# Processor pipelines

Design and performance

Instruction-level parallelism (ILP)

- To some degree, instructions may be evaluated in parallel
- ILP is done implicitly; no programmer intervention required!

Technique for exploiting ILP	Topic	Week
Pipelining (including forwarding and hazards)	Pipelined processors	2
Multiple-issue processors	Instruction scheduling	4
Dynamic instruction scheduling	Instruction scheduling	4
Branch prediction	Speculation	5

A simplified single-cycle datapath
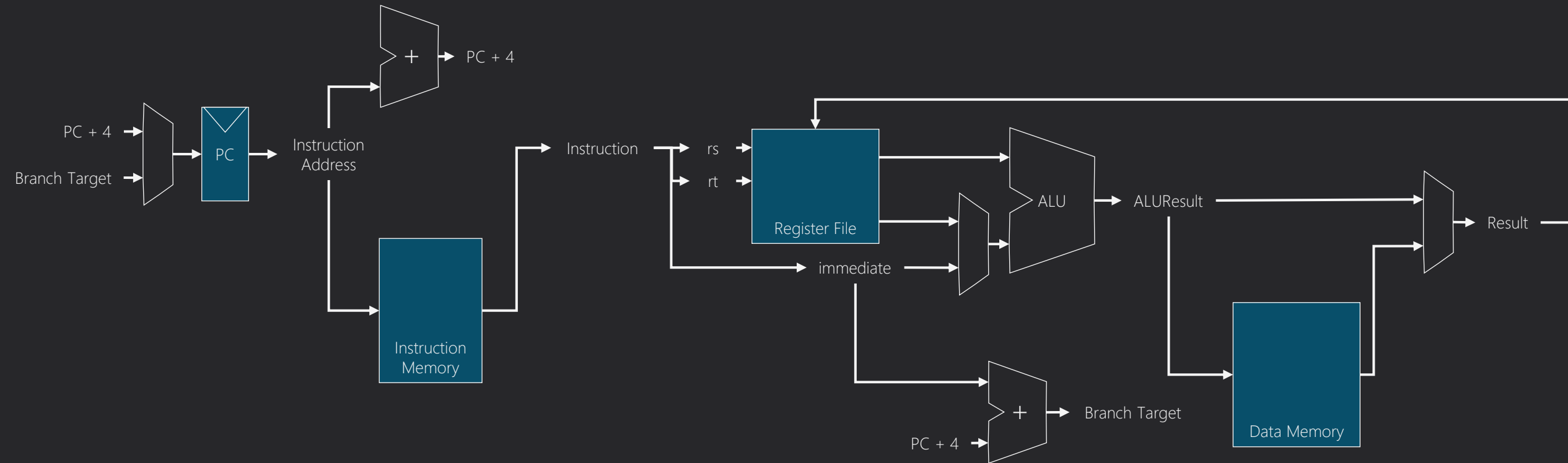
Breaking up the single-cycle datapath

4
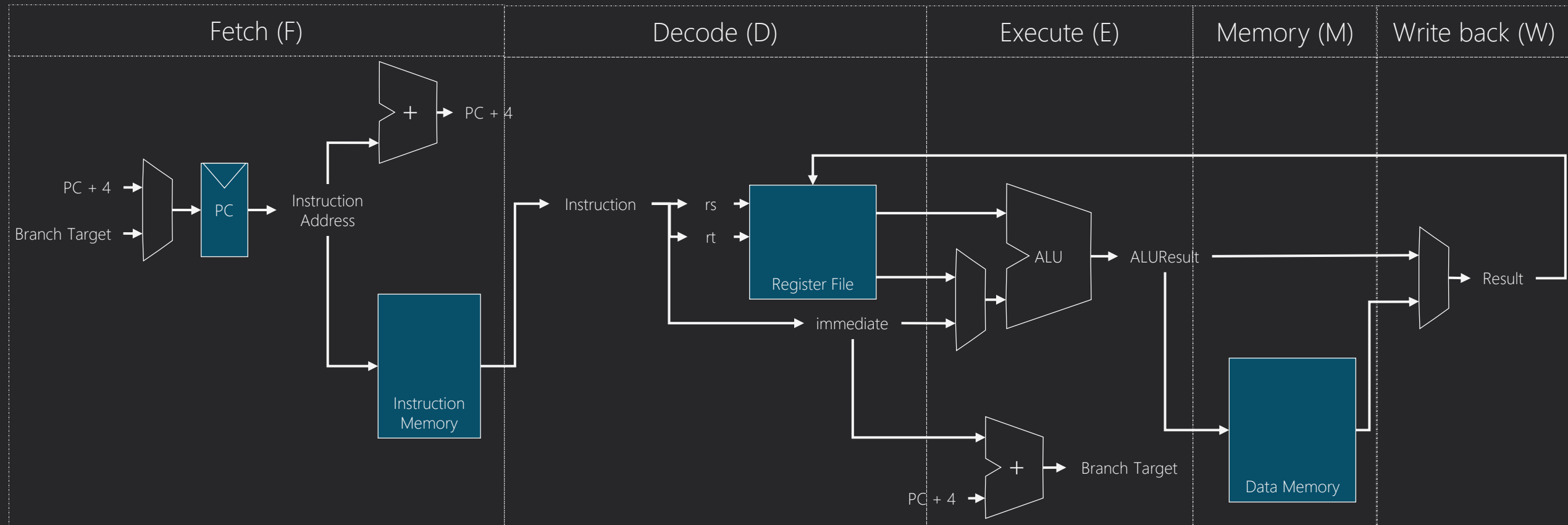
Different instructions use parts of the datapath

