VLSI System Design (Graduate Level)
Fall 2012

HOMEWORK II-Part I

` Due day: 9:00am Nov. 8 (Thursday), 2012

This homework is to let you familiar with Verilog and SMILE CPU. It includes the second part of a simplified multi-cycle **(SMILE)** CPU with ***17*** instructions. You will complete the CPU. It shall be noted that some problems provide skeleton codes. 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: n09299992_蕭育書_hw2pt1_report.doc**
4. All homework requirements should be upload in this file hierarchy. Your file name/folder name is labeled in color red, specifications in color black.

N2601xxxx (Your student ID number)(Folder)
- N2601xxxx.doc (Your homework report)
- P1(Folder for Problem 1)
  - top.v (top module, use "`include" to include all files related EXCEPT IM)
  - top_tb.v (testbench, use "`include" to include IM)
  - Any other Verilog files for your design
  - Any other files to support your testbench
- P2(Folder for Problem 2)
  - top.v (top module, use "`include" to include all files related EXCEPT IM,DM)
  - top_tb.v (testbench ,use "`include" to include IM and DM)
  - Any other Verilog files for your design
  - Any other files to support your testbench
- P3(Folder for Problem 3)
  - top.v (top module, use "`include" to include all files related EXCEPT IM,DM)
  - top_tb.v (testbench ,use "`include" to include IM and DM)
  - Any other Verilog files for your design
  - Any other files to support your testbench
- P4(Folder for Problem 4)
  - top.v (top module, use "`include" to include all files related EXCEPT IM,DM,ROM and MEMORY)
  - top_tb.v (testbench ,use "`include" to include IM, DM,ROM and MEMORY)
  - Any other Verilog files for your design
  - Any other files to support your testbench

