Computer Architecture
Final project: CPU

TA 蔡東翰 r12943012@ntu.edu.tw
TA 謝郡軒 r12943006@ntu.edu.tw
TA 曾品珊 pstseng@eecs.ee.ntu.edu.tw
TA 余岳龍 ylyu@eecs.ee.ntu.edu.tw

Due 23:59, 2024/6/14(Fri.)

Outline

- Announcement & Data Preparation
- Goal & Specifications
- Test Pattern
- Simulation
- Synthesizable Coding Style Check
- Report
- Submission
- Grading Policy
- Appendix

Outline

- Announcement & Data Preparation
- Goal & Specifications
- Test Pattern
- Simulation
- Synthesizable Coding Style Check
- Report
- Submission
- Grading Policy
- Appendix

Announcement

- 1 ~ 2 people / group
- Please find a representative to fill out the google form before 23:59, 5/10(Fri.)
  - https://forms.gle/MqinM4ZL8rbUYVuQ8
  - 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, 5/13 (Wed.)
  - Those who do not response will be regarded as one people in one group

Data Preparation

- Decompress CA_Final.zip
- Directory hierarchy:
  - 00_TB/
    - tb.v ➔ testbench file
    - Memory.v ➔ memory file
    - Pattern/ ➔ test pattern directory
  - 01_RTL/
    - 00_license.f ➔ EDA tool license source command
    - 01_run.f ➔ vcs/ncverilog command
    - 99_clean_up.f ➔ Command to clean temporary data
    - CHIP.v ➔ Your design
  - 02_Assembly/ ➔ Assembly files directory
  - 03_Python/ ➔ Pattern generator files directory

Outline

- Announcement & Data Preparation
- Goal & Specifications
- Test Pattern
- Simulation
- Synthesizable Coding Style Check
- Report
- Submission
- Grading Policy
- Appendix

Goal

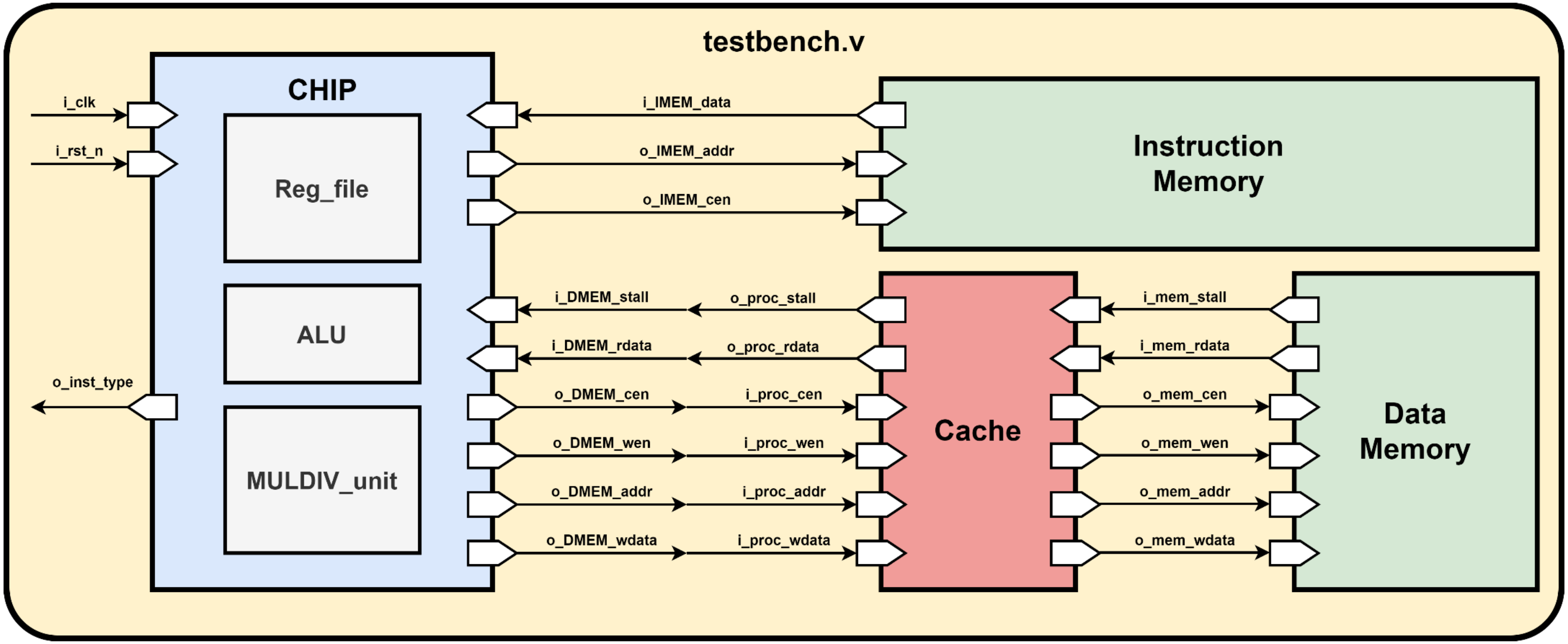
- Implement a CPU
- Add multiplication/division unit (mulDiv) to CPU (HW2)
- Handle multi-cycle operations
- Get more familiar with assembly and Verilog

- BONUS:
  - Implement L1 cache
  - What benefit cache brings from

Supporting Instructions

- Your design must **<u>at least</u>** support
  - `auipc, jal, jalr`
  - `add, sub, and, xor`
  - `addi, slli, slti, srai`
  - `lw, sw`
  - `mul`
  - `beq, bge, blt, bne`
  - `Ecall (the end of program)`
- See "Instruction_Set_Listings.pdf" for more information of machine code

Block Diagram



Specification – CHIP I/O

Signal Name	I/O	Width	Description
i_clk	I	1	Clock signal
i_rst_n	I	1	Active **low** asynchronous reset
i_IMEM_data	I	32	Instruction binary code
o_IMEM_addr	O	32	PC address
o_IMEM_cen	O	1	Set **high** to load instruction
i_DMEM_stall	I	1	Active **high** control signal that asks processor to wait
i_DMEM_rdata	I	32	32-bit output data
o_DMEM_cen	O	1	Set **high** to enable memory functions
o_DMEM_wen	O	1	Set **high** for write, **low** for read
o_DMEM_addr	O	32	Data memory address
o_DMEM_wdata	O	32	Data for writing to data memory
o_finish	O	1	Set **high** for finishing the procedure
i_cache_finish	I	1	Finish signal from cache
o_proc_finish	O	1	Finish signal to cache

Specification – CHIP I/O

- Do not modify the I/O interface!!

