mp_pipeline

Part 1

Overview (MP)

Implementation of a pipelined processor that is mostly* RV32I compliant

Questions?

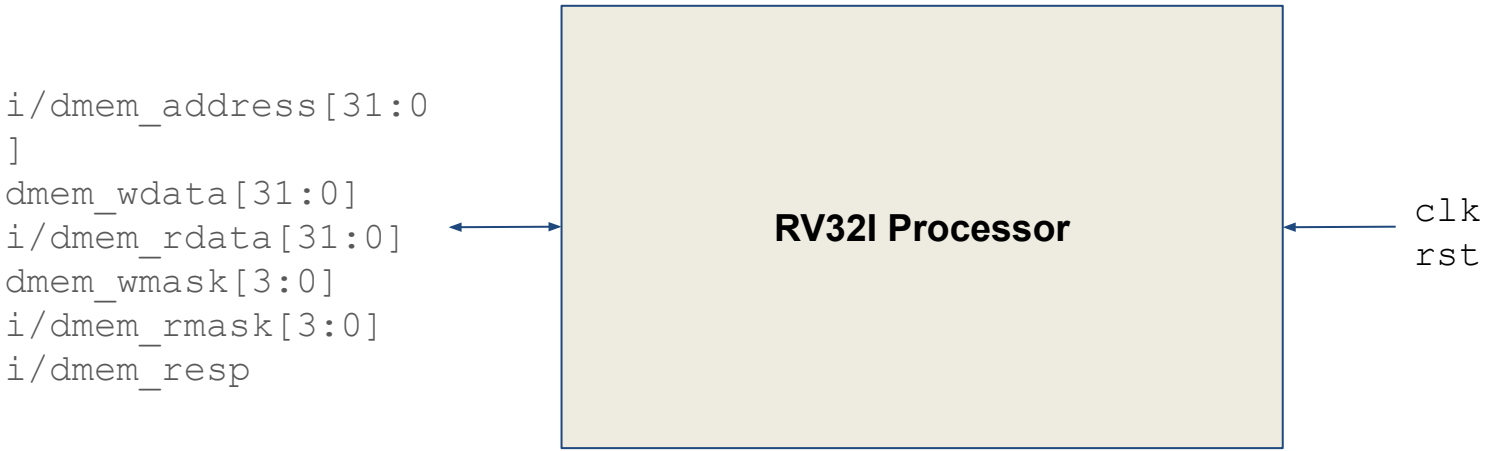
- Always refer to README/GUIDE.md first
- RISC-V Manual (ch. 2, 19, 20)
- P&H Computer Organization and Design (RISC-V ed.): Chapter 4
- Office hours/Campuswire

* You will not be required to implement FENCE*, ECALL, EBREAK, and CSRR* instructions.

Overview (Checkpoint 1)

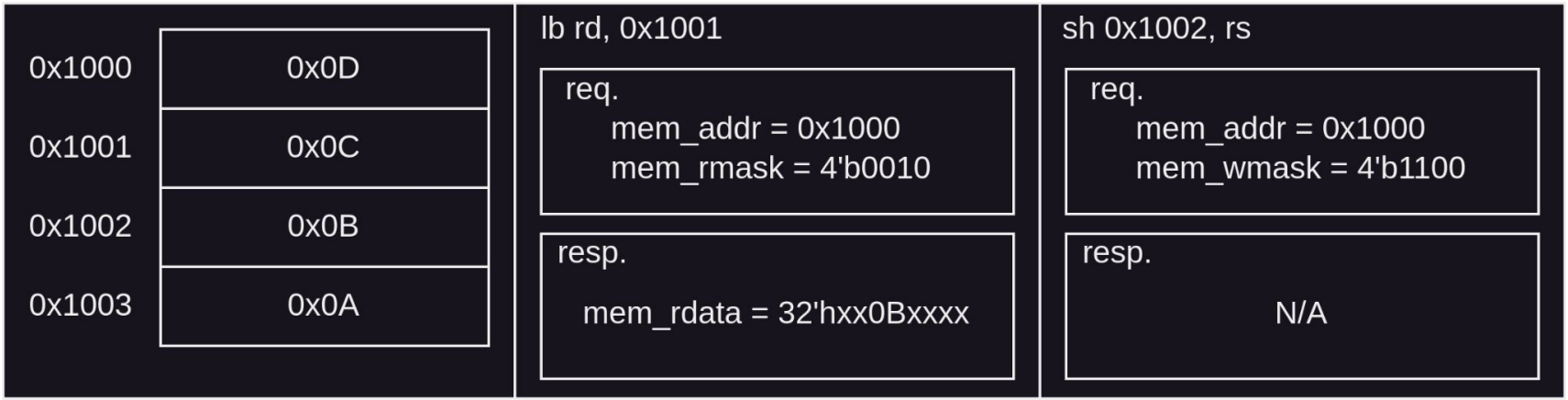
- No control flow instructions (branches & jumps)
  - No flushing
- No loads/stores
- No hazards
  - No stalls, or forwarding
- Single-cycle memory delay
- This means ALU instructions only for this checkpoint!

