Computer Architecture
Final Project: Single Cycle CPU

TA1: Chun-Yen Yao (姚鈞嚴)
r08943003@ntu.edu.tw
TA2: Ya-Cheng Ko (柯亞承)
ycko@eecs.ee.ntu.edu.tw

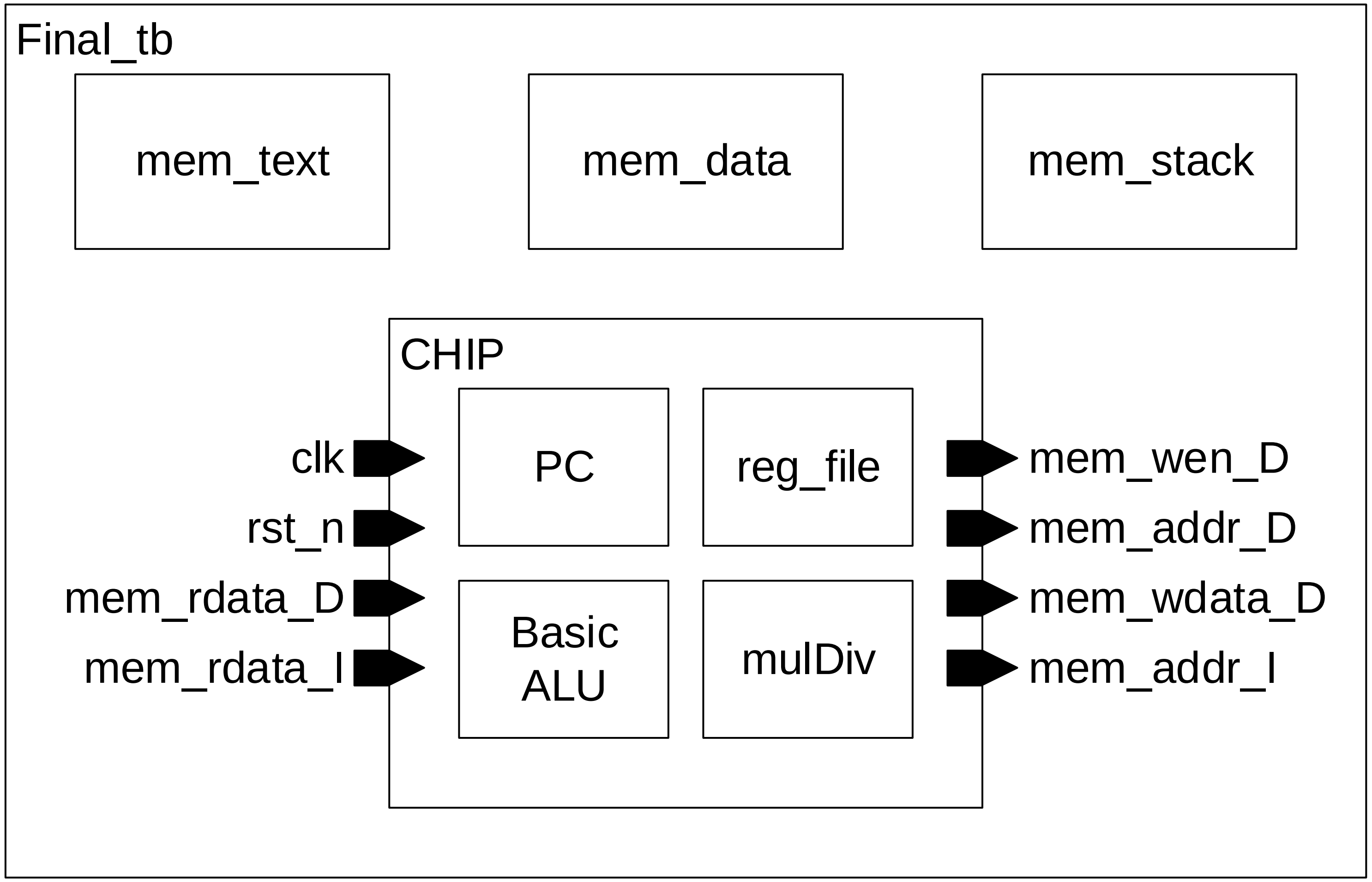
Announcement

- 1 ~ 3 people / group
- Please find a representative to fill out the google form before 8:00, 5/25 (Tue.)
  - https://forms.gle/MjAAk5jwLtCTjq216
- TA will help you find group members if you can not find any partner
  - Select “徵隊友” in the form
- The final member list will be announced before 23:59, 5/28 (Fri.)
  - Those who do not response will be regarded as one people in one group

Goal

- Implement a single cycle CPU
- Add multiplication/division unit (mulDiv) to CPU (HW2)
- Handle multi-cycle operations
- Get more familiar with assembly and Verilog
- Run your own assembly in HW1-1 on your CPU (Bonus)

