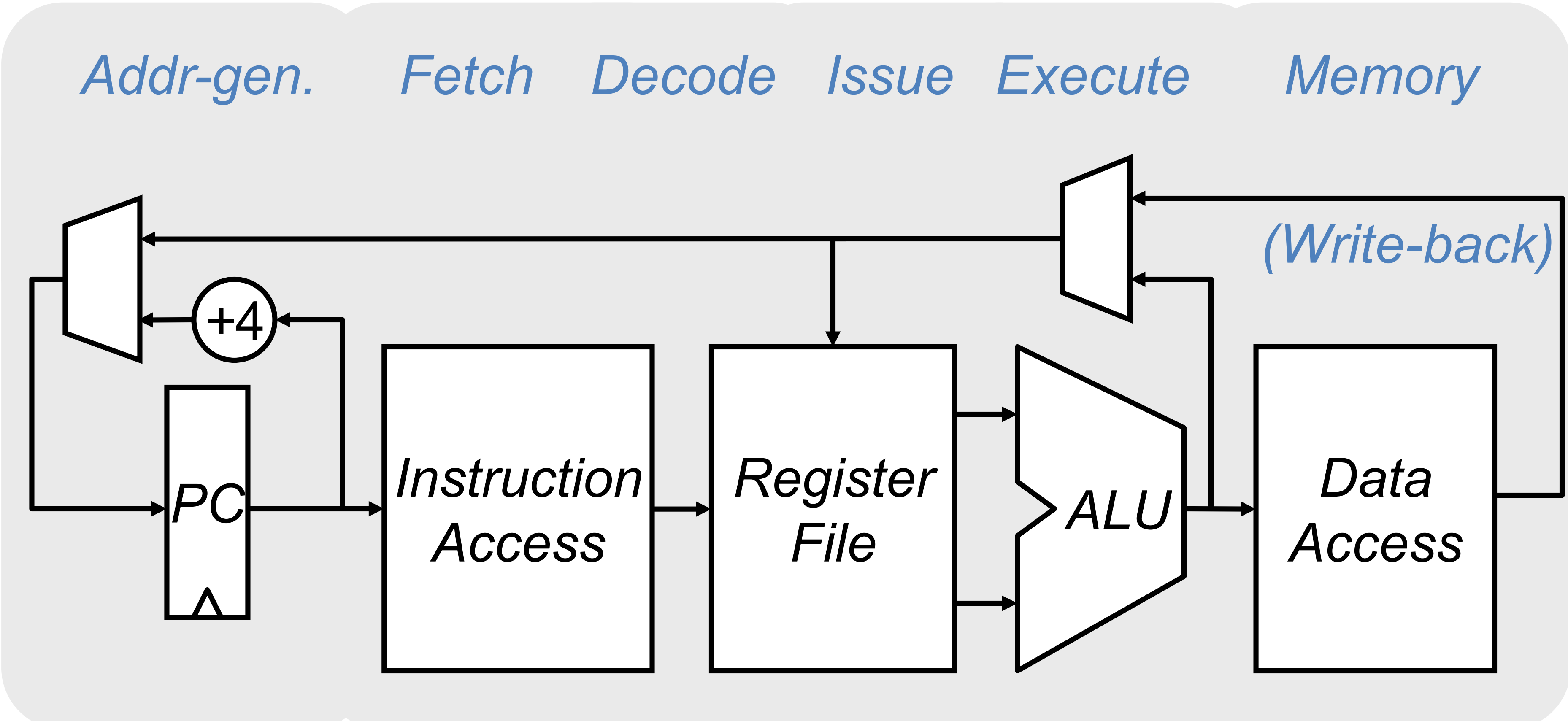
CSE 502: Computer Architecture

Core Pipelining

Generic Instruction Cycle

- Steps in processing an instruction:
  - Instruction Fetch (**IF_STEP**)
  - Instruction Decode (**ID_STEP**)
  - Operand Fetch (**OF_STEP**)
  - Execute (**EX_STEP**)
  - Result Store or Write Back (**RS_STEP**)
- Actions per instruction at each stage given by ISA
- μArch determines how HW implements the steps