Top-level Interface



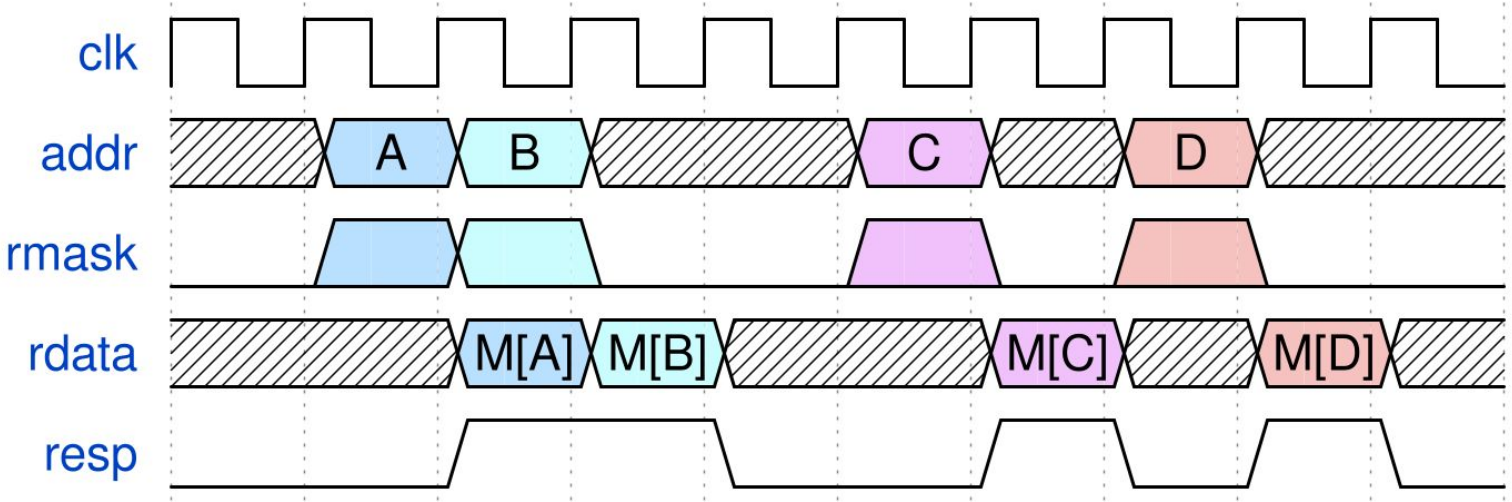
Memory Alignment

- Memory access must be 4-byte aligned
- Use mask to indicate validity of byte



Memory Interface

- Simulating pipelined cache behavior
  - Incurring one cycle delay (more on this later)



Why pipeline?

	Clock Frequency	Throughput (IPC)
Single-cycle	Low	High
Multi-cycle	High	Low
Pipeline	High	High