Specification

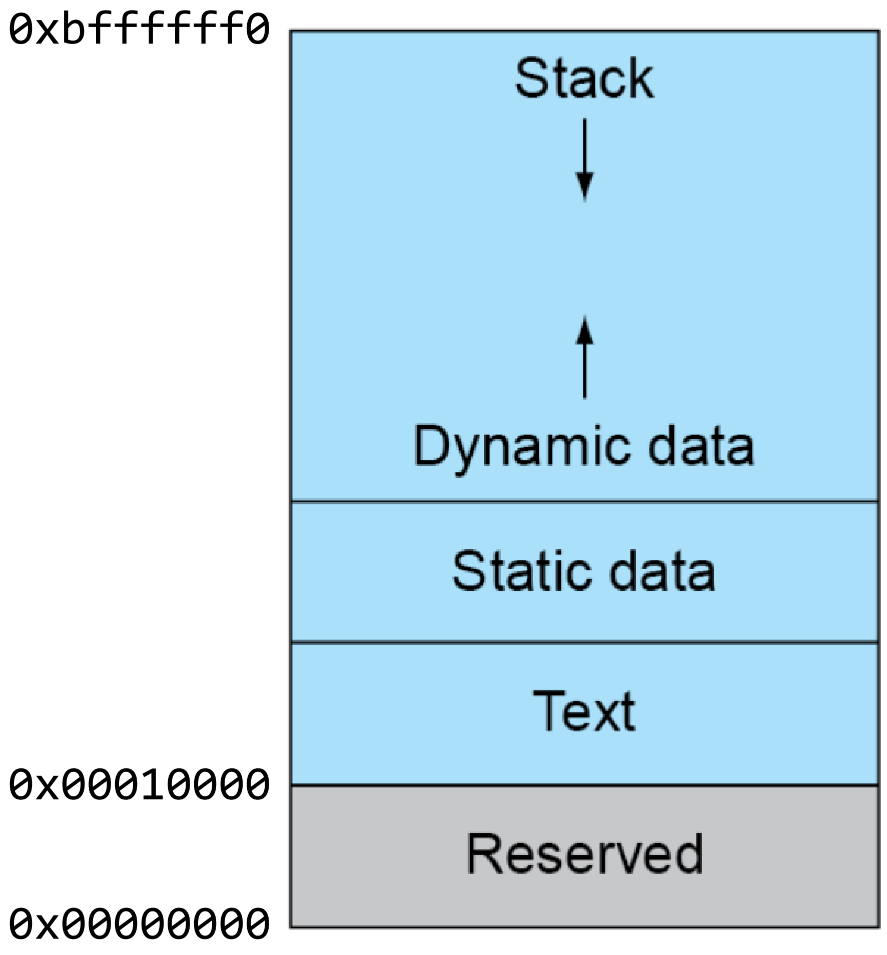


Port Definition

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

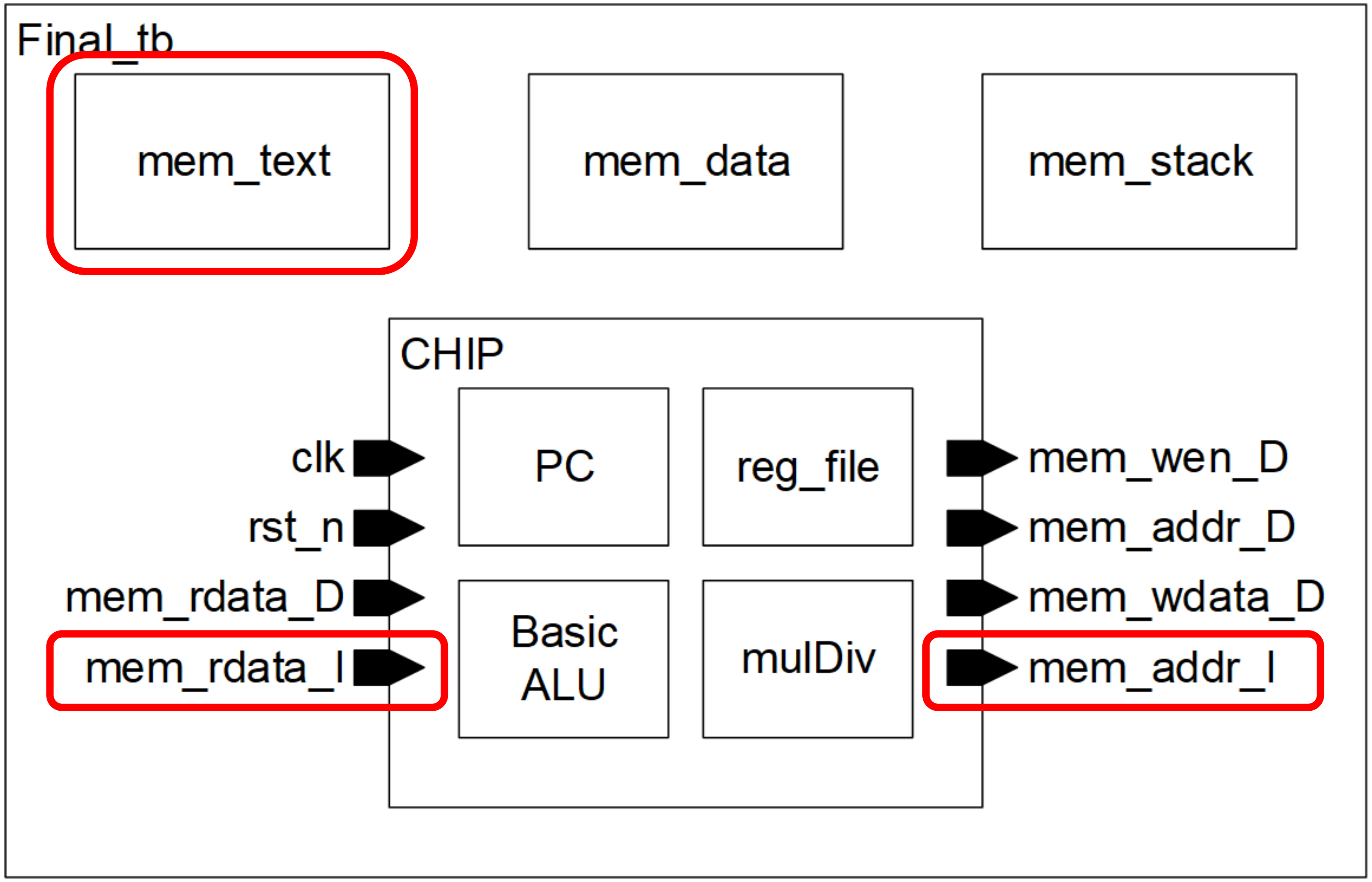
Memory Layout

