mp_pipeline

Part 2

Announcements

- How are things?
- Patch released for mp_pipeline on Monday, 2/5
    - Details on site
- Asking questions
    - 4 sources for potential answers
        - Website, Campuswire, Discord, MP docs
        - Use ctrl+f or search functions of unique keywords
            - W/ search, you may not need to wait minutes/hours/days/. . . for us to reply w/ an answer
    - When asking, tell us where you have looked for an answer
        - E.g “what does resp do?” -> “I have seen the attached memory timing diagram for the MP, but I still don’t quite understand when resp is raised”
        - Tells us where things may need clarification
- Going into the MP contents of the session
    - CP1 was conceptually light but heavy on code
    - CP2 is conceptually heavy but lighter* on code
        - Don’t confuse lighter for light! Less code than CP1 doesn’t imply little or no code!

Checkpoint 2 Overview

- New instructions
  - Memory instructions
    - Loads
    - Stores
  - Control flow instructions
    - Branches
    - Jumps
- Hazard detection and resolution
  - Data hazards & forwarding
  - Control hazards & flushing

New Instructions

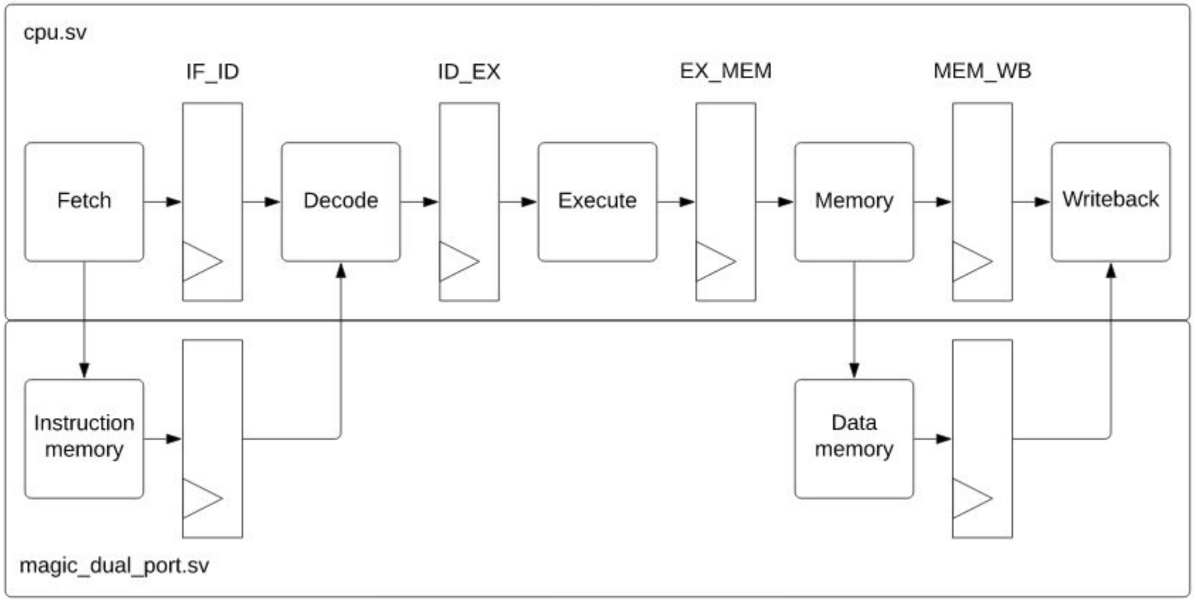
Memory Overview

Memory instructions are instructions that interact with memory
. . . . (beyond the Fetch stage)

- Memory instruction execution in the pipeline
- Memory model
  - Memory timing
  - Mask signals and resp
- Alignment of memory transactions

Execution of a Memory Instruction

- The hardware and data needed to complete a load or store instruction
  - Think about the following
    - What are the interfacing signals of the memory I care about in each instruction?
    - How does my pipeline need to massage data that goes to or comes from memory?
    - When and where of signals



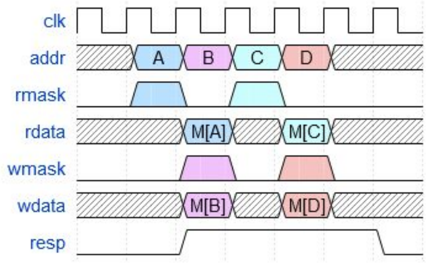
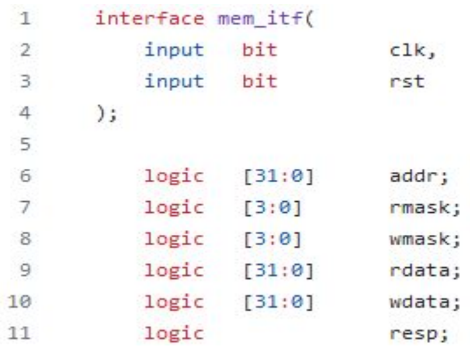


Figure 4: Mixed read and write without stalling



```
interface mem_itf(
    input   bit         clk,
    input   bit         rst
);

    logic   [31:0]      addr;
    logic   [3:0]       rmask;
    logic   [3:0]       wmask;
    logic   [31:0]      rdata;
    logic   [31:0]      wdata;
    logic               resp;
```

Execution of a Memory Instruction

- Signals - what are they and what hardware determines/drivers them?
  - **address**: Target word for memory (ensure word aligned); ALU calculates
  - **rmask**: Relevant on loads - two approaches
    - Read everything and throw out rest
    - Read only what you want (use thrown out bits from raw address calculated by ALU)
  - **wmask**: Relevant on stores - Write only bytes dictated by instruction (again using thrown out addr bits)
  - **rdata**: Data from memory to regfile
  - **wdata**: Data sent from regfile to memory
  - **resp**: Can ignore for this checkpoint (no stalls)!

- Hardware - what do we need?
  - ALU for address calc (already here from CP1)
  - Shift logic (for sub-word data)
    - Stores - shift write data to appropriate section matching wmask
    - Loads - shift read data to lowest bits of registers (maybe matching rmask)

Memory Model and Timing Diagrams

So many scenarios to consider . . .