Instruction	Fetch	Decode	Execute	Memory	Write back
add	✓	Read rs and rt	Compute [rs] + [rt]	-	Write result to rd
Branch (e.g., beq)	✓	Read rs and rt	Compute branch target Compute [rs] – [rt] (take branch if 0)	-	-
jump	✓	Take jump	-	-	-
load	✓	Read rs	Compute [rs] + imm (address)	Read Mem[address]	Write data to rd
store	✓	Read rs and rt	Compute [rs] + imm	Mem[address] = [rt]	-

Temporal parallelism: a recap

- Pipeline stages must be balanced
    - i.e., have similar latencies

- A token must pass through all stages of the pipeline before it is complete

- Stages are divided by pipeline registers
    - Contain the data needed for the next stage's combinational element(s)

- Each stage worsens latency but improves throughput

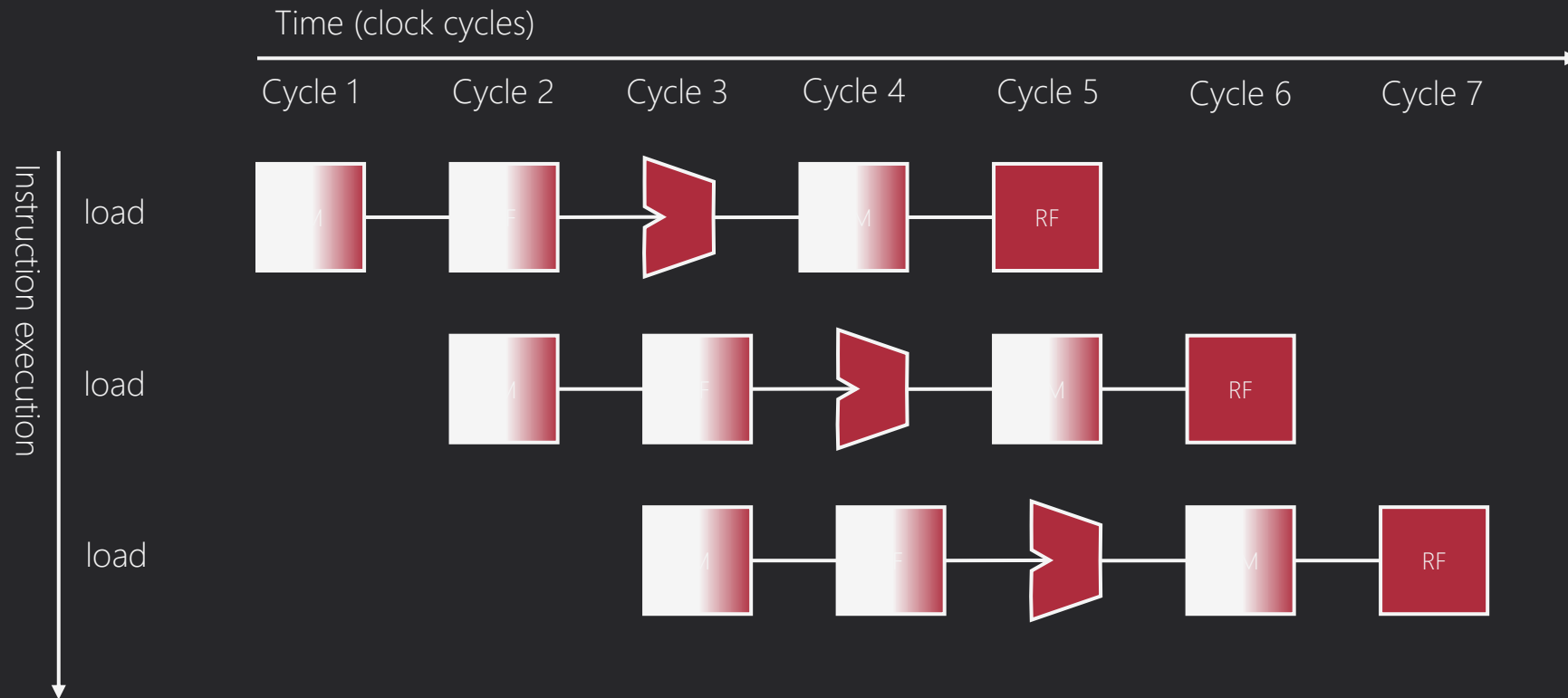- There is a limit to how many stages can be added