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



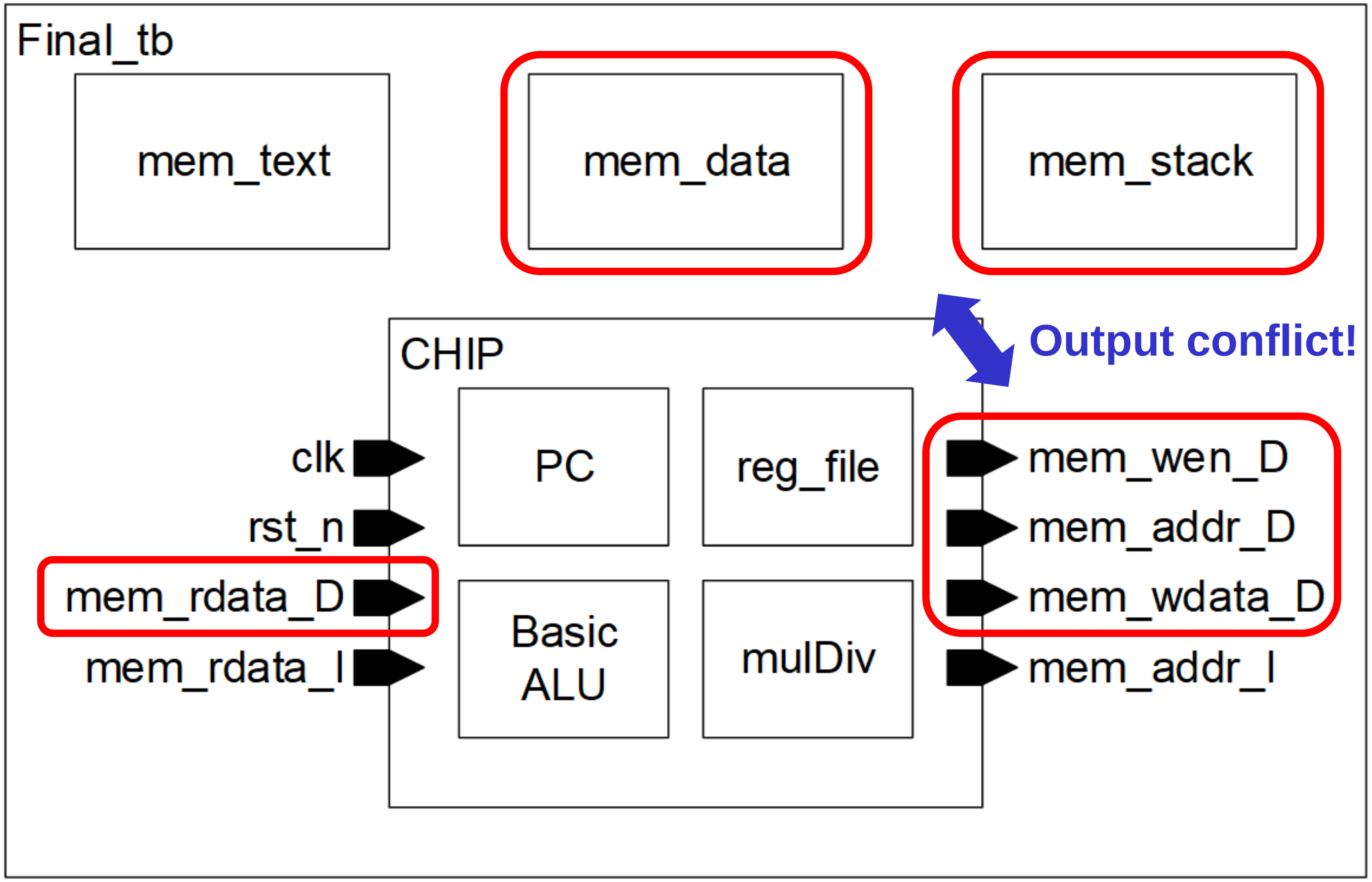
Relate Memory to Testbench (1/4)

- Instruction (text) memory



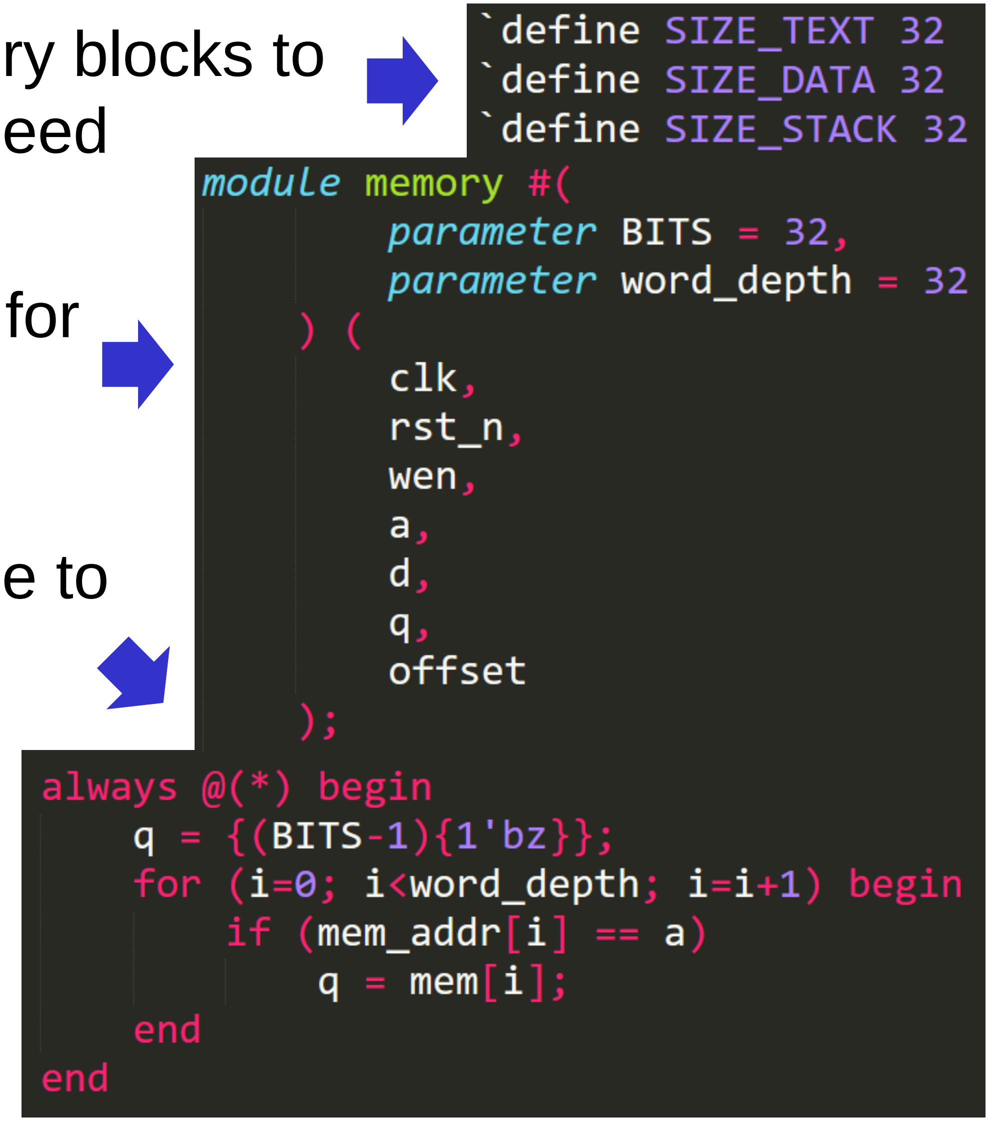
Relate Memory to Testbench (2/4)

