

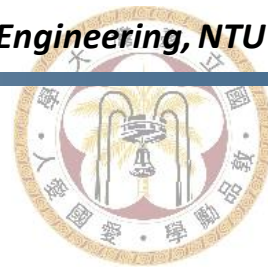
Computer-Aided VLSI System Design

Homework 2: Simple MIPS CPU

Graduate Institute of Electronics Engineering, National Taiwan University

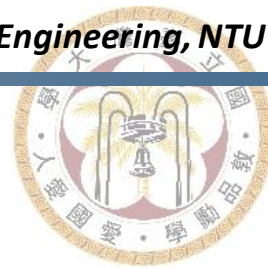


NTU GIEE



Goal

- In this homework, you will learn
 - How to write testbench
 - How to design FSM
 - How to use IP
 - Generate patterns for testing

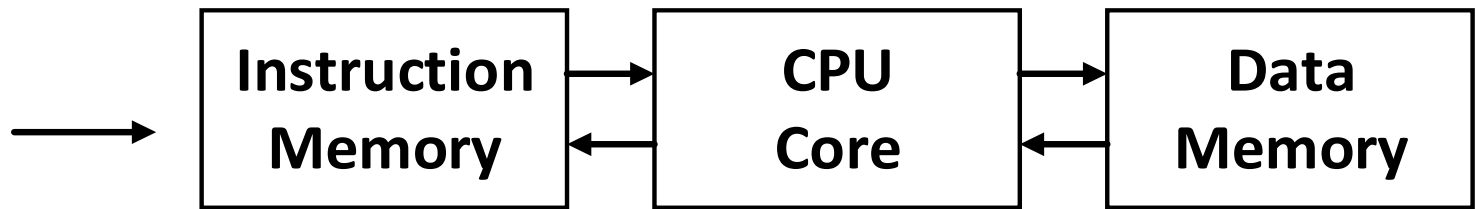


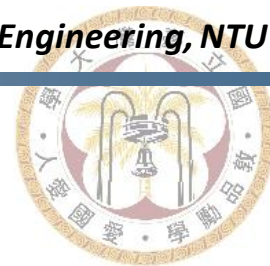
Introduction

- 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.