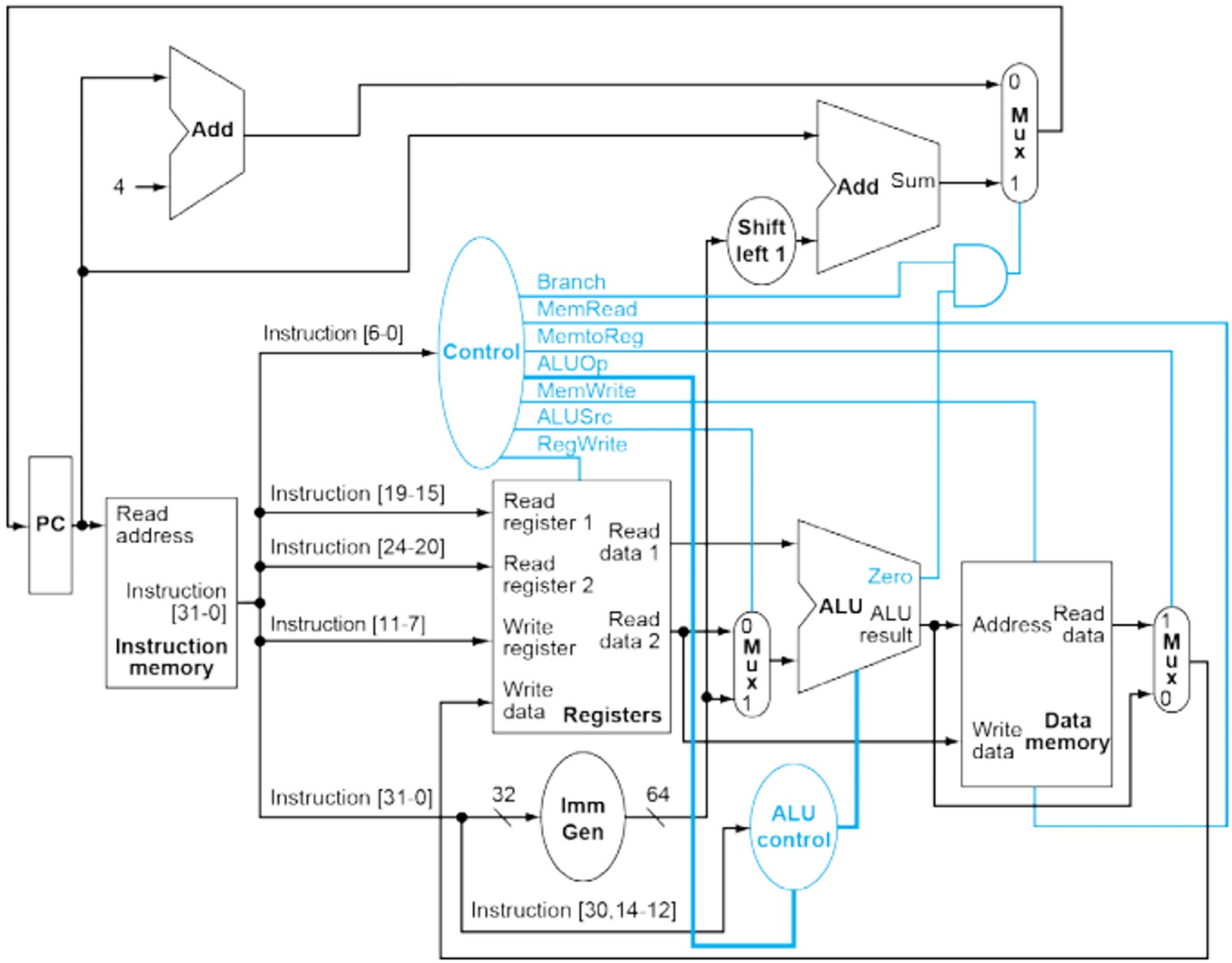
```
//------------------------- DO NOT MODIFY THE I/O INTERFACE!! ---------------------------//
module CHIP #(                                                                           //
    parameter BIT_W = 32                                                                 //
)(                                                                                       //
    // clock                                                                             //
        input               i_clk,                                                       //
        input               i_rst_n,                                                     //
    // instruction memory                                                                //
        input  [BIT_W-1:0]  i_IMEM_data,                                                 //
        output [BIT_W-1:0]  o_IMEM_addr,                                                 //
        output              o_IMEM_cen,                                                  //
    // data memory                                                                       //
        input               i_DMEM_stall,                                                //
        input  [BIT_W-1:0]  i_DMEM_rdata,                                                //
        output              o_DMEM_cen,                                                  //
        output              o_DMEM_wen,                                                  //
        output [BIT_W-1:0]  o_DMEM_addr,                                                 //
        output [BIT_W-1:0]  o_DMEM_wdata,                                                //
    // finnish procedure                                                                 //
        output              o_finish,                                                    //
    // cache                                                                             //
        input               i_cache_finish,                                              //
        output              o_proc_finish                                                //
);                                                                                       //
//------------------------- DO NOT MODIFY THE I/O INTERFACE!! ---------------------------//
```

Specification – Other Description

- All inputs are synchronized with the negative edge clock.
- All outputs should be synchronized at clock rising edge.
- You should reset all your outputs when i_rst_n is **low**. Active low asynchronous reset is used and only once.
- The runtime of the design should be within 10000 cycles
- **The operators “ * ” and “ / “ are forbidden except for index**

Review – Architecture

- Not complete (does not include jal, jalr, …)



TODO

- Parameters declaration
  - Instructions
  - Opcode
  - …

```
// ----------------------------------
// Parameters
// ----------------------------------

    // TODO: any declaration
```

- Wires & Registers declaration
  - PC
  - …

```
// ----------------------------------
// Wires and Registers
// ----------------------------------

    // TODO: any declaration
        reg [BIT_W-1:0] PC, next_PC;
```

TODO

- Continuous Assignment
  - …
- Submodules
  - Register file
  - ALU
  - …

```
// ------------------------------------
// Continuous Assignment
// ------------------------------------

    // TODO: any wire assignment

// ------------------------------------
// Submoddules
// ------------------------------------

    // TODO: Reg_file wire connection
    Reg_file reg0(
        .i_clk  (i_clk),
        .i_rst_n(i_rst_n),
        .wen    (),
        .rs1    (),
        .rs2    (),
        .rd     (),
        .wdata  (),
        .rdata1 (),
        .rdata2 ()
    );
```