A simple pipeline

Assume a load-store architecture where all instructions are independent

A pipeline for independent instructions

- Insert pipeline registers between the stages of the single-cycle datapath
  - The four pipeline registers are: F/D, D/E, E/M, M/W
  - Clocked every cycle

- The pipeline registers must store all of a stage's outputs that are needed in a later stage
  - e.g., to compute the branch target in E, the PC + 4 value computed in F needs to be propagated through the pipeline

Visualizing pipelined datapath use

Time (clock cycles)

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7

Instruction execution

load

load

load

- Shaded region shows when a structure is being used
  - Full cycle
  - First half of cycle
  - Second half of cycle

- The register file is:
  - Read in the second half of cycle
  - Written to in first half of cycle

- No duplication of hardware is needed

The pipeline registers

Stage	Inputs	Outputs	Propagate
Fetch	Branch target (from E/M)	PC + 4 Instruction data	Nothing
Decode			
Execute			
Memory			
Write back			

The pipeline registers

Stage	Inputs	Outputs	Propagate
Fetch	Branch target (from E/M)	PC + 4 Instruction data	Nothing
Decode	Instruction data (from F/D)	Values of rs and rt Immediate Register address rd	PC + 4
Execute	PC + 4 (from D/E) Values of rs and rt Immediate	Branch target ALU Result	Value of rt Register address rd
Memory	ALU Result (address) Value of rt	Data read from memory	ALU Result (data) Register address rd
Write back	ALU Result Data read from memory	(may update register file at register address rd)	Nothing

A pipeline diagram

Time (cycles) →

Time (instructions) ↓

	1	2	3	4	5	6	7	8	9
addi r1, r2, 1	F	D	E	M	W				
lw r3, 0(r4)		F	D	E	M	W			
sw r5, 8(r6)			F	D	E	M	W		
add r7, r8, r9				F	D	E	M	W	
mul r10, r11, r12					F	D	E	M	W

Data dependencies

Defining dependences and identifying pipeline hazards

Dependencies versus hazards

Data dependencies

- May have dependencies between register operands or memory operands

- A property of software

- For example,

  ```
  add r1, r2, r3 # r1 = r2 + r3
  add r4, r1, r1 # r4 = r1 + r1
  ```

Data hazards

- When an instruction in the pipeline depends on another instruction that hasn't finished yet

- A data hazard in one implementation may not exist in another
  - e.g., there are no data hazards in a single-cycle processor

Data dependencies

- A and B are instructions, and
- A appears earlier than B in the instruction stream

Dependency	Description	Potential problem
Read after write (RAW)	B needs the result from A	• If B does not wait for A, then • it reads an old value
Write after read (WAR)	B writes to an operand that is read by A	• If B finishes before A, then • A reads a future value
Write after write (WAW)	B writes to the same operand as A	• If B finishes before A, then • A overwrites a future value

Data hazards in our five-stage pipeline

Hazard	Possible?	Explanation
RAW	Yes	Several possibilities exist!
WAR	No	Only the decode stage reads from the register file. All instructions reach decode in program order, and those values are propagated through pipeline registers.
WAW	No	Only the write back stage updates the register file. All instructions reach write back in program order.

RAW: add needs result of add

	1	2	3	4	5	6	7	8	9
addi r1, r2, 4	F	D	E	M	W				
add r3, r1, r4		F	D	E	M	W			

	1	2	3	4	5	6	7	8	9
addi r1, r2, 4	F	D	E	M	W				
NOOP		F	D	E	M	W			
NOOP			F	D	E	M	W		
NOOP				F	D	E	M	W	
add r3, r1, r4					F	D	E	M	W

- Dependency: r1
- addi writes r1 in cycle 5
- add reads r1 in cycle 3

- Solution?
  - The compiler adds "no op" instructions

The hazard detection unit

- Detect the data hazard in hardware.
  - i.e., don't rely on compiler/programmer adding NOOPs

- Where should we detect data hazards?
  - Decode stage.

- Stall the instruction in the decode stage.
  - i.e., don't read the register file until the data is ready.

The impact of stalling

	1	2	3	4	5	6	7	8	9
addi r1, r2, 4	F	D	E	M	W				
add r3, r1, r4		F	D*	D*	D*	D	E	M	W
lw r7, 0(r8)						F	D	E	M

Suppose that RAW hazards occur 20% of the time. Assuming no other penalties, what is the CPI of our five-stage pipeline?

