



Computer Architecture

Final Project: Single Cycle CPU

姚鈞嚴 r08943003@ntu.edu.tw

陳瑋廷 r08943022@ntu.edu.tw

王 琰 b05901016@ntu.edu.tw



Announcement

- ◆ 2 ~ 3 people / group
- ◆ Please find a representative to send your member list before 13:00, 5/29
 - ◆ E-mail: r08943003@ntu.edu.tw
- ◆ TA will help you find group members if you does not find any partner
- ◆ The final member list will be announced before 23:59, 5/29

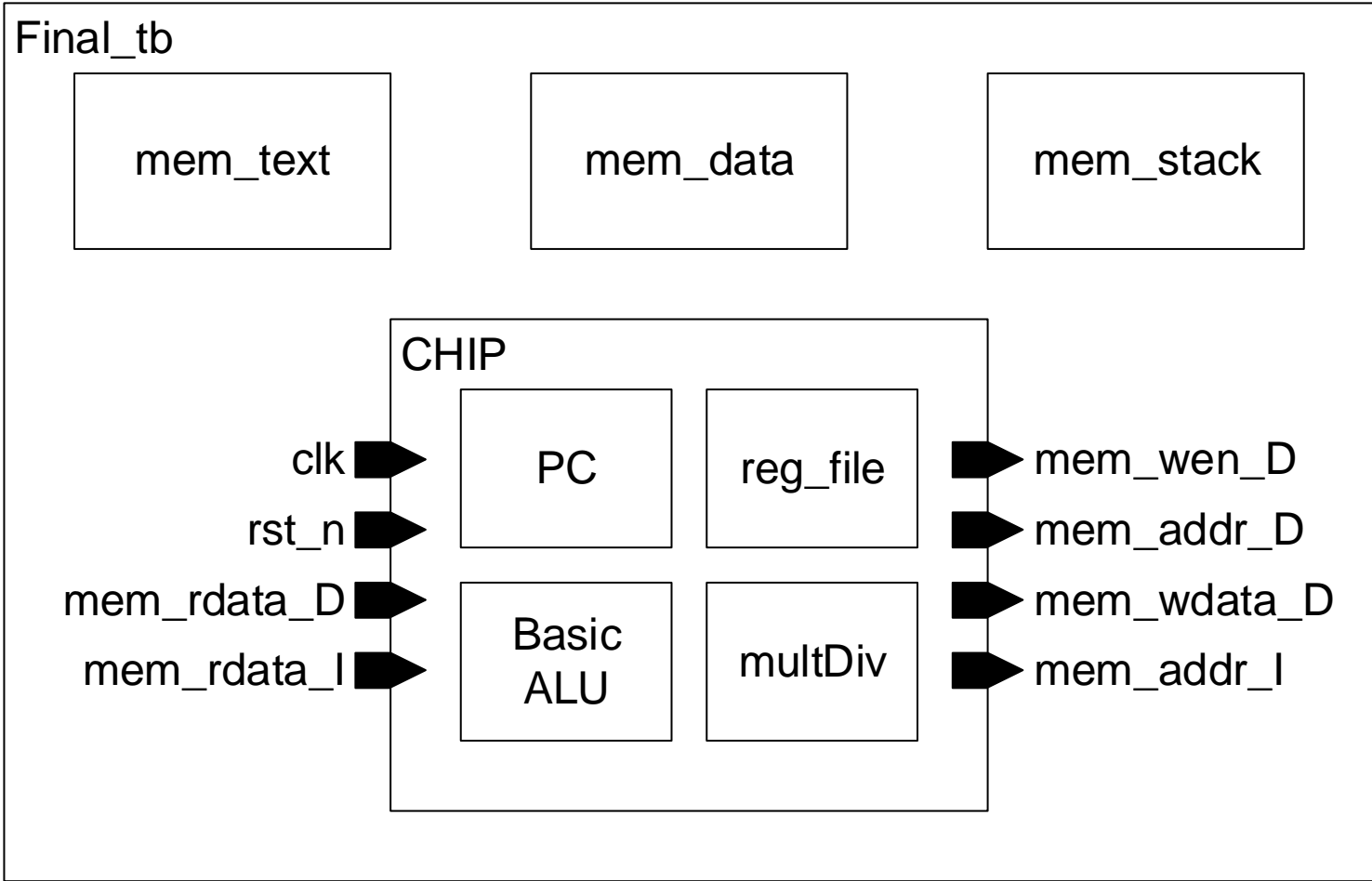


Goal

- ◆ Implement a single cycle CPU
- ◆ Add multiplication/division unit (multDiv) to CPU (HW3)
- ◆ Handle multi-cycle operations
- ◆ Get more familiar with assembly and Verilog
- ◆ Run your own assembly in HW1-1 on your CPU (Bonus)