- Data/stack memory



Relate Memory to Testbench (3/4)

- Reduce size of memory blocks to improve simulation speed
- Define offset address for each memory block
- Define high impedance to avoid output conflict
- **Not synthesizable coding style!**



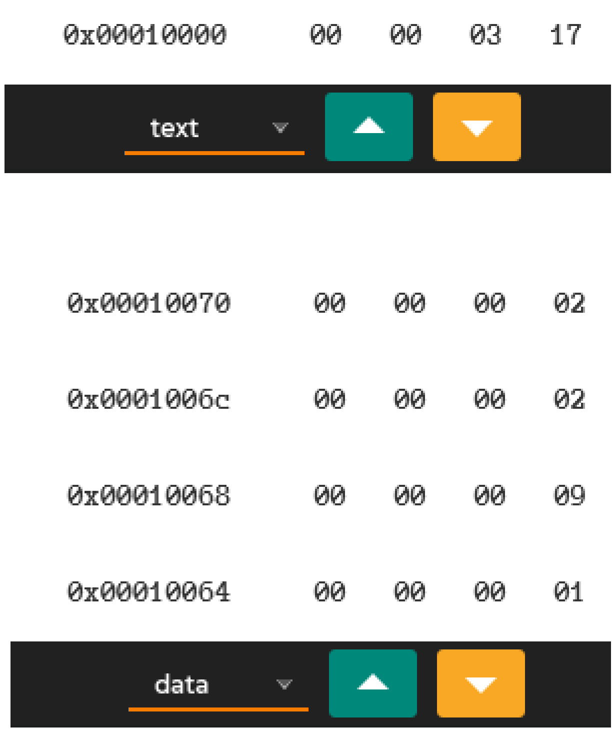
```
`define SIZE_TEXT 32
`define SIZE_DATA 32
`define SIZE_STACK 32
```

```
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
```

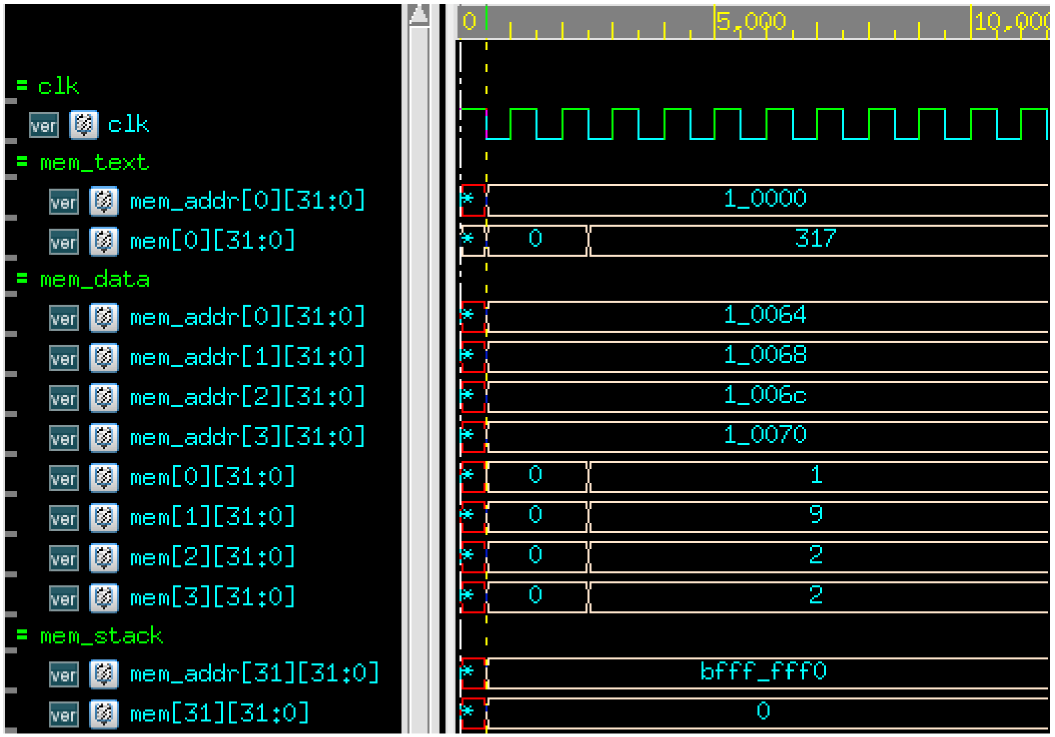
Relate Memory to Testbench (4/4)

