Computer-Aided VLSI System Design
Homework 2: Simple MIPS CPU

TA: 林奕廷 r09943017@ntu.edu.tw

TA: 羅宇呈 f08943129@ntu.edu.tw

Due Tuesday, Oct. 18, 14:00

Data Preparation

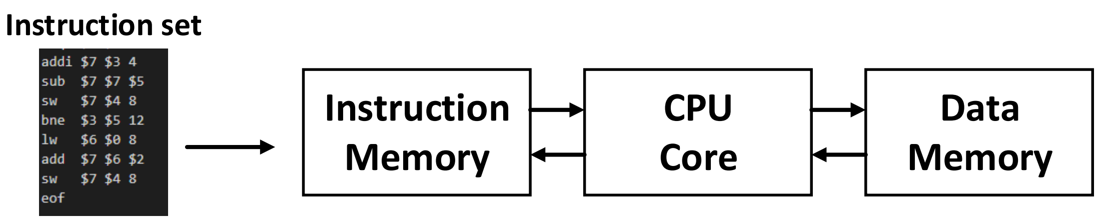
1. Decompress 1111_hw2.tar with following command

```
tar -xvf 1111_hw2.tar
```

Folder	File	Description
00_TESTBED	inst_mem.vp	Module of instruction memory (protected)
	data_mem.vp	Module of data memory (protected)
	define.v	File of definition
	testbed_temp.v	Testbed template
00_TESTBED/PATTERN/p*	inst.dat	Pattern of instructions in binary format
	inst_assembly.dat	Corresponding assembly code of the instruction pattern
	data.dat	Pattern of final data in data memory
	status.dat	Pattern of corresponding status
01_RTL	core.v	Your design
	rtl.f	File list
	01_run	NCVerilog command
	99_clean_up	Command to clean temporary data