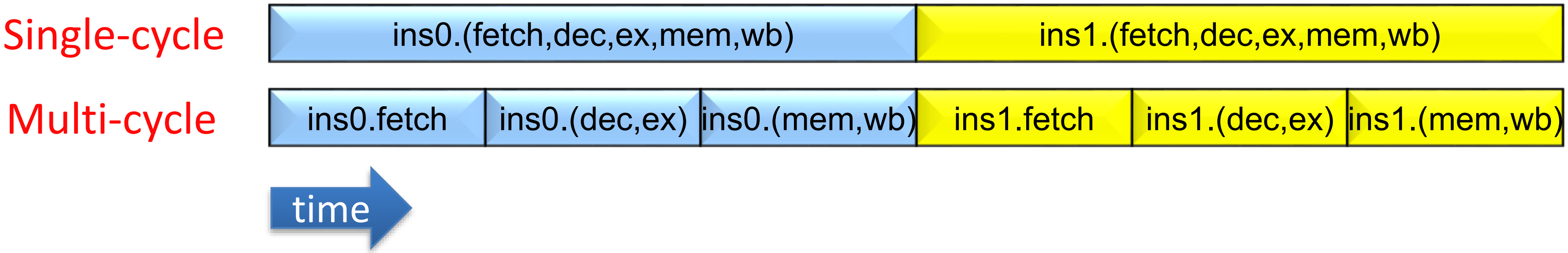
Prototypical Processor Organization



Datapath vs. Control Logic

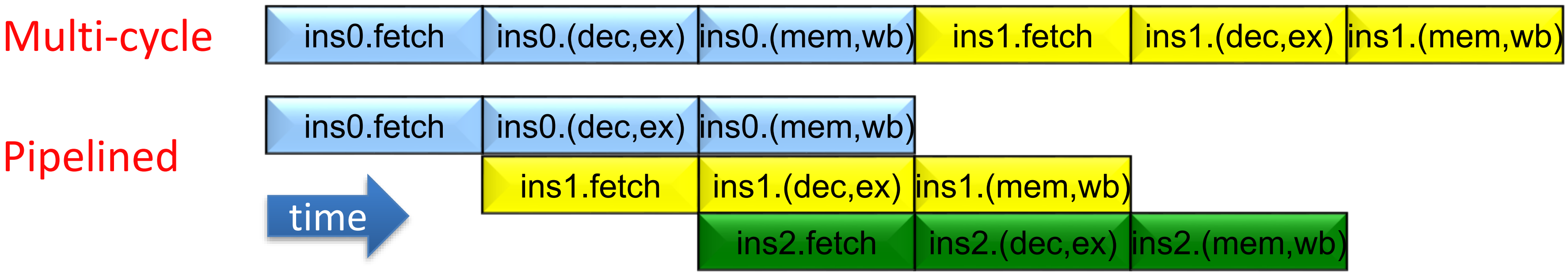
- ***Datapath*** is HW components and connections
  - Determines the *static* structure of processor
- ***Control logic*** controls data flow in datapath
  - Control is determined by
    - Instruction words
    - State of the processor
    - Execution results at each stage

Single-Instruction Datapath



- Process one instruction at a time
- *Single-cycle* control: hardwired
  - Low CPI (1)
  - Long clock period (to accommodate slowest instruction)
- *Multi-cycle* control: typically micro-programmed
  - Short clock period
  - High CPI
- Can we have both low CPI and short clock period?
  - Not if datapath executes only one instruction at a time
  - No good way to make a single instruction go faster

