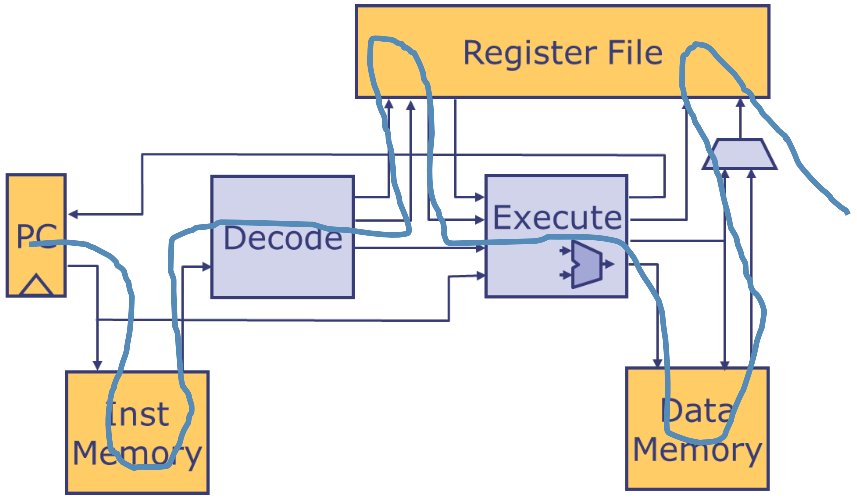
Multicycle processors and Realistic Memories

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

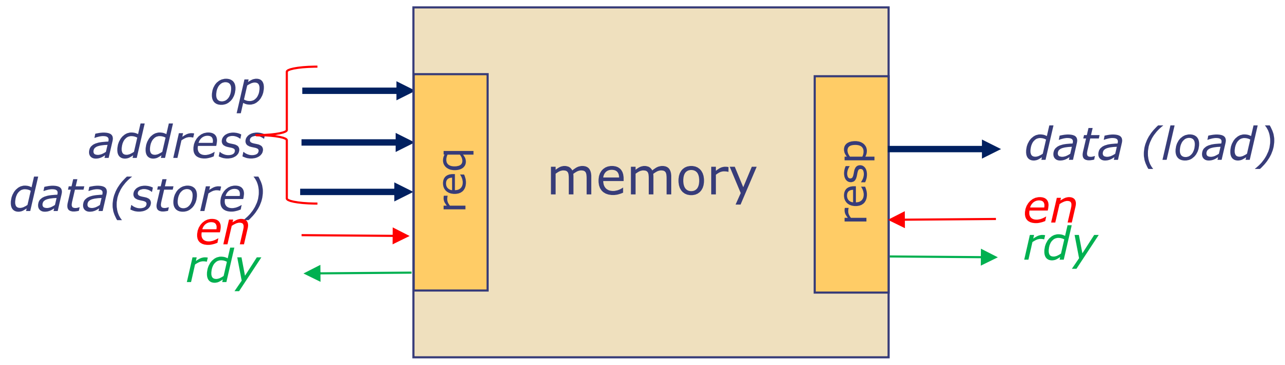
Single-cycle processor
Clock Speed



- Clock speed depends upon the longest combinational path between two state elements
  - $t_{Clock} > t_M + t_{DEC} + t_{RF} + t_{ALU} + t_M + t_{WB}$ slo…w
- We can breakdown the execution in multiple phases and execute each phase in one cycle
  - $t_{Clock} > \max\{t_M, t_{DEC} + t_{RF}, t_{ALU}, t_M, t_{WB}\}$
  - Clock will be faster but each instruction will take multiple cycles!

… but some times multicycle implementations are unavoidable

Realistic Memory Interface
Request/Response methods



No response for Stores;
Load responses come back in the requested order

```
interface Memory;
   method Action req(MemReq req);
   method ActionValue#(Word) resp();
endinterface
typedef struct {MemOp op; Word addr; Word data;}
        MemReq deriving(Bits, Eq);
typedef enum {Ld, St} MemOp deriving(Bits, Eq);
```

```
m.req(MemReq{op:Ld, addr:a, data:dwv});
m.req(MemReq{op:St, addr:a, data:v});
let data <- m.resp();
```

Request/Response methods must be called from separate rules

```
interface Memory;
   method Action req(MemReq req);
   method ActionValue#(Word) resp();
endinterface
typedef struct {MemOp op; Word addr; Word data;}
        MemReq deriving(Bits, Eq);
typedef enum {Ld, St} MemOp deriving(Bits, Eq);
```

default value

```
rule doFetch if (state == Fetch);
       m.req(MemReq{op:Ld, addr:pc, data:dwv});
       state <= Execute; x <= …;
endrule

rule doExecute if (state == Execute);
       let inst <- mem.resp;
       … decode(inst);… read x
endrule
```