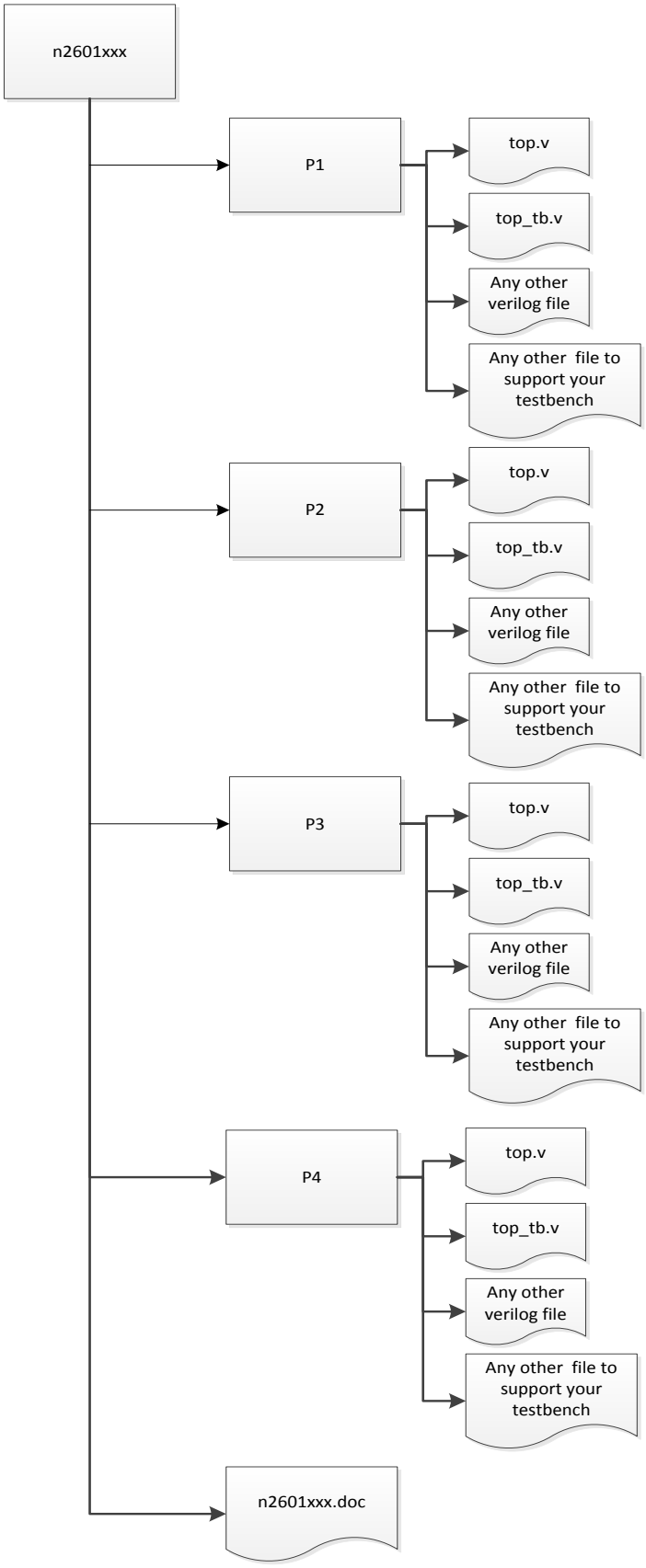


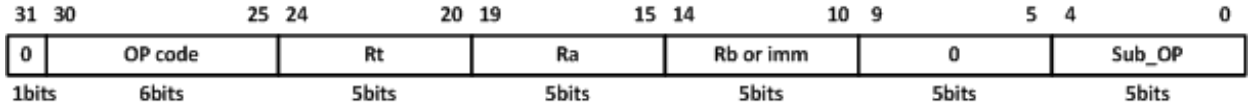
Fig.1 File hierarchy for Homework submission

Exercise

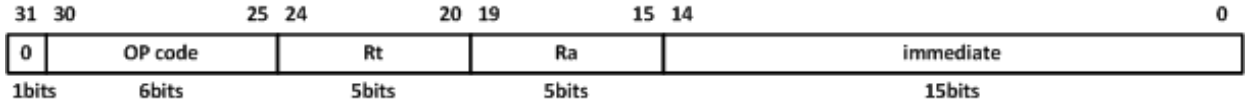
SMILE CPU has the following instruction format

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

*** Note that NOP and SRLI has the same OP + sub OP. This is durable. If you exam their DST, SRC1 and SRC2, you will be able to find their differences and find a proper way to implement.

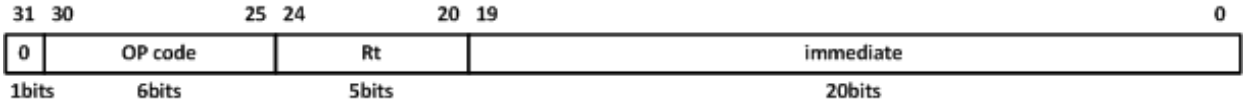


OP code	Mnemonics	DST	SRC1	SRC2	Sub OP	Description
100000	NOP	00000	00000	00000	01001	No operation
100000	ADD	$Rt	$Ra	$Rb	00000	Rt = Ra + Rb
100000	SUB	$Rt	$Ra	$Rb	00001	Rt = Ra - Rb
100000	AND	$Rt	$Ra	$Rb	00010	Rt = Ra & Rb
100000	OR	$Rt	$Ra	$Rb	00100	Rt = Ra | Rb
100000	XOR	$Rt	$Ra	$Rb	00011	Rt = Ra ^ 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