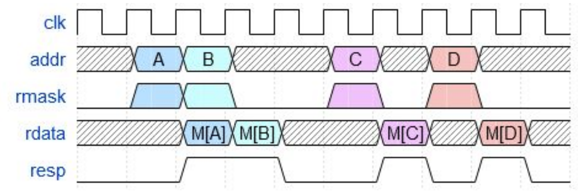


Figure 2: Read without stalling

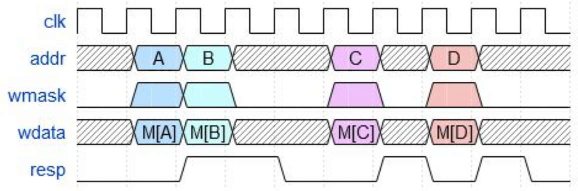


Figure 3: Write without stalling

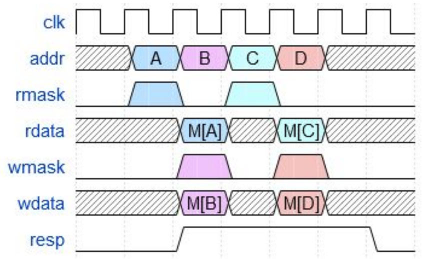


Figure 4: Mixed read and write without stalling

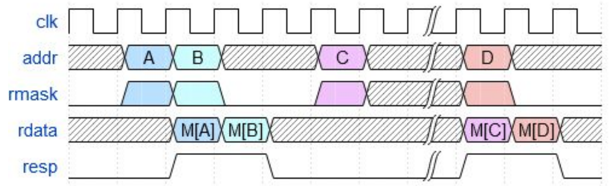


Figure 5: Read with stalling

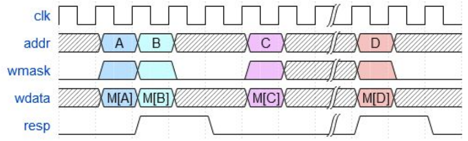


Figure 6: Write with stalling

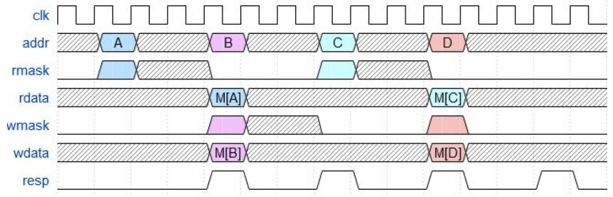


Figure 7: Mixed read and write with stalling

Memory Model and Timing Diagrams

So many scenarios to consider . . .

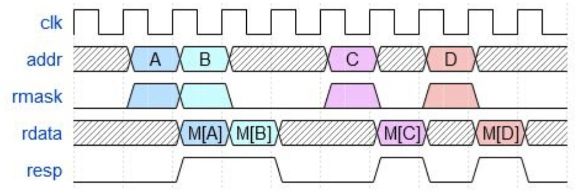


Figure 2: Read without stalling

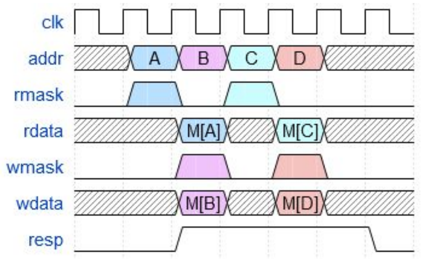


Figure 4: Mixed read and write without stalling

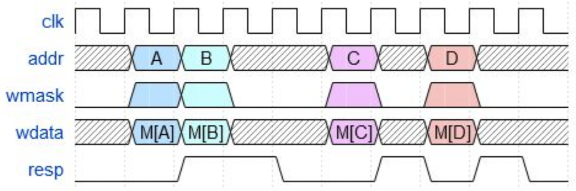


Figure 3: Write without stalling

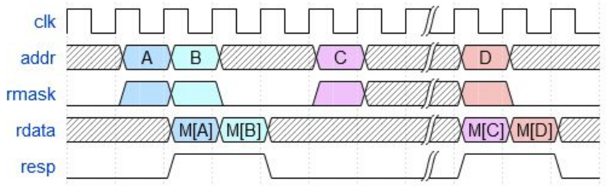
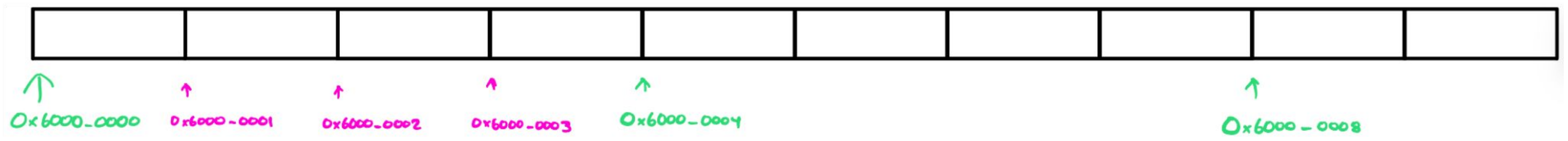


Figure 5: Read with stalling

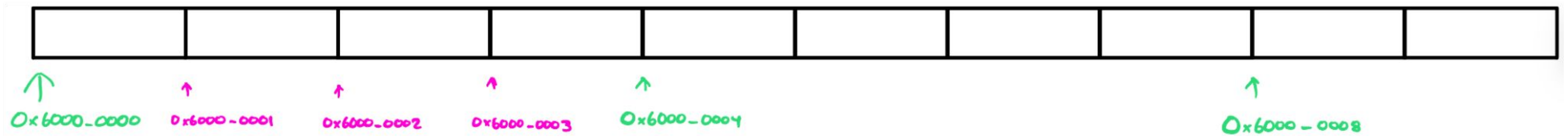
When/Where Memory?

- From timing diagram, it is clear that we need at least two cycles to complete a memory transaction
    - Crucial to utilize pipeline of memory and pipeline of our CPU to maintain high IPC
    - Implies that a memory transaction is spread across multiple stages
        - What parts of the memory transaction partitioned into different cycles?
            - Initiation of memory request by CPU
                - Stores - Request to write to location and data to write
                - Loads - Request to read from location
            - Response from memory that request is complete
                - Stores - Response that write is complete
                - Loads - Response that read is complete and presentation of data
        - What stages should handle these parts?

Memory Alignment

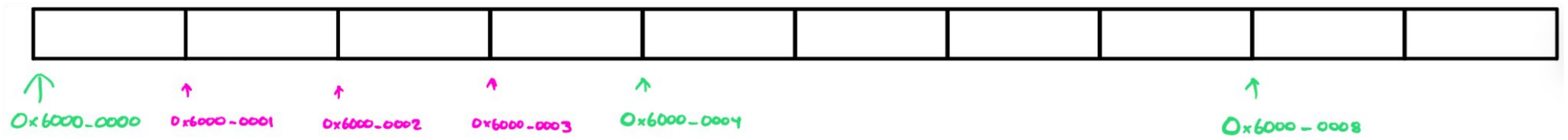


Memory Alignment



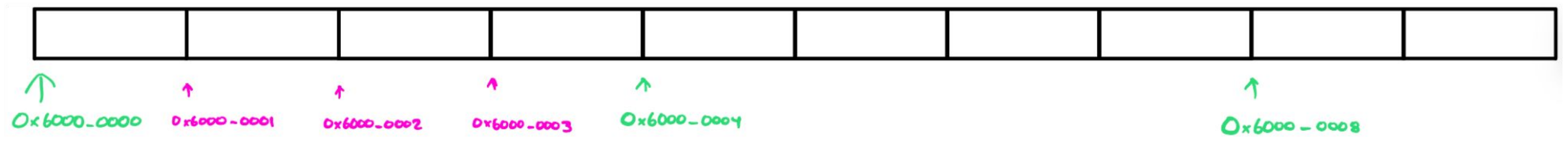
- All addresses leaving your CPU and going to memory must be word aligned!