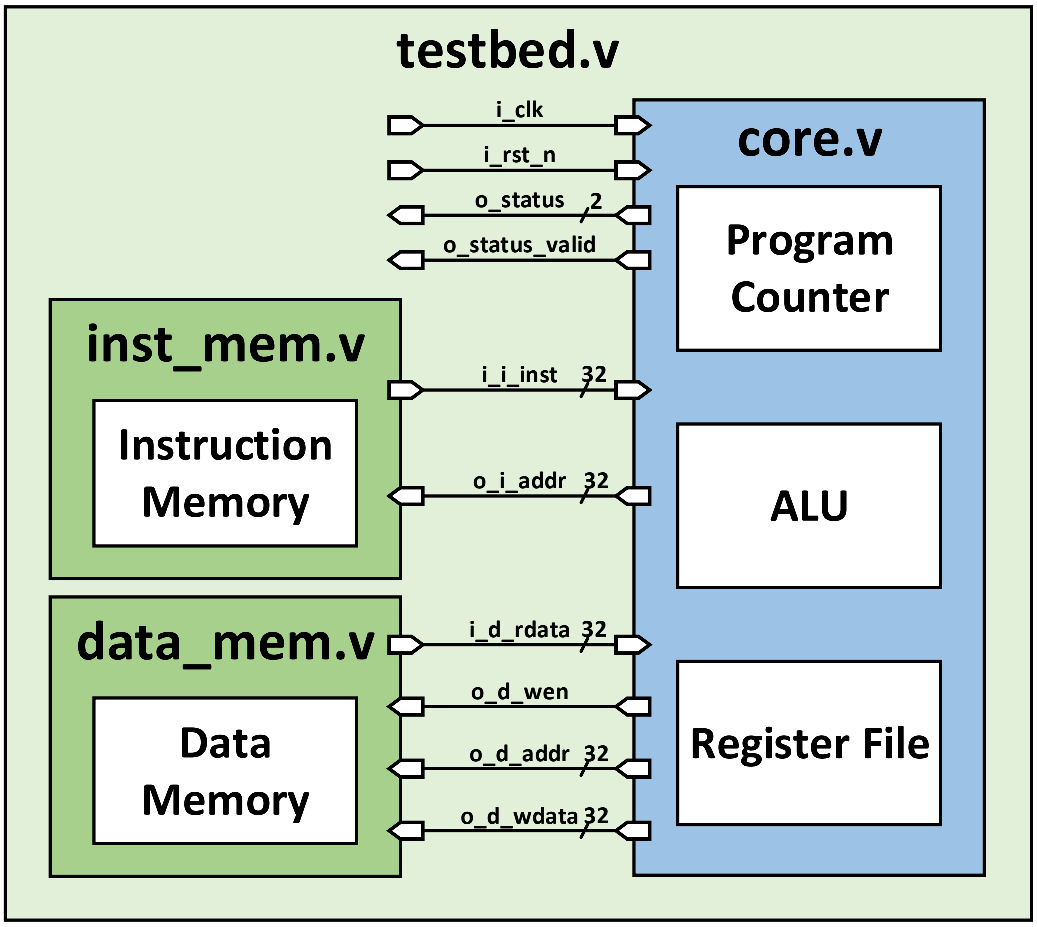
Introduction

Central Processing Unit (CPU) is one of the most important core in a computer system. In this homework, you are asked to design a simple CPU which consists of a program counter, an ALU, and a register file. The instruction set of the simple CPU is similar to MIPS ISA. Since the files of testbed (inst_mem.v, data_mem.v) are protected, you have to write the testbed by our own to test your design.



```
addi $7 $3 4
sub  $7 $7 $5
sw   $7 $4 8
bne  $3 $5 12
lw   $6 $0 8
add  $7 $6 $2
sw   $7 $4 8
eof
```

Block Diagram



Specifications

1. Top module name: core
2. Input/output description:

Signal Name	I/O	Width	Simple Description
i_clk	I	1	Clock signal in the system.
i_rst_n	I	1	Active **low** asynchronous reset.
o_i_addr	O	32	Address from program counter (PC)
i_i_inst	I	32	Instruction from instruction memory
o_d_wen	O	1	Write enable of data memory Set **low** for reading mode, and **high** for writing mode
o_d_addr	O	32	Address for data memory
o_d_wdata	O	32	Data input to data memory
i_d_rdata	I	32	Data output from data memory
o_status	O	2	Status of the core after processing each instruction
o_status_valid	O	1	Set **high** if core is ready to output status

3. All outputs should be synchronized at clock **rising** edge.
4. You should set all your outputs and register files to zero when i_rst_n is **low**. Active

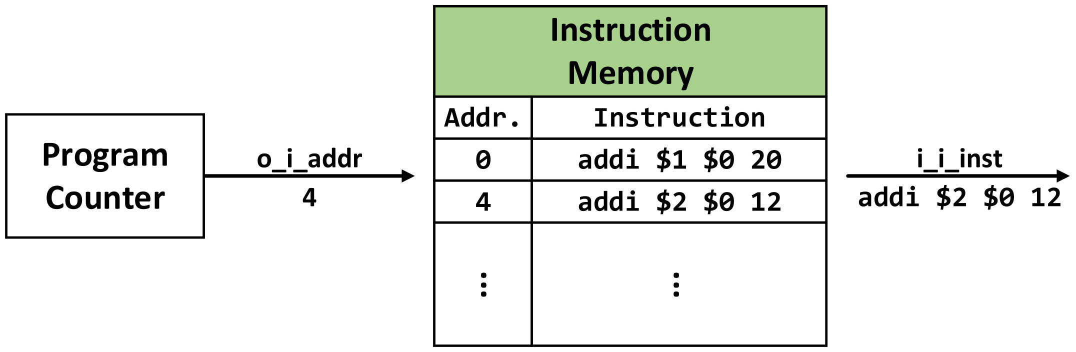
low asynchronous reset is used.

5. Instruction memory and data memory are provided. All values in memory are reset to zero.
6. You should create **32-bit registers** in register file.
7. After sending o_i_addr to instruction memory, the core can receive the corresponding i_i_inst at the next rising edge of the clock.
8. To load data from the data memory, set o_d_wen to **0** and o_d_addr to relative address value. i_d_rdata can be received at the next rising edge of the clock.
9. To save data to the data memory, set o_d_wen to **1**, o_d_addr to relative address value, and o_d_wdata to the written data. At the next rising edge of the clock, the data is written to memory.
10. Your o_status_valid should be set to **high** for only **one cycle** for every o_status.
11. The testbed will get your output at negative clock edge to check the o_status if your o_status_valid is **high**.
12. When you set o_status_valid to **high** and o_status to **3**, stop processing. The testbed will check the data in data memory with golden data.
13. If overflow happened, stop processing and raise o_status_valid to **high** and set o_status to **2**. The testbed will check the data in data memory with golden data.
14. **Less than 1024** instructions are provided for each pattern.
15. The whole processing time can't exceed **120000** cycles.