- Use "D*" to indicate that decode cannot proceed (stalled)

- If add is stalled, then the next instruction cannot be fetched
  - Otherwise, it would overwrite the data in F/D

The impact of stalling

	1	2	3	4	5	6	7	8	9
addi r1, r2, 4	F	D	E	M	W				
add r3, r1, r4		F	D*	D*	D*	D	E	M	W
lw r7, 0(r8)						F	D	E	M

- Use "D*" to indicate that decode cannot proceed (stalled)

- If add is stalled, then the next instruction cannot be fetched
  - Otherwise, it would overwrite the data in F/D

Suppose that RAW hazards occur 20% of the time. Assuming no other penalties, what is the CPI of our five-stage pipeline?

Recall: $CPI = CPI_{base} + CPI_{penalty}$

The base CPI is 1.

Whenever we encounter a raw hazard, we stall for 3 cycles.

So, $CPI = 1 + (0.2 \times 3) = 1.6$

Solving data hazards with forwarding

Also called "bypassing"

Register forwarding

- Solve some data hazards by forwarding results in pipeline registers back to the execute stage
    - If the data hazard cannot be solved with forwarding, the pipeline must stall

- Extra hardware is needed
    - Multiplexers for the ALU operand inputs
    - More logic in the hazard detection unit

Revising our earlier RAW hazard

	1	2	3	4	5	6	7	8	9
addi r1, r2, 4	F	D	E	M	W				
add r3, r1, r4		F	D	E	M	W			

- Observe that the operation r2 + 4 finishes in cycle 3

- So, in cycle 4, the result of r2 + 4 is available in the E/M pipeline register

D/E

ALU

ALUResult

immediate

E/M

ALUResult of previous instruction
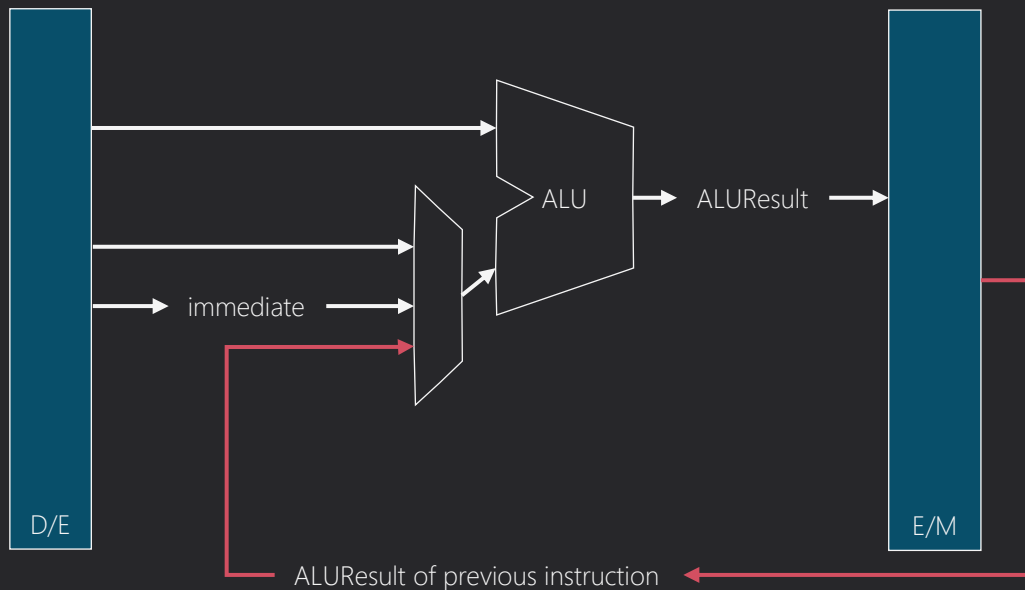
Using a write back to execute bypass

	1	2	3	4	5	6	7	8	9
add r1, r2, r3	F	D	E	M	W				
lw r4, 0(r5)		F	D	E	M	W			
sub r6, r1, r7			F	D	E	M	W		

- Dependency: r1
- add writes r1 in cycle 5
- sub reads r1 in cycle 4

- The result from cycle 3 is in...
  - E/M in cycle 4
  - M/W in cycle 5

- Note: the value for r1 in D/E on cycle 5 is stale

An example of load-to-use

	1	2	3	4	5	6	7	8	9
lw r1, 0(r5)	F	D	E	M	W				
add r3, r1, r4		F	D*	D	E	M	W		

- Load instructions don't have the result until the end of the memory stage.

Suppose that we add *full bypassing* to our five-stage pipeline. RAW hazards occur 20% of the time. And 50% of those RAW hazards are load-to-use. Assuming no other penalties, what is the CPI of our five-stage pipeline?