- In Jupiter
- In Testbench



```
0x00010000    00   00   03   17
```

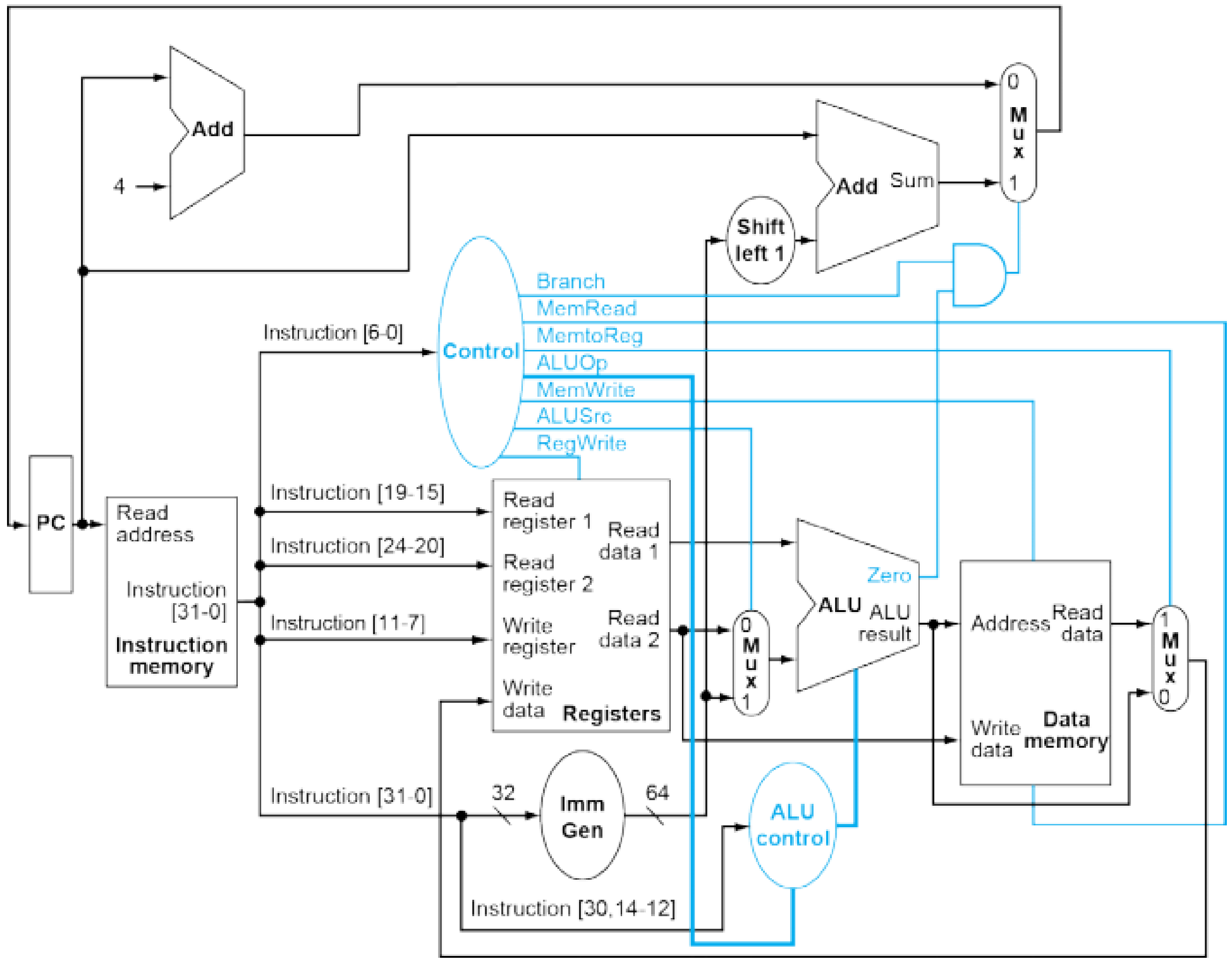
```
0x00010070    00   00   00   02
0x0001006c    00   00   00   02
0x00010068    00   00   00   09
0x00010064    00   00   00   01
```



```
= clk
  clk
= mem_text
  mem_addr[0][31:0]    1_0000
  mem[0][31:0]         0 | 317
= mem_data
  mem_addr[0][31:0]    1_0064
  mem_addr[1][31:0]    1_0068
  mem_addr[2][31:0]    1_006c
  mem_addr[3][31:0]    1_0070
  mem[0][31:0]         0 | 1
  mem[1][31:0]         0 | 9
  mem[2][31:0]         0 | 2
  mem[3][31:0]         0 | 2
= mem_stack
  mem_addr[31][31:0]   bfff_fff0
  mem[31][31:0]        0
```

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”

31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	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`"

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
MULDIV	multiplier	multiplicand	MUL/MULH[[S]U]	dest	OP

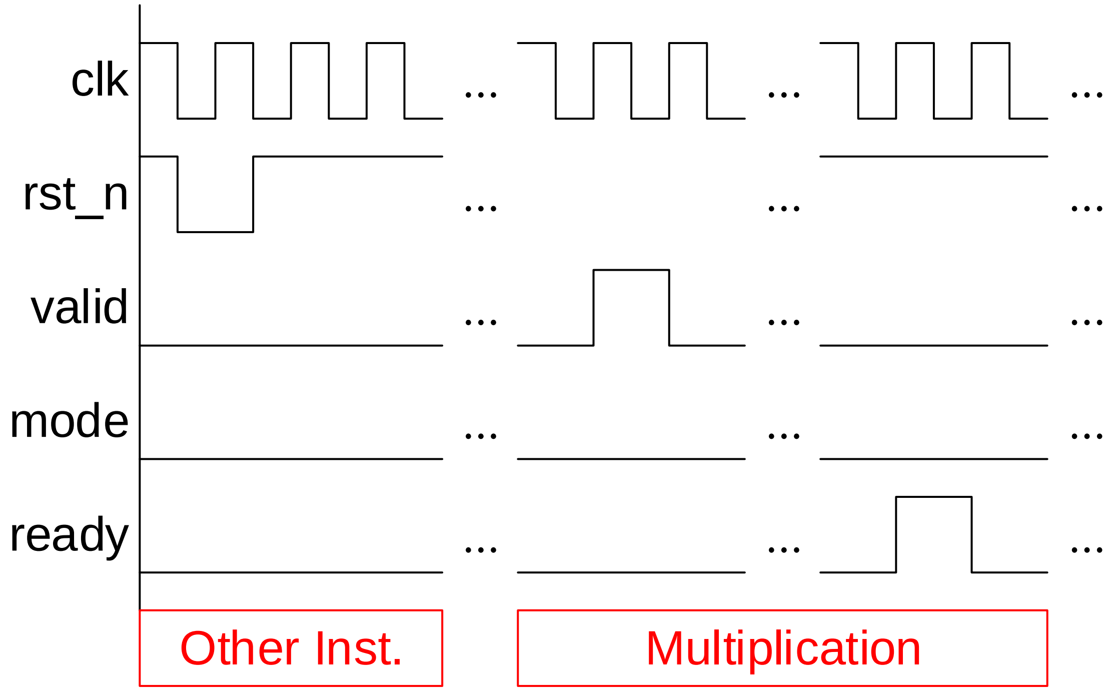
- Not included in RV32I
- Store the lower 32-b result (rs1 × rs2) to rd

- Example: mul x10, x10, x6
  - x10 = 0x00000001, x6 = 0x00000002
  - 0x00000001 × 0x00000002 = 0x00000002
  - Store 0x00000002 in x10