Register File

- Do not modify this part !!!
- Initial values
  - X0 stores constant 0
  - X2 stores stack pointer
  - X3 stores global pointer
  - Others are 0

```verilog
module Reg_file(i_clk, i_rst_n, wen, rs1, rs2, rd, wdata, rdata1, rdata2);

    parameter BITS = 32;
    parameter word_depth = 32;
    parameter addr_width = 5; // 2^addr_width >= word_depth

    input i_clk, i_rst_n, wen; // wen: 0:read | 1:write
    input [BITS-1:0] wdata;
    input [addr_width-1:0] rs1, rs2, rd;

    output [BITS-1:0] rdata1, rdata2;

    reg [BITS-1:0] mem [0:word_depth-1];
    reg [BITS-1:0] mem_nxt [0:word_depth-1];

    integer i;

    assign rdata1 = mem[rs1];
    assign rdata2 = mem[rs2];

    always @(*) begin
        for (i=0; i<word_depth; i=i+1)
            mem_nxt[i] = (wen && (rd == i)) ? wdata : mem[i];
    end

    always @(posedge i_clk or negedge i_rst_n) begin
        if (!i_rst_n) begin
            mem[0] <= 0;
            for (i=1; i<word_depth; i=i+1) begin
                case(i)
                    32'd2: mem[i] <= 32'hbffffff0;
                    32'd3: mem[i] <= 32'h10008000;
                    default: mem[i] <= 32'h0;
                endcase
            end
        end
        else begin
            mem[0] <= 0;
            for (i=1; i<word_depth; i=i+1)
                mem[i] <= mem_nxt[i];
        end
    end
endmodule
```

TODO: Always blocks

- Combinational circuits
- Sequential circuits
- …

```
// ------------------------------------------------------------
// Always Blocks
// ------------------------------------------------------------

    // Todo: any combinational/sequential circuit

    always @(posedge i_clk or negedge i_rst_n) begin
        if (!i_rst_n) begin
            PC <= 32'h00010000; // Do not modify this value!!!
        end
        else begin
            PC <= next_PC;
        end
    end
endmodule
```

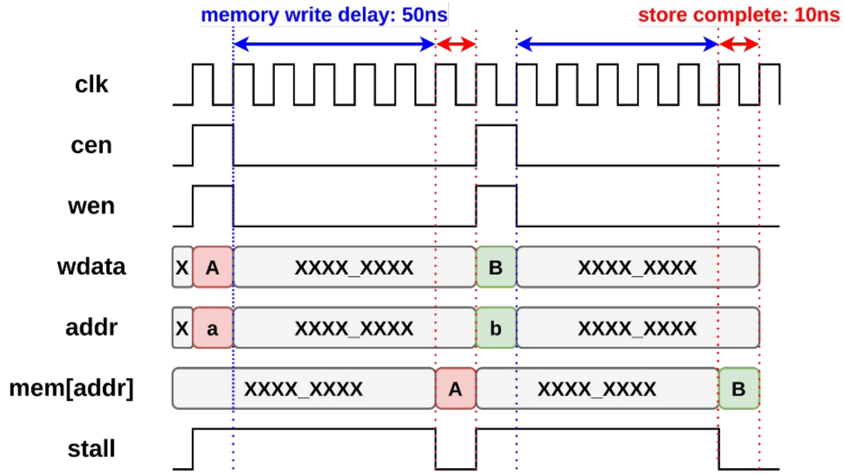
TODO: MUL

- Your HW2

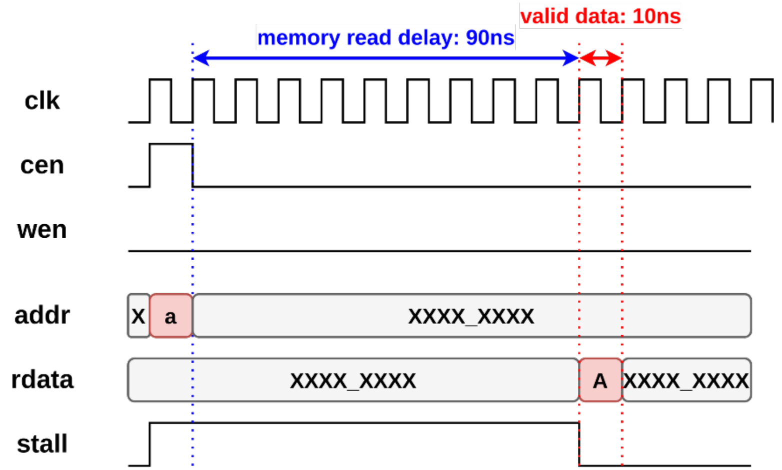
```
module MULDIV_unit(
    // TODO: port declaration
    );
    // Todo: HW2
endmodule
```

Supplement: Memory control signals

Function	cen	wen
Hold	0	X
Read	1	0
Write	1	1



Write wavefrom



Read wavefrom

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

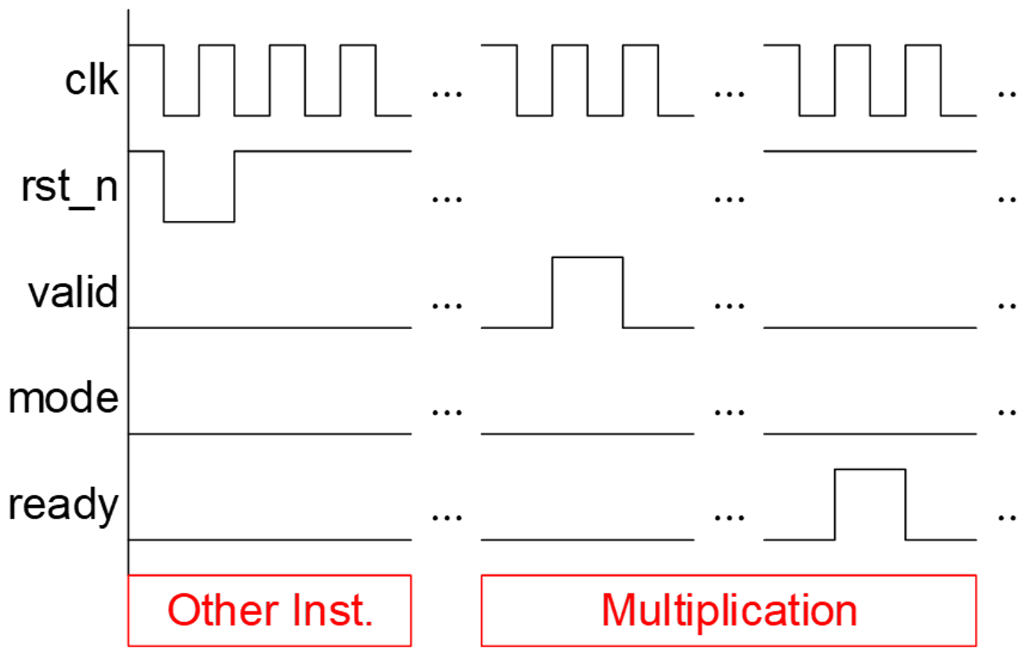
- Not included in RV32I
- Store the lower 32-b result (rs1 × rs2) to rd

