

HOMEWORK I (Part I)

Due day: midnight Oct. 7 (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

HOMEWORK I (Part I)

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

1. (20 points) Write and verify Verilog for a 4-bit ripple-carry style adder/subtractor using basic gates with the following specification:

- a. Number format: 2's complement
- b. Inputs: A (a1, a2, a3, a4), B(b1, b2, b3, b4), addsub
- c. Outputs: S(s1, s2, s3, s4), cout, ovud
- d. (a4, b4 and s4) are the most significant bits for A, B and S, respectively.
- e. If addsub = 0, A+B will be performed, otherwise, A-B will executed.
- f. If the result is out of range, ou_vd will be flagged to logic 1, otherwise, 0.
- g. Hierarchy design is permitted, such as half_adder and full_adder may be used, however, they must be specified.
- h. Proper explanation of your design is required for full credits.
- i. A figure (block diagram with logic gates) shall be draw to depict your design in the end.
- j. Verify your code with a testbench. This testbench needs to show all major situations for overflow plus other six situations.

PS:1. If it is signed numbers, then the 4'b1000=-8

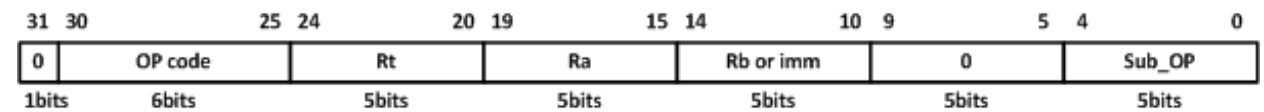
2. If it is unsigned numbers, then the 4'b1000=8

3. In your report you need indicated operations of signed or unsigned

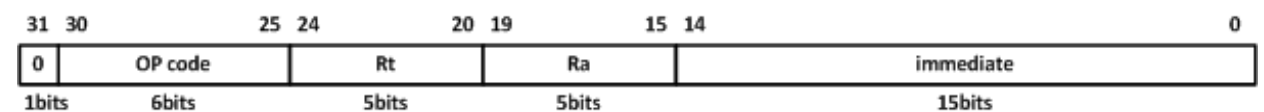
HOMEWORK I (Part I)

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

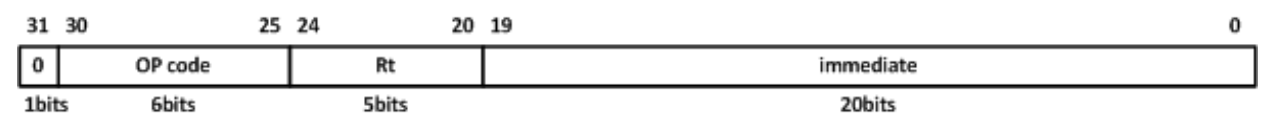
☞ Data-processing (SE:sign-extended, ZE:zero-extended)



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	$Rt = Ra \wedge Rb$
100000	SRLI	\$Rt	\$Ra	imm	01001	Shift right : $Rt = Ra \gg imm$
100000	SLLI	\$Rt	\$Ra	imm	01000	Shift left : $Rt = Ra \ll imm$
100000	ROTRI	\$Rt	\$Ra	imm	01011	Rotate right : $Rt = Ra \gg imm$



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	$Rt = Ra \wedge ZE(imm)$



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

HOMEWORK I (Part I)

2. (30 points) Write and verify a 32-bit ALU that could perform the following arithmetic and logic operations:

- Logic operations: AND, OR, XOR, NOP
- Arithmetic operations: ADD, SUB
- Shift & Rotate operations: SLLI (Shift Left Immediate), SRLI (Shift Right Immediate) and ROTRI (Rotate Right Immediate)

The ALU unit shown in Fig. 2 has two data inputs (SCR1 and SCR2) and one data output (ALU_result) along with one output flag, ALU_Overflow. Controlling signals are *opcode*, *sub_opcode* and *Enable_execute*. *opcode* and *sub_opcode* will be used to select which operation to be executed

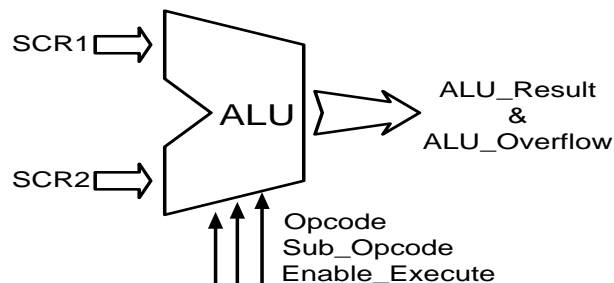


Fig. 2 Conceptual Diagram of the ALU

This ALU consists of three different combinational circuit building blocks:

- Add/Sub to perform addition and subtraction arithmetic operations
- Logic block to perform logical operations
- Barrel Shifter to perform shift and rotate operations

HOMEWORK I (Part I)

Its major portion of the Verilog may look like as follows.