Pipelined Datapath

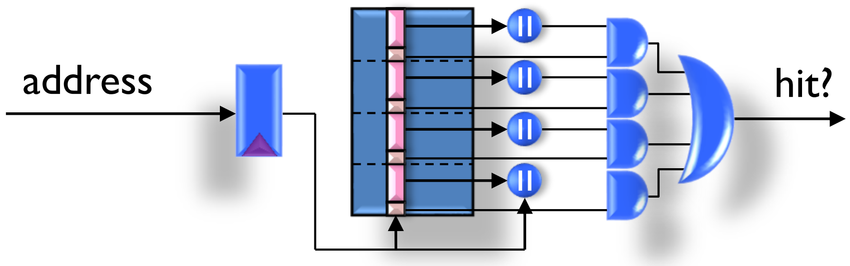


- Start with multi-cycle design
- When insn0 goes from stage 1 to stage 2
  … insn1 starts stage 1
- Each instruction passes through all stages
  … but instructions enter and leave at faster ***rate***

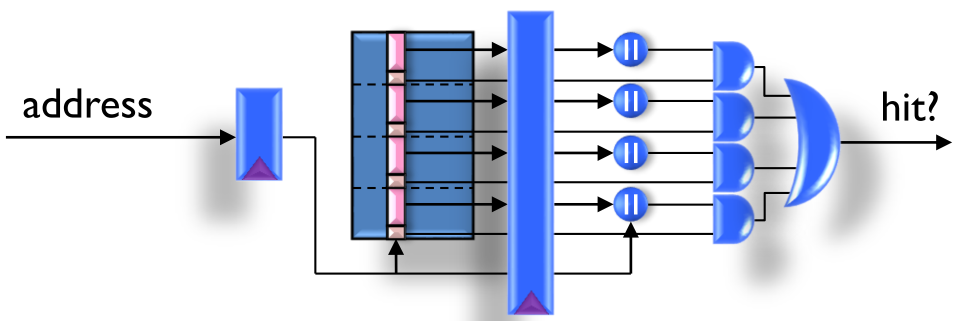
Style	Ideal CPI	Cycle Time (1/freq)
Single-cycle	1	Long
Multi-cycle	> 1	Short
Pipelined	**1**	**Short**