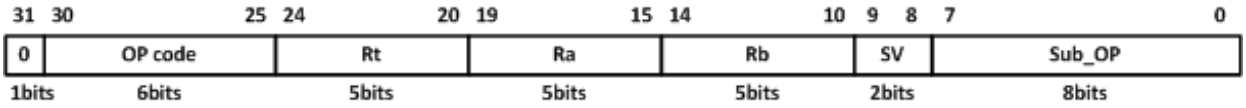
OP code	Mnemonics	DST	SRC1	SRC2	Description
101000	ADDI	$Rt	$Ra	imm	Rt = Ra + SE(imm)
101100	ORI	$Rt	$Ra	imm	Rt = Ra | ZE(imm)
101011	XORI	$Rt	$Ra	imm	Rt = Ra ^ ZE(imm)
000010	LWI	$Rt	$Ra	imm	Rt = Mem[Ra + (imm << 2)]
001010	SWI	$Rt	$Ra	imm	Mem[Ra + (imm << 2)] = Rt

HOMEWORK II-Part I



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

☞Load and store



OP code	Mnemonics	DST	SRC1	SRC2	Sub OP	Description
011100	LW	$Rt	$Ra	$Rb	00000010	Rt = Mem[Ra + (Rb << sv)]
011100	SW	$Rt	$Ra	$Rb	00001010	Mem[Ra + (Rb << sv)] = Rt

1. **(20/250) Add in a 2^{10}x32bit instruction memory as shown in Figure 1-1 into the CPU system that designed in HW1-Part2-P4.**
   a. **Redesign your CPU with the revised controller and add-on components, such as a program counter, or multiplexers etc. so that the modified CPU system could read the sequence of the machine codes from the instruction memory. NOTE that the instruction memory shall be separated from the kernel since the memory will NOT be synthesized later.**
   b. **Please load the following machine code shown in Figure 1-2 into the instruction memory and verify the correctness of the corresponding operations.**
   c. **Devise another two verification programs by yourself with at least 20 lines each and use them to verify your CPU.**
   d. **Modify your code to comply with nLint within 70% of your code, i.e., the number of errors & warnings in total shall not exceed 30% of the number of lines in your code.**

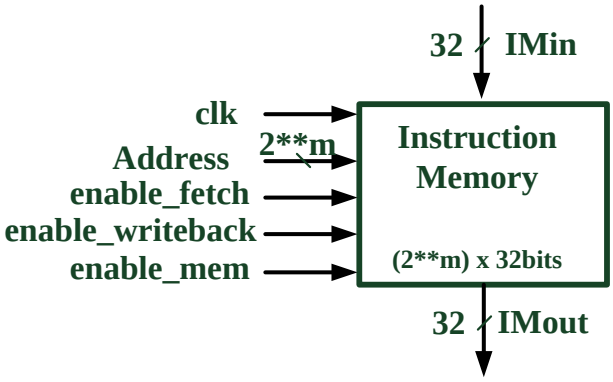


Figure 1-1 Conceptual Diagram of Instruction Memory

```
0_100010_00000_00000000000000000100     //MOVI   R0=20'd4
0_101000_00000_00000_000000000001101    //ADDI   R0=R0 + 4'b1101
0_101100_00001_00000_000000000000010    //ORI    R1=R0 | 4'b0010
0_101011_00001_00001_000000000000111    //XORI   R1=R1 ^ 4'b0111
0_100000_00000_00000_00000_00000_01001  //NOP
0_100000_00001_00000_00001_00000_00000  //ADD    R1=R0 + R1
0_100000_00001_00001_00000_00000_00001  //SUB    R1=R1 - R0
0_100000_00001_00001_00000_00000_00010  //AND    R1=R1 & R0
0_100000_00000_00000_00001_00000_00100  //OR     R0=R0 | R1
0_100000_00010_00000_00001_00000_00011  //XOR    R2=R0 ^ R1
0_100000_00010_00000_00011_00000_01001  //SRLI   R2=R0 SRL(3)
0_100000_00010_00010_00011_00000_01000  //SLLI   R2=R2 SLL(3)
0_100000_00010_00000_11101_00000_01011  //ROTRI  R2=R0 ROTR(29)
```

Figure 1-2

You can load the machine code by the following procedures:

- Added the following block into the testbench to load mins.prog into Instruction memory
  - initial
  - begin : prog_load
    $readmemb("mins.prog",u_memory_Instruction.mem_data1);
  - end
- The system task $readmemb (reads in binary) and $readmemh (reads in hex) will read in a file.