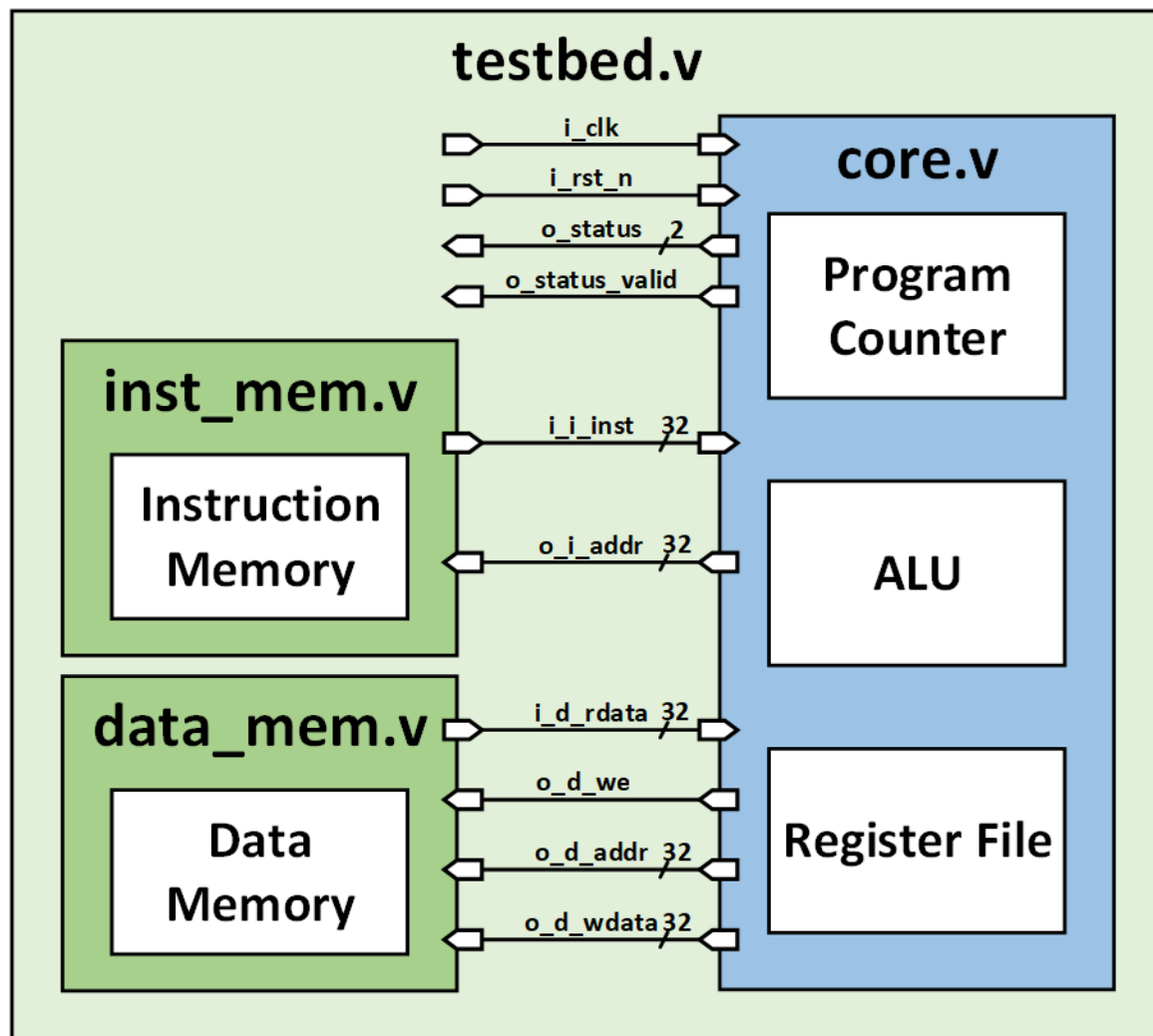
Instruction set

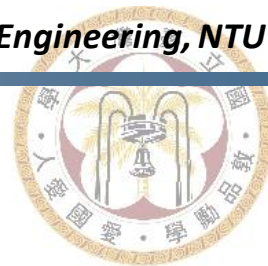
```
addi $7 $3 4
sub  $7 $7 $5
sw   $7 $4 8
bne  $3 $5 12
lw   $6 $0 8
add  $7 $6 $2
sw   $7 $4 8
eof
```





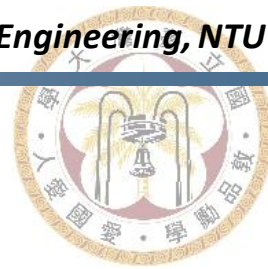
Block Diagram





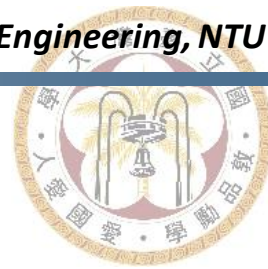
Input/Output

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_v alid	O	1	Set high if ready to output status



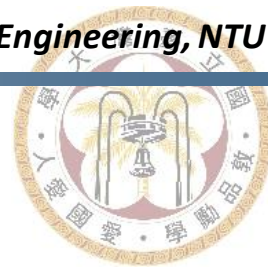
Specification (1)

- All outputs should be synchronized at clock **rising** edge.
- You should set all your outputs and register file to be zero when `i_rst_n` is **low**. Active low asynchronous reset is used.
- Instruction memory and data memory are provided. All values in memory are reset to be zero.
- You should create **32 unsigned 32-bit registers** in register file.
- 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.



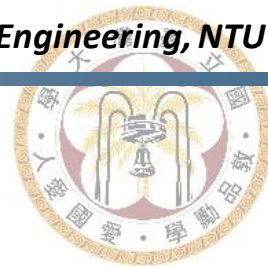
Specification (2)

- 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.
- 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.
- Your o_status_valid should be turned to **high** for only **one cycle** for every o_status.
- The testbench will get your output at negative clock edge to check the o_status if your o_status_valid is **high**.



Specification (3)

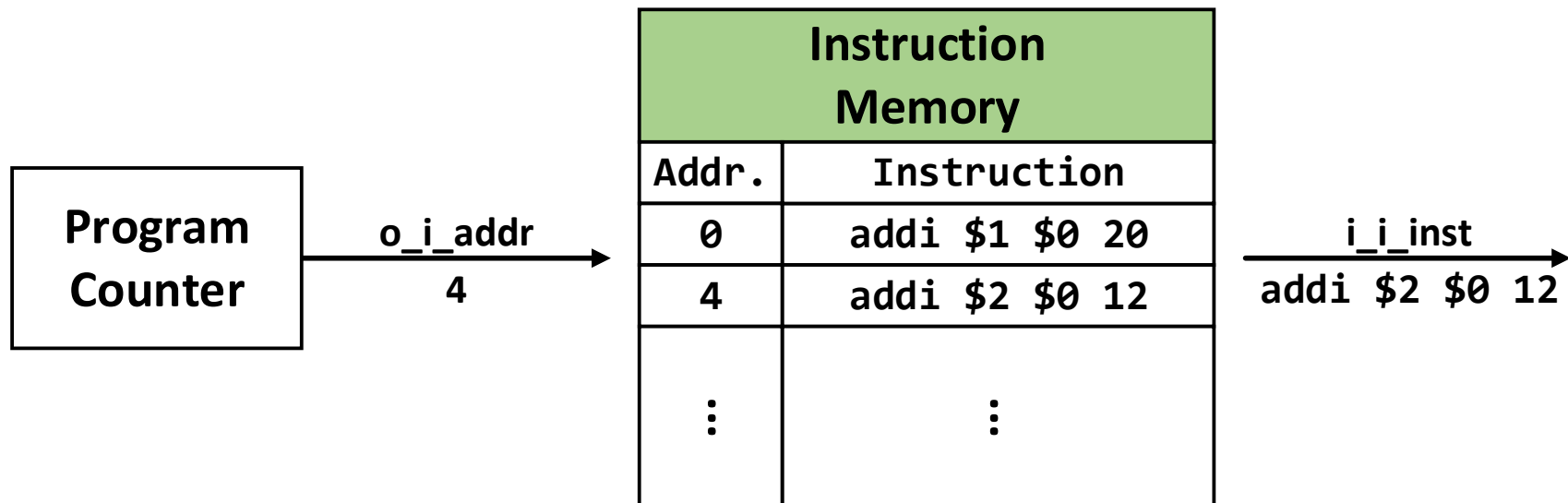
- When you set o_status_valid to **high** and o_status to **3**, stop processing. The testbench will check your data memory value with golden data.
- If overflow happened, stop processing and raise o_status_valid to **high** and set o_status to **2**. The testbench will check your data memory value with golden data.
- Less than **1024** instructions are provided for each pattern..
- The whole processing time can't exceed **120000** cycles.