Design Description

1. Program counter is used to control the address of the instruction memory.

   `$pc = $pc + 4` for every instruction (except for **`beq`**, **`bne`**)



2. Register file contains 32 registers (each register is 32-bit) for operation.

3. Instruction mapping:

   a. **R-type**

[31:26]	[25:21]	[20:16]	[15:11]	[10:0]
opcode	$s2	$s3	$s1	Not used

31 **0**

   b. **I-type**

[31:26]	[25:21]	[20:16]	[15:0]
opcode	$s2	$s1	im

31 **0**

   c. **EOF**

[31:26]	[25:0]
opcode	Not used

31 **0**

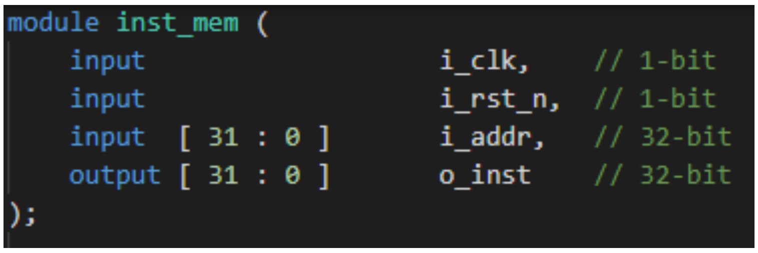
4. The followings are the instructions you need to design for this homework:

Operation	Assembly	Opcode	Type	Meaning	Note
Add	`add`	6'd1	R	`$s1 = $s2 + $s3`	Signed operation
Subtract	`sub`	6'd2	R	`$s1 = $s2 - $s3`	Signed operation
Add unsigned	`addu`	6'd3	R	`$s1 = $s2 + $s3`	Unsigned operation
Subtract unsigned	`subu`	6'd4	R	`$s1 = $s2 - $s3`	Unsigned operation
Add immediate	`addi`	6'd5	I	`$s1 = $s2 + im`	Signed operation
Load word	`lw`	6'd6	I	`$s1 = Mem[$s2 + im]`	Unsigned operation
Store word	`sw`	6'd7	I	`Mem[$s2 + im] = $s1`	Unsigned operation
AND	`and`	6'd8	R	`$s1 = $s2 & $s3`	Bit-wise
OR	`or`	6'd9	R	`$s1 = $s2 | $s3`	Bit-wise
NOR	`nor`	6'd10	R	`$s1 = ~($s2 | $s3)`	Bit-wise
Branch on equal	`beq`	6'd11	I	`if($s1==$s2), $pc = $pc + 4 + im;`	PC-relative; Unsigned

				`else, $pc = $pc + 4`	operation
Branch on not equal	`bne`	6’d12	I	`if($s1!=$s2), $pc = $pc + 4 + im;` `else, $pc = $pc + 4`	PC-relative; Unsigned operation
Set on less than	`slt`	6’d13	R	`if($s2<$s3), $s1 = 1;` `else, $s1 = 0`	Signed operation
End of File	`eof`	6’d14	EOF	**`Stop processing`**	Last instruction in the pattern

Note: Use two’s complement arithmetic for signed operations.

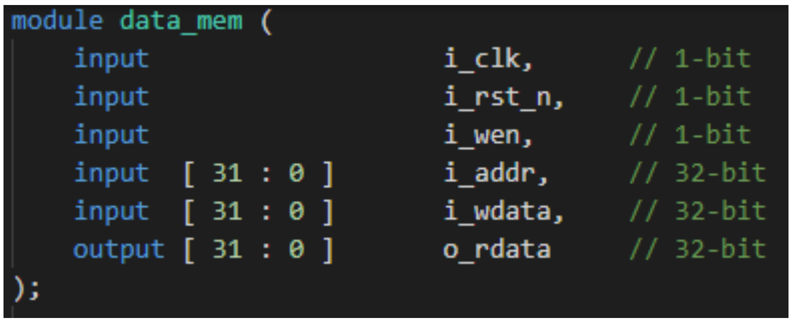
5. Interface of instruction memory (size: 1024×32 bit)
   - i_add[11:2] for address mapping in instruction memory.



```
module inst_mem (
    input                   i_clk,    // 1-bit
    input                   i_rst_n,  // 1-bit
    input  [ 31 : 0 ]       i_addr,   // 32-bit
    output [ 31 : 0 ]       o_inst    // 32-bit
);
```

6. Interface of data memory (size: 64×32 bit)
   - i_add[7:2] for address mapping in data memory,
   - To fetch data of data memory in your **testbed** (not in your RTL code!), use the following instance name.

```
u_data_mem.mem_r[i]
```



```
module data_mem (
    input                   i_clk,     // 1-bit
    input                   i_rst_n,   // 1-bit
    input                   i_wen,     // 1-bit
    input  [ 31 : 0 ]       i_addr,    // 32-bit
    input  [ 31 : 0 ]       i_wdata,   // 32-bit
    output [ 31 : 0 ]       o_rdata    // 32-bit
);
```

7. Overflow may happen
   - **Situation1**: Overflow happens at arithmetic instructions (`add`, `sub`, `addu`, `subu`, `addi`).
   - **Situation2**: The address of data/instruction memory is out of the memory size (Do not consider the case if instruction address is beyond eof, but the address mapping is still in the size of instruction memory).