- **Your mulDiv can support this instruction!**

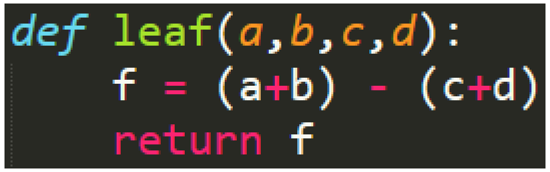
Multi-Cycle Operation

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

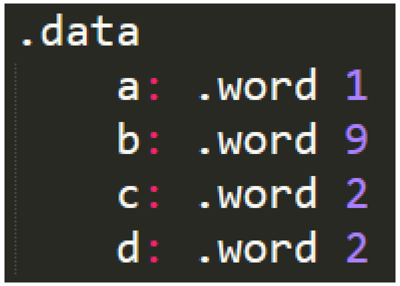


Test Pattern 1: Leaf Example

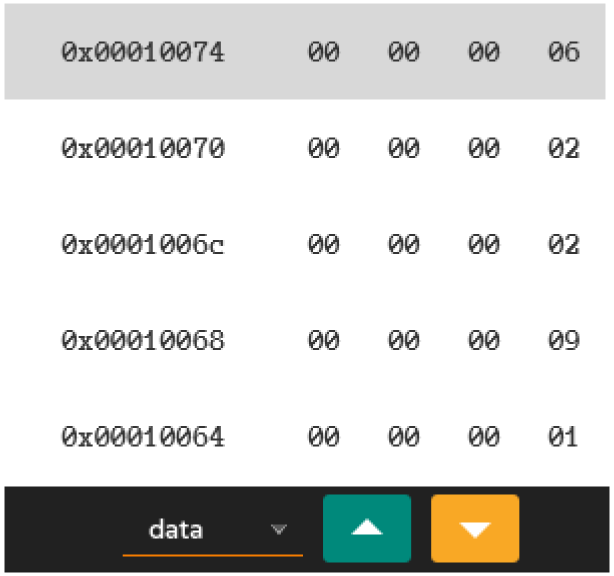
- Modified from lecture slides
- The procedure loads a,b,c,d from 0x00010064–0x00010070, and stores the result to 0x00010074
- Run simulation:
  - `$ ncverilog Final_tb.v +define+leaf +access+r`



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



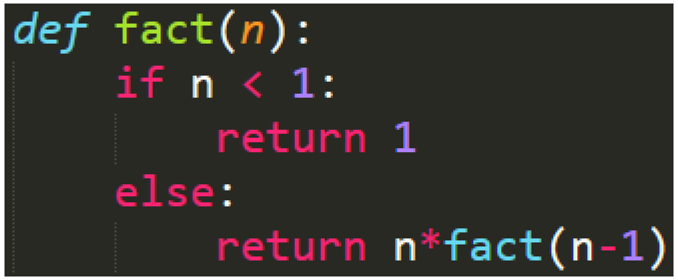
```
.data
    a:  .word 1
    b:  .word 9
    c:  .word 2
    d:  .word 2
```



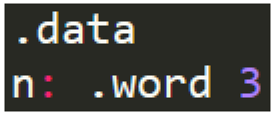
```
0x00010074    00    00    00    06
0x00010070    00    00    00    02
0x0001006c    00    00    00    02
0x00010068    00    00    00    09
0x00010064    00    00    00    01
```

Test Pattern 2: Fact

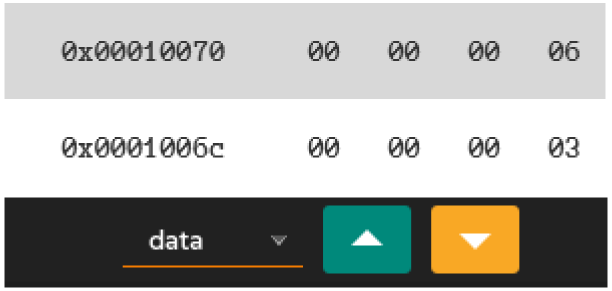
- Modified from lecture slides
- The procedure loads n from 0x0001006c, and stores the result to 0x00010070
- Run simulation:
  - `$ ncverilog Final_tb.v +define+fact +access+r`



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



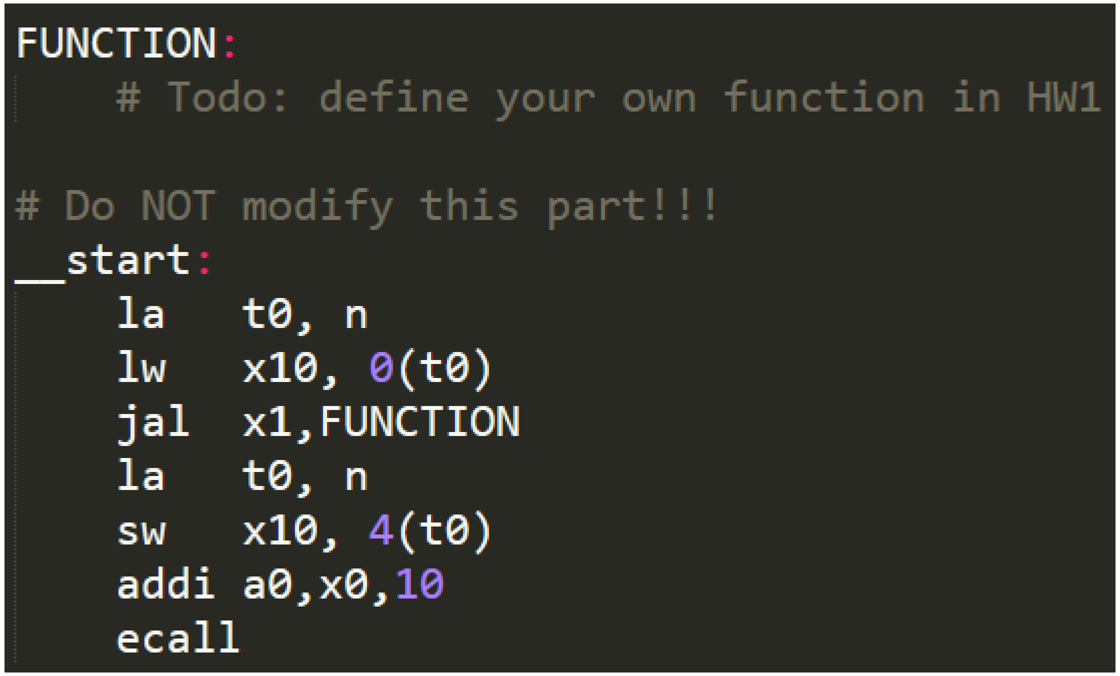
```
.data
n:  .word 3
```



```
0x00010070    00    00    00    06
0x0001006c    00    00    00    03
```