- It takes 2 arguments, the file name and the memory structure name. It reads the file into the memory structure.

Skeleton code for instruction memory :

```
module IM(clk, rst, IM_address, enable_fetch, enable_write, enable_mem, IMin, IMout);

parameter data_size=32;
parameter mem_size=1024;

input clk, rst, enable_fetch, enable_write, enable_mem;
input [9:0]IM_address;
input [data_size-1:0]IMin;


output [data_size-1:0]IMout;

reg [data_size-1:0]IMout;
reg [data_size-1:0]mem_data[mem_size-1:0];

integer i;

always@(posedge clk)
begin
  if(rst)begin
    for(i=0;i<mem_size;i=i+1)
    mem_data[i]<=0;
    IMout<=0;
  end
  else if(enable_mem)begin
    if(enable_fetch)begin
      IMout<=mem_data[IM_address];
    end
    else if(enable_write)begin
      mem_data[IM_address] <= IMin;
    end
  end
end

endmodule
```

Report Requirements

a. Proper explanation of your design is required for full credits.

b. A figure (block diagram with logic gates) shall be draw to depict your design in the end.

c. Verify your code with testbenches other than designated ones. These testbenches need to show all major situations. Show your snapshot of waveform for different cases in your reports and illustrate the correctness of your results.

d. Report the number of lines of your code, the final results of running nLint and 3~5 most frequent warning/errors in your code. Describe how you modify your code to comply with the nLint.

2. (20/250) Add in a 2^{12}x32bit data memory as shown in Figure 2-1 into the CPU system designed in HW2_P1.

a. Redesign your CPU with the revised controller and any feasible add-on components so that the modified CPU system could read the sequence of the machine codes from the instruction memory and access/store data in the data memory. NOTE that the data memory shall be separated from the kernel since the memory will NOT be synthesized later.

b. Please load the mins.prog as shown in Figure 2-2 (you have to convert it to machine code first by yourself) in the next page into the instruction memory and verify the correctness of your code.

c. Devise another two verification programs by yourself with at least 20 lines each and use them to verify your CPU.

d. Modify your code to comply with nLint within 80% of your code, i.e., the number of errors & warnings in total shall not exceed 20% of the number of lines in your code.

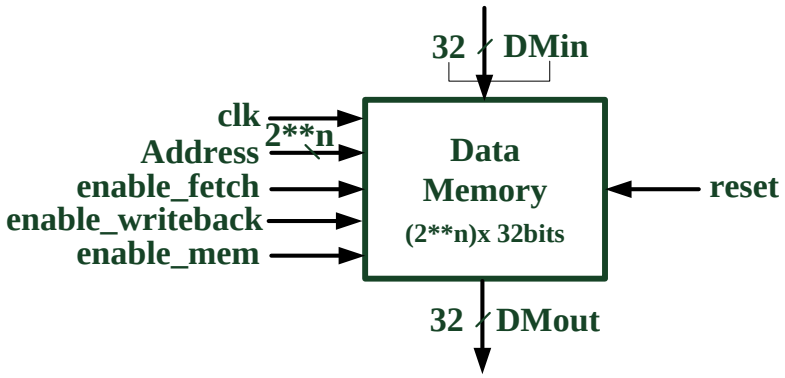


Figure 2-1 Conceptual Diagram of Data Memory

Skeleton code for data memory :

```
module DM(clk, rst, enable_fetch, enable_writeback, enable_mem, DMin, DMout, DM_address);

parameter data_size=32;
parameter mem_size=4096;

input clk;
input rst;
input enable_fetch;
input enable_writeback;
input enable_mem;
input [data_size-1:0]DMin;
input [11:0]DM_address;
output [data_size-1:0]DMout;
reg [data_size-1:0]DMout;
reg [data_size-1:0]mem_data[mem_size-1:0];

integer i;
always@(posedge clk)
begin
  if(rst)begin
    for(i=0;i<mem_size;i=i+1)
      mem_data[i]<=0;
      DMout<=0;
  end
  else if(enable_mem==1)begin
    if(enable_fetch==1)
      DMout<=mem_data[DM_address];
    else if(enable_writeback==1)
      mem_data[DM_address]<=DMin;
  end
end
endmodule
```

mins.prog content :