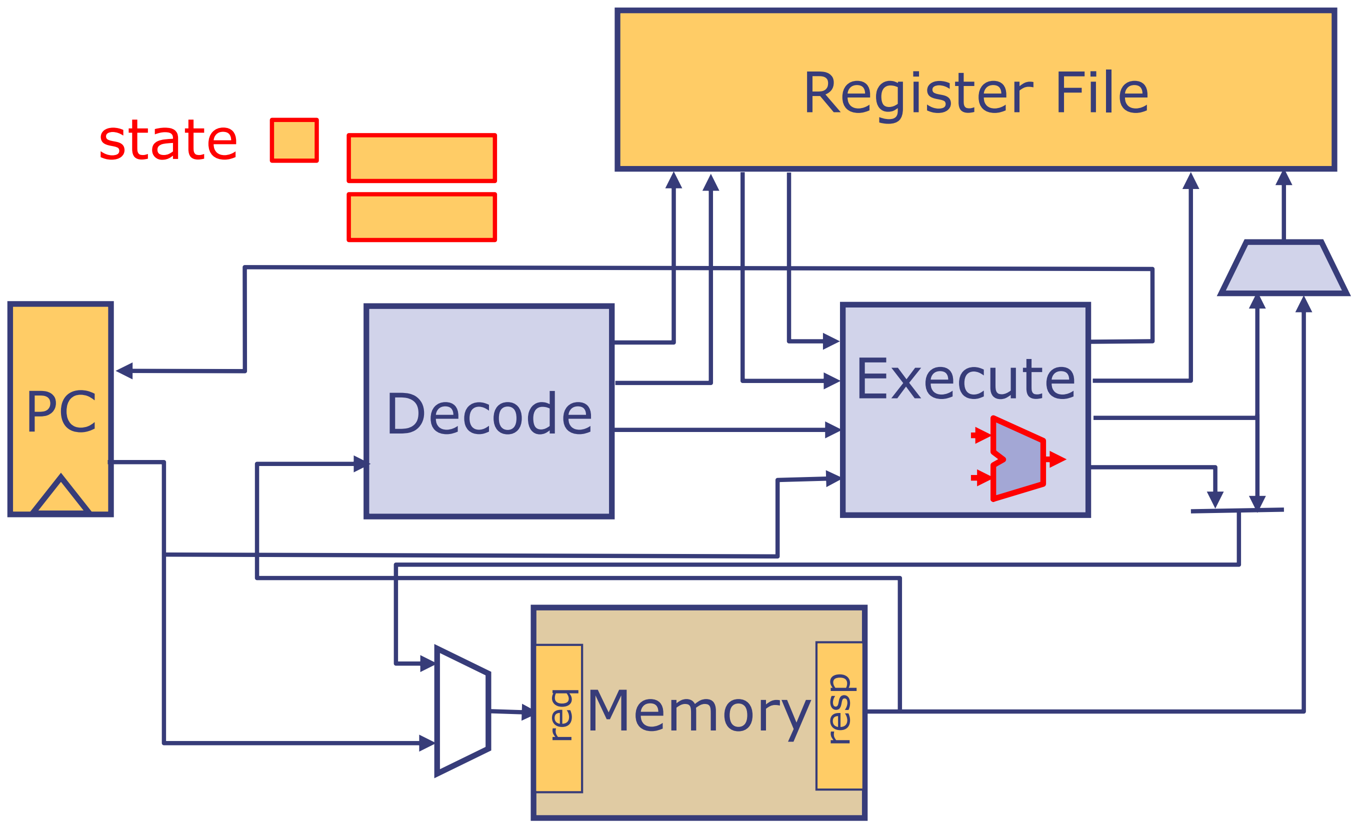
Often we need to hold the state of a partially executed instruction in new state elements between cycles

Processor with realistic memory multicycle

```
module mkProcMulticycle(Empty);
    Instantiate pc, rf, mem, and registers to hold the state of a
    partially executed instruction
    rule doFetch if (state == Fetch);
       Initiate instruction fetch; go to Execute
    rule doExecute if (state == Execute);
       let inst <- mem.resp;
       if instruction is not memory type, execute it; go to Fetch
       else initiate memory access;
       if Store, go to Fetch (Store); if Load, go to LoadWait
    rule doLoadWait if (state == LoadWait);
       Wait for the load value; update rf; go to Fetch
endmodule
```

Multicycle ALU's

any multicycle, floating point ALU operations



- Multicycle ALU's can be viewed as request/response modules
- Instructions can be further classified after decoding as simple 1 cycle, multicycle (e.g., multiply) or memory access

Processor with realistic memory and multicycle ALUs

```
module mkProcMulticycle(Empty);
    Instantiate pc, rf, mem, and registers to hold the state of a
    partially executed instruction
    rule doFetch if (state == Fetch);
       Initiate instruction fetch; go to Execute
    rule doExecute if (state == Execute);
       let inst <- mem.resp;
       if instruction is not memory type, execute it; go to Fetch
       else initiate memory access;
       if instruction is memory type, initiate memory access
       if Store, go to Fetch (Store); if Load, go to LoadWait
       if multicycle instruction; initiate it; go to MCWait
    rule doLoadWait if (state == LoadWait);
       Wait for the load value, update rf, go to Fetch
    rule doMCWait if (state == MCWait);
       Wait for MC value, update rf, go to Fetch
endmodule
```