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

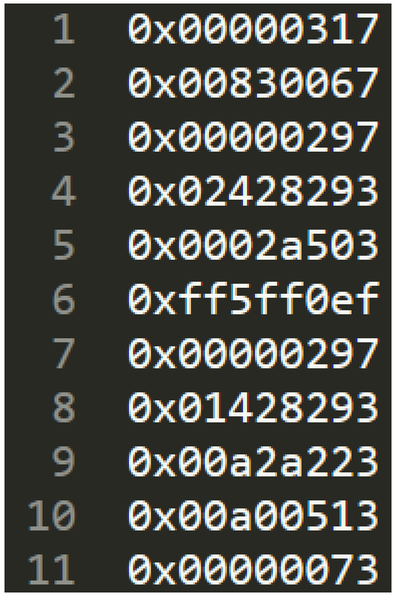
- Design your assembly first (hw1.s)
  - $T(n) = \begin{cases} 8T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 4n, & n \geq 2 \\ 7, & n = 1 \end{cases}$
  - Example: $T(7) = 572, T(77) = 2187508$
  - **Use recursive function**



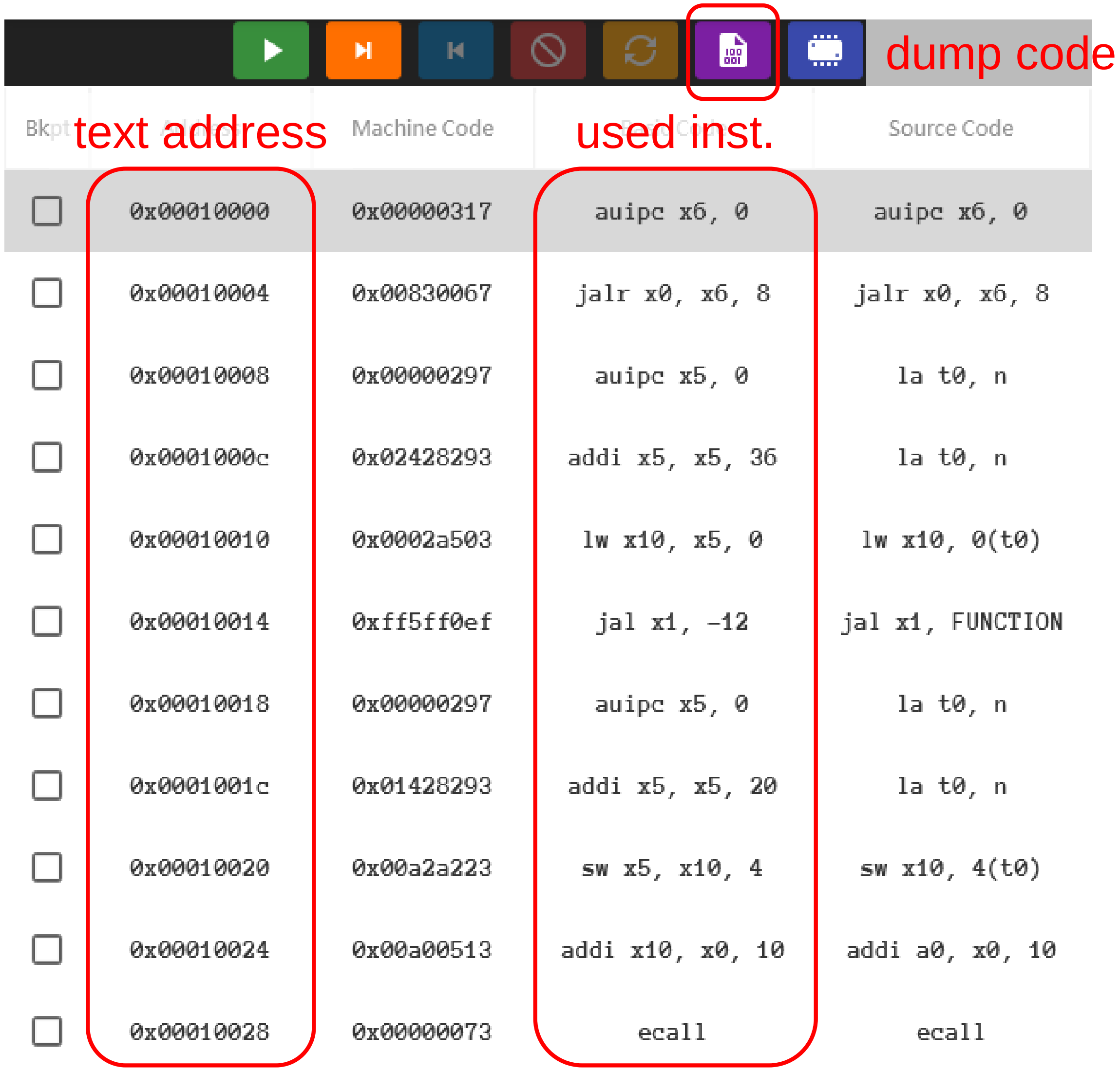
```
FUNCTION:
    # Todo: define your own function in HW1

# 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



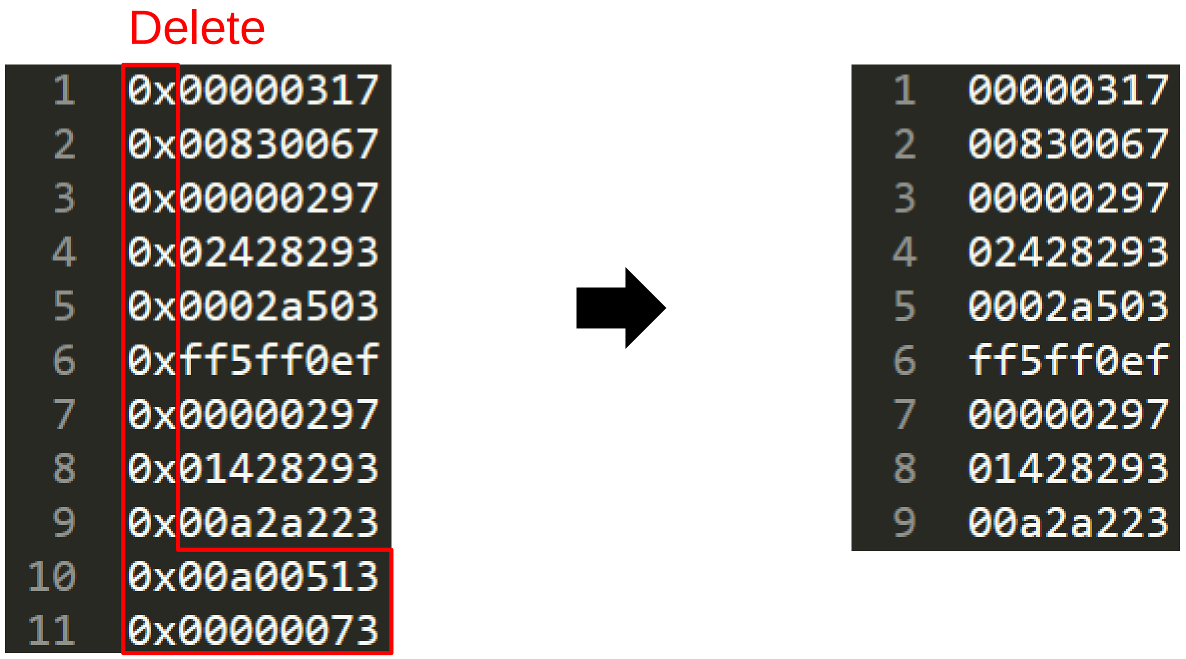
```
0x00000317
0x00830067
0x00000297
0x02428293
0x0002a503
0xff5ff0ef
0x00000297
0x01428293
0x00a2a223
0x00a00513
0x00000073
```



Bkpt	Address	Machine Code	Basic Code	Source Code
☐	0x00010000	0x00000317	auipc x6, 0	auipc x6, 0
☐	0x00010004	0x00830067	jalr x0, x6, 8	jalr x0, x6, 8
☐	0x00010008	0x00000297	auipc x5, 0	la t0, n
☐	0x0001000c	0x02428293	addi x5, x5, 36	la t0, n
☐	0x00010010	0x0002a503	lw x10, x5, 0	lw x10, 0(t0)
☐	0x00010014	0xff5ff0ef	jal x1, -12	jal x1, FUNCTION
☐	0x00010018	0x00000297	auipc x5, 0	la t0, n
☐	0x0001001c	0x01428293	addi x5, x5, 20	la t0, n
☐	0x00010020	0x00a2a223	sw x5, x10, 4	sw x10, 4(t0)
☐	0x00010024	0x00a00513	addi x10, x0, 10	addi a0, x0, 10
☐	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`



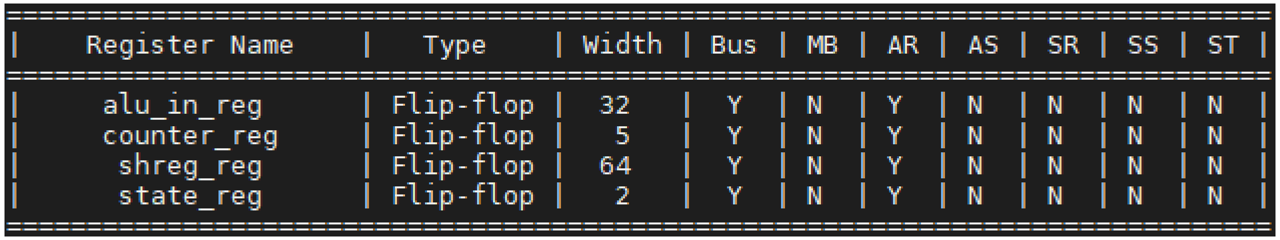
```
0x00000317
0x00830067
0x00000297
0x02428293
0x0002a503
0xff5ff0ef
0x00000297
0x01428293
0x00a2a223
0x00a00513
0x00000073
```