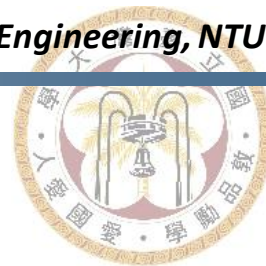


Program Counter

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

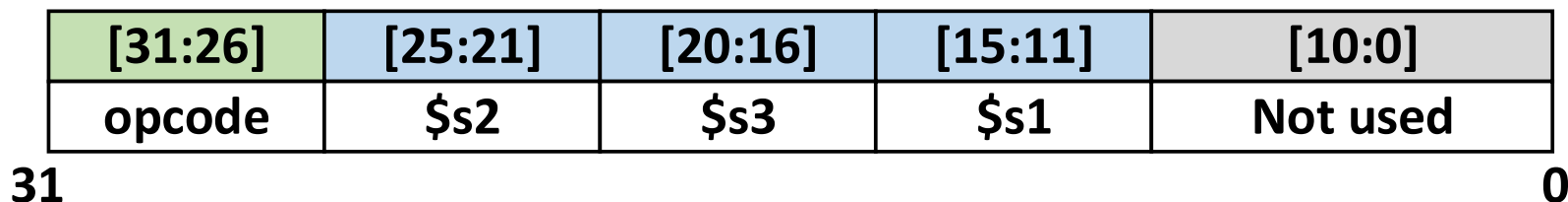
\$pc = \$pc + 4 for every instruction (except **beq**, **bne**, **blt**, **bge**, **bltu**, **bgeu**)