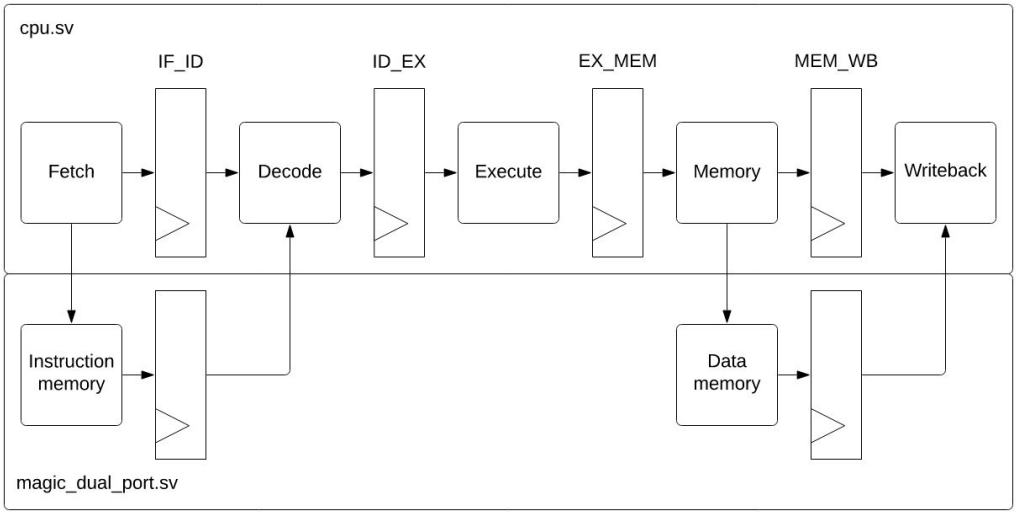
Pipeline Stages

5-stage pipeline

- Fetch
- Decode
- Execute/ALU
- Memory
- Writeback/commit

Each stage takes one clock cycle

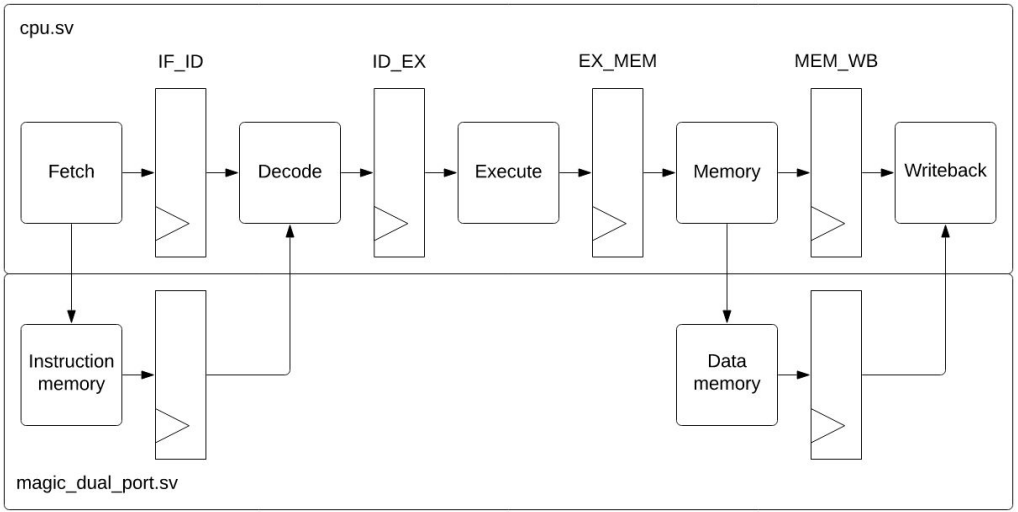
Works in parallel (on different instructions)



Pipeline Stages (Fetch)

Fetch

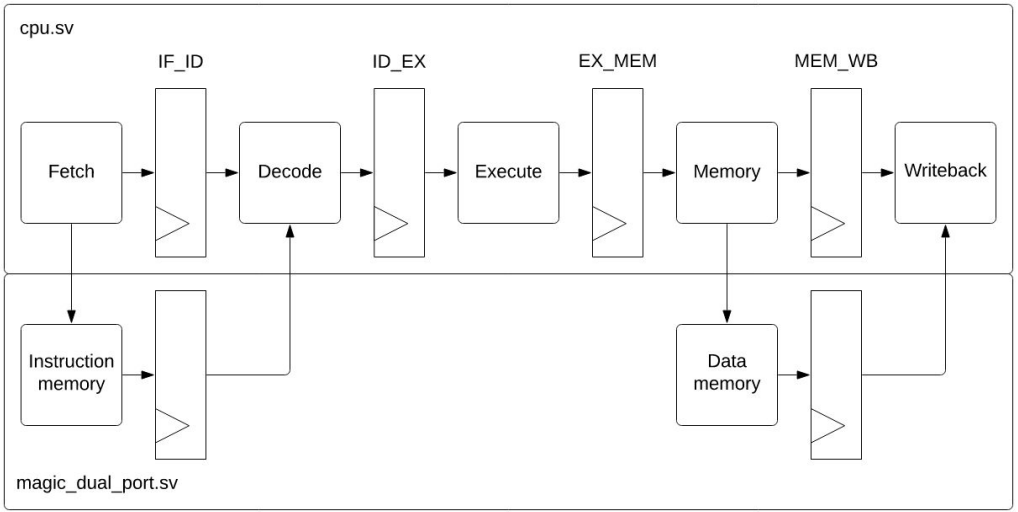
- Send request to memory for instruction
- Determine what the possible* next PC is
- DOES NOT receive memory response