```
00000317
00830067
00000297
02428293
0002a503
ff5ff0ef
00000297
01428293
00a2a223
```

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

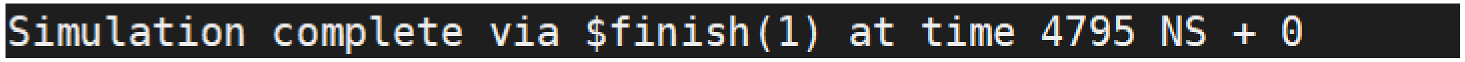


```
===============================================================================
|     Register Name      |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|      alu_in_reg        | Flip-flop |  32   |  Y  | N  | 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:
  - `$ dv -no_gui`
  - `design_vision> read_verilog CHIP.v`
- Exit:
  - `design_vision> 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 = 0, b = 6, c = 1, d = 5
  - Fact: n = 3
  - (Bonus) HW1: n = 7

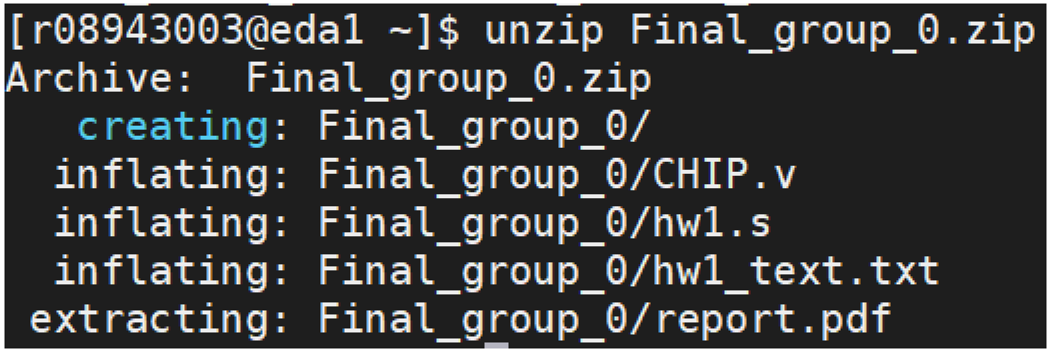


```
Simulation complete via $finish(1) at time 4795 NS + 0
```

- Describe your observation
- Snapshot the “Register table” in Design Compiler (p. 22)
- List a work distribution table

Submission

- Deadline: 6/15 (Tue.) 8:00 am
  - 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 20 %)
  - Leaf
    - Default: 15 %
    - Change test pattern: 15 %
  - Fact
    - Default: 20 %
    - Change test pattern: 20 %
  - HW1 (bonus)
    - Default: 10 %
    - Change test pattern: 10 %
- Report: 30 %
  - Content: 20 %
  - Snapshots: 5 %
  - Work distribution: 5 %