Memory Alignment

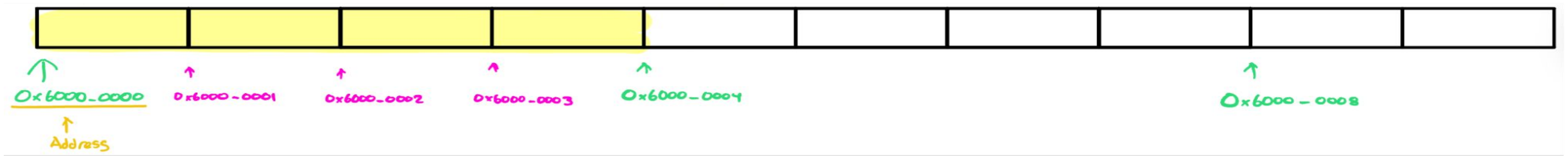


- All addresses leaving your CPU and going to memory must be word aligned!
  - Think of the address as a “coarse” target

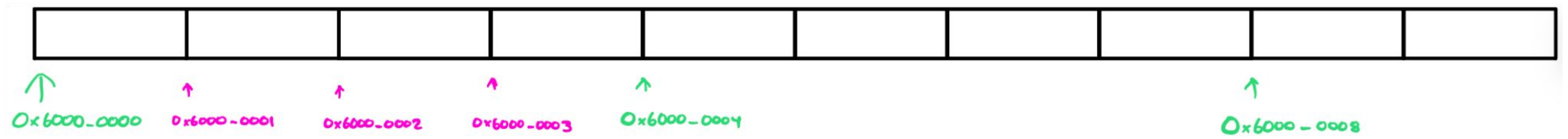
Memory Alignment



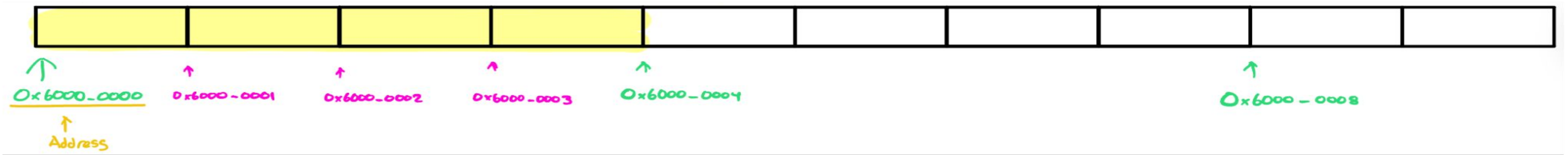
- All addresses leaving your CPU and going to memory must be word aligned!
    - Think of the address as a “coarse” target



Memory Alignment

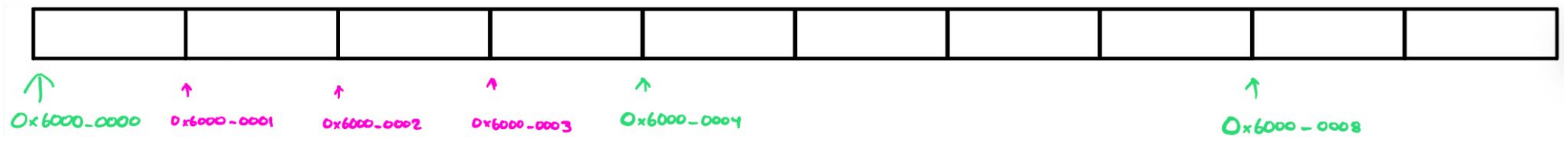


- All addresses leaving your CPU and going to memory must be word aligned!
  - Think of the address as a “coarse” target

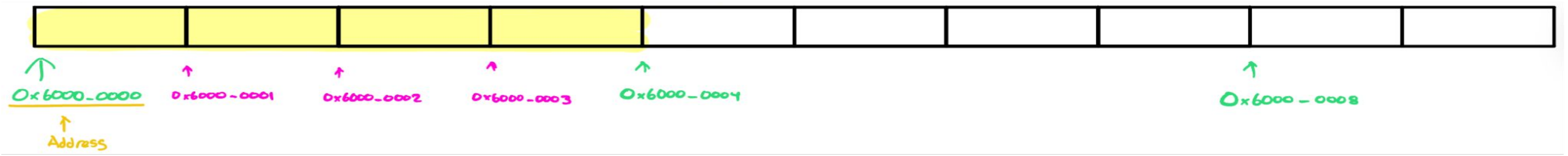


  - Think of the mask as a “fine” target
  - The (w)/(r)mask bits indicate which of the four bytes from the word selected by address is being (w)ritten to or (r)ead from

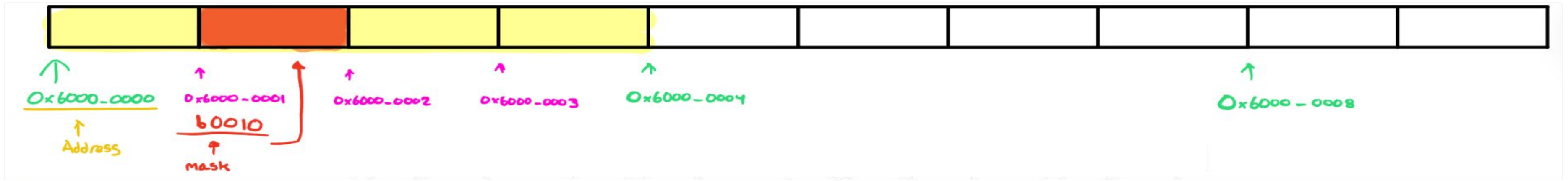
Memory Alignment



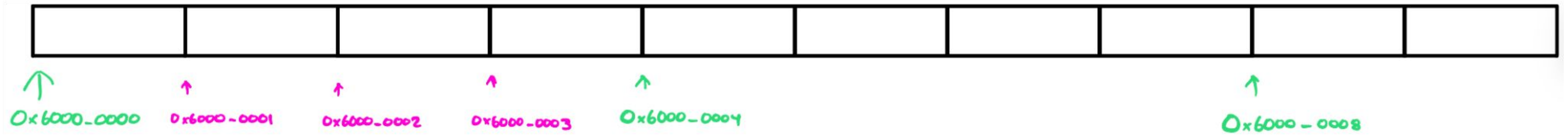
- All addresses leaving your CPU and going to memory must be word aligned!
  - Think of the address as a "coarse" target



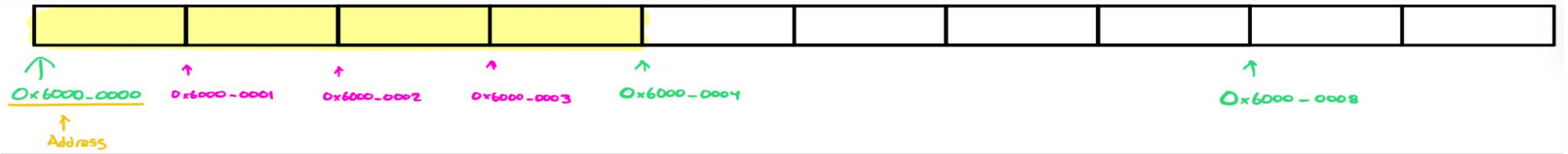
  - Think of the mask as a "fine" target
  - The (w)/(r)mask bits indicate which of the four bytes from the word selected by address is being (w)ritten to or (r)ead from