★0.	ADDI	R1=R1+ 4'b1001
★1.	XORI	R1=R1^ 4'b1010
★2.	MOVI	R0=20'd3
★3.	SW	M0=R0
★4.	ORI	R0=R0 | 4'b0100
★5.	AND	R1=R1 & R0
★6.	LW	R0=M0
★7.	NOP	
★8.	ADD	R1=R0 + R1
★9.	OR	R1=R1 | R0
★10.	SUB	R1=R1 – R0
★11.	SW	M19=R1
★12.	SRLI	R2=R0 SRL(1)
★13.	SLLI	R2=R2 SLL(3)
★14.	LW	R1=M23
★15.	AND	R1=R1 & R3
★16.	SW	M35=R2

Figure 2-2

3. (45/250) Add performance counters into the CPU system designed in HW2_P2. You MUST follow the module specifications in Table 3-1.

A real processor usually uses performance counter to tack events within the processor and to understand its performance as well as status. The performance counter is usually update during the writeback stage. SMILE will have two counters.

- **Cycle count (128 bits)– the number of cycles since the processor was last reset. The cycle count is incremented every cycle.**
- **Instruction count (64 bits) – the number of actual instructions executed since the processor was last reset. The instruction count is incremented every instruction.**

a. Redesign your CPU with the revised controller and any feasible add-on components so that the modified CPU system could update cycle count and instruction count.

b. Revise testbenches that perform verification programs in HW2-P1.2 and write another verification program with highly data dependency and massive data memory load/store, such as the program shown below. ALL verification programs MUST be longer than 20 lines, then use the verification programs to verify your CPU . The testbenches MUST be able to show the results as PASS or FAIL , and print DEBUG information when errors occur. At the end of execution, please show the information of performance counters, including instruction count and cycle count.

★1.	LW	R1=M1
★2.	LW	R2=M2
★3.	AND	R1=R1 & R2
★4.	SW	M2=R1
★5.	.....	

c. Modify your code to comply with nLint within 80% of your code, *i.e.*, the number of errors & warnings in total shall not exceed 20% of the number of lines in your code.

d. The testbenches and verification programs will be used to verify designs of other classmates. You will gain more credits if your testbench is able

to detect errors for designs by other classmates. Note that this part will be performed by TAs.

Table 3-1

Module Type	**Specifications**		
Top Module	module name	top(…)	
	sub-module name	Regfile regfile1(…)	
	system signal	Input	clk rst
	connect with IM	Input	instruction
		output	IM_read IM_write IM_enable IM_address
	connect with DM	Input	DM_out
		output	DM_read DM_write DM_enable DM_in DM_address
	for performance counter	output	Cycle_cnt Ins_cnt
Register file	The internal variable for storing data of your register file module MUST be declaired as “reg [31:0] rw_reg [31:0]”		

4. (45/250) Add one ROM and initialization routines to HW2_P3.

In an embedded system, when the reset signal is asserted, the ROM will gain control over the processor and behave as a booting program. The ROM will first initialize the hardware, especially the memory subsystem, and load an OS from an external storage (here, the external storage is mimicked by a file).

In this problem, the ROM will initialize instruction memory (IM) and data memory (DM). Then the ROM will load the instructions from the external storage (Memory) to IM address 0x080. Next, the ROM will return the control to the processor controller. The processor will execute instruction starting from the IM 0x80. [Hint: ROM is still one kind of memory learned earlier. It can be set to read data (or commands) out sequentially once activated and stop at certain position. An example of ROM is provided with the problem.]