An example of load-to-use

	1	2	3	4	5	6	7	8	9
lw r1, 0(r5)	F	D	E	M	W				
add r3, r1, r4		F	D*	D	E	M	W		

- Load instructions don't have the result until the end of the memory stage.

Suppose that we add *full bypassing* to our five-stage pipeline. RAW hazards occur 20% of the time. And 50% of those RAW hazards are load-to-use. Assuming no other penalties, what is the CPI of our five-stage pipeline?

$$CPI = 1 + (0.5 \times 0.2 \times 1) = 1.1$$

Example: compiler optimizations for data hazards

- Compiler can re-order instructions to reduce stalls for different microarchitectures.
  - Constraint: The compiler needs to ensure proper execution.

Consider the high-level language code:

```
a = b + c;
d = e + f;
```

Compile the code into assembly for a load-store architecture
1. Without any optimizations
2. With optimizations for a 5-stage pipeline with full bypassing

Solution: compiler optimizations for data hazards

No optimizations

```
lw r2, 4(sp)
lw r3, 8(sp)
add r1, r2, r3
sw r1, 0(sp) // a = b + c
lw r5, 16(sp)
lw r6, 20(sp)
add r4, r5, r6
sw r4, 12(sp) // d = e + f
```

With optimizations (full bypassing)

Solution: compiler optimizations for data hazards

No optimizations

```
lw r2, 4(sp)
lw r3, 8(sp)
add r1, r2, r3
sw r1, 0(sp)   // a = b + c
lw r5, 16(sp)
lw r6, 20(sp)
add r4, r5, r6
sw r4, 12(sp)  // d = e + f
```

With optimizations (full bypassing)

```
lw r2, 4(sp)
lw r3, 8(sp)
lw r5, 16(sp)
add r1, r2, r3
lw r6, 20(sp)
sw r1, 0(sp)   // a = b + c
add r4, r5, r6
sw r4, 12(sp)  // d = e + f
```

Control hazards

Dealing with branches

Conditional branches cause control hazards

- Our pipeline fetches a new instruction every cycle
  - Usually PC + 4, but for taken branches needs to be Branch Taken

- But when do we know if a branch instruction is taken?
  - The delay to determine the right instruction to fetch is a control hazard

- If the branch is taken...
  - Then the fetch stage was reading the wrong addresses

- If the branch is not taken
  - Then the fetch stage was reading the correct addresses

Predict branches are not taken

- The hardware assumes that branch instructions won't be taken

- If the prediction is wrong, then there are (older) instructions in our pipeline that should not have been executed!

- We need to *flush* the pipeline of these older instructions
  - Essentially turns the instructions into no-ops
  - This is a performance hit

Stalling vs. predicting not taken

Performance with stalls

- Suppose branches occur 20% of the time and that we must stall for 2 cycles to resolve the branch. Assuming no other penalties, what is the CPI of our five-stage pipeline?

Performance with not taken prediction

- Consider a five-stage pipeline that predicts not taken for branches. Suppose branches occur 20% of the time, and the penalty for a misprediction (flush) is 2 cycles. Assuming no other penalties, what is the CPI of our five-stage pipeline?

Stalling vs. predicting not taken

Performance with stalls

- Suppose branches occur 20% of the time and that we must stall for 2 cycles to resolve the branch. Assuming no other penalties, what is the CPI of our five-stage pipeline?

$$CPI = 1 + (0.2 \times 2) = 1.4$$

Performance with not taken prediction

- Consider a five-stage pipeline that predicts not taken for branches. Suppose branches occur 20% of the time, and the penalty for a misprediction (flush) is 2 cycles. Assuming no other penalties, what is the CPI of our five-stage pipeline?