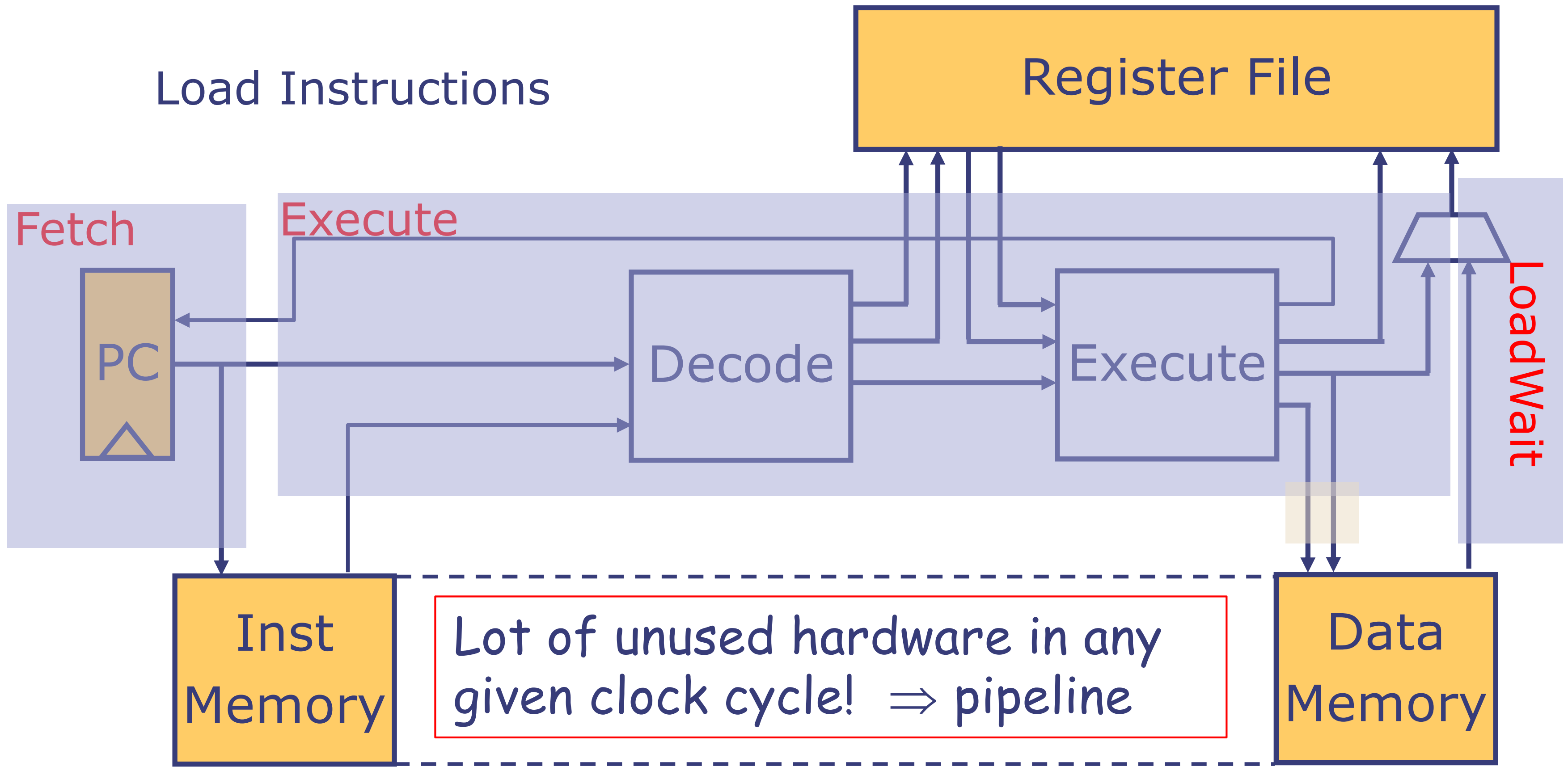
Lab 5

Reducing cycle counts further

Any disadvantage?

```
module mkProcMulticycle(Empty);
    Instantiate pc, rf, mem, and registers to hold the state of a
    partially executed instruction
    rule doFetch if (state == Fetch);
       Initiate instruction fetch; go to Execute
    rule doExecute if (state == Execute);
       let inst <- mem.resp;
       if instruction is not memory type, execute it; ~~go to Fetch~~
                                                      initiate fetch
       else initiate memory access;
       if instruction is memory type, initiate memory access
       if Store, go to Fetch (Store); if Load, go to LoadWait
       if multicycle instruction; initiate it; go to MCWait
    rule doLoadWait if (state == LoadWait);
       Wait for the load value, update rf, ~~go to Fetch~~
                                            initiate fetch
    rule doMCWait if (state == MCWait);
       Wait for MC value, update rf, ~~go to Fetch~~
                                      initiate fetch
endmodule
```

Multicycle RISC-V: *Analysis*

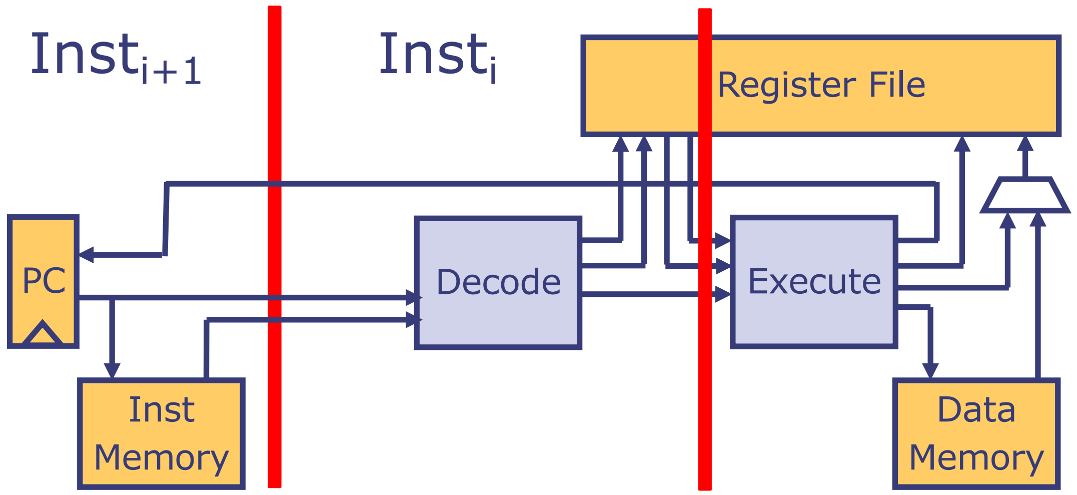


Lot of unused hardware in any given clock cycle! ⇒ pipeline

- Assuming 20% load instructions, and memory latency of one, the average number of cycles per instruction:
  - 2 x .8 + 3 x .2 = 2.2