Specification





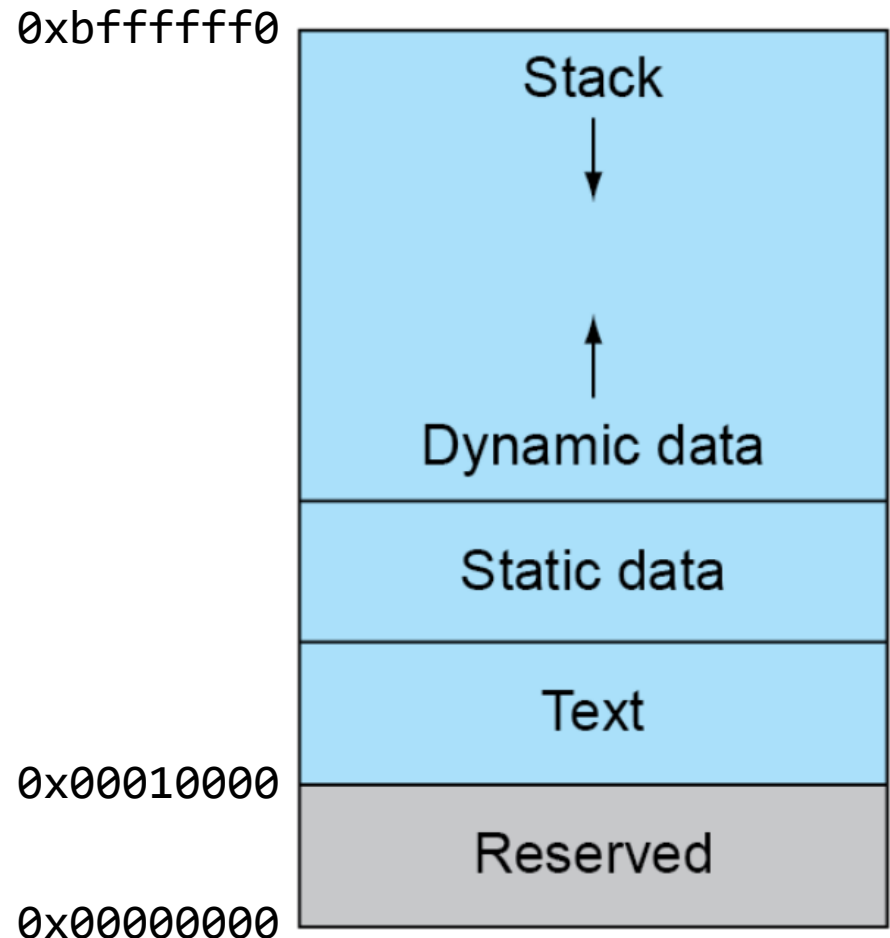
Port Definition

Name	I/O	Width	Description
clk	I	1	Positive edge-triggered clock
rst_n	I	1	Asynchronous negative edge reset
mem_wen_D	O	1	0: Read data from data/stack memory 1: Write data to data/stack memory
mem_addr_D	O	32	Address of data/stack memory
mem_wdata_D	O	32	Data written to data/stack memory
mem_rdata_D	I	32	Data read from data/stack memory
mem_addr_I	O	32	Address of instruction (text) memory
mem_rdata_I	I	32	Instruction read from instruction (text) memory