NOTE that ROM and MEMORY shall be separated from the kernel since the memory will NOT be synthesized later.

You MUST follow the module specifications in Table 4-1 to define the module.

a. Redesign your CPU so that ROM is connected by a straightforward way and could perform initialization as described.

b. Revise testbenches that perform verification programs in HW2-P1.3, then use the verification programs to verify your CPU . The testbenches MUST be able to show the results as PASS or FAIL , and print DEBUG information when errors occur. At the end of execution, please show the information of performance counters, including instruction count and cycle count.

c. Modify your code to comply with nLint within 80% of your code, *i.e.*, the number of errors & warnings in total shall not exceed 20% of the number of lines in your code.

d. The testbenches and verification programs will be used to verify designs of other classmates. You will gain more credits if your testbench is able to detect more errors for designs by other classmates. Note that this part will be performed by TAs.

HOMEWORK II-Part I

Table 4-1

Module Type	**Specifications**		
Top Module	module name	top(…)	
	sub-module name	Regfile regfile1(…)	
	system signal	input	clk rst system_enable
	connect with IM	input	instruction
		output	IM_read IM_write IM_enable IM_address
	connect with DM	Input	DM_out
		output	DM_read DM_write DM_enable DM_in DM_address
	for performance counter	output	Cycle_cnt Ins_cnt
	connect with ROM	input	rom_out
		output	rom_enable rom_read rom_address
	connect with MEMORY	input	MEM_data
		output	MEM_en MEM_read MEM_write MEM_addr
Register file	The internal variable for storing data of your register file module MUST be declaired as “reg [31:0] rw_reg [31:0]”		