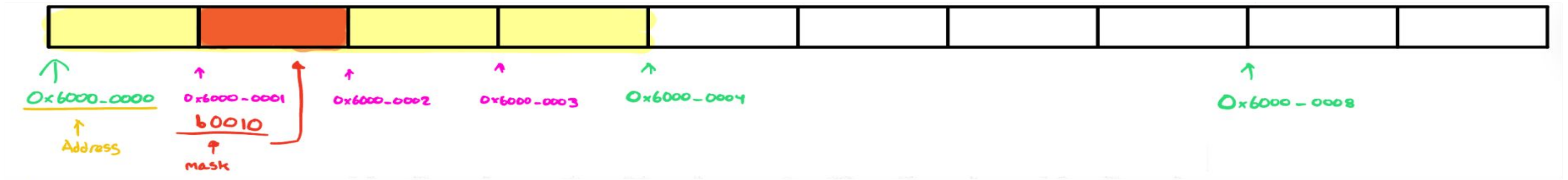
Memory Alignment



- All addresses leaving your CPU and going to memory must be word aligned!
    - Think of the address as a "coarse" target



- Think of the mask as a "fine" target
- The (w)/(r)mask bits indicate which of the four bytes from the word selected by address is being (w)ritten to or (r)ead from



- **We will not test you on misaligned memory accesses**!
    - You do not need to implement logic for the trap signal
        - If you see this triggered in RVFI, your memory access is not aligned
    - Ensure that your random testbench doesn't generate misaligned addresses
        - Revisit mp_verif docs for refresher

Calculated Raw Address Convention

- The calculated / raw address from your ALU should obey the following alignment rules (**NOT** the final output address to mem)
    - lw, sw - Word aligned -> 32-bit aligned -> 4-byte aligned -> Bottom 2 bits are zero
    - lh, sh - Half aligned -> 16-bit aligned -> 2-byte aligned -> Bottom bit is zero
    - lb, sb - Byte aligned -> 8-bit aligned -> 1-byte aligned -> Any address goes

- The above does not need to be enforced by your hardware
    - This is just the convention we follow with our tests and you should too
- For RVFI, your mem_rdata should NOT be shifted!
    - Report exactly what you read from memory and your mask
    - RVFI will shift accordingly internally when determining what is written into the regfile

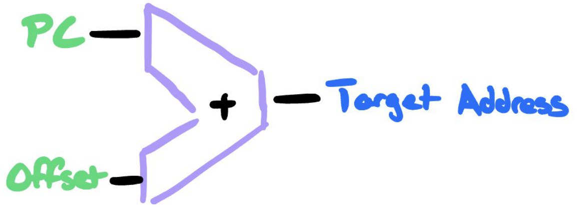
Control Flow Instructions

Control flow instructions are instructions that can modify the PC

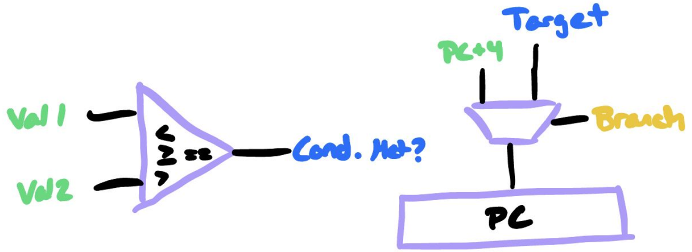
- Control flow instruction execution in the pipeline
- Where to determine the branch

Execution of a Control Flow Instruction

- What are the parts of executing a control flow instruction?
  - Evaluating the condition (if a conditional branch)
  - Calculating the target address
  - Writing into the PC

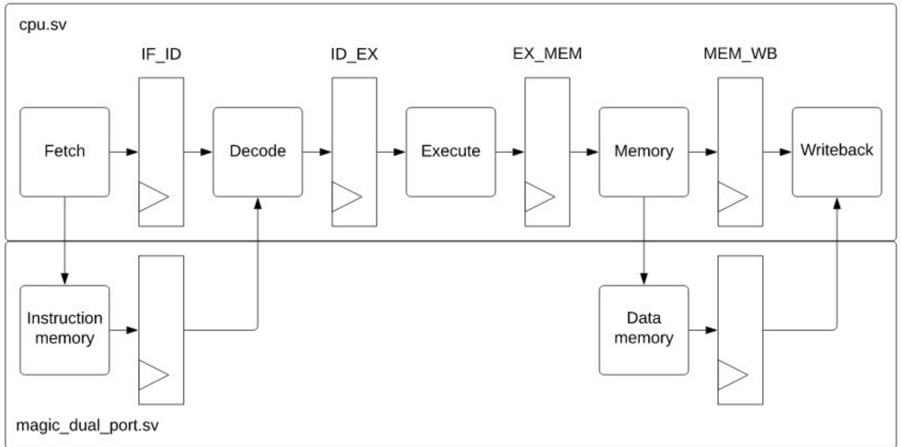


- What does the hardware look like for these steps?
  - Evaluating the condition -> A comparator
  - Calculating the target address -> An adder (the ALU)
  - Writing into the PC -> Mux to override the typical PC+4 write



When/Where Branch?

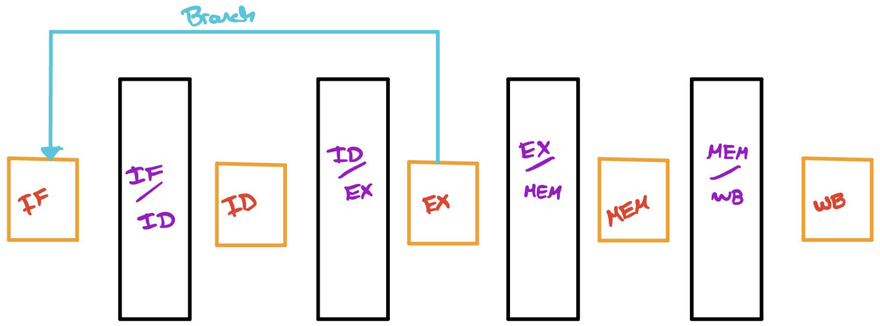
- What stage do we know all the information necessary to complete a branch?
  - Execute
    - Has comparator to evaluate condition
    - Has ALU to compute branch target address
- When should we *actually* branch (write to the PC)?
  - Other instructions wait until Write-Back to change state
    - But wait, actually stores don't . . .
    - So why do branches need to wait until Write-Back if we have everything?
      - Wouldn't it be nice if we could branch earlier?