Memory Layout

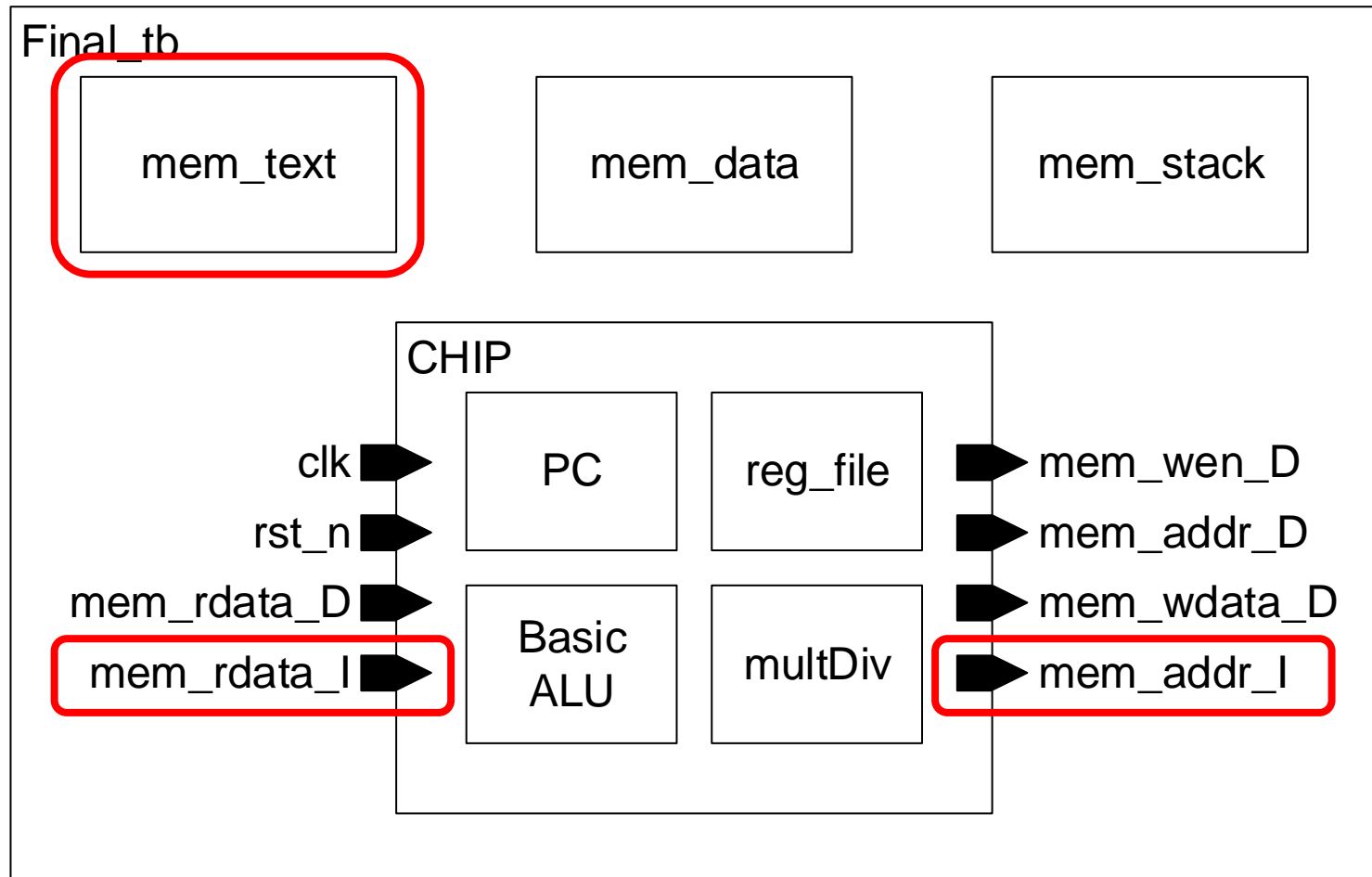
- ◆ In Jupiter simulator
 - ◆ Text
 - ◆ Program code
 - ◆ Data
 - ◆ Variables, arrays, etc.
 - ◆ Stack
 - ◆ Automatic storage





Relate Memory to Testbench (1/4)