Mullticycle memory latency will make this number much worse

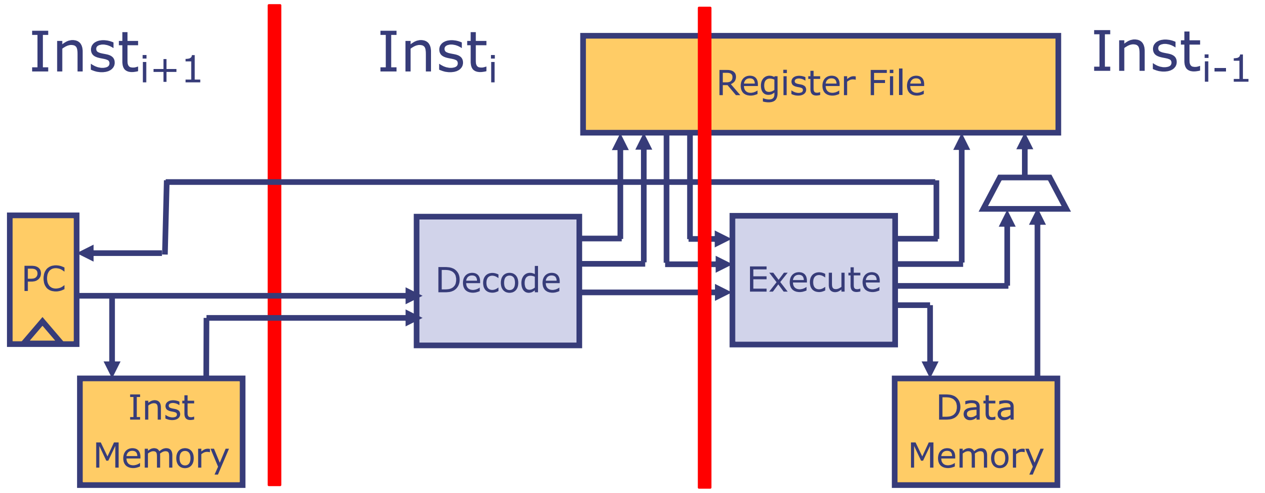
Pipeline the processor to increase its throughput



- Pipelining processor provides the ultimate challenge in computer architecture
  - Requires speculative execution of instructions to pipeline at all!
  - Requires dealing with a variety of feedbacks in the pipeline
    - Easy to make the processor functionally wrong!

The goal is always to achieve highest performance but within a given area and power budget

New problems in pipelining instructions over arithmetic pipelines



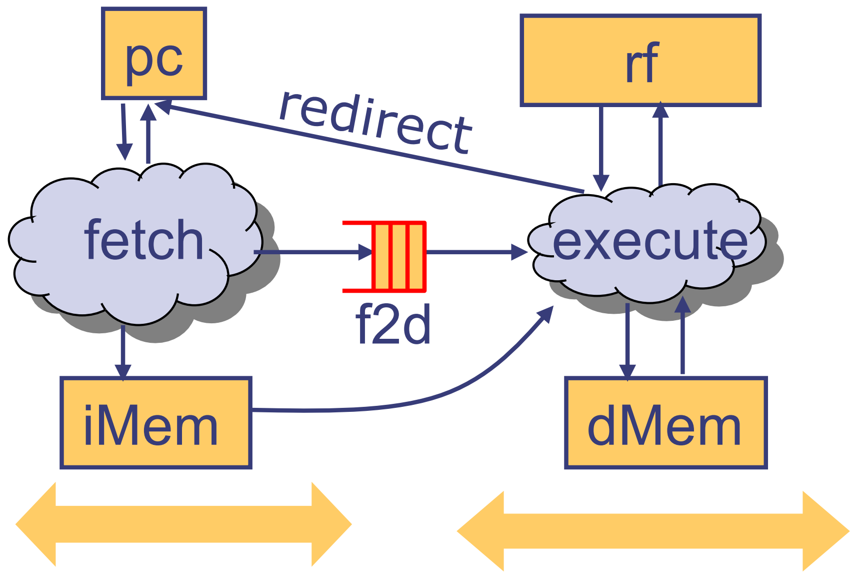
- *Control hazard:* pc for Inst_{i+1} is not known until at least Inst_i is decoded. So which instruction should be fetched?
  - Solution: *Speculate and squash* if the prediction is wrong
- *Data hazard:* Inst_i may be dependent on Inst_{i-1}, and thus, it must wait for the effect of Inst_{i-1} on the state of the machine (pc, rf, dMem) to take place
  - Solution: *Stall* instruction Inst_i until the dependency is resolved
  - Number of stalls can be reduced by *bypassing,* that is by providing additional datapaths

Plan

1. Develop a two-stage pipeline by providing a solution for *control hazards*
2. Develop a three-stage pipeline by also providing a solution for *data hazards*