Pipeline can have as many insns ***in flight*** as there are stages

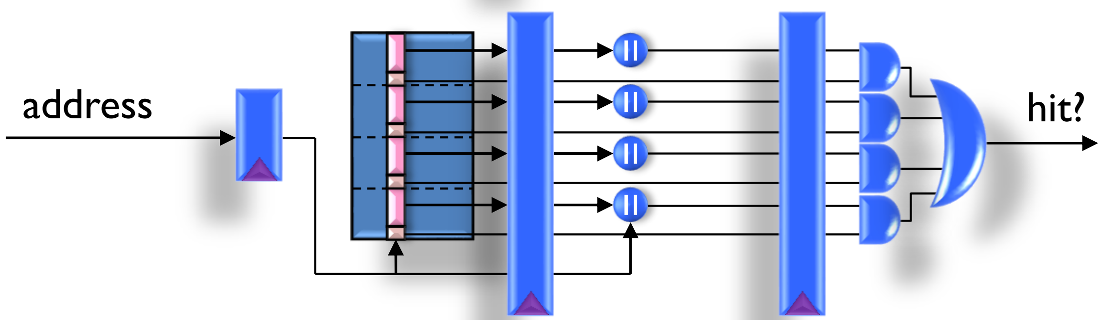
Pipeline Examples



Stage delay = n
Bandwidth = $\sim(^1/_n)$



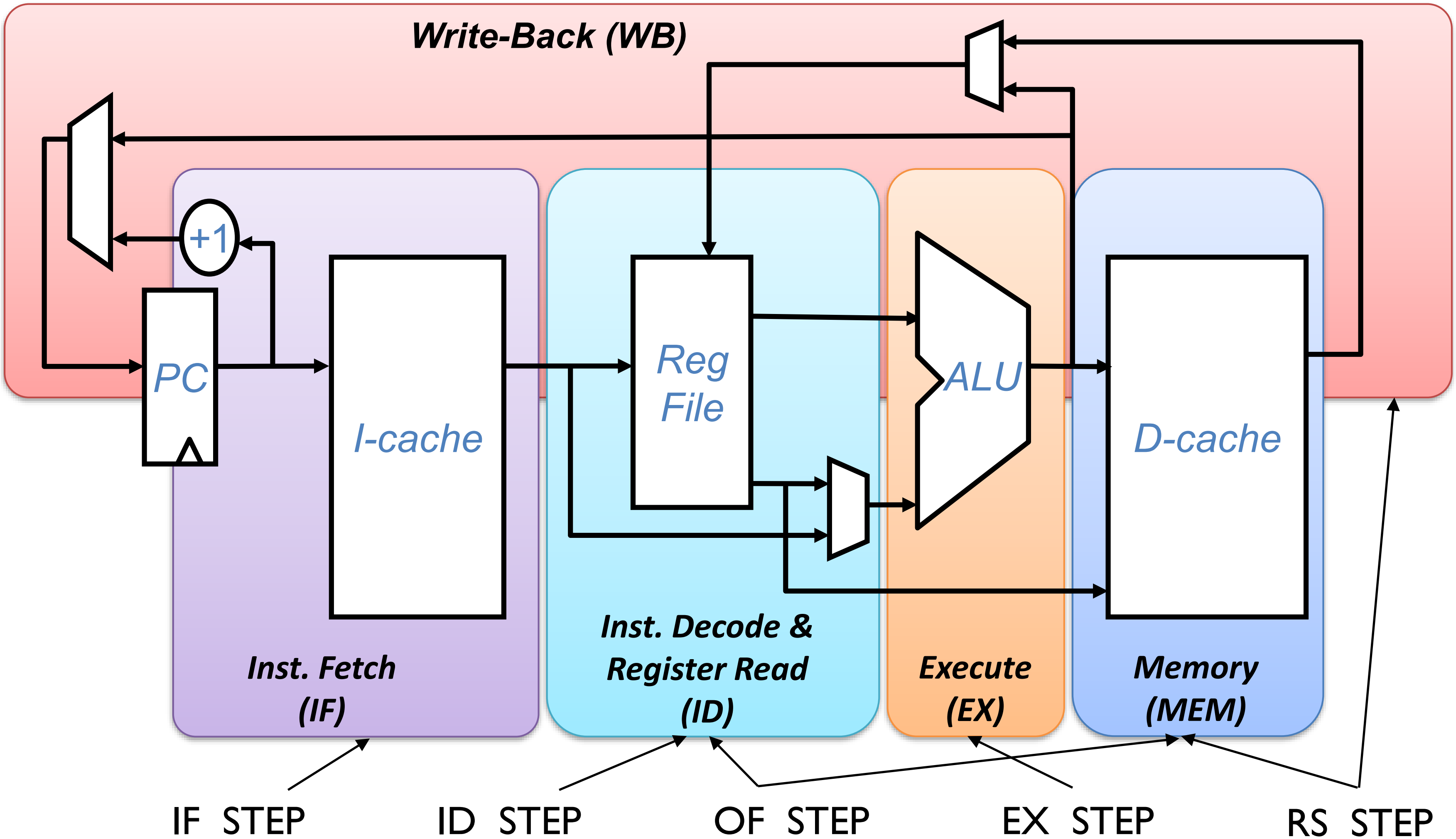
Stage delay = $^n/_2$
Bandwidth = $\sim(^2/_n)$