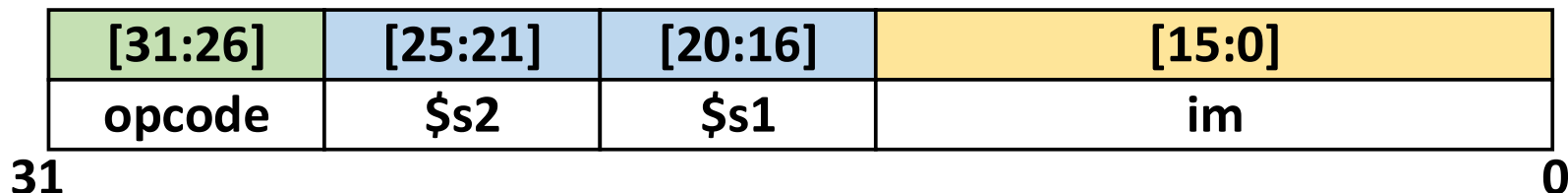


Instruction mapping

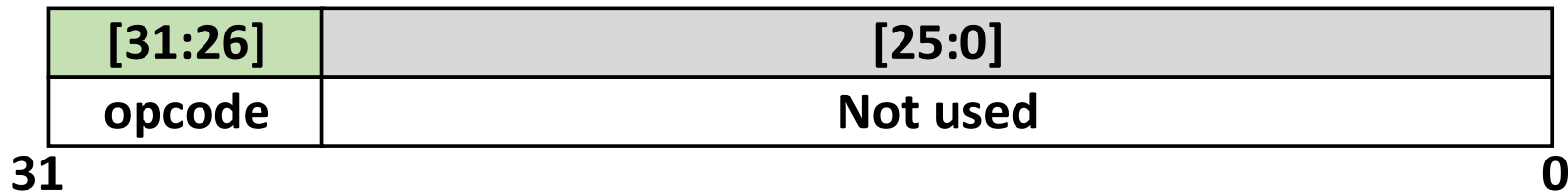
▪ R-type

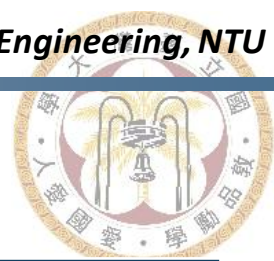


▪ I-type



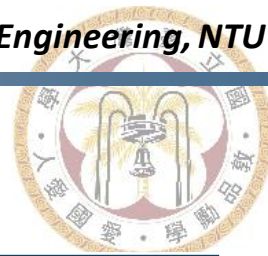
▪ EOF





Instruction

Operation	Assemble	Opcode	Type	Meaning	Note
Add	add	6'd0	R	$\$s1 = \$s2 + \$s3$	Signed Operation
Subtract	sub	6'd1	R	$\$s1 = \$s2 - \$s3$	Signed Operation
Add unsigned	addu	6'd2	R	$\$s1 = \$s2 + \$s3$	Unsigned Operation
Subtract unsigned	subu	6'd3	R	$\$s1 = \$s2 - \$s3$	Unsigned 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
XOR	xor	6'd9	R	$\$s1 = \$s2 \wedge \$s3$	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($\$s1 \neq \$s2$), $\$pc = \$pc + im$; else, $\$pc = \$pc + 4$	PC-relative Unsigned Operation

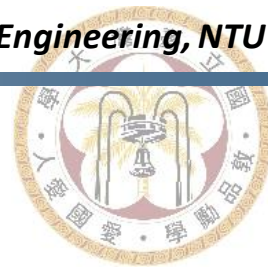


Instruction (cont'd)

Operation	Assemble	Opcode	Type	Meaning	Note
Set on less than	slt	6'd12	R	if(\$s2<\$s3), \$s1 = 1; else, \$s1 = 0	Signed Operation
Shift left logical	sll	6'd13	R	\$s1 = \$s2 << \$s3	Unsigned Operation
Shift right logical	srl	6'd14	R	\$s1 = \$s2 >> \$s3	Unsigned Operation
Branch to less than	blt	6'd15	I	if(\$s1<\$s2), \$pc = \$pc + im; else, \$pc = \$pc + 4	PC-relative Signed Operation
Branch to greater or equal	bge	6'd16	I	if(\$s1>=\$s2), \$pc = \$pc + im; else, \$pc = \$pc + 4	PC-relative Signed Operation
Branch to less than unsigned	bltu	6'd17	I	if(\$s1<\$s2), \$pc = \$pc + im; else, \$pc = \$pc + 4	PC-relative Unsigned Operation
Branch to greater or equal unsigned	bgeu	6'd18	I	if(\$s1>=\$s2), \$pc = \$pc + im; else, \$pc = \$pc + 4	PC-relative Unsigned Operation
End of File	eof	6'd19	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**.