Many code fragments from the multicycle implementation are resuable

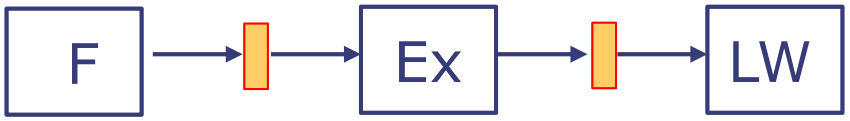
Control hazard



We will offer a solution that is independent of how many cycles each stage takes

- Fetch stage initiates instruction fetches and sends them to Execute stage via f2d. It speculatively updates pc to pc+4
- Execute stage picks up instruction from f2d and executes it. It may take one or more cycles to do this
- These two stages operate independently except in case of a branch misprediction when Execute redirects the pc to the correct pc

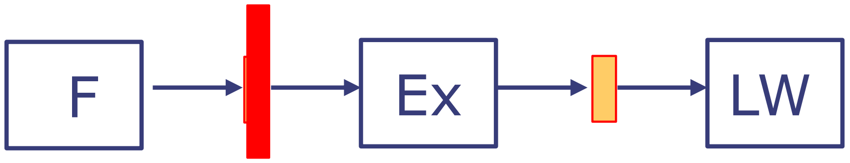
Timing diagrams and bubbles



Multicycle Processor

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
Fetch	I_0		I_1		I_2			I_3			
Ex		I_0		I_1		I_2			I_3		
LW							I_2				

Execution of
I_0; I_1; I_2; I_3; …
only I_2 is a load instruction



Two-stage Pipeline

prediction

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
Fetch	I_0	I_1	I_2	I_3							
Ex		I_0	I_1	I_2	●	I_3					
LW					I_2						

bubble: Ex/LW can hold only one instruction

Suppose I_0 is a branch instruction which jumps to I_7 instead of I_1

Fetch	I_0	I_1	I_7	I_8	
Ex		I_0	$\cancel{I_1}$	I_7	I_8
LW					

Is speculation correct?

squash

If squashing takes more than one cycle then I_7 will get further delayed

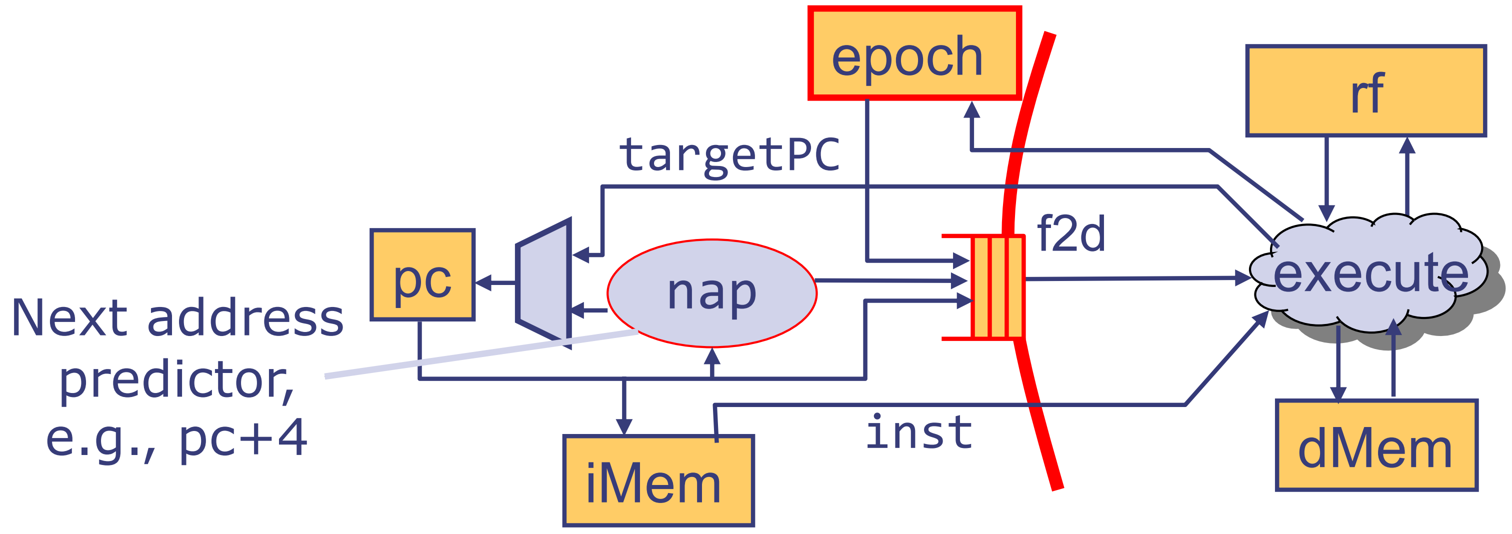
How to detect a misprediction?

- We initiate a fetch for the instruction at pc, and make a prediction for the next pc (ppc)
- The instruction at pc carries the prediction (ppc) with it as it flows through the pipeline
- At the Execute stage we know the real next pc. It is a *misprediction* if the next pc ≠ ppc

What does it mean to squash a partially executed instruction?