   Once an overflow happens, the testbed stops and checks the data memory.

8. 4 types of o_status:

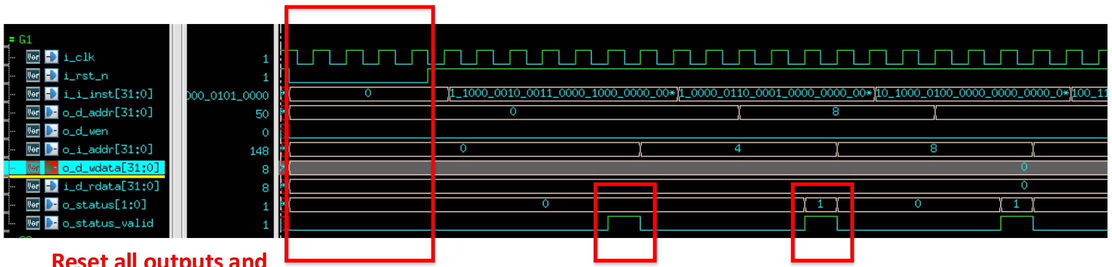
o_status[1:0]	Definition
2’d0	R_TYPE_SUCCESS
2’d1	I_TYPE_SUCCESS
2’d2	MIPS_OVERFLOW
2’d3	MIPS_END

9. Last instruction would be `eof` for every pattern.

10. There is no unknown opcode in the pattern.

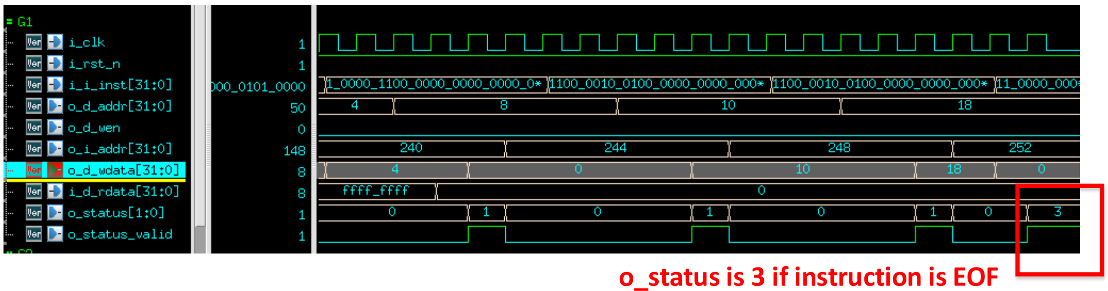
Sample Waveform

1. Status check

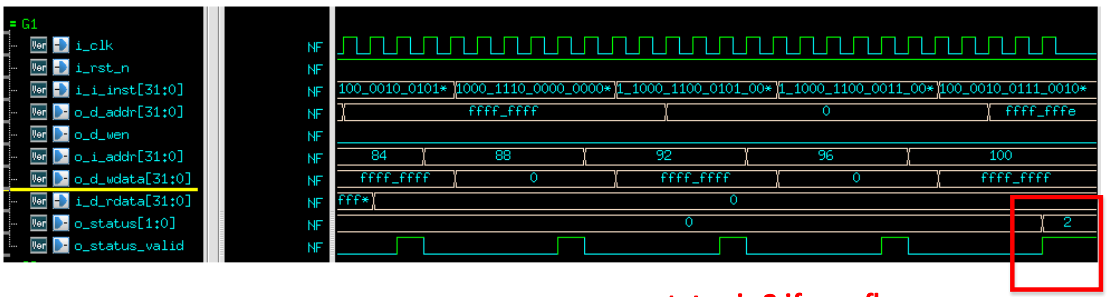


Reset all outputs and register file to 0

o_status is 0 if R-type instruction success,
o_status is 1 if I-type instruction success