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

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

Supplement: 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



Supplement: Instruction "ecall"

	31 25	24 20	19 15	14 12	11 7	6 0
I-type	imm[11:0]		rs1	funct3	rd	opcode

- Environment call: represent the end of procedure here
- Example:
  - `addi a0, x0, 10` → `00a00513`
  - `ecall` → `00000073 (machine code)`

- **Pull up** the signal **o_finish** when all tasks are done inside the processor and cache if implemented
- The testbench will check the answer and golden after **o_finish** is pull up

Outline

- Announcement & Data Preparation
- Goal & Specifications
- Test Pattern
- Simulation
- Synthesizable Coding Style Check
- Report
- Submission
- Grading Policy
- Appendix

Test Pattern I0: Leaf Example

- Modified from lecture slides
- The procedure loads a,b,c,d from 0x00010064 0x00010070, and stores the result to 0x00010074
- Simulation:

```
vcs ../00_TB/tb.v CHIP.v -full64 -R -\
debug_access+all +v2k +notimingcheck +define+I0
```

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

```
.data
    a: .word 5
    b: .word 6
    c: .word 8
    d: .word 0
.text
.globl __start
```

0x00010074	00	00	00	03
0x00010070	00	00	00	00
0x0001006c	00	00	00	08
0x00010068	00	00	00	06
0x00010064	00	00	00	05

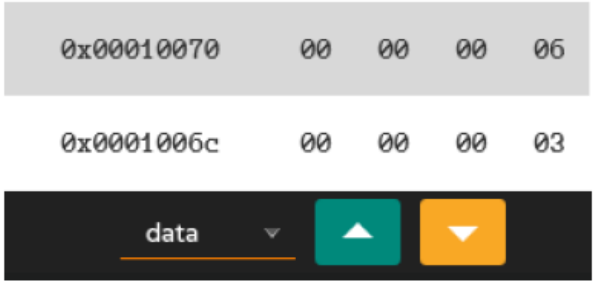
Test Pattern I1: Fact

- Modified from lecture slides
- The procedure loads n from 0x0001006c, and stores the result to 0x00010070
- Simulation:

```
vcs ../00_TB/tb.v CHIP.v -full64 -R -\
debug_access+all +v2k +notimingcheck +define+I1
```

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

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



Test Pattern I2: HW1

- Design your assembly first (hw1.s)
  - $$T(n) = \begin{cases} 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + 10n + 3, & if\ n \geq 4 \\ 3, & n = 1, 2, 3 \end{cases}$$
  - E.g., T(6) = 72, T(23) = 419
  - **Implement with recursive function only**
- The instructions should be generated by yourself to test this pattern, TA will run this part by TA's golden one.

```
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
```

Test Pattern I2: HW1

- Go to simulator
- Dump code → binary file

dump code

Bkpt	text address	Machine Code	used inst.	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

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

Test Pattern I2: HW1

- Modify the code and save as: 00_TB/Pattern/I2/mem_I.dat
- Test pattern generation:

Delete

```
0xfb5ff0ef
0x00000297
0x01428293
0x00a2a223
0x000a0513
0x00000073
```

```
fb5ff0ef
00000297
01428293
00a2a223
000a0513
00000073
```

- Simulation

```
vcs ../00_TB/tb.v CHIP.v -full64 -R -\
debug_access+all +v2k +notimingcheck +define+I2
```

Test Pattern I3: Sorting

- This procedure sorts N numbers and stores them in order back to memory
- The procedure loads N from 0x000100c0, and sorts numbers in memory banks start at 0x000100c4
- Simulation:

```
vcs ../00_TB/tb.v CHIP.v -full64 -R -\
debug_access+all +v2k +notimingcheck +define+I2
```

```python
def sort(v, n):
    for i in range(n):
        for j in range(i-1,-1,-1):
            if v[j] > v[j+1]:
                v[j], v[j+1] = v[j+1], v[j]
    return v
```

```
.data
    n:  .word 5
    a:  .word 3
    b:  .word 1
    c:  .word 5
    d:  .word 2
    e:  .word 4
```

0x000100d4	00	00	00	04
0x000100d0	00	00	00	02
0x000100cc	00	00	00	05
0x000100c8	00	00	00	01
0x000100c4	00	00	00	03
0x000100c0	00	00	00	05

data

0x000100d4	00	00	00	05
0x000100d0	00	00	00	04
0x000100cc	00	00	00	03
0x000100c8	00	00	00	02
0x000100c4	00	00	00	01
0x000100c0	00	00	00	05

data

Pattern Generation

- Three python codes provided:
  - I0_leaf_gen.py
  - I1_fact_gen.py
  - I2_hw1_gen.py
  - I3_sort_gen.py
- TA will change the variables in *_gen.py to generate new test patterns when testing your CPU design

Outline

- Announcement & Data Preparation
- Goal & Specifications
- Test Pattern
- Simulation
- Synthesizable Coding Style Check
- Report
- Submission
- Grading Policy
- Appendix

Simulation

- There are two files in folder named “code”
  - CHIP.v (your project)
  - tb.v (testbench)
  - memory.v (memory file)
- To run simulation, you should run source command in advance
  - `$ source 00_license.f` (use given file)

Simulation (cont.)

- Verilog simulation
  - `$ source 01_run.f [I0/I1/I2/I3]`
  - TA will run your code with following format of command in 01_run.sh :

```
vcs ../00_TB/tb.v CHIP.v -full64 -R -\
debug_access+all +define+$1 +v2k +notimingcheck
```

  - The word in the block "$1" is the instruction set of test pattern
    - Ex. `$ source 01_run.f I0`
  - Make sure to pass every given sets without any error messages.