Stage delay = $^n/_3$
Bandwidth = $\sim(^3/_n)$

Increases throughput at the expense of latency

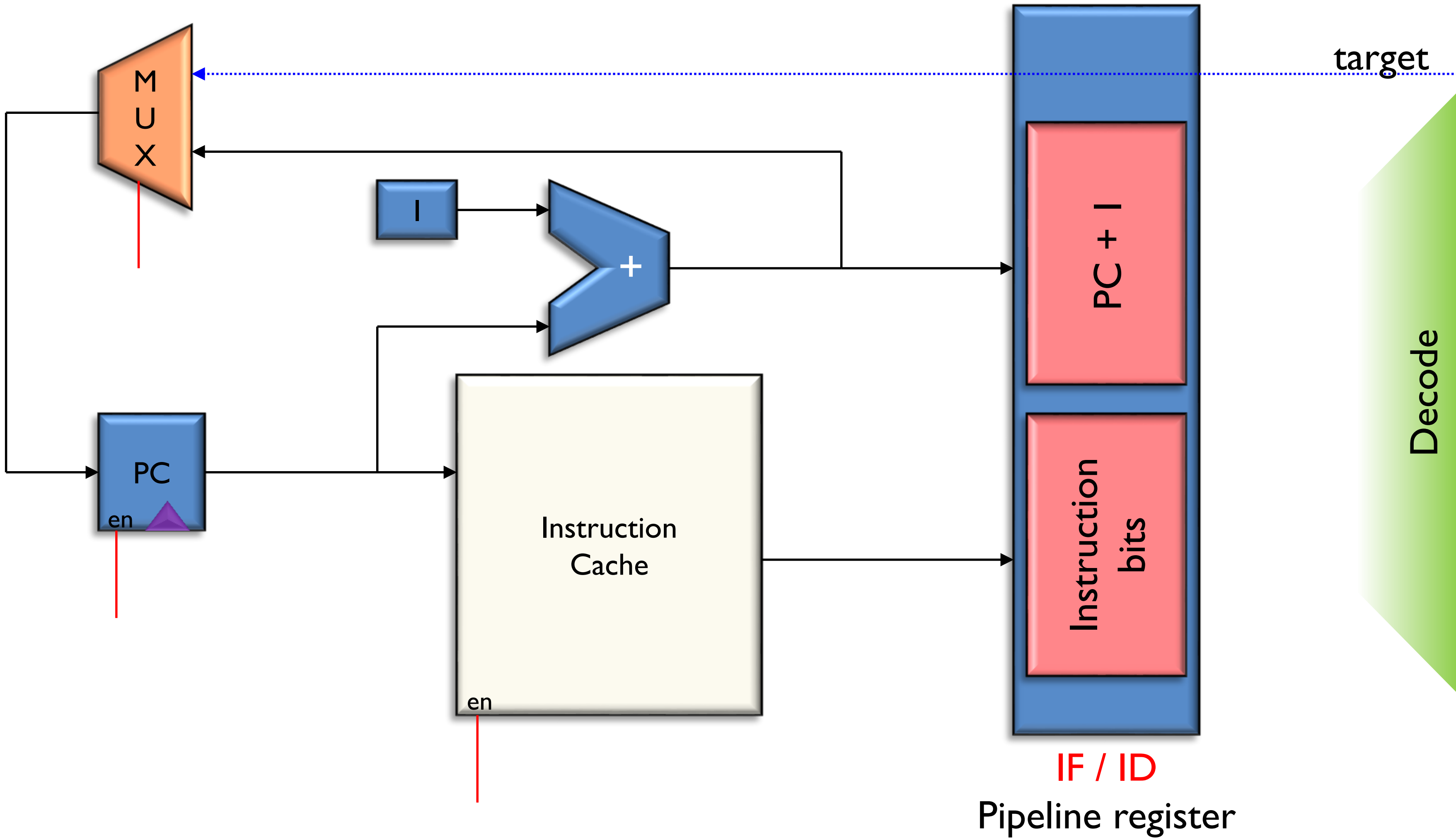
5-Stage MIPS Datapath



Stage 1: Fetch

- Fetch instruction from instruction cache
  - Use PC to index instruction cache
  - Increment PC (assume no branches for now)
- Write state to the pipeline register (IF/ID)
  - The next stage will read this pipeline register