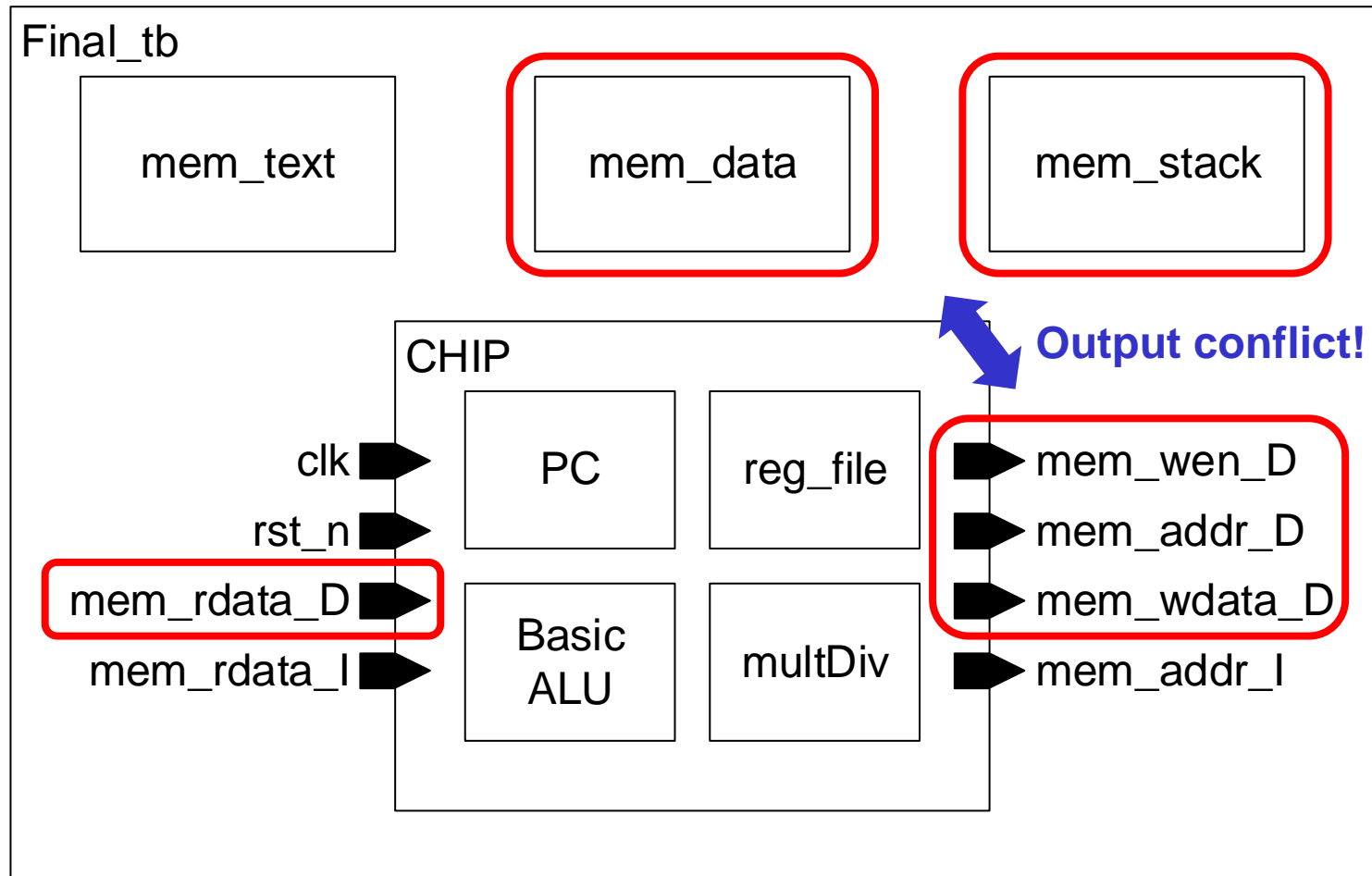
◆ Instruction (text) memory





Relate Memory to Testbench (2/4)

◆ Data/stack memory





Relate Memory to Testbench (3/4)

- ◆ Reduce size of memory blocks to improve simulation speed



```
`define SIZE_TEXT 32  
`define SIZE_DATA 32  
`define SIZE_STACK 32
```

- ◆ Define offset address for each memory block



- ◆ Define high impedance to avoid output conflict



- ◆ **Not synthesizable coding style!**

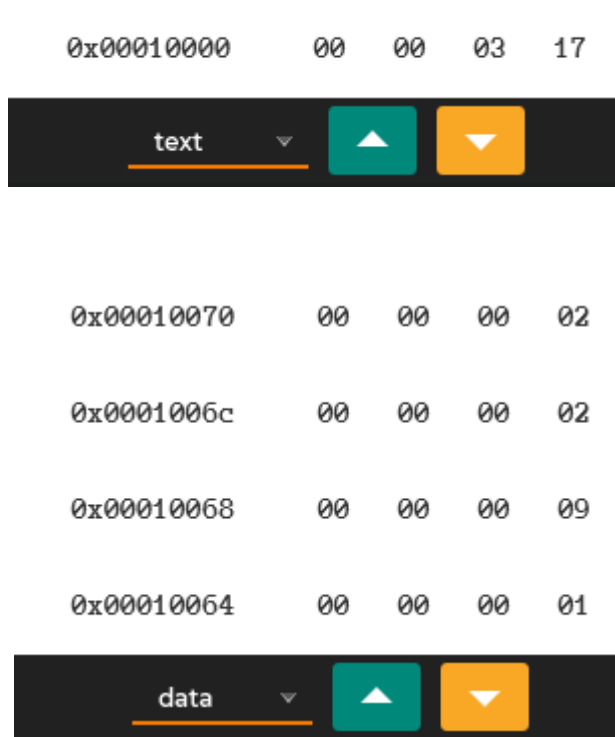
```
module memory #(  
    parameter BITS = 32,  
    parameter word_depth = 32  
) (  
    clk,  
    rst_n,  
    wen,  
    a,  
    d,  
    q,  
    offset  
);
```

```
always @(*) begin  
    q = {(BITS-1){1'bz}};  
    for (i=0; i<word_depth; i=i+1) begin  
        if (mem_addr[i] == a)  
            q = mem[i];  
    end  
end
```