Outline

- Announcement & Data Preparation
- Goal & Specifications
- Test Pattern
- Simulation
- Synthesizable Coding Style Check
- Report
- Submission
- Grading Policy
- Appendix

Synthesizable 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
- **Make sure there is no latches and errors in your design**
- Check by Design Compiler
- Command:
  - `$ dv -no_gui`
  - `design_vision> read_verilog HW2.v`
- Exit:
  - `design_vision> exit`

Outline

- Announcement & Data Preparation
- Goal & Specifications
- Test Pattern
- Simulation
- Synthesizable Coding Style Check
- Report
- Submission
- Grading Policy
- Appendix

Report

- Record the execution cycle number of each instruction set
  - Ex:

```
Success!
The test result is .....PASS :)
Total execution cycle :                                 131
```

Instruction Set	Execution cycle
I0	131
I1	…
I2	…
I3	…

- Snapshot the "Register table" in Design Compiler

Report

- Work description
  - Draw the block diagram of 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, div …)
  - Describe your observation
- [BONUS] Cache design
  - Briefly describe your cache architecture
  - Describe how your cache improves time performance
    - Ex:

Instruction Set	Without Cache	With Cache	Speedup
I0	75	75	1
I1	680	658	1.03
I2	202	180	1.12
I3	525	338	1.55

- List a work distribution table

Outline

- Announcement & Data Preparation
- Goal & Specifications
- Test Pattern
- Simulation
- Synthesizable Coding Style Check
- Report
- Submission
- Grading Policy
- Appendix

Submission

- **Deadline: 23:59, 2023/06/14(Fri.)**
  - Late submission: 50 % reduction per day
- Upload Final_group_<group_id>_vk.zip to NTUCOOL
  - (k is the number of version, k =1,2,…)
  - Final_group_<group_id>_vk.zip
    - Final_group_<group_id>/
      - CHIP.v
      - report.pdf
  - TA will only check the last version of your homework.

Outline

- Announcement & Data Preparation
- Goal & Specifications
- Test Pattern
- Simulation
- Synthesizable Coding Style Check
- Report
- Submission
- Grading Policy
- Appendix

Grading Policy

- Synthesizable design check before grading
- **(60%)** Test pattern
  - Each instruction set: 12%
    - Default: 8%
    - Change test pattern: 4 %
  - Hidden pattern: 12%
- **(20%)** Execution time performance
  - This would be graded after getting full credit from test pattern
- **(20% + 5%)** Report
  - 20% CHIP, 5% cache
- **(15% bonus)** cache implementation

Total 20% bonus for cache

- Other rules:
  - Lose **10-point** for any wrong format rule. Don’t compress all homework folder.

Dos and DONTs for the TAs

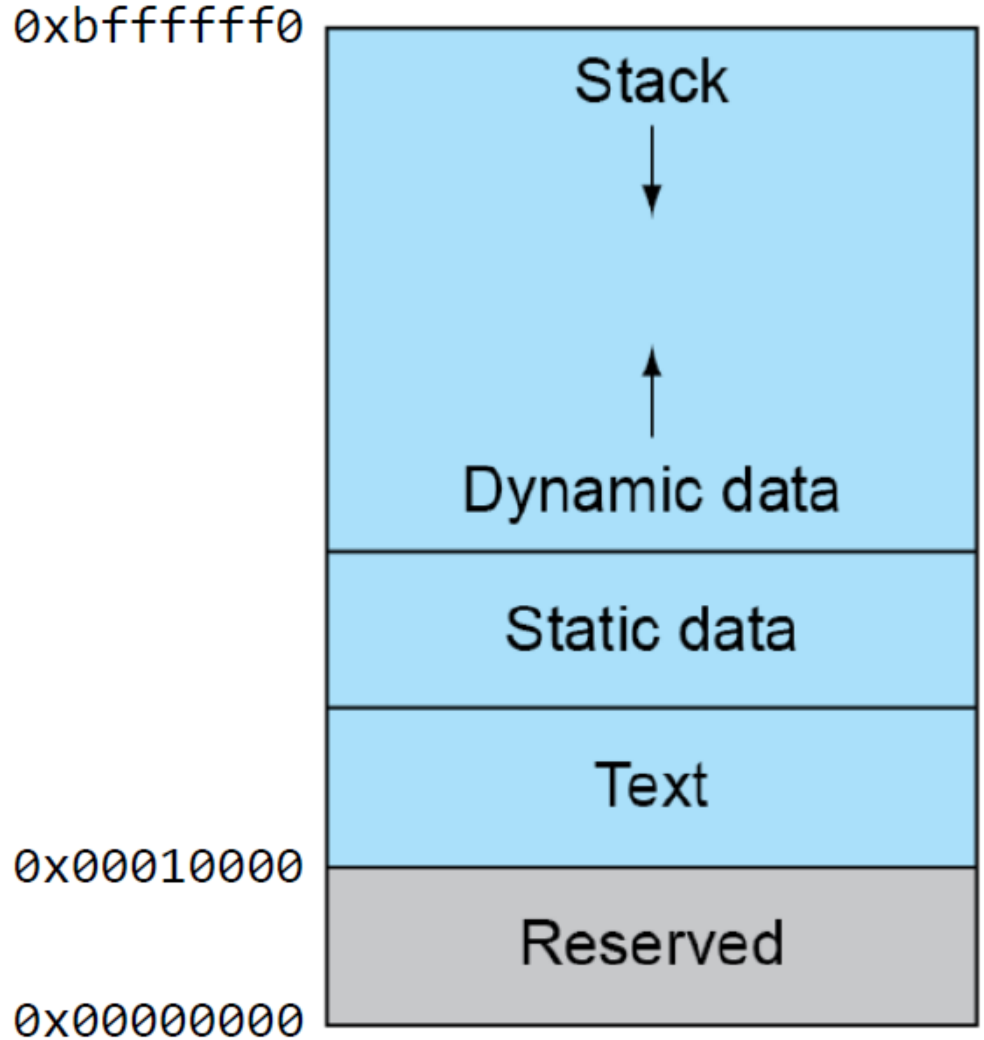
- TAs are happy to help, but they will NOT debug for you.
- TAs do NOT answer questions not related to the course.
- If you want to discuss with TAs face-to-face, please email the TAs to schedule an appointment instead of stopping by the lab directly.