$$CPI = 1 + (0.2 \times P \times 2) = 1 + 0.4 \times P$$

... where P is the probability a branch is taken

Branch prediction and performance

35
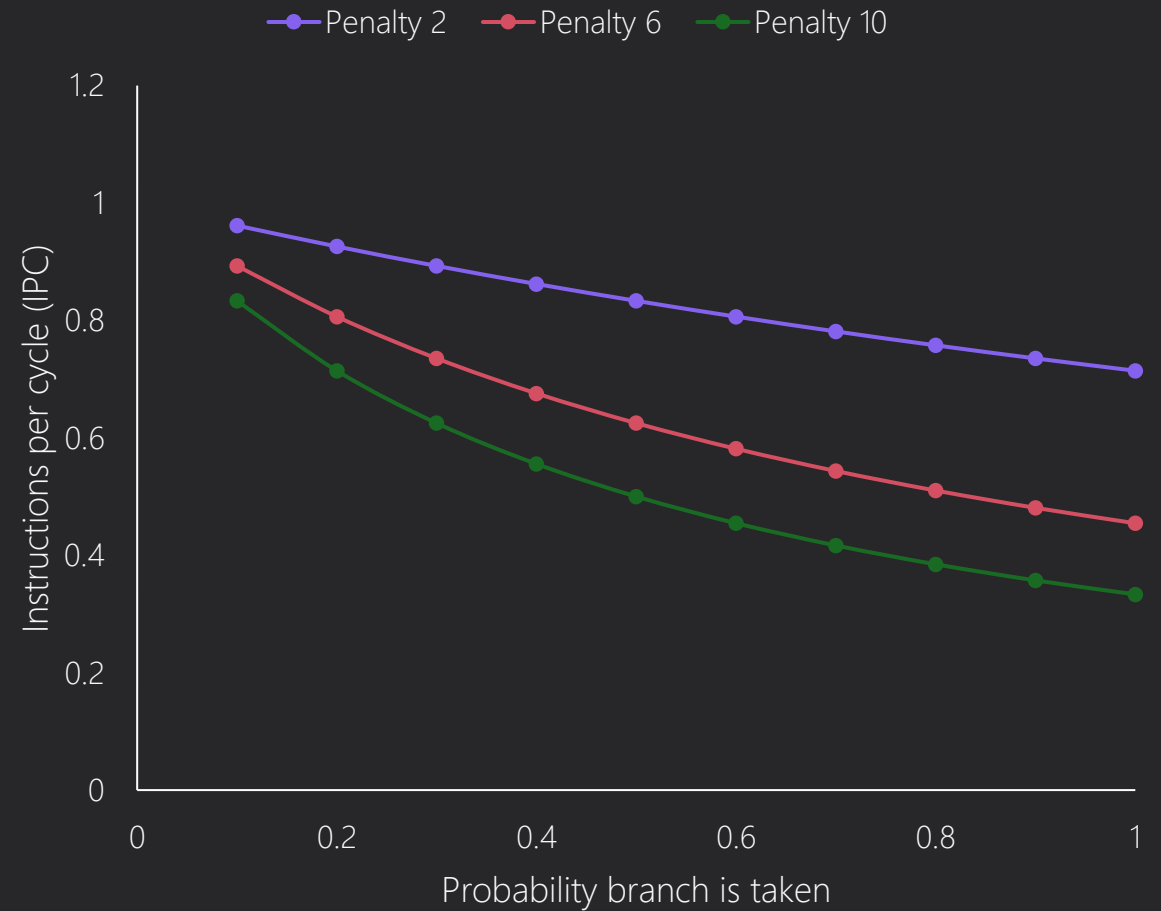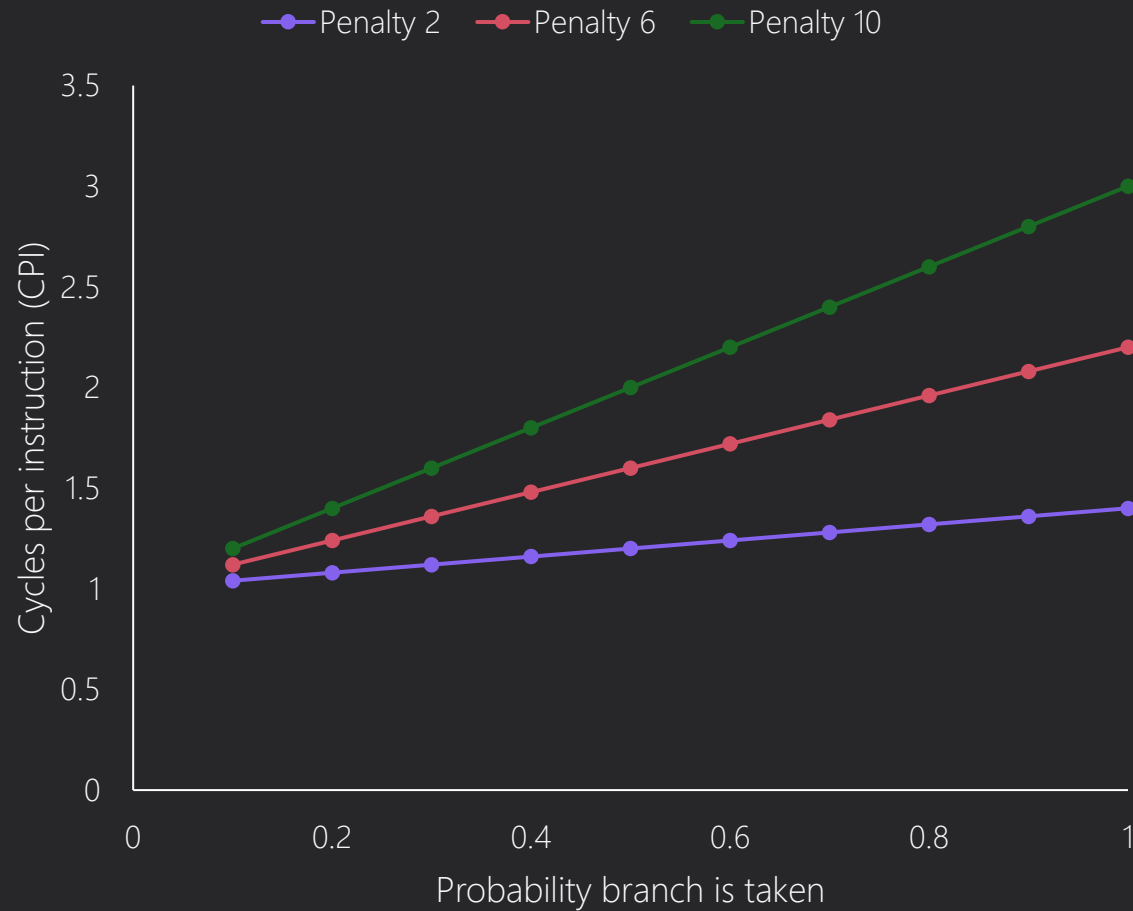
Example: compiler optimizations for branches