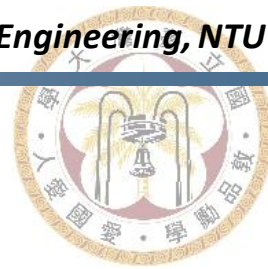


Memory IP

- Instruction memory
 - Size: 1024×32 bit
 - `i_addr[11:2]` for address mapping in instruction memory
- Data memory
 - Size: 64×32 bit
 - `i_addr[7:2]` for address mapping in data 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  
);
```

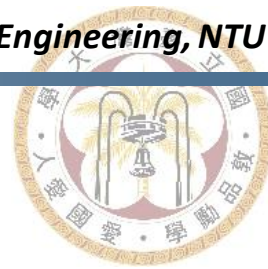
```
module data_mem (  
    input          i_clk,  
    input          i_rst_n,  
    input          i_we,  
    input [ 31 : 0 ] i_addr,  
    input [ 31 : 0 ] i_wdata,  
    output [ 31 : 0 ] o_rdata  
);
```



Status

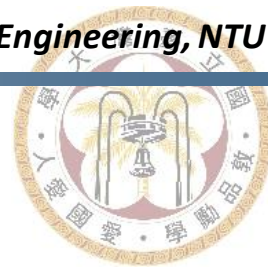
- 4 statuses of o_status

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



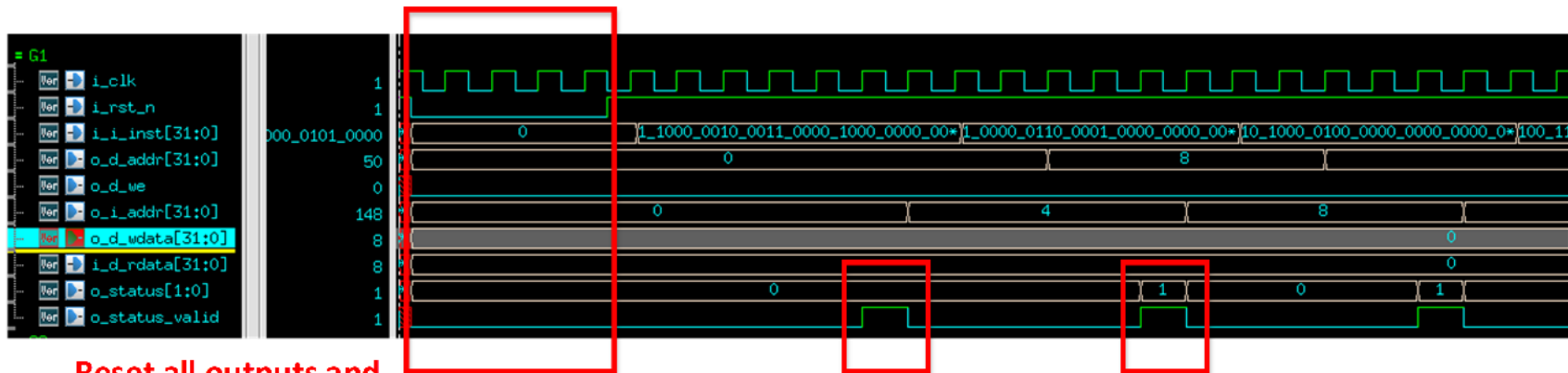
Overflow

- Overflow may be happened.
 - **Situation1**: Overflow happened at arithmetic instructions (add, sub, addu, subu, addi)
 - **Situation2**: If output address are 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)



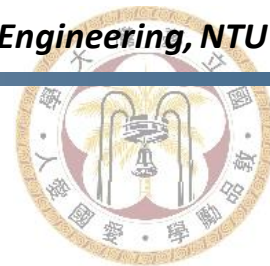
Waveform

- Status Check



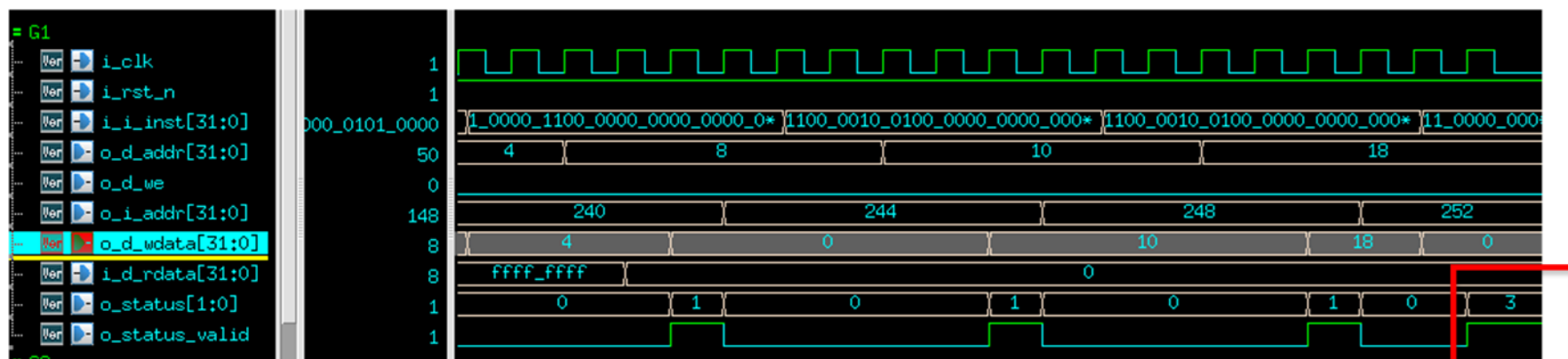
Reset all outputs and register file to 0

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

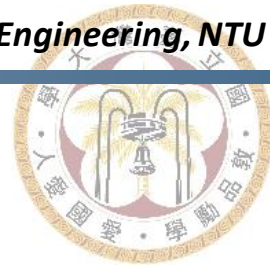


Waveform

■ Status Check

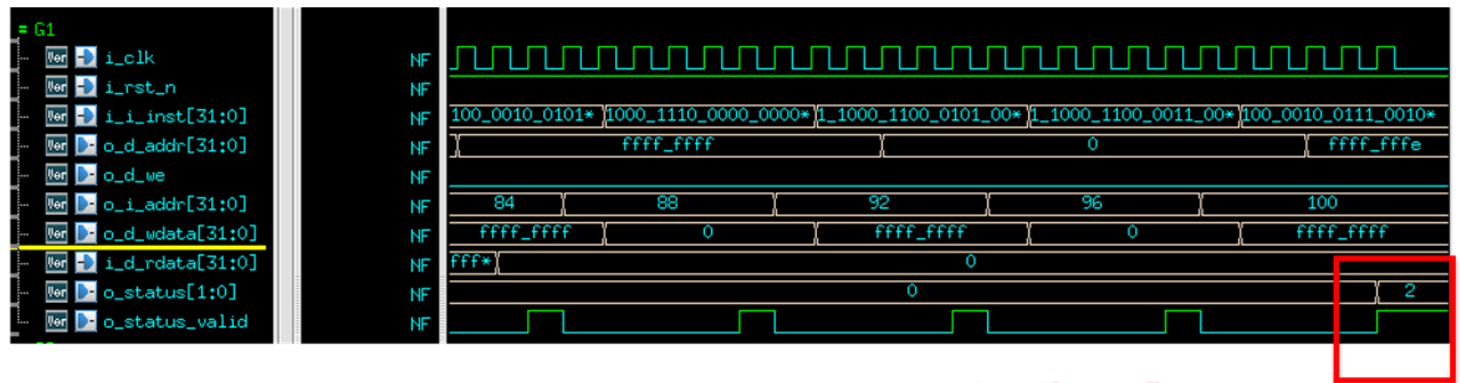


o_status is 3 if instruction is EOF

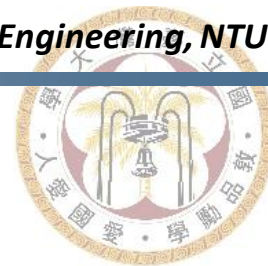


Waveform