Cache Implementation & Memory Layout

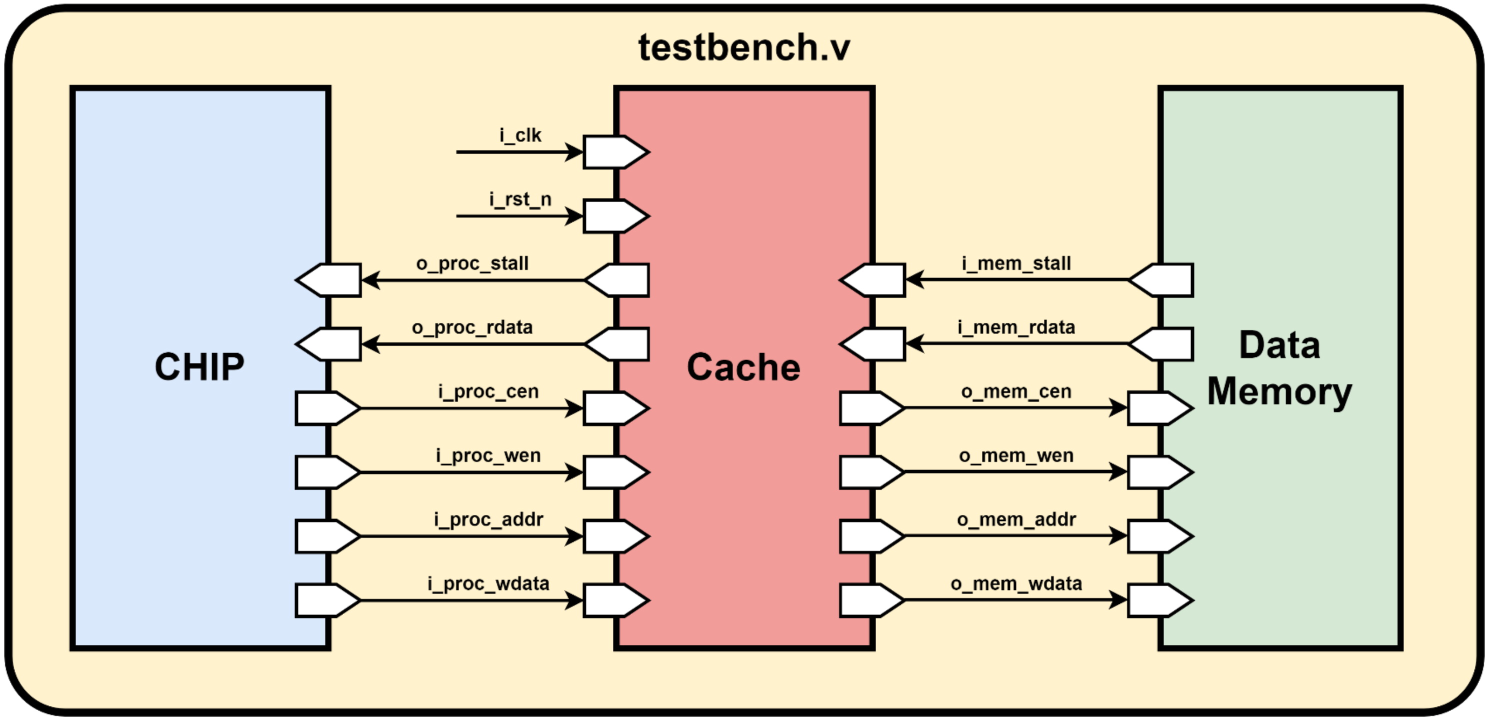
APPENDIX

Memory Layout

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



Specification – Cache I/O



Signal Name	I/O	Width	Description
i_clk	I	1	Clock signal
i_rst_n	I	1	Active **low** asynchronous reset

Specification – Cache I/O

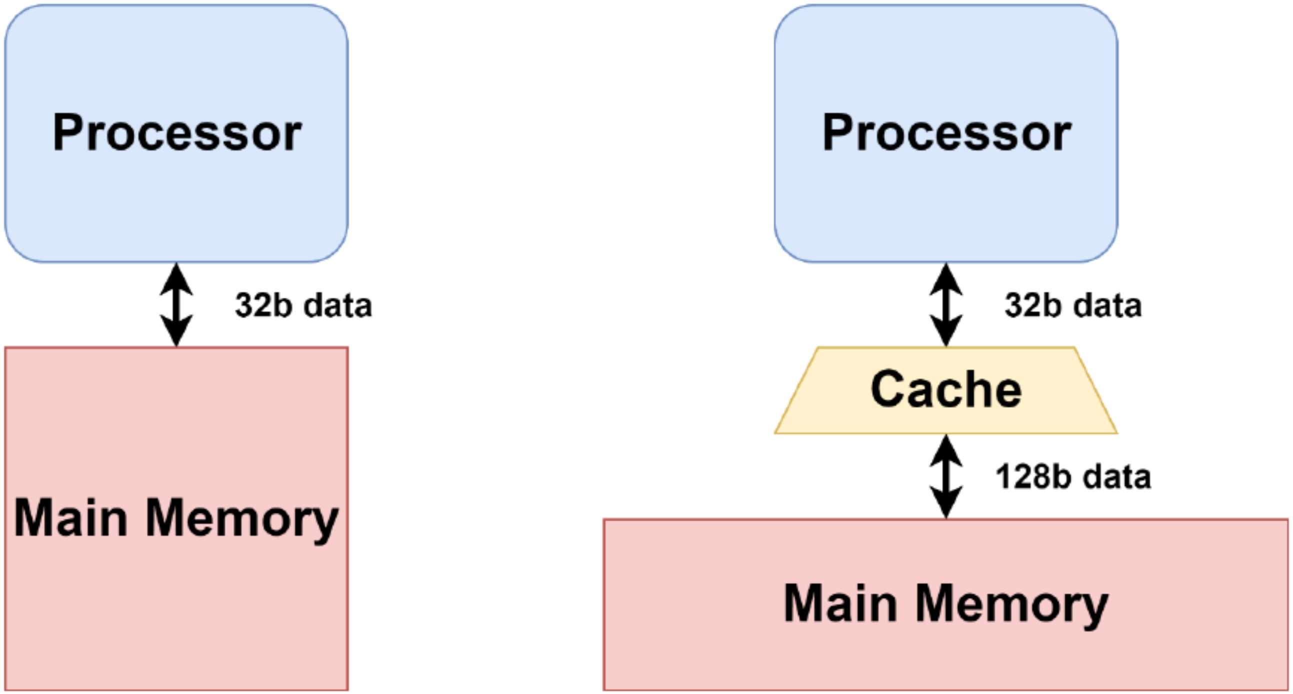
Signal Name	I/O	Width	Description
i_proc_cen	I	1	Active **high** enable signal for read and write
i_proc_wen	I	1	Active **high** enable signal for write
i_proc_addr	I	32	Data memory address
i_proc_wdata	I	32	Data bus for writing to memory
o_proc_rdata	O	32	Data that processor to access from cache
o_proc_stall	O	1	Active **high** control signal that asks processor to wait
o_mem_cen	O	1	Set **high** to enable memory functions
o_mem_wen	O	1	Set **high** for write, **low** for read
o_mem_addr	O	32	Data memory address
o_mem_wdata	O	128	Data bus for writing to memory
i_mem_rdata	I	128	Data that cache to access from memory
i_mem_stall	I	1	Active **high** control signal that asks cache to wait
o_cache_available	O	1	Set this value to 1 if the cache is implemented

Specification – Cache I/O

Signal Name	I/O	Width	Description
i_proc_finish	I	1	Finish signal from processor (To tell the cache to store all data back to the main memory)
o_cache_finish	O	1	Finish signal to cache (To tell the processor that all data is stored back to the main memory)
i_offset	I	32	Signal for memory offset (You can choose to use or not to use)