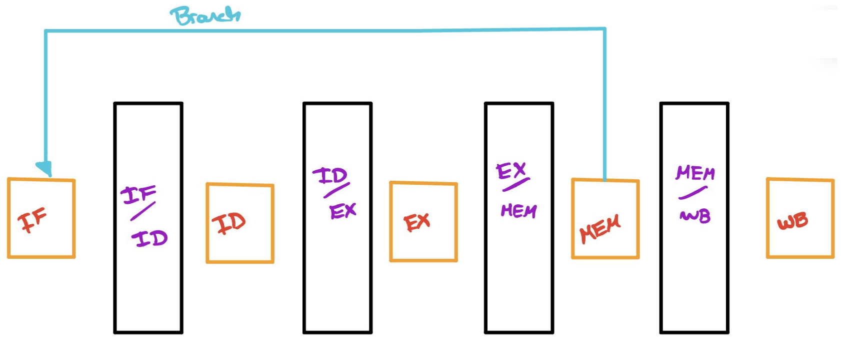


When/Where Branch?

- Branch in EX
  - Less speculative instructions in pipeline
    - -> Lower misprediction penalty
  - Longer combinational path
    - But is it the critical path?

- Branch in MEM
  - More speculative instructions in pipeline
    - -> Higher misprediction penalty
  - Split combinational path
  - **Beware**: Memory stores initiated in EX
    - Can you really flush these?

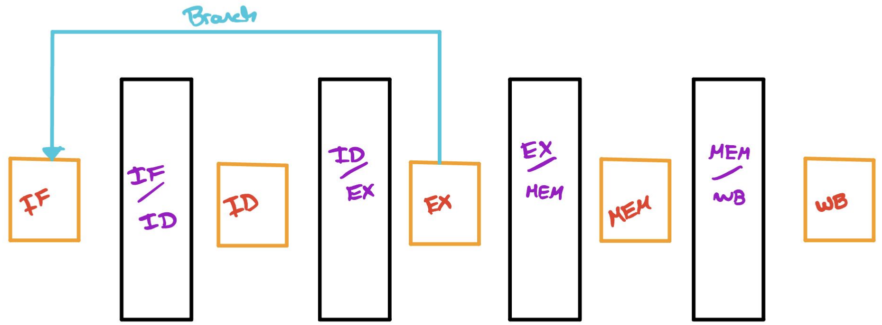




- Regardless of where you decide to branch, remember to update pc_wdata in RVFI!
  - CP1 had this as constant PC+4; now we overwrite this if branch occurs

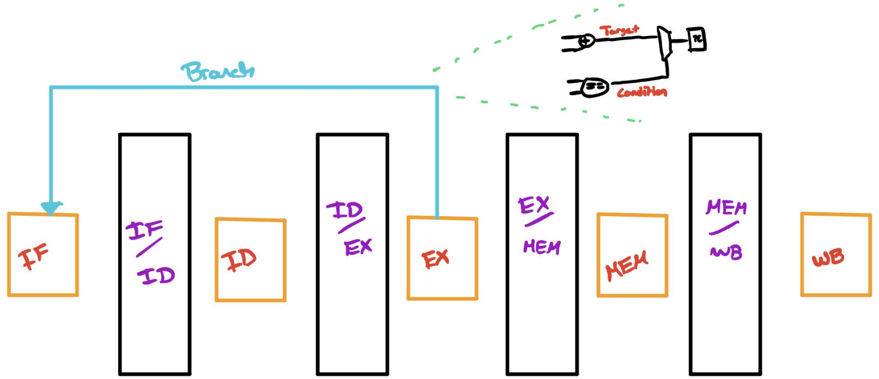
When/Where Branch?

- Branch in EX
  - Less speculative instructions in pipeline
    - -> Lower misprediction penalty
  - Longer combinational path
    - But is it the critical path?



When/Where Branch?

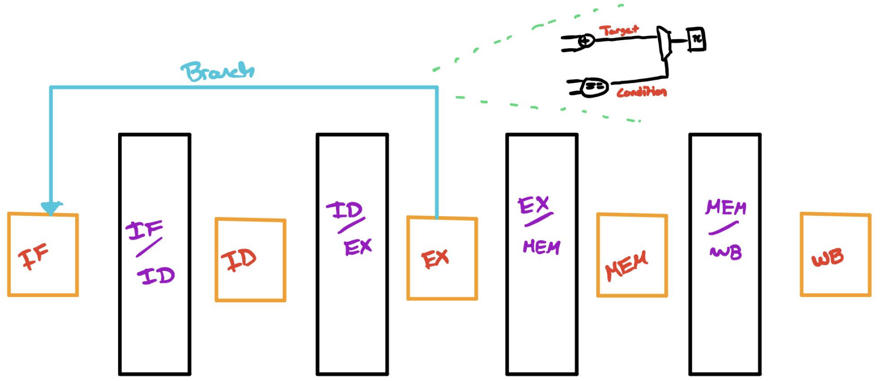
- Branch in EX
  - Less speculative instructions in pipeline
    - -> Lower misprediction penalty
  - Longer combinational path
    - But is it the critical path?

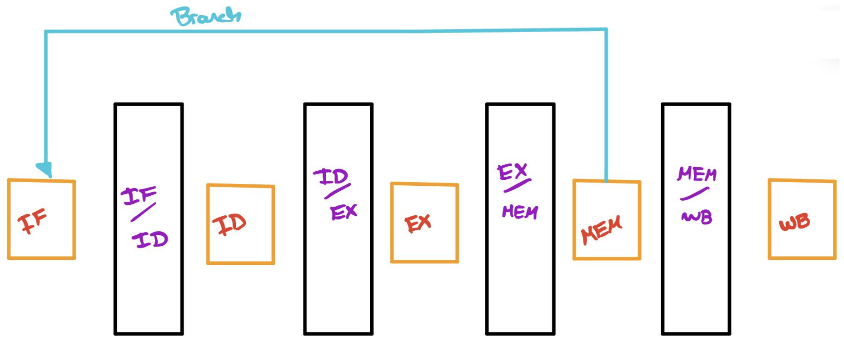


When/Where Branch?

- Branch in EX
  - Less speculative instructions in pipeline
    - -> Lower misprediction penalty
  - Longer combinational path
    - But is it the critical path?

- Branch in MEM
  - More speculative instructions in pipeline
    - -> Higher misprediction penalty
  - Split combinational path
  - **Beware**: Memory stores initiated in EX
    - Can you really flush these?





