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

TA：周子皓 r11943133@ntu.edu.tw **Due Tuesday, Oct. 17, 13:59**

Data Preparation

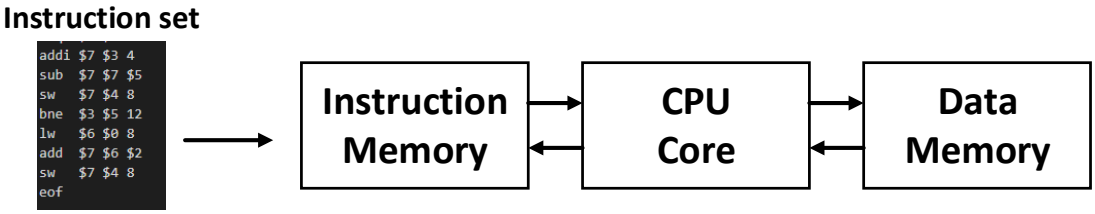
1. Decompress 1121_hw2.tar with following command

```
tar -xvf 1121_hw2.tar
```

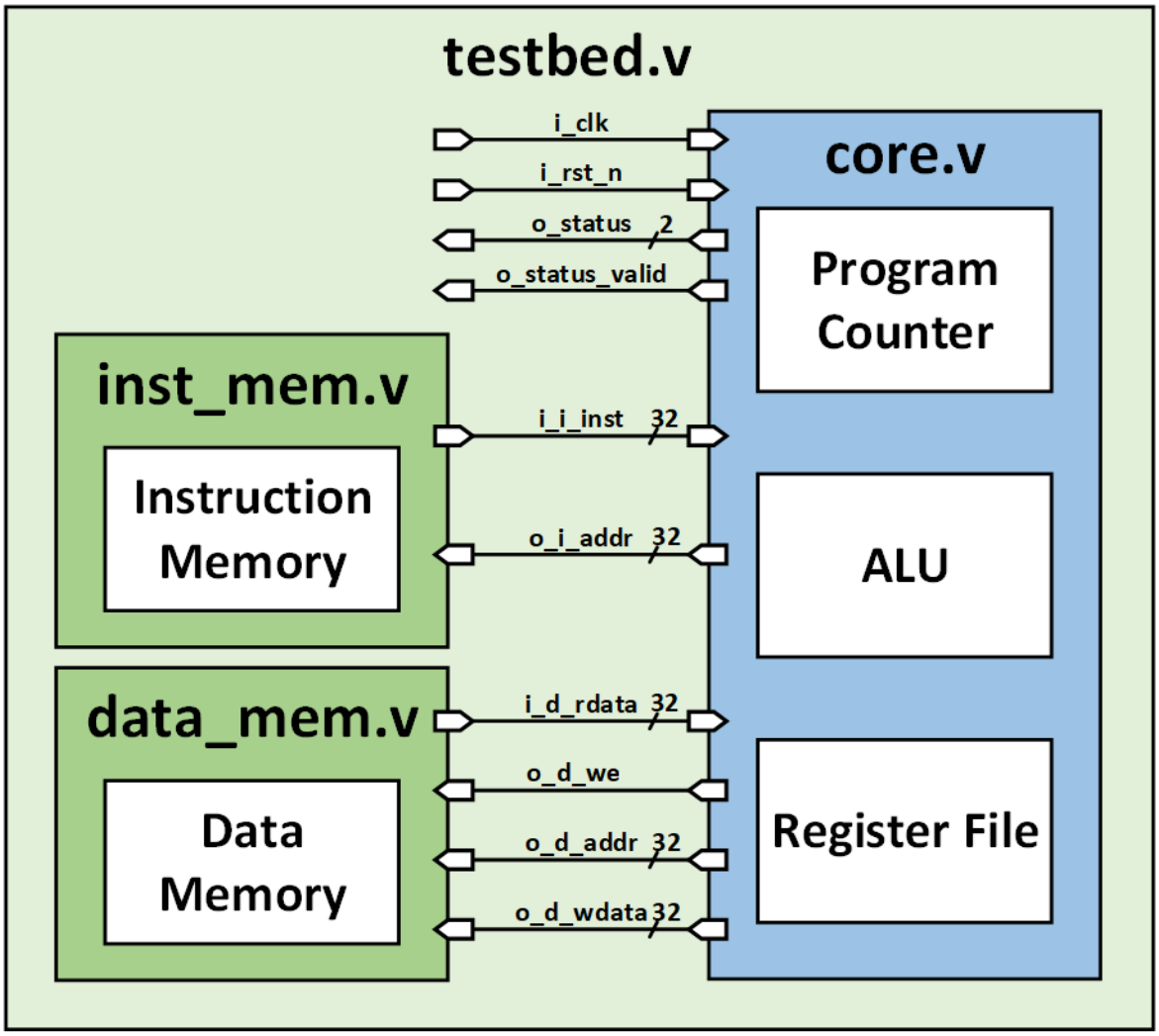
Folder	File	Description
00_TESTBED	inst_mem.vp	Module of instruction memory (protected)
	data_mem.vp	Module of data memory (protected)
	define.v	File of definition
	testbed_temp.v	Testbench template
00_TESTBED/ PATTERN/p*	inst.dat	Pattern of instruction in binary format
	inst_assemble.dat	Corresponding 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
	flist.v	File list (for spyglass check)
	lint.tcl	Spyglass script
	01_run	VCS command
	02_lint	Spyglass command
	99_clean	Command to clean temporary data