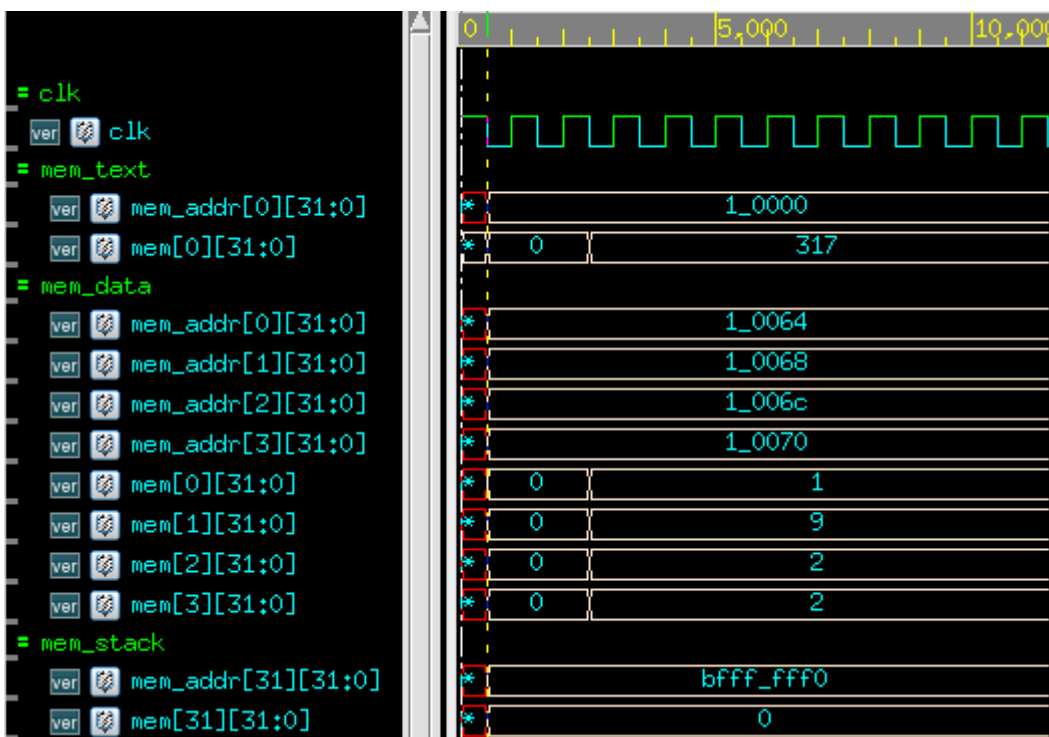


Relate Memory to Testbench (4/4)