- Identify loops and, if possible, re-write them as repeating instructions
  - Called "loop unrolling"

- Trade-off
  - Reduce number of branches
  - Increase code size

Consider the high-level language code for adding two vectors, A and B:

```
for (int i = 0; i < 2; i++)
    A[i] = A[i] + B[i];
```

Compile the code into assembly for a load-store architecture:

- Using a branch for the loop

- Using no branches

Solution: compiler optimizations for branches

With branches

```
addi r6, r0, 2      // i = 2
loop:
    lw r3, 0(r1)
    lw r4, 0(r2)
    add r5, r3, r4   // r5 = A[i] + B[i]
    sw r5, 0(r1)     // A[i] = r5
    addi r1, r1, 4
    addi r2, r2, 4
    subi r6, r6, 1
    bnez r6, loop
```

Without branches

Solution: compiler optimizations for branches

With branches

```
addi r6, r0, 2          // i = 2

loop:
    lw r3, 0(r1)
    lw r4, 0(r2)
    add r5, r3, r4      // r5 = A[i] + B[i]
    sw r5, 0(r1)        // A[i] = r5
    addi r1, r1, 4
    addi r2, r2, 4
    subi r6, r6, 1
    bnez r6, loop
```

Without branches

```
lw r3, 0(r1)

lw r4, 0(r2)

add r5, r3, r4      // r5 = A[0] + B[0]

sw r5, 0(r1)        // A[0] = r5

addi r1, r1, 4

addi r2, r2, 4


lw r3, 0(r1)

lw r4, 0(r2)

add r5, r3, r4      // r5 = A[1] + B[1]

sw r5, 0(r1)        // A[1] = r5

addi r1, r1, 4

addi r2, r2, 4
```
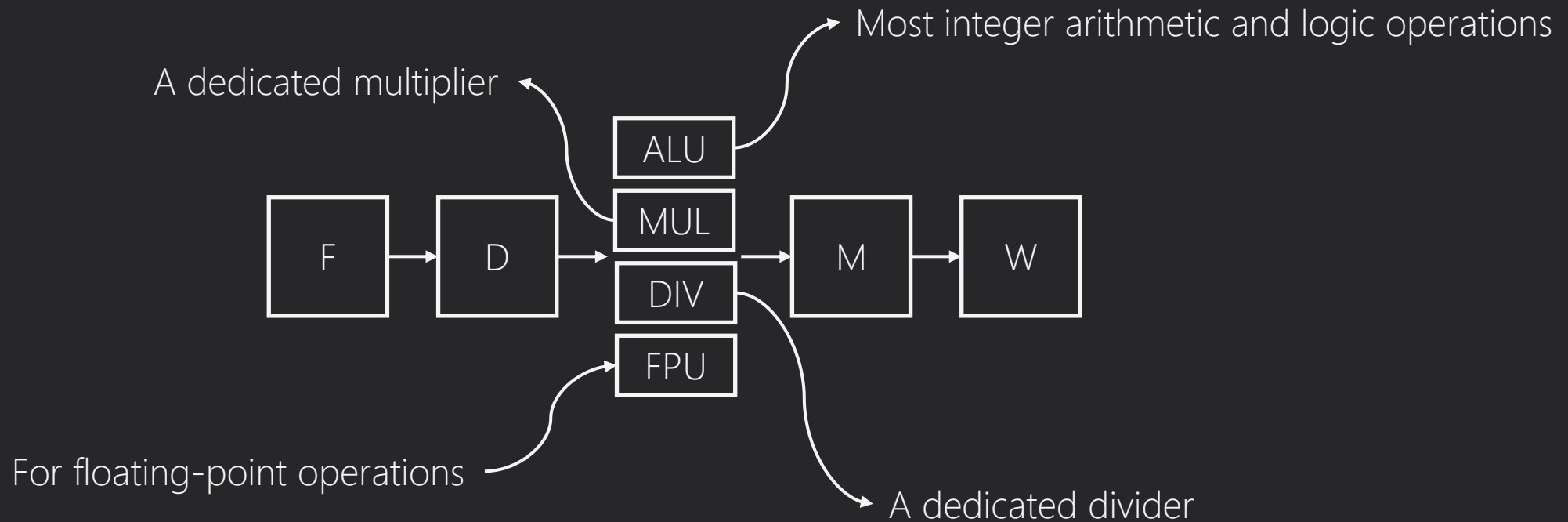
Out-of-order execution