Add/Sub block (add sub module in reference code1 for ALU):

```
assign result=(control == 0)?(operand2+operand1):(operand2-operand1);
```

Logic block

```
and_result=src2&src1;
or_result=src2|src1;
xor_result=src2^src1;
```

Shift block (Barrel Shifter module in reference code1 for ALU):

```
output [31:0] result; // result operand
reg [95:0] rotate_reg;
input [31:0] in1,in2; // input operand; in1: # of shifts; in2: original data
rotate_reg={in2,in2,in2};
```

✦ Rotate Operation:

```
rotate_reg=(direction)?rotate_reg>>in1:rotate_reg<<in1;
result=rotate_reg[63:32];
```

✦ Shift Operation:

```
result=(direction)?in2>>in1:in2<<in1;
```

Reference code for ALU

```
module alu(alu_result,alu_overflow,scr1,scr2,opcode,sub_opcode,enable_execute,reset);
    parameter NOP=5'b01001,ADD=5'b00000,SUB=5'b00001,AND=5'b00010,
            OR=5'b00100,XOR=5'b00011,SRLI=5'b01001,SLLI=5'b01000,
            ROTRI=5'b01011;
    output reg [31:0]alu_result;
    output reg alu_overflow;

    input [31:0]scr1,scr2;
    input [5:0]opcode;
    input [4:0]sub_opcode;
    input reset;
    input enable_execute;

    reg [63:0]rotate;
    reg a,b;

    always @ ( * )begin
        if(reset)begin
            alu_result=32'b0;
            alu_overflow=1'b0;
        end
        else if(enable_execute)begin
            case(opcode)
                6'b100000 : case (sub_opcode)
                    NOP : begin
                        alu_result=32'b0;
                        alu_overflow=1'b0;
                    end
                    ADD : begin
                        {a,alu_result[30:0]}=scr1[30:0]+scr2[30:0];
                        {b,alu_result[31]}=scr1[31]+scr2[31]+a;
                        alu_overflow=a ^ b;
                    end
                    SUB : begin
                        {a,alu_result[30:0]}=scr1[30:0]-scr2[30:0];
                        {b,alu_result[31]}=scr1[31]-scr2[31]-a;
                        alu_overflow=a ^ b;
                    end
                endcase
            endcase
        end
    end
```

HOMEWORK I (Part I)

```
AND    : begin
        alu_overflow=1'b0;
        alu_result=scr1&scr2;
    end
OR      : begin
        alu_overflow=1'b0;
        alu_result=scr1|scr2;
    end
XOR     : begin
        alu_overflow=1'b0;
        alu_result=scr1^scr2;
    end
SRLI    : begin
        alu_overflow=1'b0;
        alu_result=scr1>>scr2;
    end
SLLI    : begin
        alu_overflow=1'b0;
        alu_result=scr1<<scr2;
    end
ROTRI   : begin
        alu_overflow=1'b0;
        rotate={scr1,scr1}>>scr2;
        alu_result=rotate[31:0];
    end
default : begin
        alu_overflow=1'b0;
        alu_result=32'b0;
    end
endcase

6'b101000 : begin
    {a,alu_result[30:0]}=scr1[30:0]+scr2[30:0];
    {b,alu_result[31]}=scr1[31]+scr2[31]+a;
    alu_overflow=a ^ b;
end
6'b101100 : begin
    alu_overflow=1'b0;
    alu_result=scr1|scr2;
end
6'b101011 : begin
    alu_overflow=1'b0;
    alu_result=scr1^scr2;
end
default : begin
    alu_overflow=1'b0;
    alu_result=32'b0;
end
endcase
end
else begin
    alu_result=32'b0;
    alu_overflow=1'b0;
end
end
endmodule
```

HOMEWORK I (Part I)

//////////////////////////////////// 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

////////////////////////////////////