◆ In Jupiter



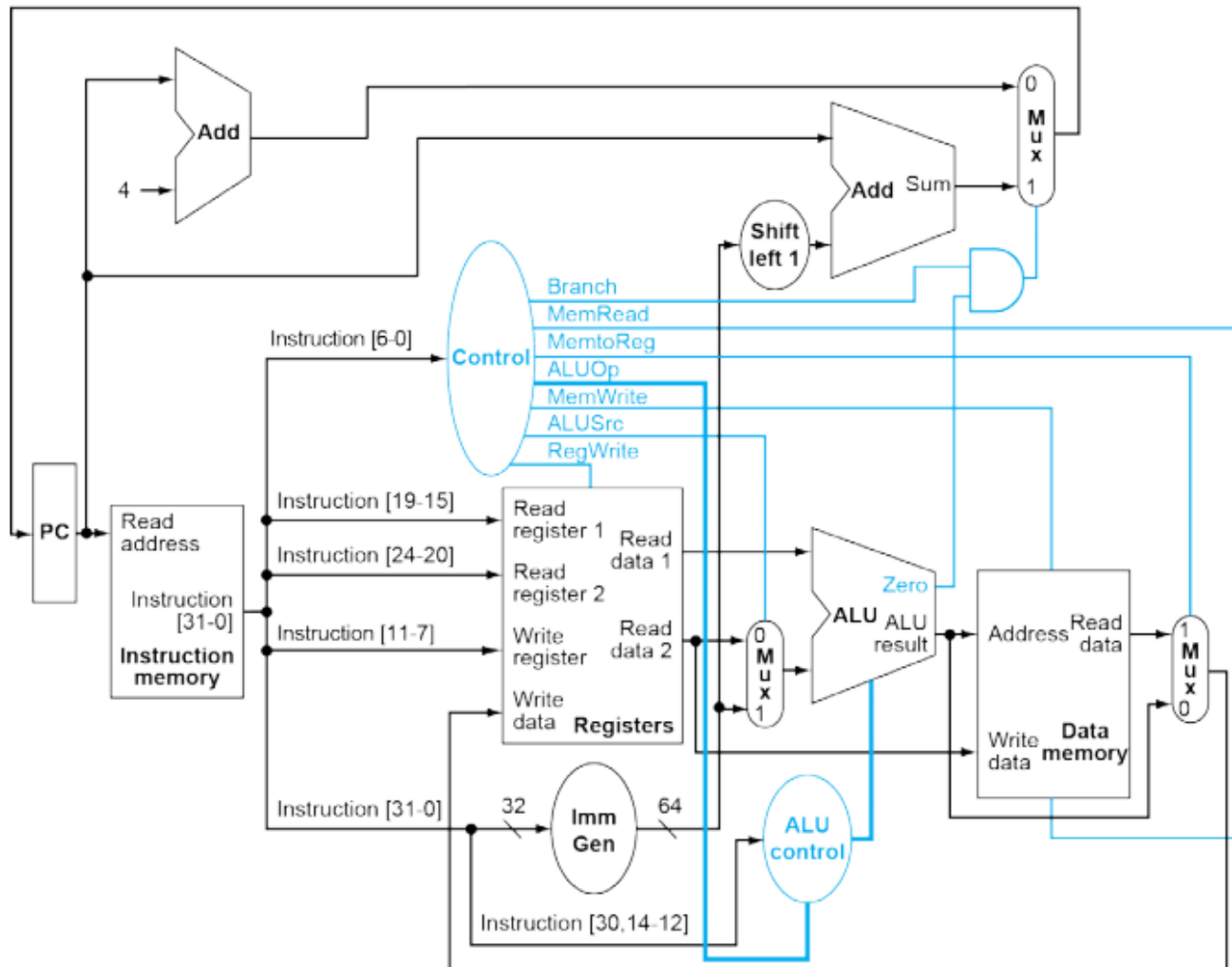
◆ In Testbench





Architecture

- ◆ Not complete (does not include jal, jalr, ...)



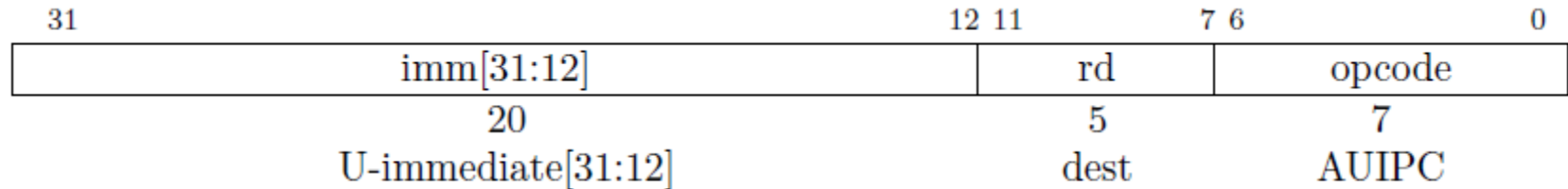


Supporting Instructions

- ◆ Your design must at least support
 - ◆ auipc, jal, jalr
 - ◆ beq, lw, sw
 - ◆ addi, slti, add, sub
 - ◆ mul
- ◆ For **bonus** challengers
 - ◆ srai, slli, ... (observe which instructions do you use)
- ◆ See “Instruction_Set_Listings.pdf” for more information of machine code



Supplement: Instruction “auipc”

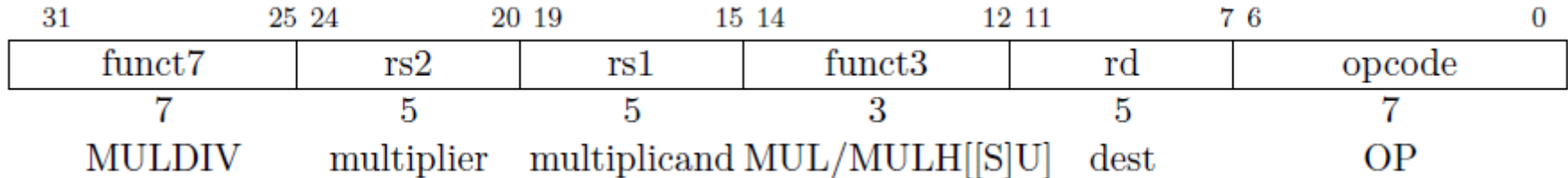


- ◆ Add upper immediate to PC, and store the result to rd
 - ◆ auipc rd, U-immediate

- ◆ Example: auipc x5, 1 (PC = 0x0001001c)
 - ◆ $0x0001001c + 0x00001000 = 0x0001101c$
 - ◆ Store 0x0001101c in x5



Supplement: Instruction “mul”