- Regardless of where you decide to branch, remember to update pc_wdata in RVFI!
  - CP1 had this as constant PC+4; now we overwrite this if branch occurs

Hazards

Control Hazards

- No oracle to determine correct branch 100% of time
  -> Leads to speculation that may be incorrect
- Resolutions
  - **Easy/Poor***: Do not submit any instructions to the pipeline until branch resolves
  - **More Involved/Better**: Submit instructions from one branch path and remove them from the pipeline if they were not correct
- Why an asterisk on the “Poor”?
  - In the event of bad prediction accuracy, you waste a lot of power (not time) on work from instructions that just end up being thrown out

Branch Predictor & Flushing

- The proactive approach requires a branch predictor
  - But it doesn’t require a complicated one. . .
- Static not-taken branch predictor is the simplest and what we recommend
  - Can do others if desired, but you have enough work this MP
  - Will also be able to explore branch predictor design in mp_bp
- Misprediction -> Bad instructions in pipeline
  - Must delete these
  - How?
    - Activate the reset signal to clear the contents of the relevant pipeline registers
    - Set valid bit to 0 and have all control signals ANDed with it

Data Hazards

- Attempt to access data that is still in the pipeline
- Resolutions
  - **Easy/Poor**: Stall dependent instructions at Decode until Write-Back of dependent instruction is complete
  - **More Involved/Better**: Forward the data from the dependent instruction between stages before completing Write-Back stage
    - **This is the necessary implementation to meet the >= 0.99 IPC requirement**!

Forwarding

- Forwarding is emulating a premature write back (Write-back -> Decode)
    - Write from a stage that may not be from Write-back
    - Write to a stage that may not be Decode
- The hardware required to perform a write back must then be duplicated into other stages
    - What hardware is this?
        - Registers containing values to write (embedded in the pipeline register)
        - Muxes to select which value to write
- Duplicated hardware may not need all inputs duplicated
    - Think what data can actually be forwarded from that stage

Forwarding

- We know what hardware is present, but when do we know to use this hardware?
  - The control logic to do this shouldn't be long . . .
  - See section 4.7 of Computer Organization and Design RISC-V edition