Pipeline Stages (Decode)

Decode

- Receives response from memory
- Decodes the instruction
  - opcode?
  - registers?
  - Immediates?
- Create all control signals for later stages
  - Control words



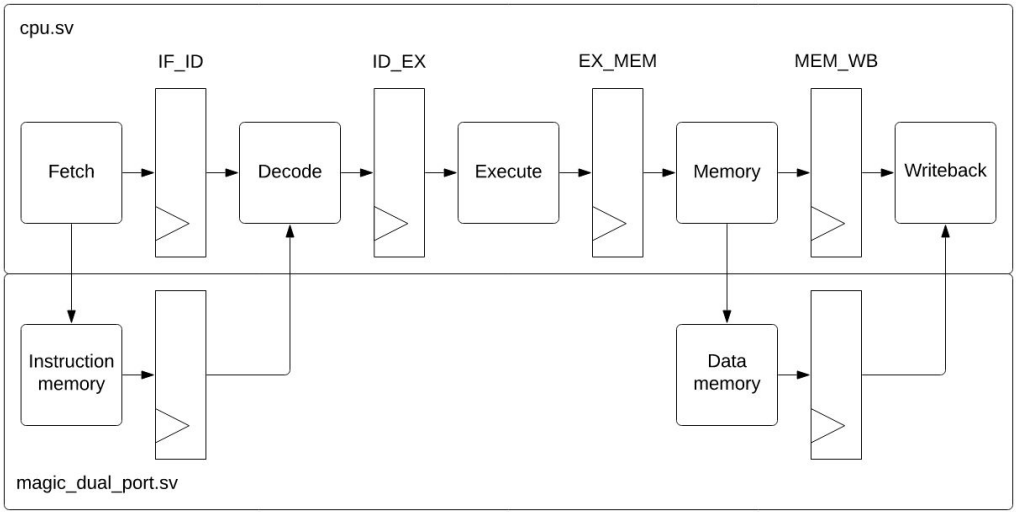
Pipeline Stages (Execute & Memory)

Execute/ALU

- Performs all necessary calculations
  - ALU
  - CMP
  - Multiplier (M-extension)

Memory

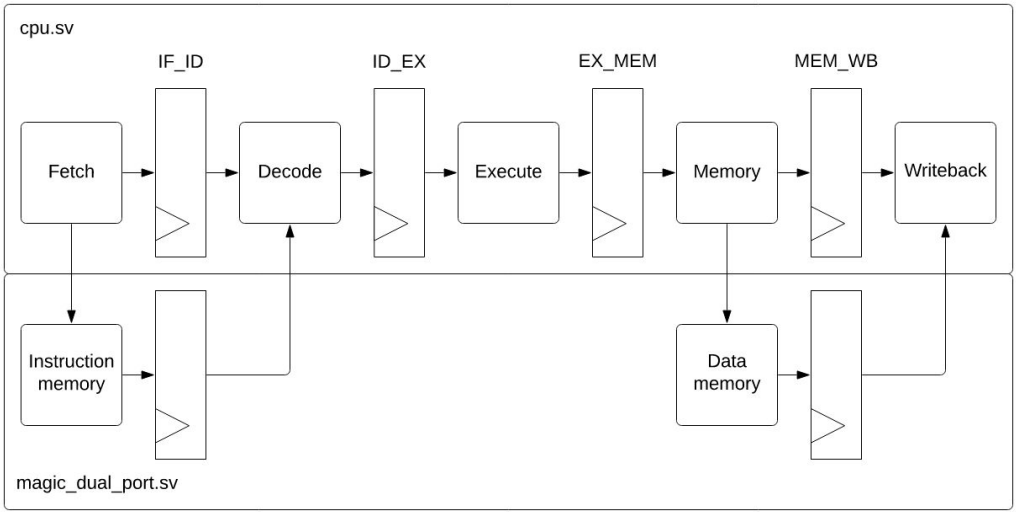
- Send request to data memory



Pipeline Stages (Writeback/Commit)

Writeback/Commit

- Receives memory response (if any)
- Commits (writes) instruction results to registers/architectural state