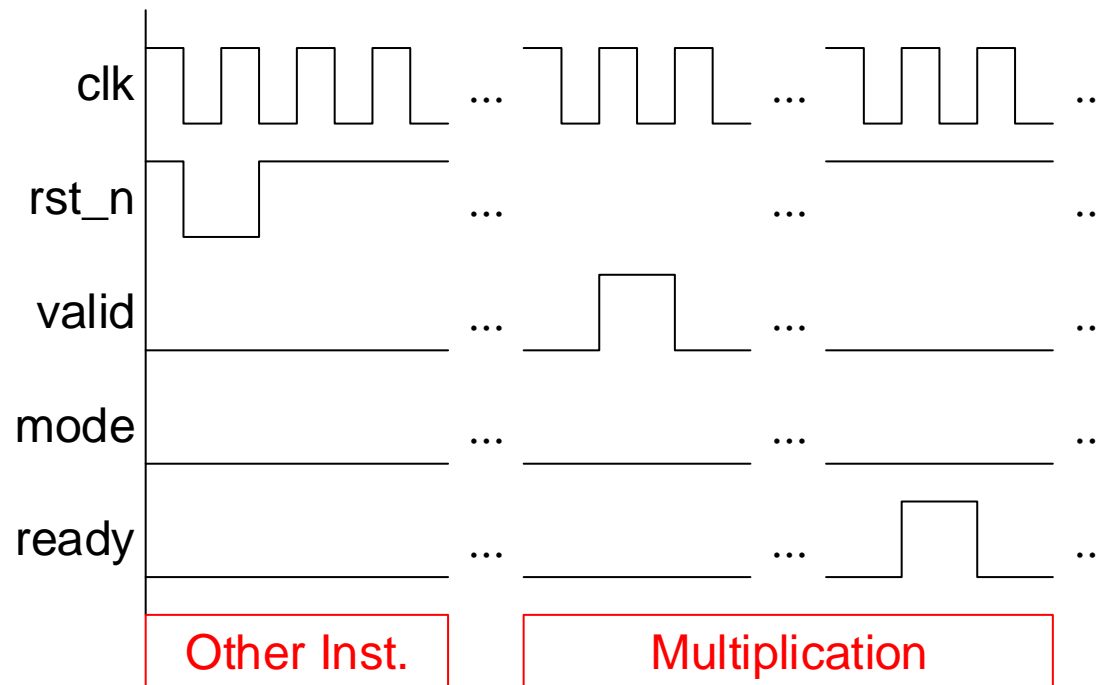


- ◆ Not included in RV32I
- ◆ Store the lower 32-b result ($rs1 \times rs2$) to rd
- ◆ Example: mul x10, x10, x6
 - ◆ $x10 = 0x00000001$, $x6 = 0x00000002$
 - ◆ $0x00000001 \times 0x00000002 = 0x00000002$
 - ◆ Store $0x00000002$ in x10
- ◆ **Your multDiv can support this instruction!**



Multi-Cycle Operation

- ◆ Once CPU decodes mul operation, issue valid to your multDiv
- ◆ Once CPU receives ready, store the lower 32-b result to rd
- ◆ You might have to design FSM in your CPU





Test Pattern 1: Leaf Example

- ◆ Modified from lecture slides
- ◆ The procedure loads a,b,c,d from 0x00010064–0x00010070, and stores the result to 0x00010074
- ◆ Run simulation:
 - ◆ > ncverilog Final_tb.v +define+leaf +access+r

```
def leaf(a,b,c,d):
    f = (a+b) - (c+d)
    return f
```

```
.data
    a: .word 1
    b: .word 9
    c: .word 2
    d: .word 2
```

0x00010074	00	00	00	06
0x00010070	00	00	00	02
0x0001006c	00	00	00	02
0x00010068	00	00	00	09
0x00010064	00	00	00	01

data
▲
▼



Test Pattern 2: Fact

- ◆ Modified from lecture slides
- ◆ The procedure loads `n` from `0x0001006c`, and stores the result to `0x00010070`
- ◆ Run simulation:
 - ◆ `> ncverilog Final_tb.v +define+fact +access+r`

```
def fact(n):  
    if n < 1:  
        return 1  
    else:  
        return n*fact(n-1)
```

```
.data  
n: .word 3
```

0x00010070	00	00	00	06
0x0001006c	00	00	00	03
data ▼ ▲ ▼				



(Bonus) Test Pattern 3: HW1 (1/3)

◆ Design your assembly first (hw1.s)

- ◆
$$T(n) = \begin{cases} 4T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2n, & n \geq 2 \\ 1, & n = 1 \end{cases}$$
- ◆ Example: $T(10) = 188, T(20) = 792$
- ◆ **Use recursive function**

FUNCTION:

Todo: define your own function in HW1

Do NOT modify this part!!!

__start:

```
la    t0, n
lw    x10, 0(t0)
jal   x1, FUNCTION
la    t0, n
sw    x10, 4(t0)
addi  a0, x0, 10
ecall
```



(Bonus) Test Pattern 3: HW1 (2/3)

- ◆ Go to simulator
- ◆ Dump code → binary file

```

1  0x00000317
2  0x00830067
3  0x00000297
4  0x02428293
5  0x0002a503
6  0xff5ff0ef
7  0x00000297
8  0x01428293
9  0x00a2a223
10 0x00a00513
11 0x00000073
  