Introduction

Central Processing Unit (CPU) is the important core in the computer system. In this homework, you are asked to design a simple MIPS CPU, which contains the basic module of program counter, ALU and register files. The instruction set of the simple CPU is similar to MIPS structure. Since the files of testbench (testbed.v, inst_mem.v, data_mem.v) are protected, you also need to design the testbench to test your design.



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)

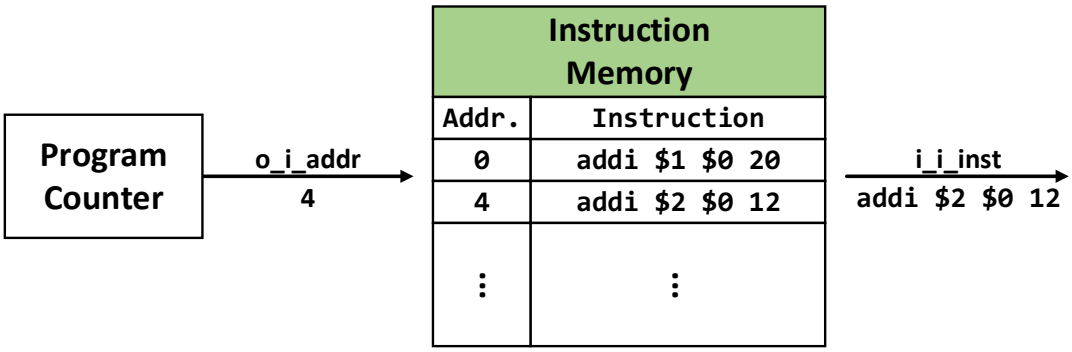
i_i_inst	I	32	Instruction from instruction memory
o_d_we	O	1	Write enable of data memory Set **low** for reading mode, and **high** for writing mode
o_d_addr	O	32	Address for data memory
o_d_wdata	O	32	Unsigned data input to data memory
i_d_rdata	I	32	Unsigned data output from data memory
o_status	O	2	Status of core processing to each instruction
o_status_valid	O	1	Set **high** if ready to output status

3. All outputs should be synchronized at clock **rising** edge.
4. You should set all your outputs and register file to be 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 be zero.
6. You should create **32 unsigned 32-bit registers** in register file.
7. After outputting 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_we 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_we to **1**, o_d_addr to relative address value, and o_d_wdata to the written data.
10. Your o_status_valid should be turned to **high** for only **one cycle** for every o_status.
11. The testbench 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 testbench will check your data memory value with golden data.
13. If overflow happened, stop processing and raise o_status_valid to **high** and set o_status to **2**. The testbench will check your data memory value with golden data.
14. **Less than 1024** instructions are provided for each pattern.
15. The whole processing time can't exceed **120000** cycles.