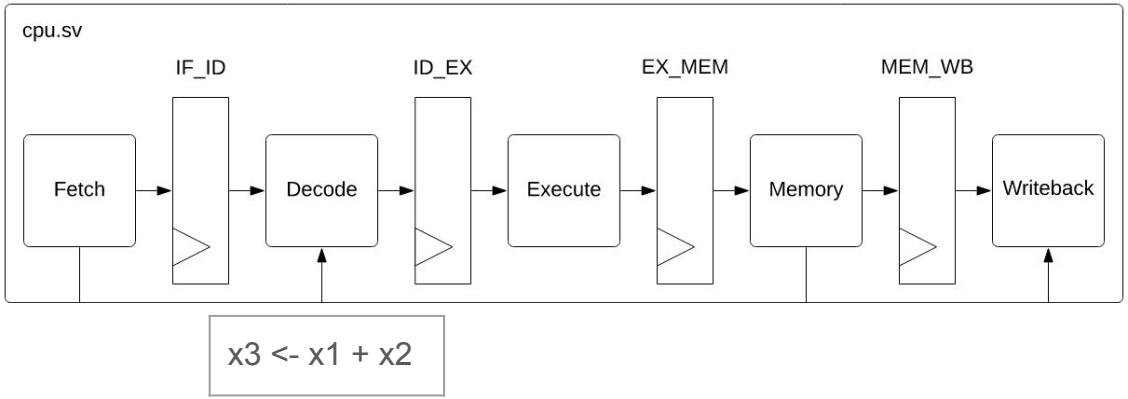
Writing your own tests

- Only reg-reg and reg-imm ALU instructions
  - i.e. add(i), and(i), sll(i), slt(i), etc.
- No forwarding
  - Insert nops between instructions that have dependencies
  - No need for nops for independent instructions
- Refer to testcode/cp1_example.s

Writing your own tests (cont.)

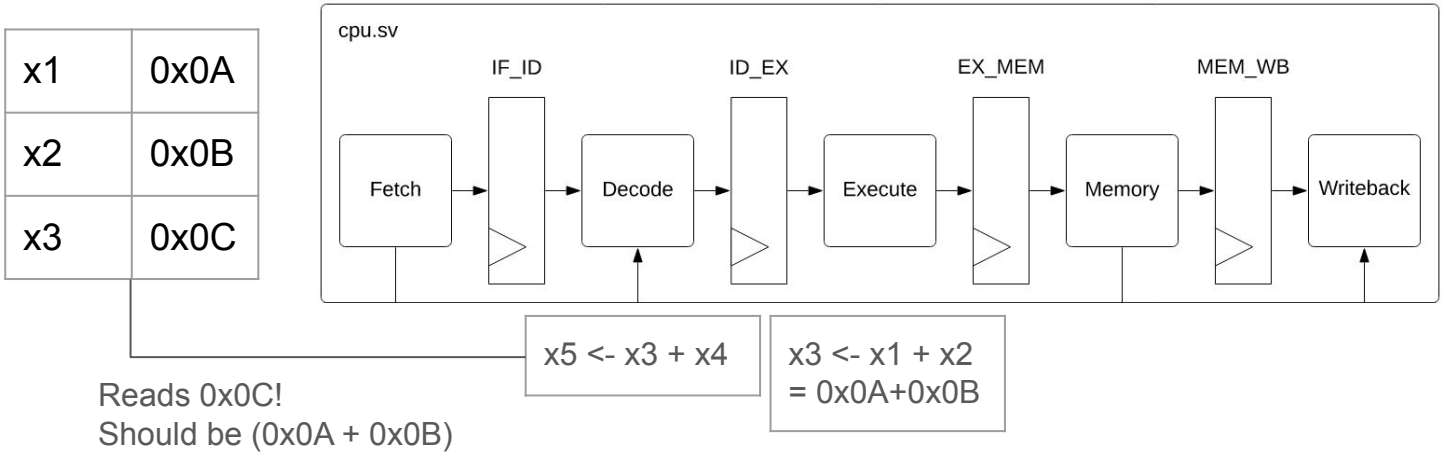
How to write test codes that avoid hazards?

x1	0x0A
x2	0x0B
x3	0x0C



Writing your own tests (cont.)

How to write test codes that avoid hazards?