- Status Check

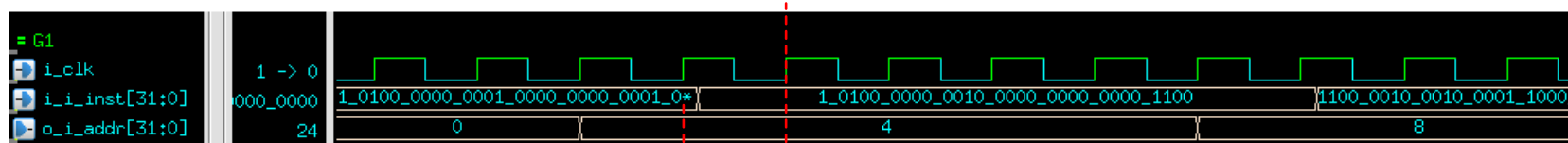


o_status is 2 if overflow occur



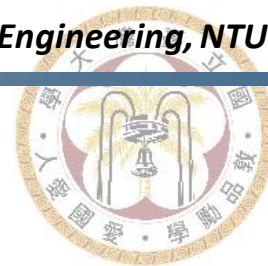
Waveform

- Read instruction from instruction memory



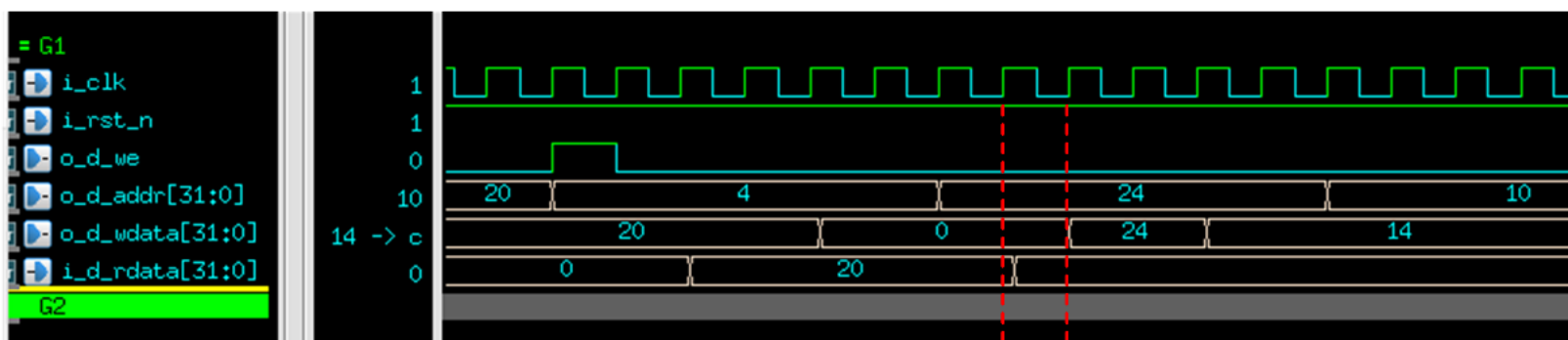
Output `o_i_addr` for relative instruction →

← Get `i_i_inst` at the next rising edge of clock



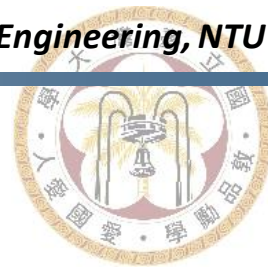
Waveform

- Load data from data memory



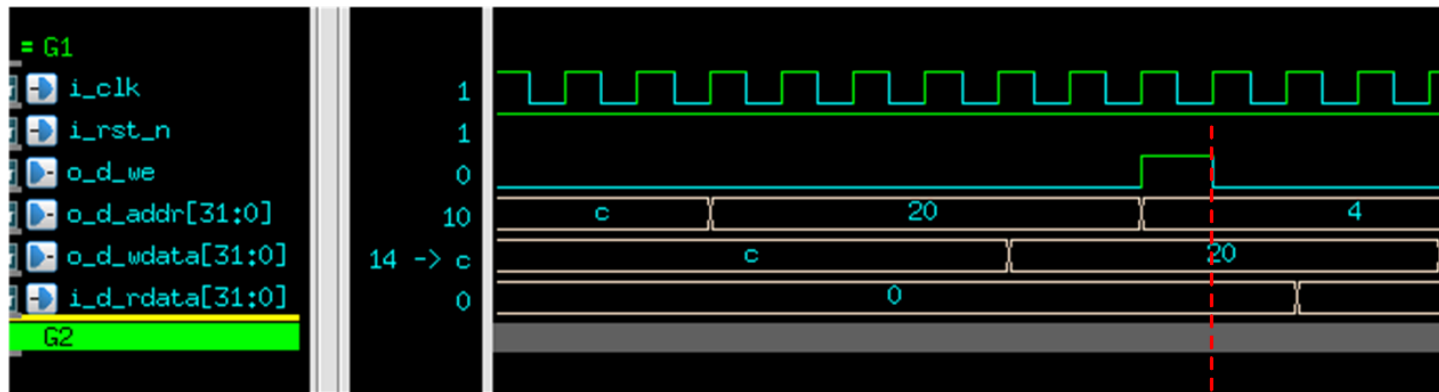
**`o_d_we` = 0, load data from
data memory at `o_d_addr` = 24**

**Receive `i_d_rdata` at
next rising edge of clock**

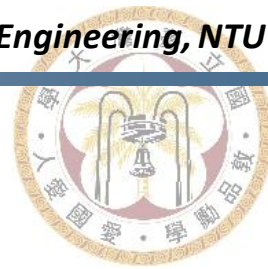


Waveform

- Save data to data memory

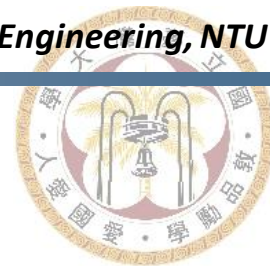


**`o_d_we = 1`, store `o_d_wdata`
to data memory at `o_d_addr = 4`**



core.v

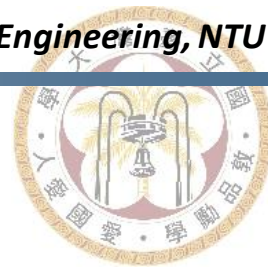
```
module core #(                                //Don't modify interface
    parameter ADDR_W = 32,
    parameter INST_W = 32,
    parameter DATA_W = 32
) (
    input          i_clk,
    input          i_rst_n,
    output [ ADDR_W-1 : 0 ] o_i_addr,
    input  [ INST_W-1 : 0 ] i_i_inst,
    output          o_d_we,
    output [ ADDR_W-1 : 0 ] o_d_addr,
    output [ DATA_W-1 : 0 ] o_d_wdata,
    input  [ DATA_W-1 : 0 ] i_d_rdata,
    output [          1 : 0 ] o_status,
    output          o_status_valid
);
```



rtl.f

- Filelist