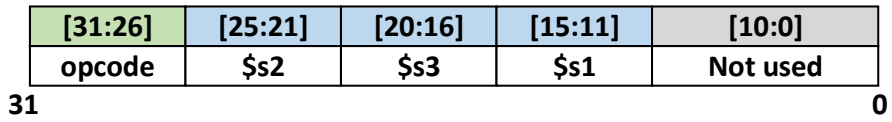
Design Description

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

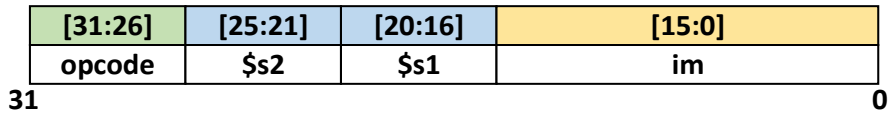
   `$pc = $pc + 4` for every instruction (except **`beq`**, **`bne`**)



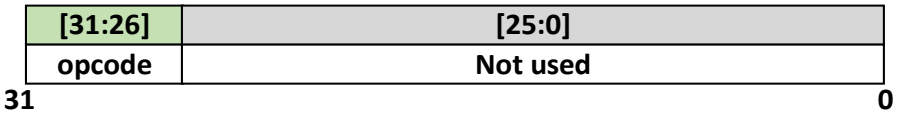
2. Register file contains 32 unsigned 32-bit registers for operation.
3. Instruction mapping
   a. **R-type**



   b. **I-type**



   c. **EOF**



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

Operation	Assemble	Opcode	Type	Meaning	Note
Add	`add`	6'd1	R	`$s1 = $s2 + $s3`	Signed Operation
Subtract	`sub`	6'd2	R	`$s1 = $s2 - $s3`	Signed Operation
Multiply	`mul`	6'd3	R	`$s1 = $s2 * $s3`	Signed Operation
Add (floating point)	`fp_add`	6'd13	R	`$s1 = $s2 + $s3`	Floating Point Operation
Subtract (floating point)	`fp_sub`	6'd14	R	`$s1 = $s2 - $s3`	Floating Point Operation
Multiply (floating point)	`fp_mul`	6'd15	R	`$s1 = $s2 * $s3`	Floating Point Operation

Add immediate	addi	6'd4	I	`$s1 = $s2 + im`	Signed Operation
Load word	lw	6'd5	I	`$s1 = Mem[$s2 + im]`	Signed Operation
Store word	sw	6'd6	I	`Mem[$s2 + im] = $s1`	Signed Operation
AND	and	6'd7	R	`$s1 = $s2 & $s3`	Bit-wise
OR	or	6'd8	R	`$s1 = $s2 | $s3`	Bit-wise
NOR	nor	6'd9	R	`$s1 = ~($s2 | $s3)`	Bit-wise
Branch on equal	beq	6'd10	I	`if($s1==$s2), $pc = $pc + im;` `else, $pc = $pc + 4`	PC-relative Unsigned Operation
Branch on not equal	bne	6'd11	I	`if($s1!=$s2), $pc = $pc + im;` `else, $pc = $pc + 4`	PC-relative Unsigned Operation
Set on less than	slt	6'd12	R	`if($s2<$s3), $s1 = 1;` `else, $s1 = 0`	Signed Operation
Shift left logical	sll	6'd16	R	`$s1 = $s2 << $s3`	Unsigned Operation
Shift right logical	srl	6'd17	R	`$s1 = $s2 >> $s3`	Unsigned Operation
End of File	eof	6'd18	EOF	`Stop processing`	Last instruction in the pattern

Note: The notation of `im` in I-type instruction is **2's complement**.

Note: Signed operations indicates that the data in register file are expressed in **2's complement**.

5. Interface of instruction memory (size: 1024×32 bit)
   - i_addr[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_addr[7:2] for address mapping in data memory
   - To fetch data of data memory in your testbench, use following instance name

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

```
module data_mem (
    input             i_clk,
    input             i_rst_n,
    input             i_we,
    input  [ 31 : 0 ] i_addr,
    input  [ 31 : 0 ] i_wdata,
    output [ 31 : 0 ] o_rdata
);
```

7. Overflow may be happened.
   - **Situation1**: Overflow happened at arithmetic instructions (`add`, `sub`, `addi`)

   - **Situation2**: If output address is mapped to unknown address in data/instruction memory. (Do not consider the case if instruction address is beyond eof, but the address mapping is in the size of instruction memory)

