(posedge clock)
  begin
    if(reset)
      current_state = S0;
    else
      current_state = next_state;
  end

  always @(current_state)
  begin
    case(current_state)
    S0 : begin
      next_state = S1;
      enable_fetch = 0;
      enable_execute = 0;
      enable_writeback = 0;
      mux3to1_select = 0;
      mux2to1_select = 0;
      imm_reg_select = 0;
         end
        .
        .
        .
    endcase
  end

  always @(posedge enable_fetch)
  begin
    if(PC == 0)
      present_instruction = 0;
    else
      present_instruction = ir;
  end

endmodule
```

5. (Bonus 30 points) Complete a working synthesized CPU design. You need to demonstrate your design is working through the same testbenches for pre-synthesis design.

// *Reference* //

☞SMILE CPU has the following instruction format (Andes ISA)

The Andes 32bit instruction formats and the meaning of each filed are described below:

- Type-0 Instruction Format

0	Opc_6	{sub_1, imm_24}

- Type-1 Instruction Format

0	Opc_6	rt_5	imm_20	
0	Opc_6	rt_5	sub_4	imm_16

- Type-2 Instruction Format

0	Opc_6	rt_5	ra_5	imm_15
0	Opc_6	rt_5	ra_5	{sub_1, imm_14}

- Type-3 Instruction Format

0	Opc_6	rt_5	ra_5	rb_5	sub_10
0	Opc_6	rt_5	ra_5	imm_5	sub_10

- Type-4 Instruction Format

0	Opc_6	rt_5	ra_5	rb_5	rd_5	sub_5
0	Opc_6	rt_5	ra_5	imm1_5	imm2_5	sub_5

- opc_6: 6-bit opcode
- rt_5: target register in 5-bit index register set
- ra_5: source register in 5-bit index register set
- rb_5: source register in 5-bit index register set
- rd_5: destination register in 5-bit index register set
- sub_10: 10-bit sub-opcode
- sub_5: 5-bit sub-opcode
- sub_4: 4-bit sub-opcode
- sub_1: 1-bit sub-opcode
- imm_24: 24-bit immediate value, for unconditional jump instructions (J, JAL). The
- immediate value is used as the lower 24-bit offset of same 32MB memory block (new
- PC[31:0] = {current PC[31:25], imm_24, 1'b0}
- imm_20: 20-bit immediate value. Sign-extended to 32-bit for MOVI operations.
- imm_16: signed PC relative address displacement for branch instructions.
- imm_15: 15-bit immediate value. Zero extended to 32-bit for unsigned operations,
- while sign extended to 32-bit for signed operations.
- imm_14: signed PC relative address displacement for branch instructions.
- imm_5, imm1_5, imm2_5: 5-bit unsigned count value or index value

//