- A squashed instruction should have no effect on the processor state
  - must not update register file or pc
  - must not launch a Store
- These conditions are easy to ensure in our two-stage processor because there is at most one instruction in the Ex/LW state

Epoch: a method to manage control hazards



- Add an *epoch* register to the processor state
- The Execute stage changes the *epoch* whenever the pc prediction is wrong and sets the pc to the correct value
- The Fetch stage associates the current *epoch* to every instruction sent to the Execute stage
- The epoch of the instruction is examined when it is ready to execute. If the processor epoch has changed the instruction is thrown away

From multicycle to a Two-Stage Pipeline processor (1)

```
module mkProcMulticycle(Empty);
    Instantiate pc, rf, mem, and registers to hold the state of a
    partially executed instruction; epoch
    rule doFetch if ~~(state == Fetch);~~
       Initiate instruction fetch; go to Execute
    rule doExecute if ~~(state == Execute);~~
       let inst <- mem.resp;
       if instruction is not memory type, execute it; go to Fetch
       else initiate memory access;
       if Store, go to Fetch (Store); if Load, go to LoadWait
    rule doLoadWait if (state == LoadWait);
       Wait for the load value; update rf; go to Fetch
endmodule
```

But doExecute must wait if state is LoadWait

From multicycle to a Two-Stage Pipeline processor (2)

```
rule doFetch;
    iMem.req(MemReq{op: Ld, addr: pc, data: dwv});
    let ppc = nap(pc);   pc <= ppc;
    f2d.enq(F2D {pc: pc, ppc: ppc, epoch: epoch});
 endrule
```

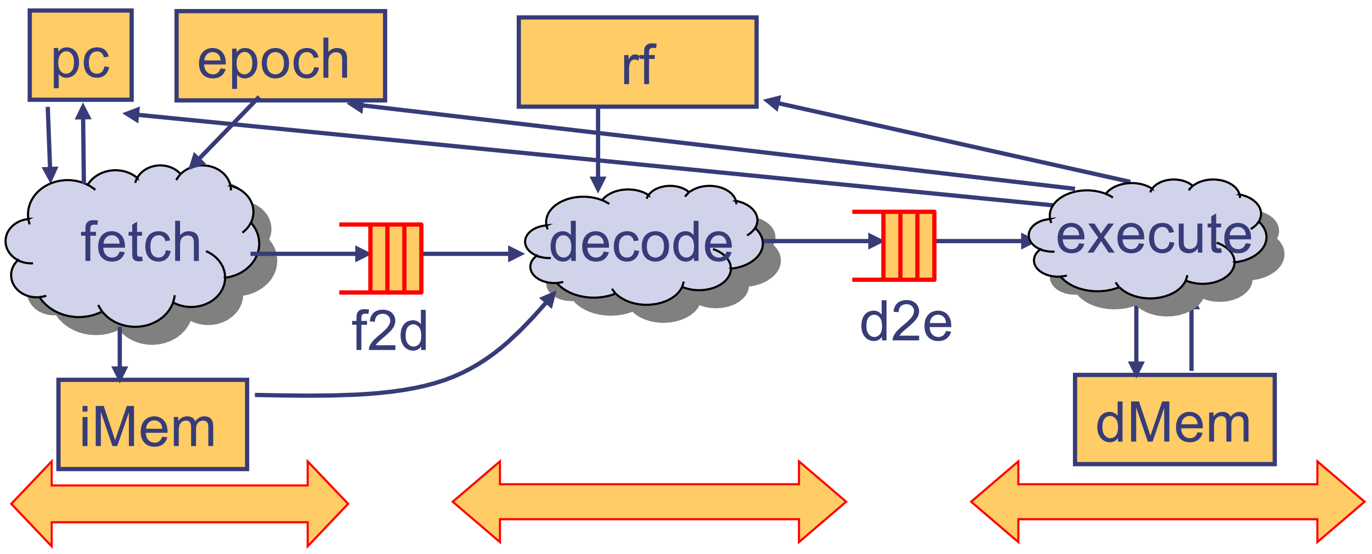
Can doFetch and doExecute execute concurrently?

solution – next time

```
rule doExecute if (state != LoadWait);
    let inst <- iMem.resp;
    let x = f2d.first; f2d.deq;
    let pcD = x.pc; let ppc = x.ppc; let epochD = x.epoch;
    if (epochD == epoch) begin  // right-path instruction
        Compute eInst from inst
        let mispred = (eInst.nextPC != ppc);
        if (mispred) begin pc <= eInst.nextPC; epoch <= !epoch; end
        Update the state;
        If a memory op, initiate memory req;
        If Ld, go to LoadWait
endrule

rule doLoadWait if (state == LoadWait); ... go to Execute ...
```