8. 4 statuses of o_status

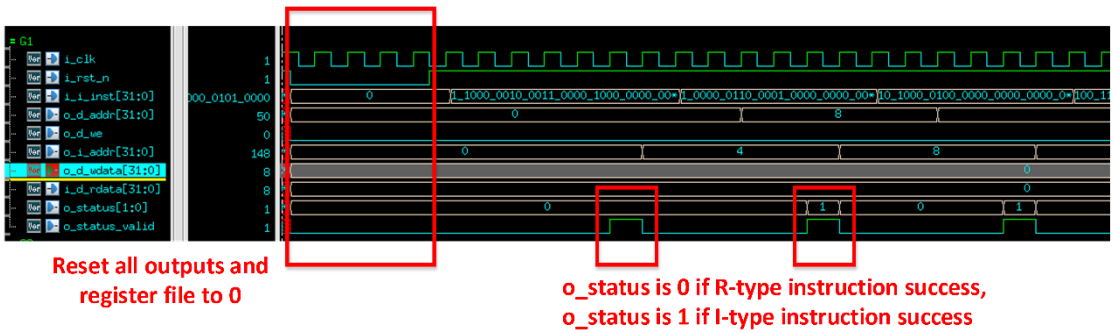
o_status[1:0]	Definition
2'd0	R_TYPE_SUCCESS
2'd1	I_TYPE_SUCCESS
2'd2	MIPS_OVERFLOW
2'd3	MIPS_END

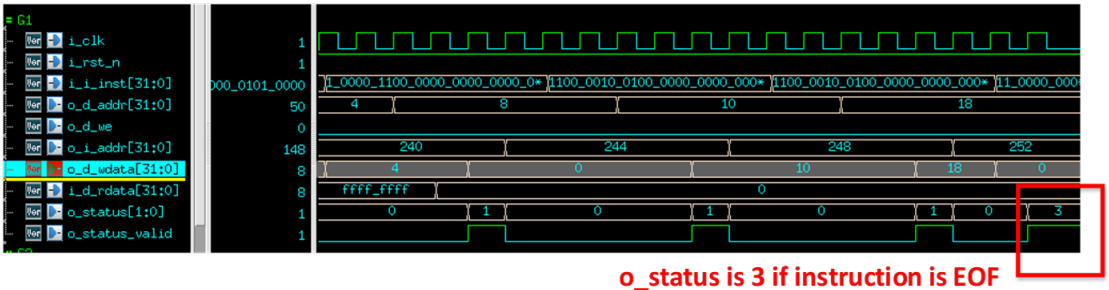
9. Last instruction would be `eof` for every pattern.

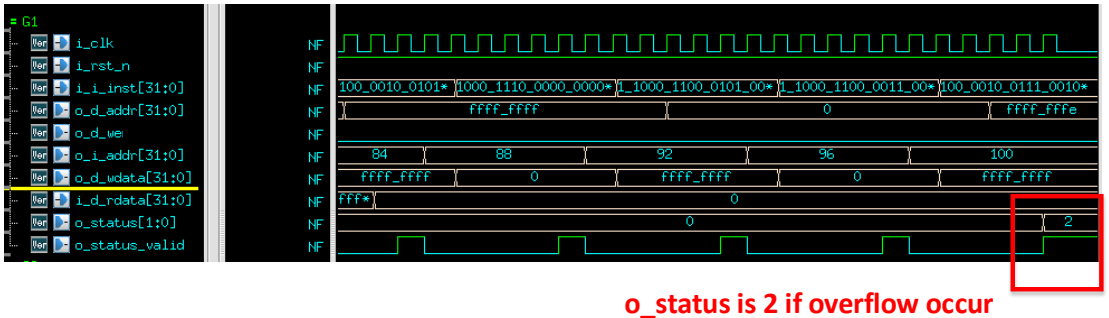
10. There is no unknown opcode in the pattern.