```
// -----  
// Simulation: HW2 simple mips CPU  
// -----  
  
// define files  
// -----  
../00_TESTBED/define.v  
  
// testbench  
// -----  
../00_TESTBED/testbed.v  
../00_TESTBED/inst_mem.vp  
../00_TESTBED/data_mem.vp  
  
// design files  
// -----  
./core.v
```



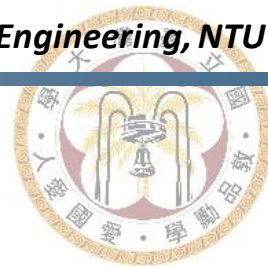
Command

- 01_run

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

- 99_clean_up

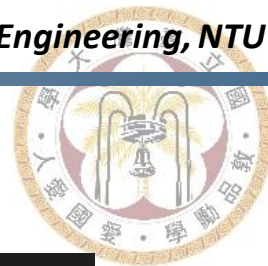
```
rm -rf INCA_libs/ ncverilog.* novas*
```

define.v

```
// opcode definition
`define OP_ADD 0
`define OP_SUB 1
`define OP_ADDU 2
`define OP_SUBU 3
`define OP_ADDI 4
`define OP_LW 5
`define OP_SW 6
`define OP_AND 7
`define OP_OR 8
`define OP_XOR 9
`define OP_BEQ 10
`define OP_BNE 11
`define OP_SLT 12
`define OP_SLL 13
`define OP_SRL 14
`define OP_BLT 15
`define OP_BGE 16
`define OP_BLTU 17
`define OP_BGEU 18
`define OP_EOF 19

// MIPS status definition
`define R_TYPE_SUCCESS 0
`define I_TYPE_SUCCESS 1
`define MIPS_OVERFLOW 2
`define MIPS_END 3
```



testbed_temp.v

- Things to add in your testbench
 - Clock
 - Reset
 - Waveform file
 - Function test
 - ...

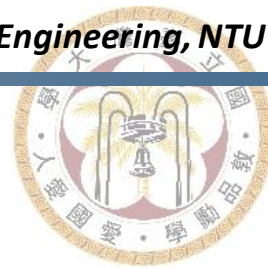
```
module testbed;

    reg clk = 0;
    reg rst_n = 1;
    wire [ 31 : 0 ] imem_addr;
    wire [ 31 : 0 ] imem_inst;
    wire            dmem_we;
    wire [ 31 : 0 ] dmem_addr;
    wire [ 31 : 0 ] dmem_wdata;
    wire [ 31 : 0 ] dmem_rdata;
    wire [ 1 : 0 ] mips_status;
    wire            mips_status_valid;
```