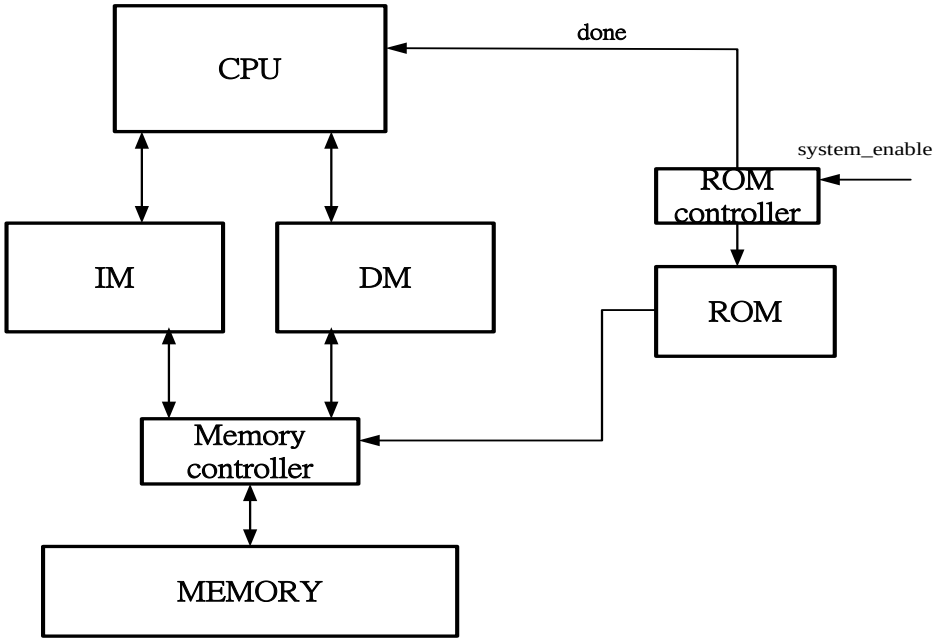


Figure 4-1 Block diagram of a controller

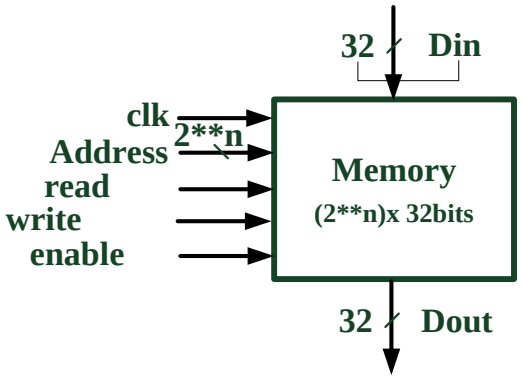


Figure 4-2 Conceptual Diagram of Memory

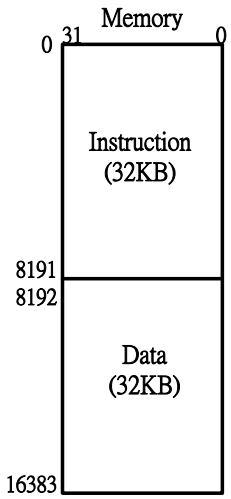


Figure 4-3 memory mapping.

Skeleton code for MEMORY :

```
module MEMORY(clk,rst,enable,read,write,address,Din,Dout);
parameter data_size=32;
parameter mem_size=16384;

input clk, rst, enable, read, write;
input [13:0] address;
input [data_size-1:0] Din;

output [data_size-1:0] Dout;

reg [data_size-1:0] Dout;
reg [data_size-1:0] mem[mem_size-1:0];

integer i;

always@(posedge clk)begin
  if(rst)begin
    for(i=0;i<mem_size;i=i+1)
    mem[i]<=0;
    Dout<=0;
  end
  else if(enable)begin
    if(read)begin
      Dout <=  mem[address];
    end
    else if(write)begin
      mem[address] <=  Din;
    end
  end
end

endmodule
```

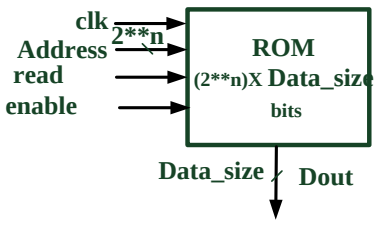
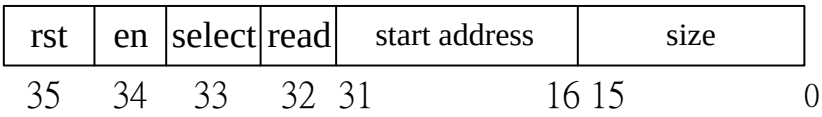


Figure 4-4 Conceptual Diagram of ROM

Table 4-2 Instruction format of ROM



notation	Description
rst	reset signal of system
en	enable signal of system
select	Selection of target 0 :IM 1:DM
read	read signal for memory
start address	start address of memory read
size	read number of words from memory

Skeleton code for ROM :

```
1  module ROM(clk, read,enable,address, dout );
2  parameter data_size=36;
3  parameter mem_size=256;
4
5  input clk, read, enable;
6  input [7:0] address;
7
8  output [data_size-1:0] dout;
9
10 reg [data_size-1:0] dout;
11 reg [data_size-1:0]mem_data[mem_size-1:0];
12
13 always@(posedge clk)begin
14   if(enable)begin
15     if(read)begin
16       dout <=  mem_data[address];
17     end
18   end
19 end
20
21 endmodule
```

HOMEWORK II-Part I

Report Requirements

a. Proper explanation of your design is required for full credits.
b. A figure (block diagram with logic gates) shall be draw to depict your design in the end.
c. Verify your code with testbenches other than the designated ones. These testbenches need to show all major situations. Show your snapshot of waveform for different cases in your reports and illustrate the correctness of your results.
d. Report the number of lines of your code, the final results of running nLint and 3~5 most frequent warning/errors in your code. Describe how you modify your code to comply with the nLint.

Note that it is strongly recommended that you can practice synthesis of your design in HW2-P1.4 with Design Compiler/Vision, it would be very helpful to you for HW2-PartII.

HOMEWORK II-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

//