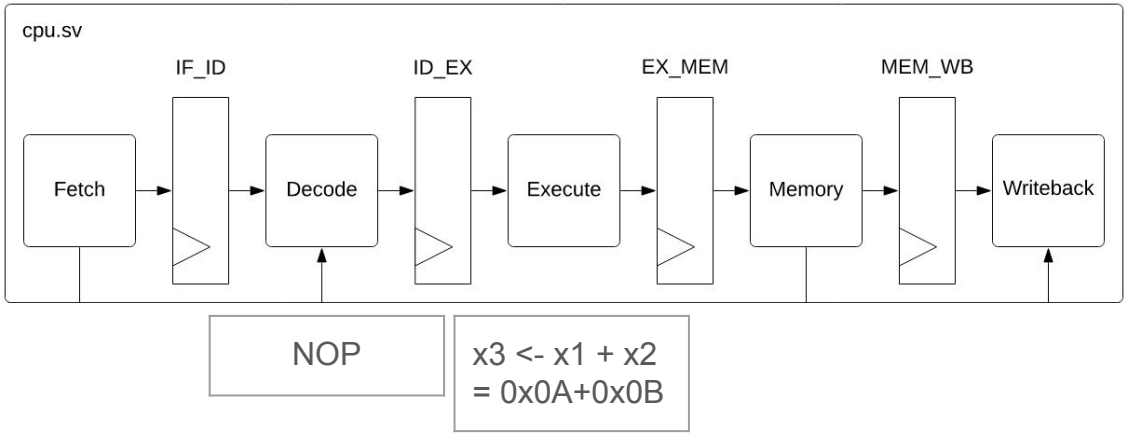


Need forwarding to resolve. For this checkpoint, you will not be tested on it.

Writing your own tests (cont.)

How to write test codes that avoid hazards?

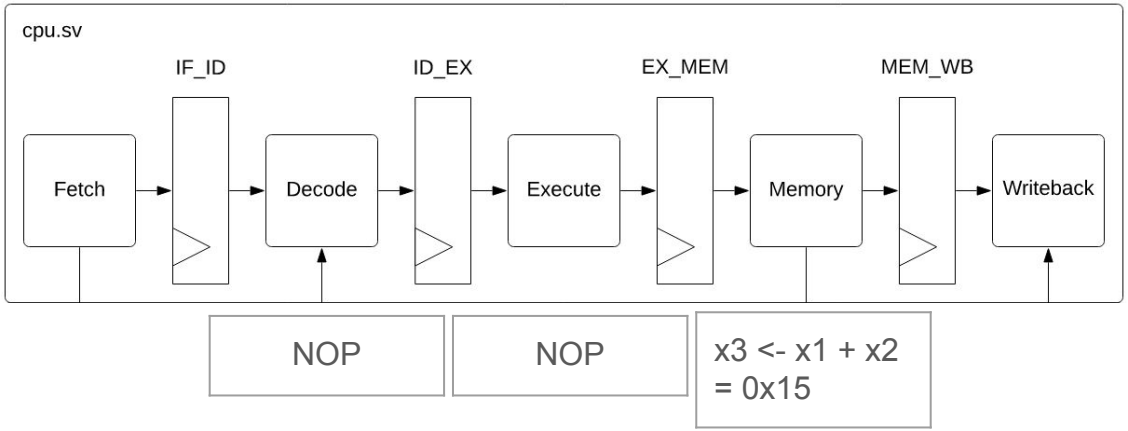
x1	0x0A
x2	0x0B
x3	0x0C



Writing your own tests (cont.)

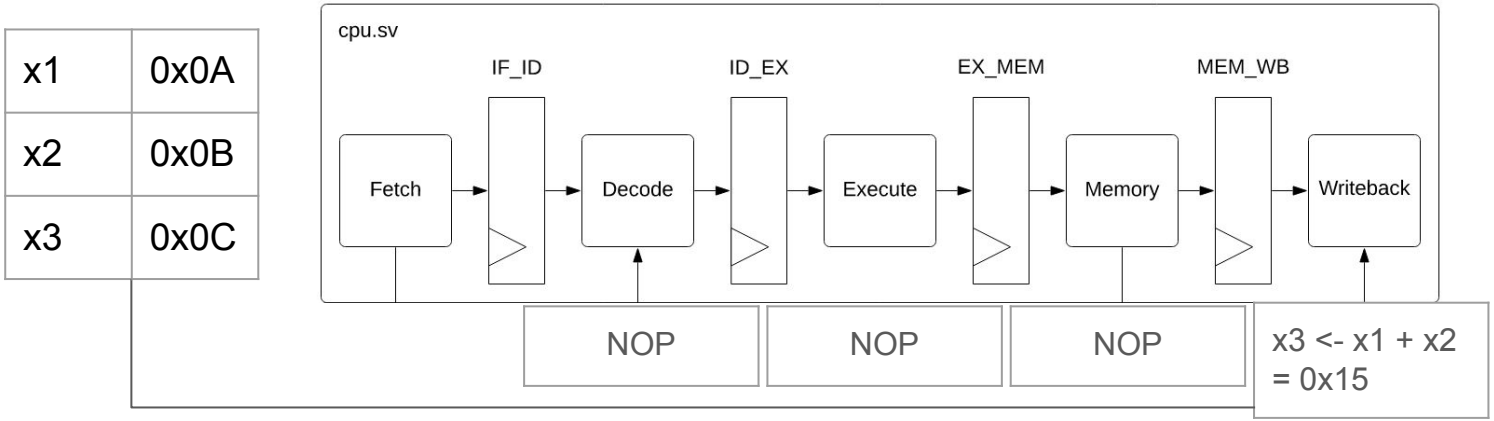
How to write test codes that avoid hazards?