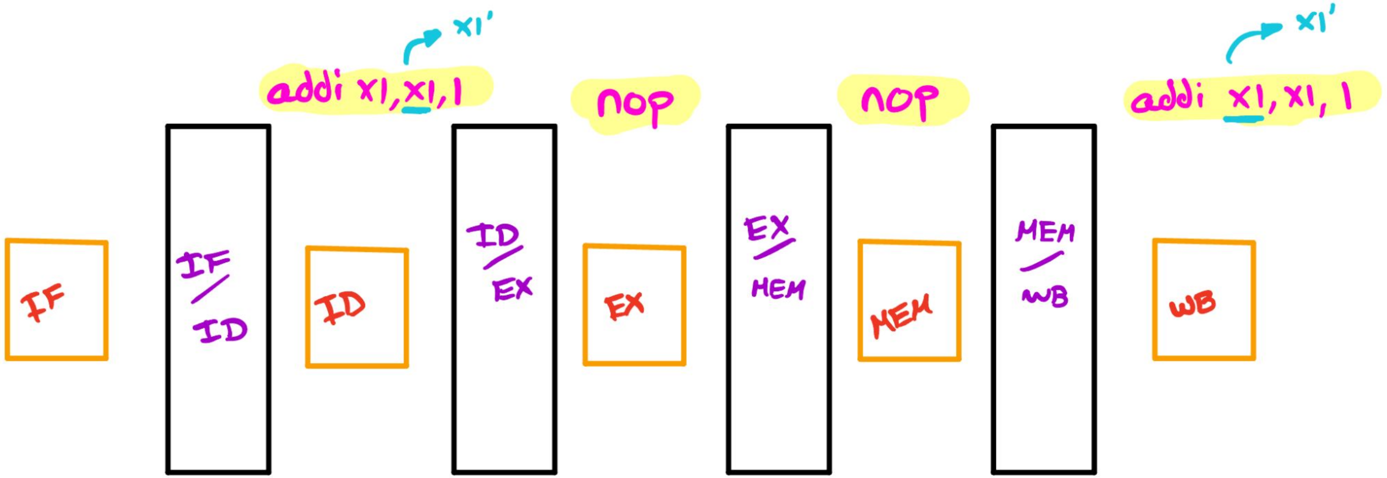
```
if   (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd ≠ 0)
and  not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
         and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
and  (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01

if   (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd ≠ 0)
and  not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
         and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
and  (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

Interesting Data Hazards

- From lecture, should know the simple forwarding cases
  - Won't review here as session already dense and focus on implementation perspective rather than theory
- We will look at the more nuanced cases here
  - Decode data hazard
  - Load data hazard

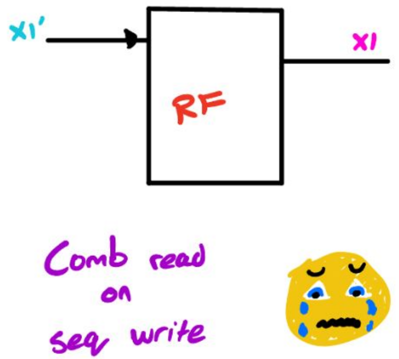
Decode Data Hazard?

- Is this a data hazard?
  - Let's assume the following regfile
    - Combinational read from the register file's registers current contents
    - Sequential write into the register file
- With these assumptions, yes we have a hazard



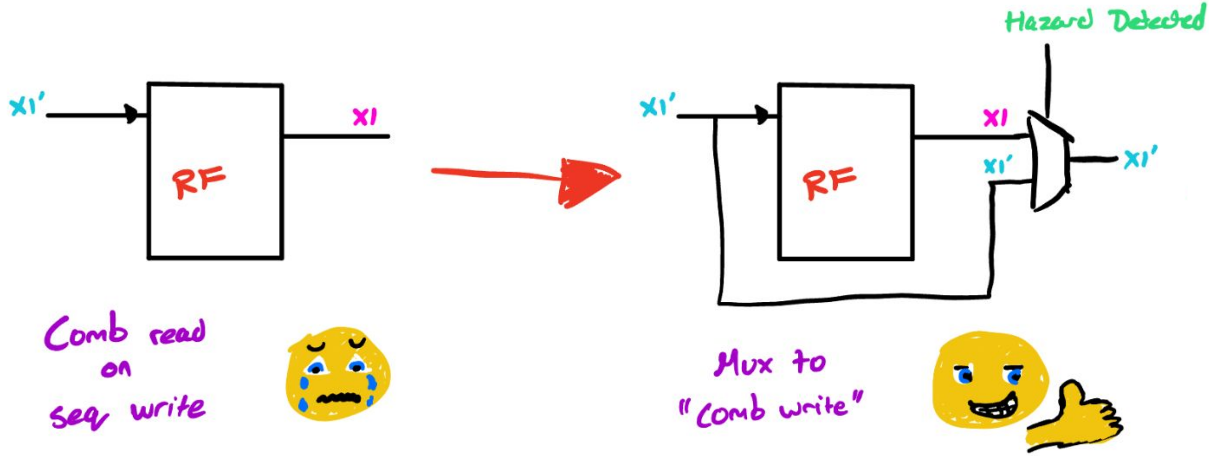
Transparent Regfile