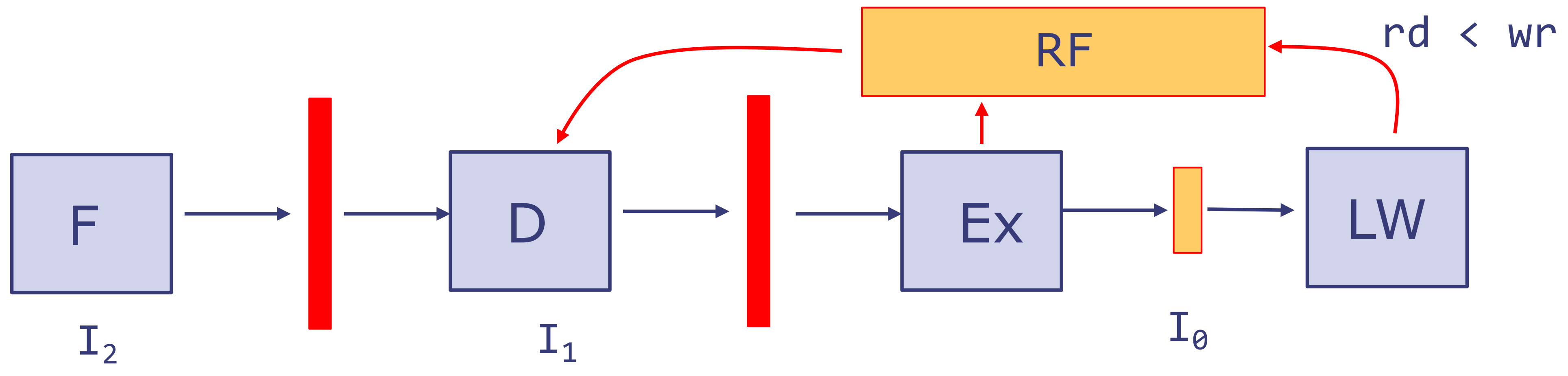
Pipelining Decode and Execute



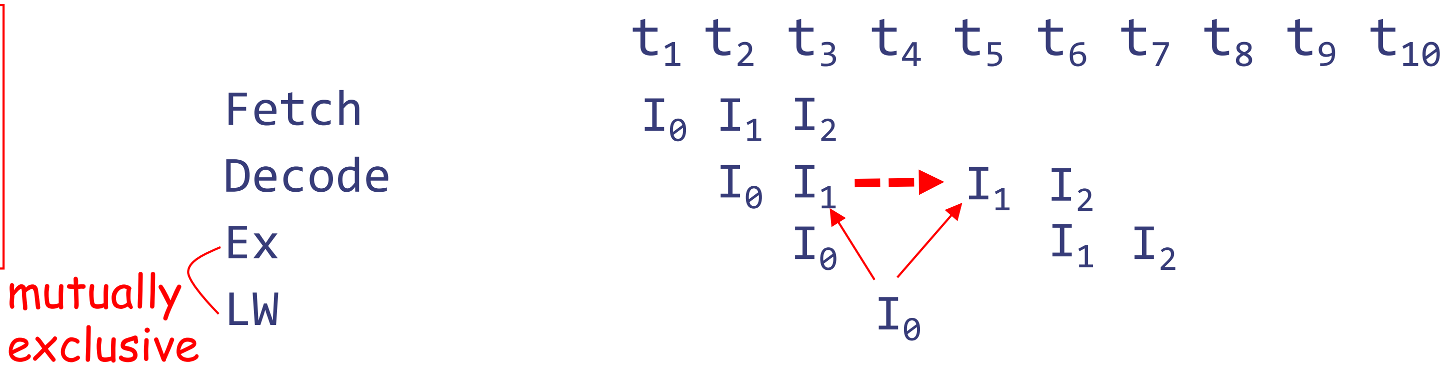
- Execute step probably has the longest propagation delay (decode + register-file read + execute)
- Separate Execute into two stages:
  - Decode and register-file-read
  - Execute – including the initiation of memory instructions
- This introduces a new problem known as a *Data Hazard*, that is, the register file, when it is read, may have stale values

Three stage pipeline

data hazard



I_0 $R1 \leftarrow$ Ld R2
I_1 $R4 \leftarrow$ R1+R2
I_2 ...



- I_1 must be stalled until I_0 updates the register file, i.e., the data hazard disappears $\Rightarrow$ *need a mechanism to stall*
- The data hazard will disappear as pipeline drains

Complication: the stalled instruction may be a wrong-path instruction

next lecture

Data Hazard

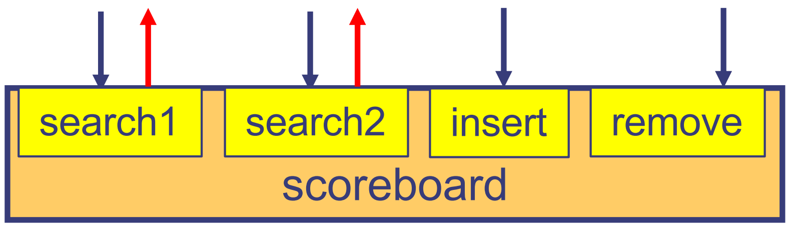
- Data hazard arises when a source register of the fetched instruction matches the destination register of an instruction already in the pipeline
- Both the source and destination registers must be valid for a hazard to exist

Dealing with data hazards
(aka read-after-write (RAW) hazard)

- Introduce a *Scoreboard* -- a data structure to keep track of the destinations of the instructions in the pipeline beyond the Decode stage
    - Initially the scoreboard is empty
- Compare sources of an instruction when it is decoded with the destinations in the scoreboard
- Stall the Decode from dispatching the instruction to Execute if there is a RAW hazard
- When the instruction is dispatched, enter its destination in the scoreboard
- When an instruction completes, delete its source from the scoreboard

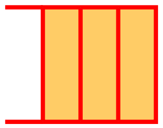
A stalled instruction will be unstalled when the RAW hazard disappears. This is guaranteed to happen as the pipeline drains.

Scoreboard



- *method insert(dst):* inserts the destination of an instruction or Invalid in the scoreboard when the instruction is decoded
- *method search1(src):* searches the scoreboard for a data hazard, i.e., a dst that matches src
- *method search2(src):* same as *search1*
- *method remove:* deletes the oldest entry when an instruction commits