x1	0x0A
x2	0x0B
x3	0x0C



Writing your own tests (cont.)

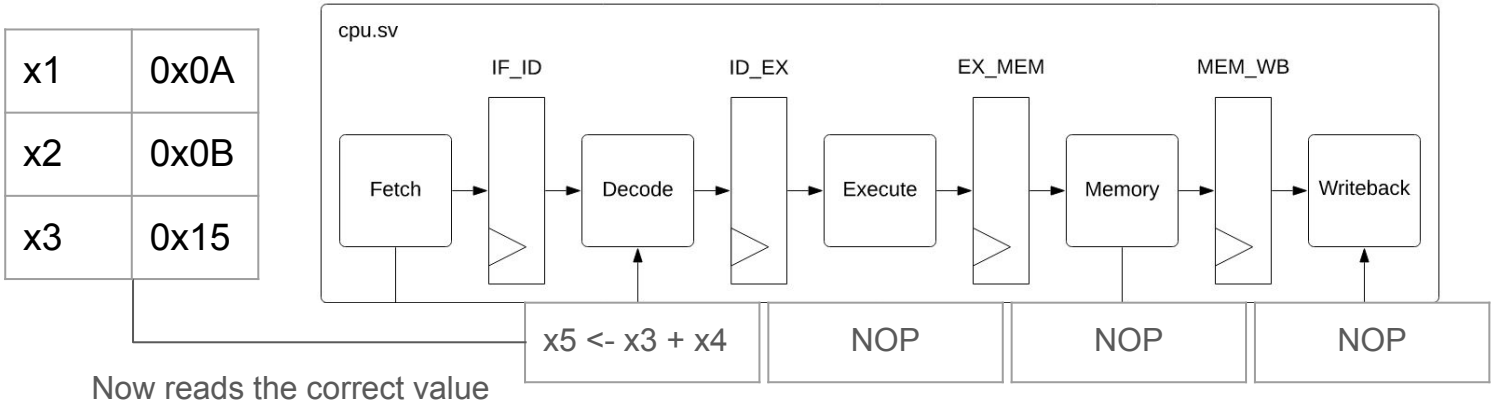
How to write test codes that avoid hazards?