Sample Waveform

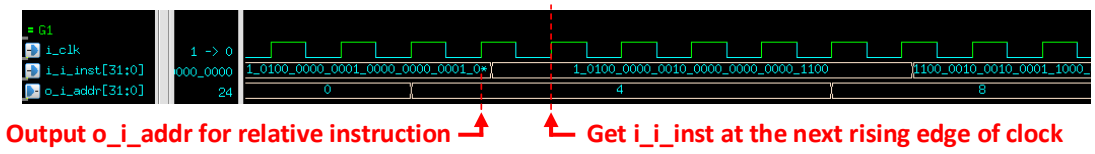
1. Status check



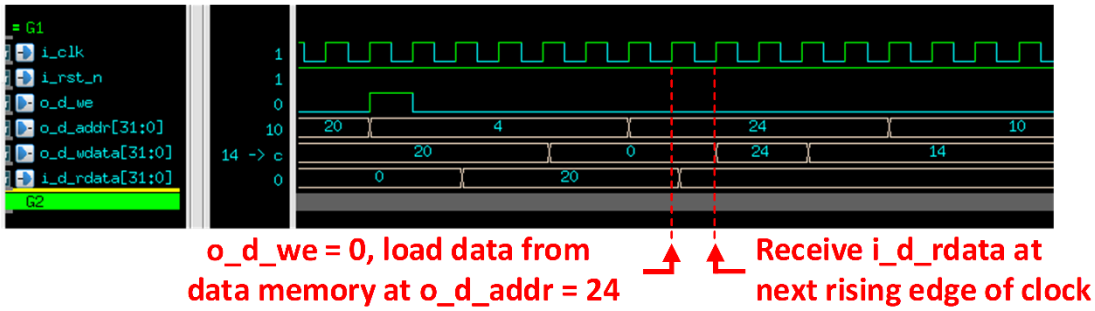




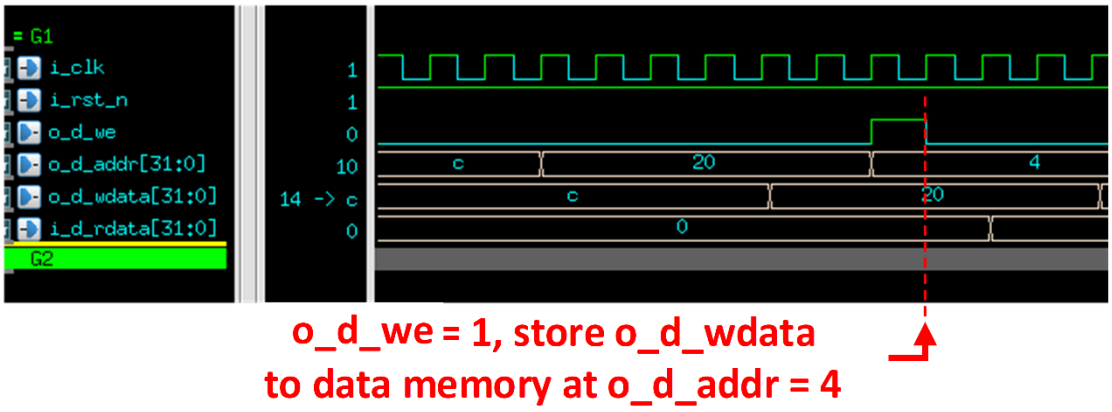
2. Read instruction from instruction memory



3. Load data from data memory



4. Save data to data memory



Testbed

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

Submission

2. Create a folder named **studentID_hw2**, and put all below files into the folder as

```
r11943133_hw2
    └── 01_RTL
        ├── rtl.f (your file list)
        ├── core.v
        ├── flist.v (your file list for lint check)
        └── all other design files (optional)
```

   Note: Use **lower case** for the letter in your student ID. (Ex. r11943133_hw2)

3. 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. r11943133_hw2_v1)

4. 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.

```
vcs -f rtl.f -full64 -R -debug_access+all +define+p0 +v2k
```

2. Pass the patterns to get full score.
   - Provided pattern: **70%** (patterns: p0, p1)
     - **30%** for each pattern (data in data memory: **15%**, status check: **15%**)
     - **10%** for spyglass check
     - **Don't implement the answers in your design directly!**
   - Hidden pattern: **30%** (20 patterns in total)
     - **1.5%** for each pattern (data & status both correct)
3. Delay submission
   - **No delay submission is allowed**
   - Lose **5 point** for any wrong naming rule. Don't compress all homework folder.

Hint

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