Memory Data Transportation

- Two transportation approaches for main memory



A. Set o_cache_available to 0: **32-bit** data transport directly to processor

B. Set o_cache_available to 1: **128-bit** data transport to cache

Specification – Cache I/O

- Do not modify the I/O interface!!

```
module Cache#(
        parameter BIT_W = 32,
        parameter ADDR_W = 32
    )(
        input i_clk,
        input i_rst_n,
        // processor interface
            input i_proc_cen,
            input i_proc_wen,
            input [ADDR_W-1:0] i_proc_addr,
            input [BIT_W-1:0]  i_proc_wdata,
            output [BIT_W-1:0] o_proc_rdata,
            output o_proc_stall,
            input i_proc_finish,
            output o_cache_finish,
        // memory interface
            output o_mem_cen,
            output o_mem_wen,
            output [ADDR_W-1:0] o_mem_addr,
            output [BIT_W*4-1:0]  o_mem_wdata,
            input [BIT_W*4-1:0] i_mem_rdata,
            input i_mem_stall,
            output o_cache_available,
        // others
        input  [ADDR_W-1: 0] i_offset
    );
```

Specification – Capacity Constraint

- The capacity of cache should be limited under 2Kb
  - Considering the tolerance, **the total register number used to implement the cache must less than 3000**
  - Check the number through design-compiler
    - Ex:

```
Inferred memory devices in process
        in routine Cache line 647 in file
                '/home/r11943138/111_2_CA/final/Final_group_20/CHIP.v'.
===============================================================================
|     Register Name     |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|     cache_reg_reg     | Flip-flop  |  976  |  Y  | N  | Y  | N  | N  | N  | N  |
|       state_reg       | Flip-flop  |   1   |  N  | N  | Y  | N  | N  | N  | N  |
===============================================================================
```

➔ $976 + 1 = 977 \leq 3000$

```
Inferred memory devices in process
        in routine Cache line 1032 in file
                '/home/r11943138/111_2_CA/final/Final_group_16/CHIP.v'.
===============================================================================
|     Register Name     |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
| data_from_mem_s_reg   | Flip-flop  |  32   |  Y  | N  | Y  | N  | N  | N  | N  |
|    cache_mem_s_reg    | Flip-flop  | 54272 |  Y  | N  | Y  | N  | N  | N  | N  |
|      state_s_reg      | Flip-flop  |   3   |  Y  | N  | Y  | N  | N  | N  | N  |
|      mode_s_reg       | Flip-flop  |   1   |  N  | N  | Y  | N  | N  | N  | N  |
===============================================================================
```