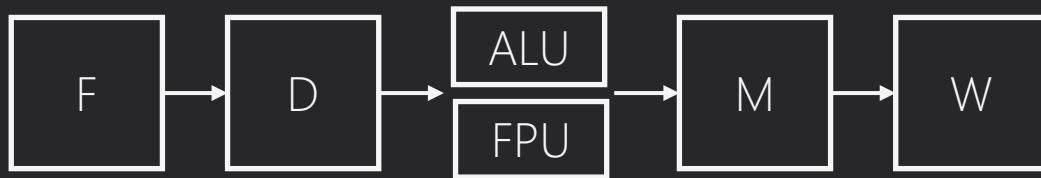Introducing functional units and structural hazards

Functional units

Most integer arithmetic and logic operations

A dedicated multiplier

| ALU |
| MUL |
| DIV |
| FPU |

F → D → M → W

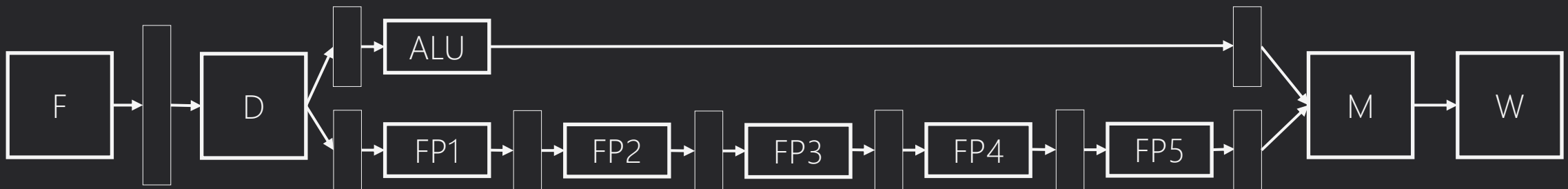For floating-point operations

A dedicated divider

Functional unit (FU) latency

- Initiation interval: minimum time for two instructions to start in a function unit

Functional unit	Latency	Initiation interval
ALU	1	1
FPU	5	Pipelined: 1 Multi-cycle: 5

Data hazards in this pipeline

RAW hazards

- Latency of operation
  - Cycles a subsequent dependent instruction must wait to avoid a RAW hazard
  - Latency of operation = Latency of FU – 1

- RAW hazard stalls are more frequent than the 5-stage pipeline
  - Why?

WAW hazards

- This pipeline allows instructions to complete out-of-order
  - How?

Example: long RAW hazard latency

	1	2	3	4	5	6	7	8	9	10	11	12
l.d f4, 0(r2)	F	D	E	M	W							
mul.d f0, f4, f6		F	D*	D	FP1	FP2	FP3	FP4	FP5	M	W	
s.d f0, 0(r2)			F*	F	D*	D*	D*	D*	D*	E	M	W

- The ".d" instruction suffix indicates floating point operations
- The "f" register name prefix indicates a floating point register

Example: structural hazard on write port

	1	2	3	4	5	6	7	8	9	10
add.d f1, f2, f1	F	D	FP1	FP2	FP3	FP4	FP5	M	W	
add.d f4, f2, f3		F	D	FP1	FP2	FP3	FP4	FP5	M	W
l.d f10			F	D	E	M	W			
l.d f12				F	D	E	M	W		
l.d f14					F	D	E	M	W	

- A structural hazard occurs when a resource is needed by 2+ instructions in the same cycle

- In cycle 9, two floating-point instructions are in write back
  - This is a structural hazard on the write port of the register file

Conclusion

And future connections

Pipeline hazards

- RAW data hazards
  - Problem: B depends on the result of A, but A has not finished yet
  - Solutions: stall, forwarding/bypassing
- Control hazards
  - Problem: with branches, there is a delay before we know what the correct next instruction address is
  - Solutions: stall, predict not taken, dynamic branch prediction (a future lecture)
- Structural hazards
  - Problem: instruction scheduling may cause contention on a hardware resource
  - Solution: stall

Cross-cutting issues

- Out-of-order completion of instructions
  - Good idea?
  - Can we do better (in hardware)?

- How do we access memory in a single cycle?
  - Next week!