```


<div> dump code </div>				
Bkpt	text address	Machine Code	used inst.	Source Code
<input type="checkbox"/>	0x00010000	0x00000317	auipc x6, 0	auipc x6, 0
<input type="checkbox"/>	0x00010004	0x00830067	jalr x0, x6, 8	jalr x0, x6, 8
<input type="checkbox"/>	0x00010008	0x00000297	auipc x5, 0	la t0, n
<input type="checkbox"/>	0x0001000c	0x02428293	addi x5, x5, 36	la t0, n
<input type="checkbox"/>	0x00010010	0x0002a503	lw x10, x5, 0	lw x10, 0(t0)
<input type="checkbox"/>	0x00010014	0xff5ff0ef	jal x1, -12	jal x1, FUNCTION
<input type="checkbox"/>	0x00010018	0x00000297	auipc x5, 0	la t0, n
<input type="checkbox"/>	0x0001001c	0x01428293	addi x5, x5, 20	la t0, n
<input type="checkbox"/>	0x00010020	0x00a2a223	sw x5, x10, 4	sw x10, 4(t0)
<input type="checkbox"/>	0x00010024	0x00a00513	addi x10, x0, 10	addi a0, x0, 10
<input type="checkbox"/>	0x00010028	0x00000073	ecall	ecall



(Bonus) Test Pattern 3: HW1 (3/3)

- ◆ Modify the code and save as: `./Verilog/hw1/hw1_text.txt`
- ◆ Test pattern generation: `./Verilog/hw1/hw1_gen.py`
- ◆ Run simulation:
 - ◆ `> ncverilog Final_tb.v +define+hw1 +access+r`

Delete

1	0x00000317		1	00000317
2	0x00830067		2	00830067
3	0x00000297		3	00000297
4	0x02428293		4	02428293
5	0x0002a503		5	0002a503
6	0xff5ff0ef		6	ff5ff0ef
7	0x00000297		7	00000297
8	0x01428293		8	01428293
9	0x00a2a223		9	00a2a223
10	0x00a00513			
11	0x00000073			



Pattern Generation

- ◆ Three python codes provided:
 - ◆ leaf_gen.py
 - ◆ fact_gen.py
 - ◆ hw1_gen.py

- ◆ TA will change the variables in *_gen.py to generate new test patterns when testing your CPU design



Coding Style Check

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
alu_in_reg	Flip-flop	32	Y	N	Y	N	N	N	N
counter_reg	Flip-flop	5	Y	N	Y	N	N	N	N
shreg_reg	Flip-flop	64	Y	N	Y	N	N	N	N
state_reg	Flip-flop	2	Y	N	Y	N	N	N	N

- ◆ All sequential elements must be **flip-flops**
- ◆ Check by Design Compiler
- ◆ Command:
 - ◆ > dc_shell
 - ◆ dc_shell> read_verilog CHIP.v
- ◆ Exit:
 - ◆ dc_shell> exit



Report

- ◆ Briefly describe your CPU architecture
 - ◆ Describe how you design the data path of instructions not referred in the lecture slides (jal, jalr, auipc, ...)
 - ◆ Describe how you handle multi-cycle instructions (mul)
 - ◆ Record total simulation time (CYCLE = 10 ns)
 - ◆ Leaf: $a = 1, b = 9, c = 2, d = 2$
 - ◆ Fact: $n = 10$
 - ◆ (Bonus) HW1: $n = 10$
- ```
Simulation complete via $finish(1) at time 4795 NS + 0
```
- ◆ Describe your observation
  - ◆ Snapshot the “Register table” in Design Compiler (p. 22)
  - ◆ List a work distribution table



# Submission

- ◆ Deadline: 7/03 (Fri.) 13:00
  - ◆ Late submission: 20 % reduction per day
- ◆ Upload Final\_group\_<group\_id>.zip to ceiba
  - ◆ Final\_group\_<group\_id>.zip
    - Final\_group\_<group\_id>/
    - Final\_group\_<group\_id>/CHIP.v
    - Final\_group\_<group\_id>/hw1.s (bonus)
    - Final\_group\_<group\_id>/hw1\_text.txt (bonus)
    - Final\_group\_<group\_id>/report.pdf
- ◆ Example

```
[r08943003@eda1 ~]$ unzip Final_group_0.zip
Archive: Final_group_0.zip
 creating: Final_group_0/
 inflating: Final_group_0/CHIP.v
 inflating: Final_group_0/hw1.s
 inflating: Final_group_0/hw1_text.txt
 extracting: Final_group_0/report.pdf
```





# Score

---

## ◆ Simulation: 70 % (+ bonus 10 %)

### ◆ Leaf

- Default: 15 %
- Change test pattern: 15 %

### ◆ Fact

- Default: 20 %
- Change test pattern: 20 %

### ◆ HW1 (bonus)

- Default: 5 %
- Change test pattern: 5 %

## ◆ Report: 30 %

- ◆ Content: 20 %
- ◆ Snapshots: 5 %
- ◆ Work distribution: 5 %