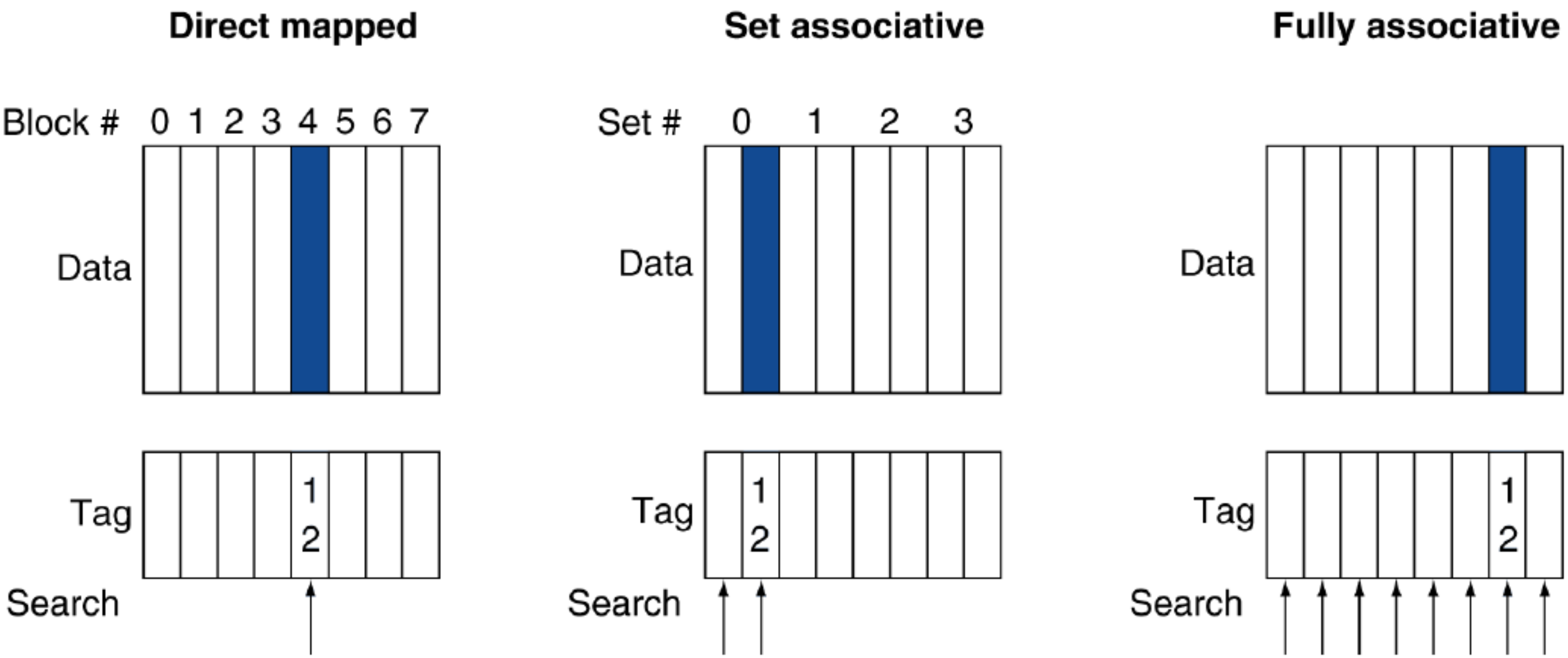
➔ $32 + 54272 + 3 + 1 > 3000$

Default connection

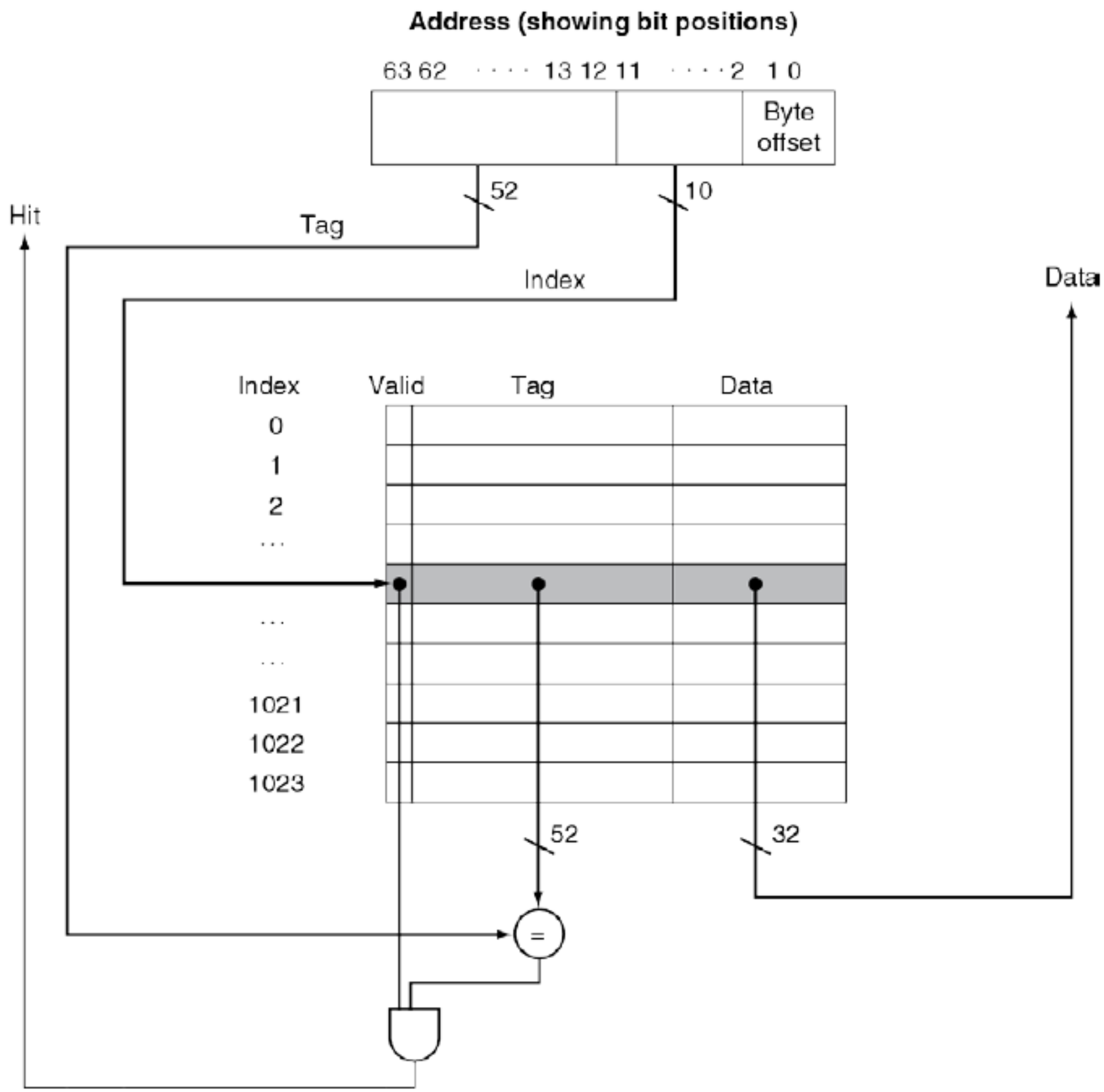
- Connect the memory and processor directly
- Remember to annotate this part if you want to design it by yourself

```
//-----------------------------------------//
//            default connection           //
assign o_mem_cen = i_proc_cen;             //
assign o_mem_wen = i_proc_wen;             //
assign o_mem_addr = i_proc_addr;           //
assign o_mem_wdata = i_proc_wdata;         //
assign o_proc_rdata = i_mem_rdata[0+:BIT_W];//
assign o_proc_stall = i_mem_stall;         //
//-----------------------------------------//
```

Review – Implement Method

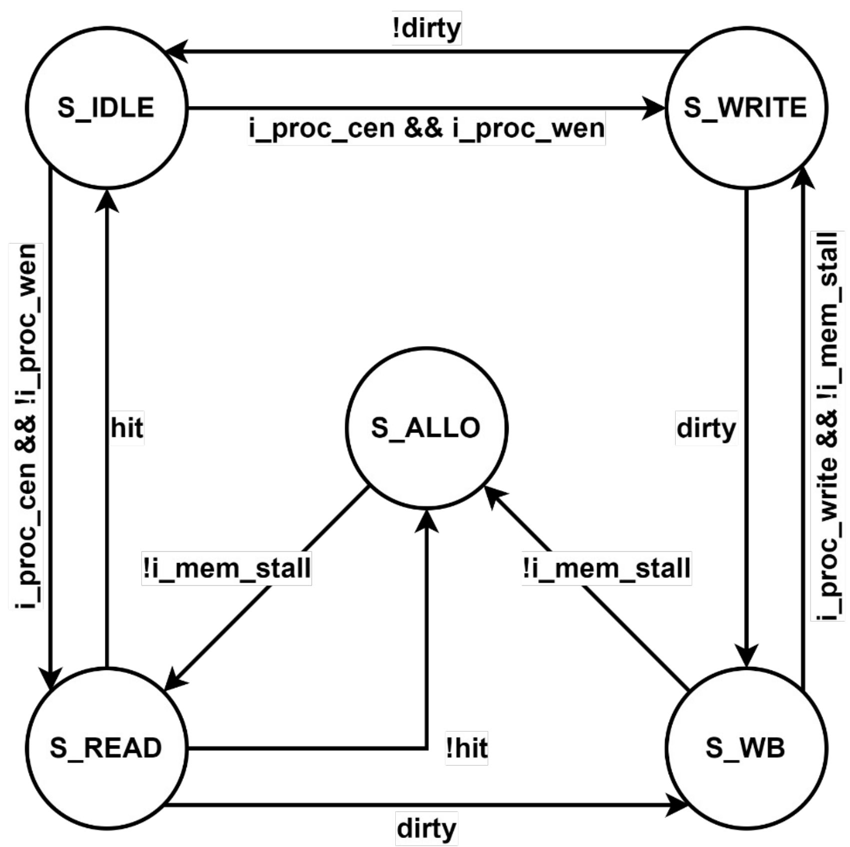


Review – Example Architecture



Example FSM

- You can design it by yourself



Stall

- When the cache needs to access data from the memory and then wait for several cycles, the i_proc_stall signal should be set high to stall the processor.
- A stall is necessary
  - **Read miss** in write back caches

Write through or write back

- Write through
  - Also update memory
  - Easy to implement
  - Longer write latency
- Write back
  - Keep tracking
  - More complex
  - More efficiently
- Write back policy
  - Least Recently Used (LRU)
  - Least Frequently Used (LFU)