Two designs for scoreboard



versus

Fifo

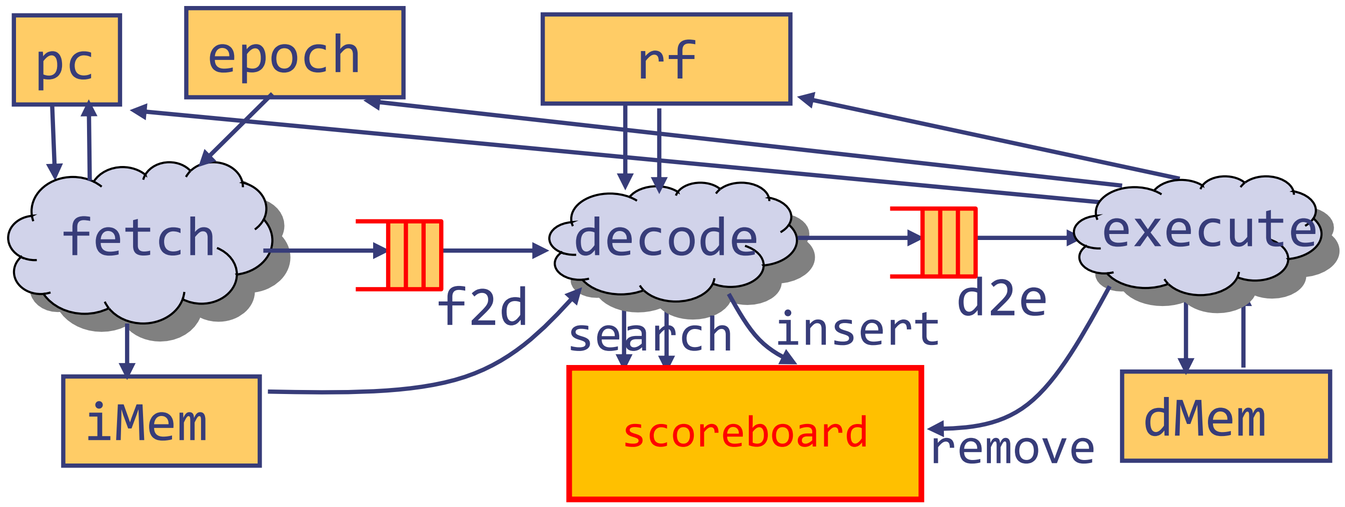
- A fifo of depth equal to the number of pipeline stages in Execute
- *Insert:* enq (dst)
- *Remove:* deq
- *Search:* compare source against each entry

Flag or counter

- One Boolean flag for each register (Initially all False)
- *Insert:* set the flag for register rd to True (block if it is already True)
- *Remove:* set the flag for register rd to False
- *Search:* Return the value of the flag for the source register

Counter design takes less hardware, especially for deep pipelines, and is more efficient because it avoids searching each element of the fifo

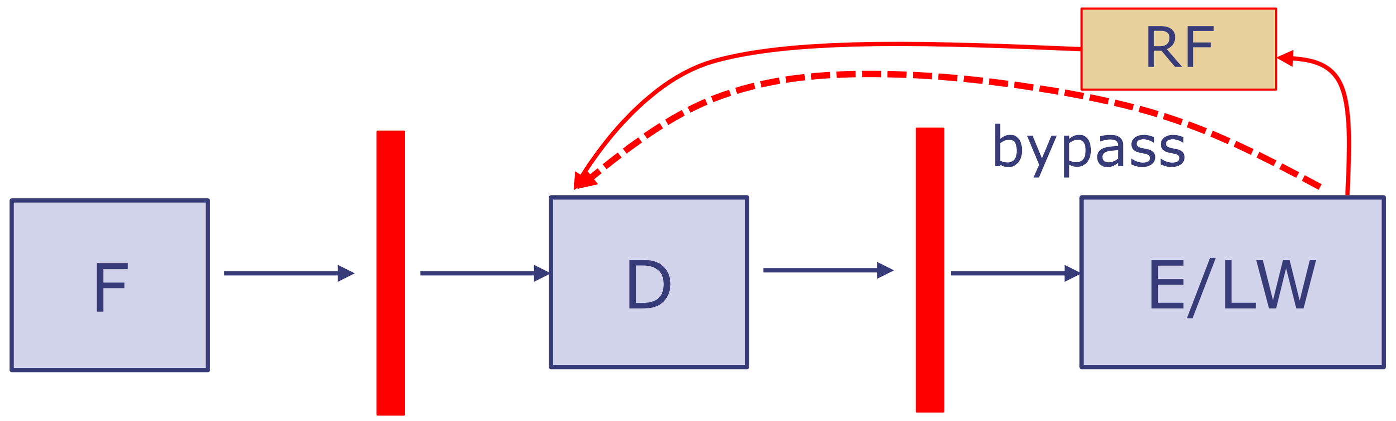
Scoreboard in the pipeline



- If search by Decode does not see an instruction in scoreboard, then that instruction must have updated the state
- Thus, when an instruction is removed from the scoreboard, its updates to Register File must be visible to the subsequent register reads in Decode
  - `remove` and `wr` should happen simultaneously
  - `search`, and `rd1, rd2` should happen simultaneously

This will require a bypass register file

Bypassing



- Bypassing is a technique to reduce the number of stalls (that is, the number of cycles) by providing extra data paths between the producer of a value and its consumer
- Bypassing introduces new combinational paths and this can increase combinational delay (and hence the clock period) and area
- The effectiveness of a bypass is determined by how often it is used

Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \underbrace{\frac{\text{Cycles}}{\text{Instruction}}}_{\text{CPI}} \cdot \underbrace{\frac{\text{Time}}{\text{Cycle}}}_{t_{Clk}}$$

- Pipelining lowers t_{Clk}. What about CPI?
- CPI = CPI_{ideal} + CPI_{hazard}
  - CPI_{ideal}: cycles per instruction if no stall
- CPI_{hazard} contributors
  - Data hazards: long operations, cache misses
  - Control hazards: branches, jumps, exceptions