```
core u_core (
    .i_clk(),
    .i_rst_n(),
    .o_i_addr(),
    .i_i_inst(),
    .o_d_we(),
    .o_d_addr(),
    .o_d_wdata(),
    .i_d_rdata(),
    .o_status(),
    .o_status_valid()
);

inst_mem u_inst_mem (
    .i_clk(),
    .i_rst_n(),
    .i_addr(),
    .o_inst()
);

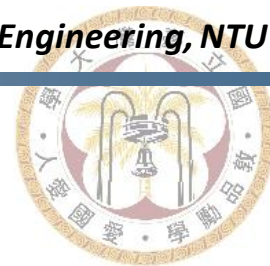
data_mem u_data_mem (
    .i_clk(),
    .i_rst_n(),
    .i_we(),
    .i_addr(),
    .i_wdata(),
    .o_rdata()
);
```



Protected Files

- The following files are protected
 - inst_mem.vp
 - data_mem.vp

```
module inst_mem (
    input          i_clk,
    input          i_rst_n,
    input  [ 31 : 0 ] i_addr,
    output [ 31 : 0 ] o_inst
);
`protected
Nd15kSQH5DT^<D9i:i7T7ceFn3@o:C2]Ke:L;dfq^QGQOG?3K:ogIe8]1ge<gcg3
lCH3E]ekmLN<RVkKa1o39E7E21a;`hJRSFMUb2pAgL?TeZdH]>^RK;KWYU@>G2G6
H[IMYG;D<[Z>[;0`?NbPoEAQM<_ZfDbp1HN@Hmq50`Q<5[53C:9UD4^:Y44]9a^e
PDH[cdHb;HPi\R4k7mA1PdY8ZpI=4?nNZgQ2I>QUg[agM4j@cT1]hnMoC<i1F9DR
[kf;]ULlecpF`H;9L2DeZa>@LdfLgfb814bWgT:_P3?ENhifQW@_Ne;gMZE9@f0A
OERY:F4d68KqAIn]N1dj4LN7_8:Uigk?9UJ9JYQM4l=Lq\TEXDQ01>Zo^SjQ=Cge
?kp68am:9p81Q1[<jSXm?;GhoPHHYKp\Q][2epXn_18k8LA5g=N7=D?=VOX<Ham8
[A:Qc;Rlp038>d9_Qk9cfk?:5hXP>LT3n=DP08A_]WPa6nA3cYZjG132qB9]I4kp
>=:4m9P`dCB8@?ip`@VR7AahIggjNR:M1:_\KXE1BF0m<Bb@ZS[^W7EheJ18mX8;
?7F`Pg\CCA8igfFUoWY@k>Yq=U3_4>E50_nJ\`aUGcfWD_89dab]cUQFF<?2P?OG
qWglWC[\iqnjC<OipHHnb<T4Sg<:UORVSVocI_g?<a@o__<PQ493cZIE;7^Sp1AQ
G<c17[ ]R\>VT]]LA\7?Uk=]\bG19MT9N;K<Y92[iK0ged92EIkQZliw>q1G]QI?5
ST06RFN<KJl@VM1EWKSmb1B5U:BaX`E7of7mq0JBgO`9k$
`endprotected
endmodule
```

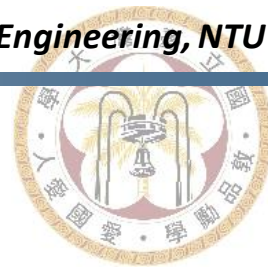


PATTERN

- Files in PATTERN are for your references

inst_assemble.dat

```
-----  
R-type  $s2  $s3  $s1  
I-type  $s2  $s1  im  
-----  
and      $1    $3    $1  
lw       $3    $1     8  
bne      $2    $0     8  
add      $7    $1    $4  
slt      $6    $5    $4  
slt      $4    $1    $1  
lw       $1    $3    12  
lw       $7    $7     4  
bne      $6    $7     8  
lw       $6    $5     8  
lw       $5    $2     8
```

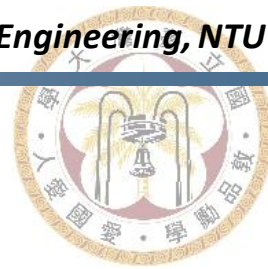


Grading Policy

- TA will run your code with following command

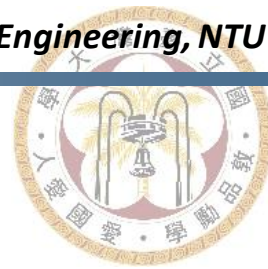
```
vcs -f rtl.f -full64 -R -debug_access+all  
+v2k +define+p0
```

- Pass the patterns to get full score
 - Provided pattern: **80%**
 - **40%** for each test (data from data memory: **20%**, status check: **20%**)
 - Hidden pattern: **20%** (20 patterns in total)
 - **1%** for each test (data & status both correct)
- **No delay submission is allowed**
- Lose **3 point** for any wrong naming rule or format for submission



Submission

- 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)
- Compress the folder **studentID_hw2** in a tar file named **studentID_hw2_vk.tar** (k is the number of version, $k = 1, 2, \dots$)



Hint

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