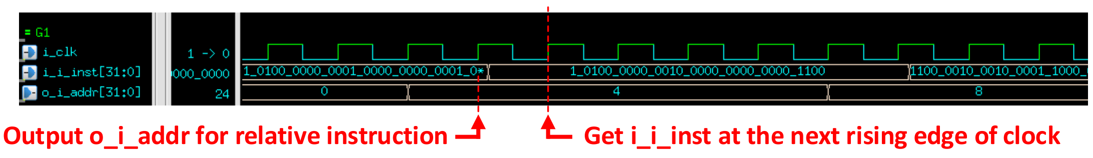


o_status is 3 if instruction is EOF

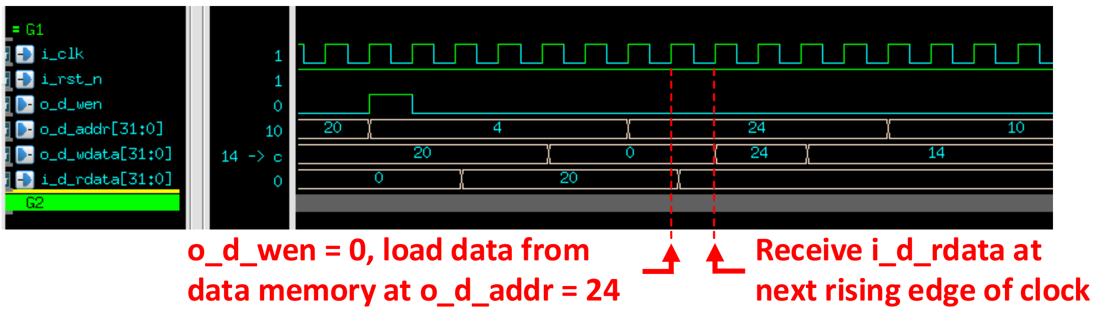


o_status is 2 if overflow occur

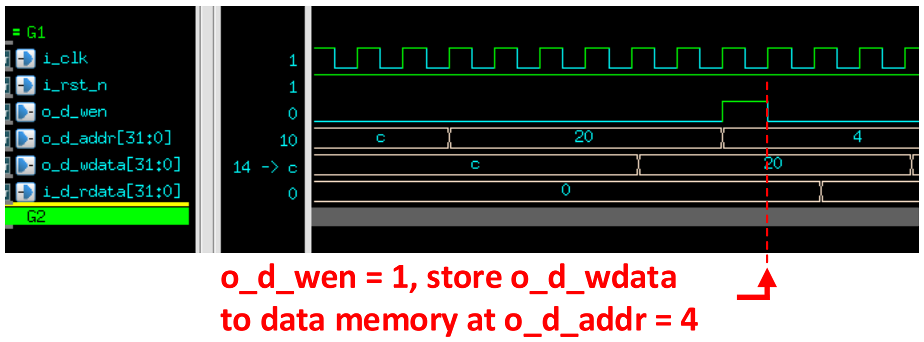
2. Read instruction from instruction memory



3. Load data from data memory


i_clk
i_rst_n
o_d_wen
o_d_addr[31:0]
o_d_wdata[31:0]
i_d_rdata[31:0]
o_d_wen = 0, load data from data memory at o_d_addr = 24
Receive i_d_rdata at next rising edge of clock

4. Save data to data memory



Testbed

1. Things to add in your testbed
   - Clock
   - Reset
   - Waveform file (.fsdb)
   - Function test
   - …

Submission

1. Create a folder named **studentID_hw2**, and put all below files into the folder:
   - **rtl.f** (your file list)
   - **core.v**
   - **all other design files** in your file list (optional)

   Note: Use **lower case** for the letter in your student ID. (Ex. r09943017_hw2)

2. Compress the folder **studentID_hw2** in a **tar file** named **studentID_hw2_v*k*.tar** **(*k* is the number of version, *k* =1,2,…)**

```
tar -cvf studentID_hw2_vk.tar studentID_hw2
```

TA will only check the last version of your homework.
Note: Use **lower case** for the letter in your student ID. (Ex. r09943017_hw2_v1)

3. Submit to NTU COOL

Grading Policy

1. TA will run your code with following format of command. Make sure to run this command with no error message.

```
ncverilog -f rtl.f +define+p0 +access+rw
```

2. Pass the patterns to get full score.
   - Provided pattern: **80%** (patterns: p0, p1)
     - **40%** for each pattern (data in data memory: **20%**, status check: **20%**)
     - **Don't implement the answers in your design directly!**
   - Hidden pattern: **20%** (20 patterns in total)
     - **1%** for each pattern (data & status both correct)
3. Delay submission:
   - In one day: **(original score)*0.6**
   - In two days: **(original score)*0.3**
   - More than two days: **0 point** for this homework
   - Lose **3 point** for any wrong naming rule

Hint

1. Design your FSM with following states
   - Idle
   - Instruction Fetching
   - Instruction decoding
   - ALU computing/Load data
   - Data write-back
   - Next PC generation
   - End of processing