Writes back the correct value

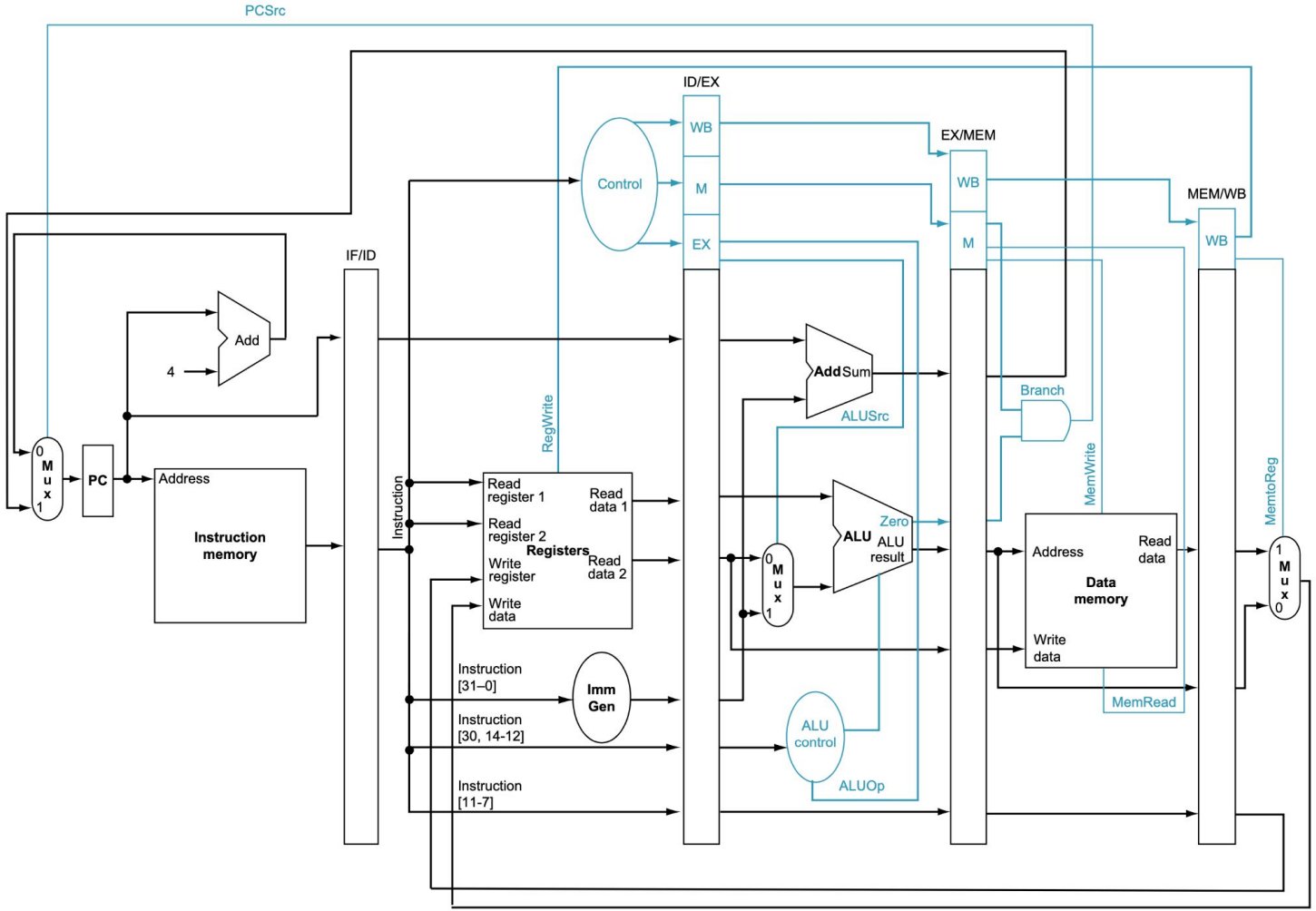
Writing your own tests (cont.)

How to write test codes that avoid hazards?



Using mp_verif Random Testbench

- Copy over random_tb, randinst, instr_cg
- Different interface (dual port)
- Add nops


PCSrc
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Control
IF/ID
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Instruction [31–0]
Instruction [30, 14-12]
Instruction [11-7]
Imm Gen
Add
Sum
ALUSrc
Branch
MemWrite
MemtoReg
ALU
Zero
ALU result
ALU control
ALUOp
Address
Read data
Data memory
Write data
MemRead
1
M
u
x
0

Advice on how to organize your RTL

```
if_id_t if_id_reg, if_id_reg_next;


always_ff @(posedge clk) begin
  if_id_reg <= if_id_reg_next;
end


id_stage id_stage_i (
  .if_id(if_id_reg),
  .id_ex(id_ex_reg_next)
);
```

```
typedef struct packed {
        logic   [31:0]      inst;
        logic   [31:0]      pc;
        logic   [63:0]      order;
        ctrl_word_t         cword;
} id_ex_reg_t;


typedef struct packed {
        alu_m1_sel_t        alu_m1_sel;
} ctrl_word_t;
```

RVFI Monitor

- Runs a golden processor in parallel to check your commits
- Verifies architectural states
- DOES NOT verify memory states (Use spike)
- Uses signals generated through the pipeline
  - The final signal resides in writeback stage
- Put your hierarchical reference in `hvl/rvfi_reference.json`
  - Starts with `dut.`
  - Ex. `dut.wb_stage.rs1`
- Refer to rvfi.md for the purpose of each signal
- Use x0 for those instruction that does not use a register, both read and write

Spike

- Golden software model
- Produces golden commit log
- Compare against your own commit log
  - Produced by `monitor.sv`

Questions?