- Writes are sequential, reads are combinational
  - But a read on a sequential write kinda makes your read sequential



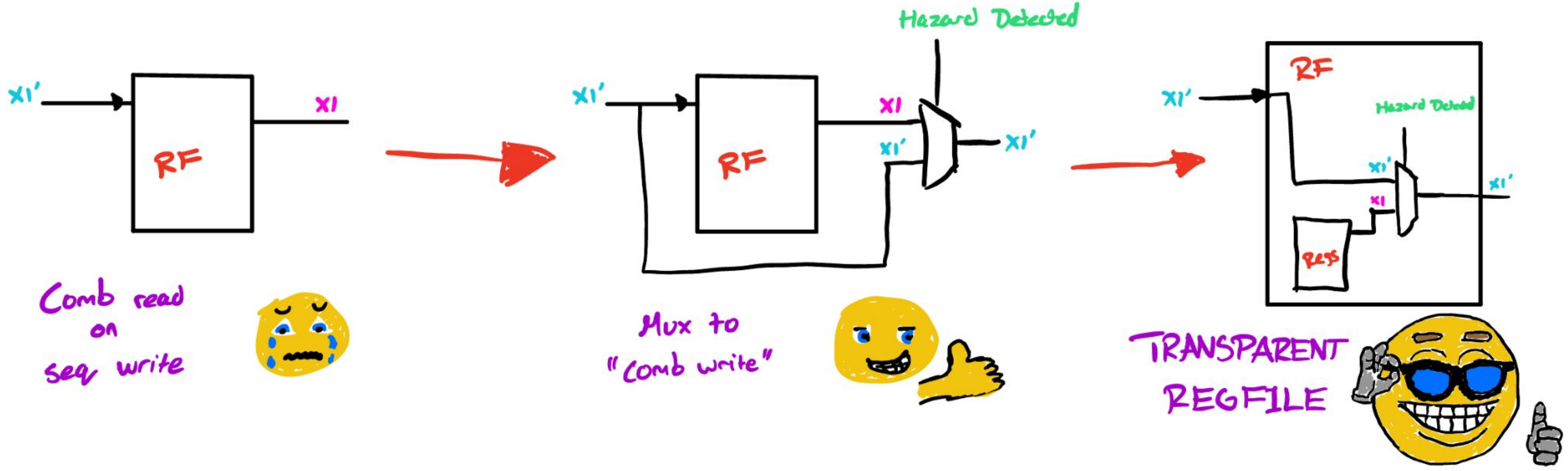
Transparent Regfile

- Writes are sequential, reads are combinational
    - But a read on a sequential write kinda makes your read sequential



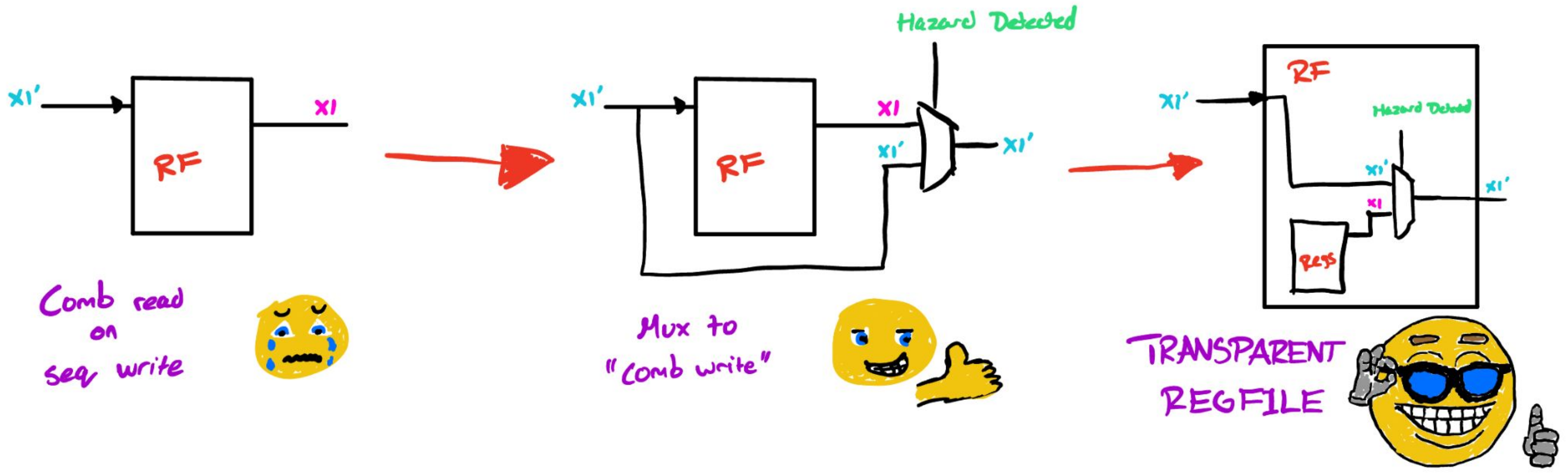
Transparent Regfile

- Writes are sequential, reads are combinational
  - But a read on a sequential write kinda makes your read sequential
    - Need a mux



Transparent Regfile

- Writes are sequential, reads are combinational
  - But a read on a sequential write kinda makes your read sequential
    - Need a mux

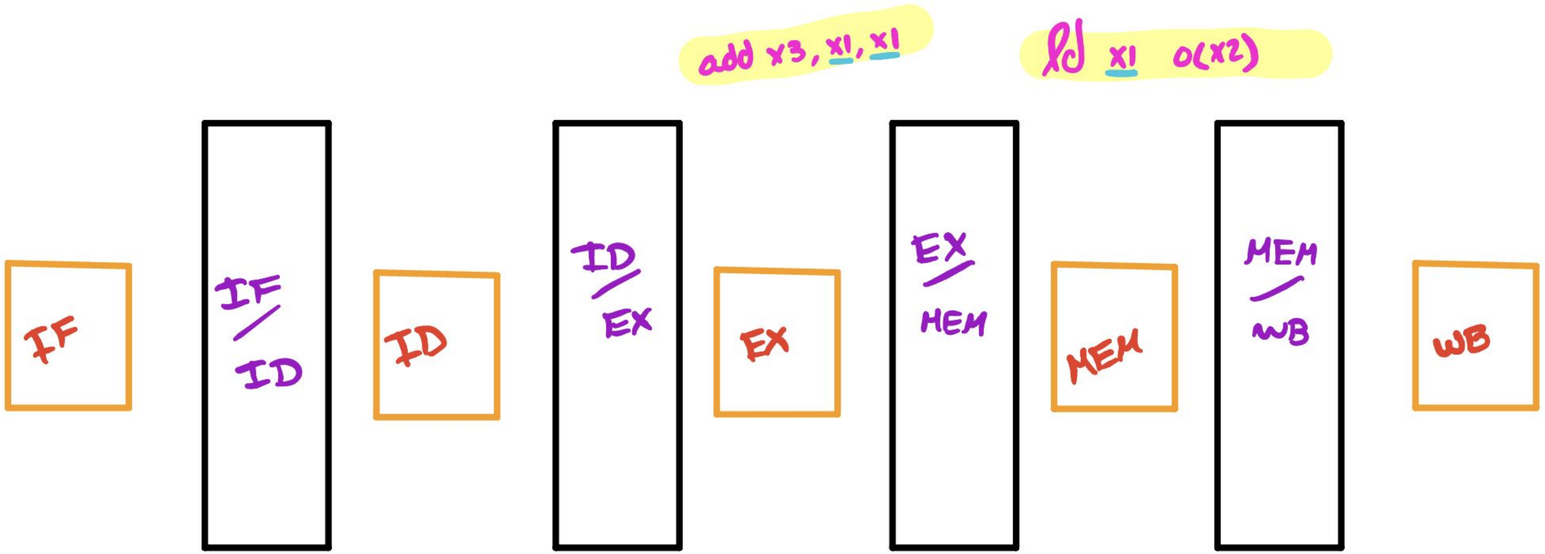


Do NOT do a posedge write and negedge read

- All always_ff sensitivity lists should only include “posedge clk”

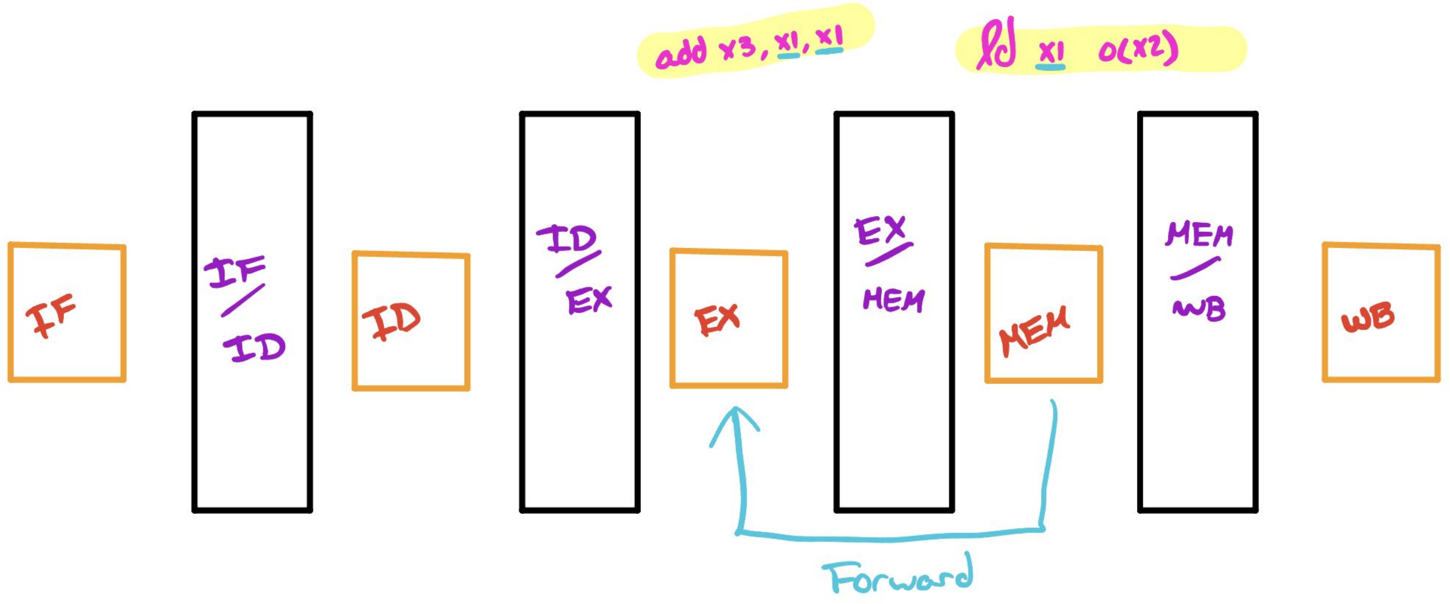
Load Data Hazard

- Is this a hazard?



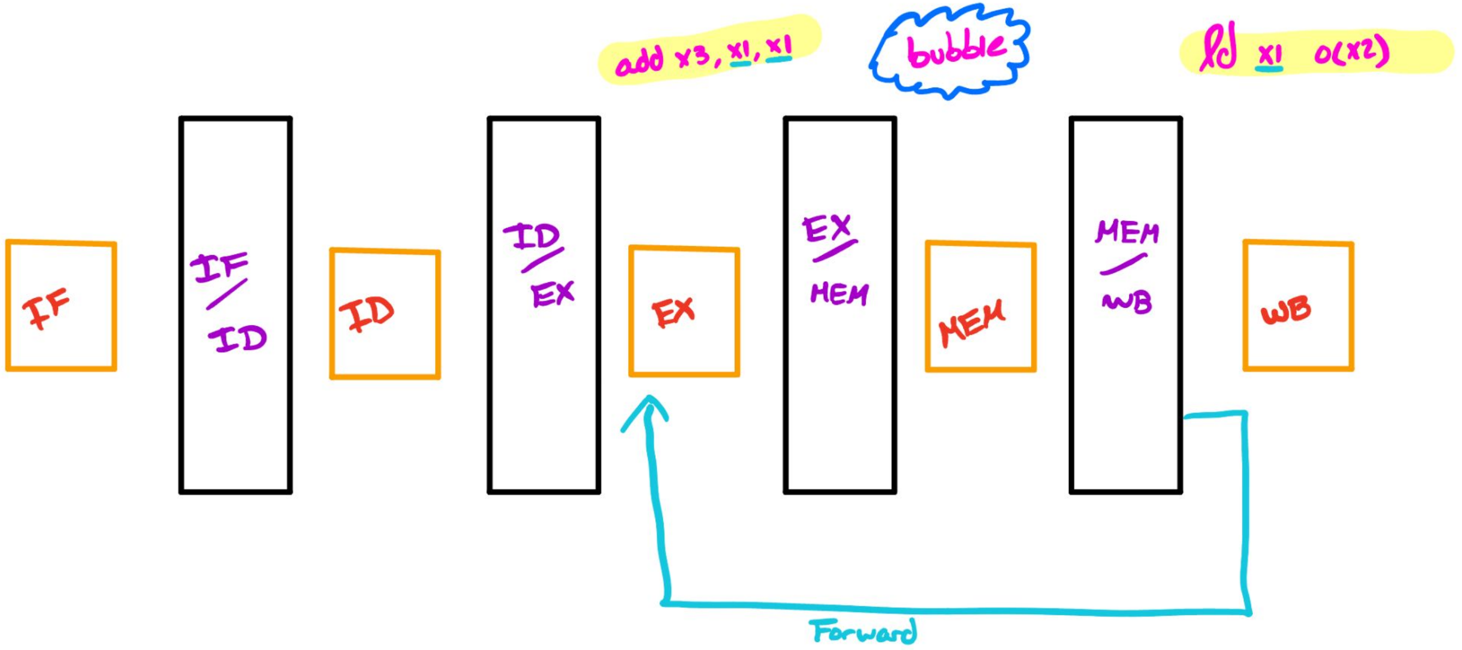
Load Data Hazard

- So forward like this?



Load Data Hazard

- Don’t actually know what to forward until after instruction leaves MEM
  - Stall EX and push load into WB and forward from there
  - Need to insert bubble in MEM



Good Luck!