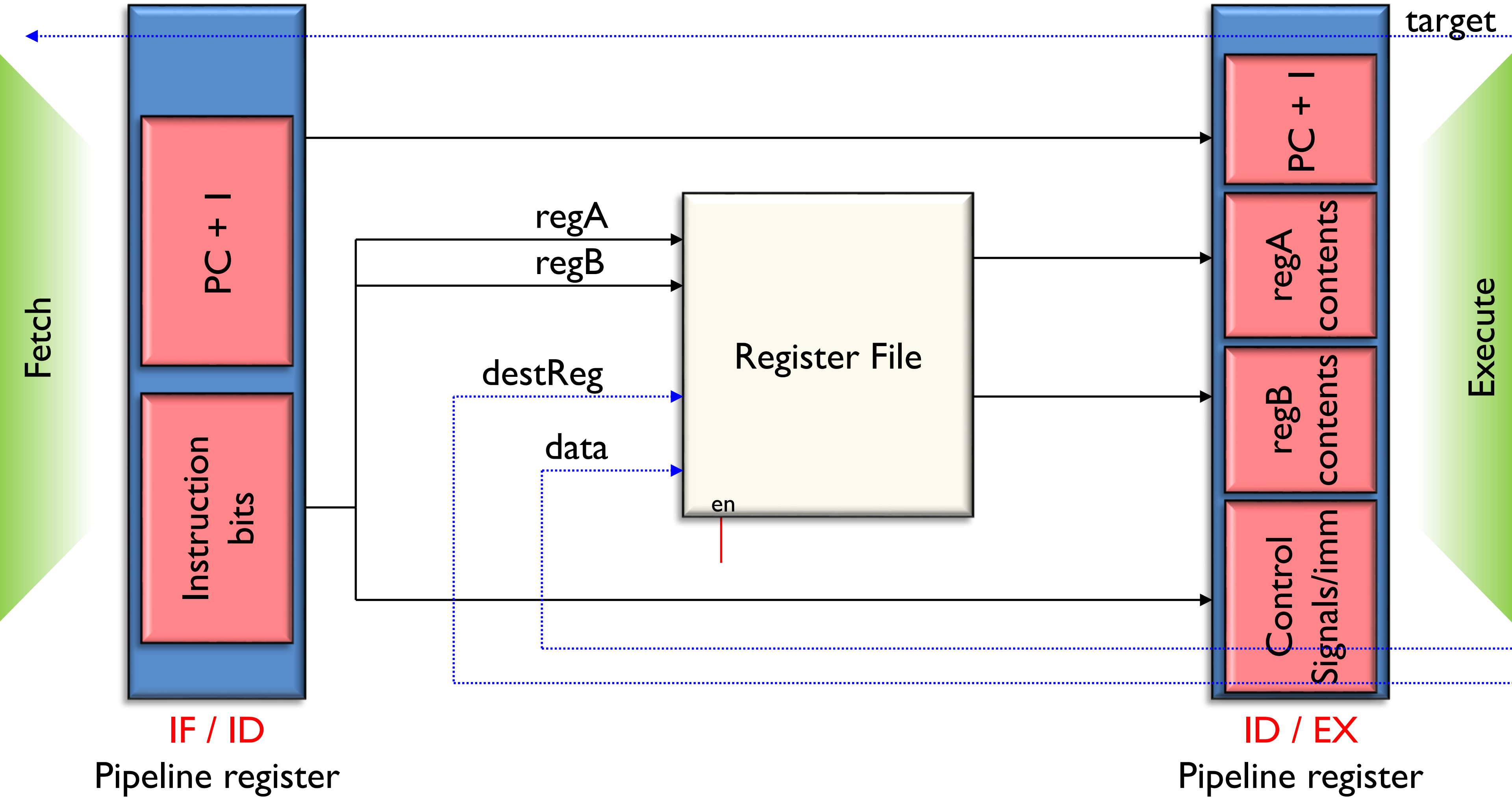
Stage 1: Fetch Diagram



Stage 2: Decode

- Decodes opcode bits
  - Set up Control signals for later stages
- Read input operands from register file
  - Specified by decoded instruction bits
- Write state to the pipeline register (ID/EX)
  - Opcode
  - Register contents, immediate operand
  - PC+1 (even though decode didn’t use it)
  - Control signals (from insn) for opcode and destReg

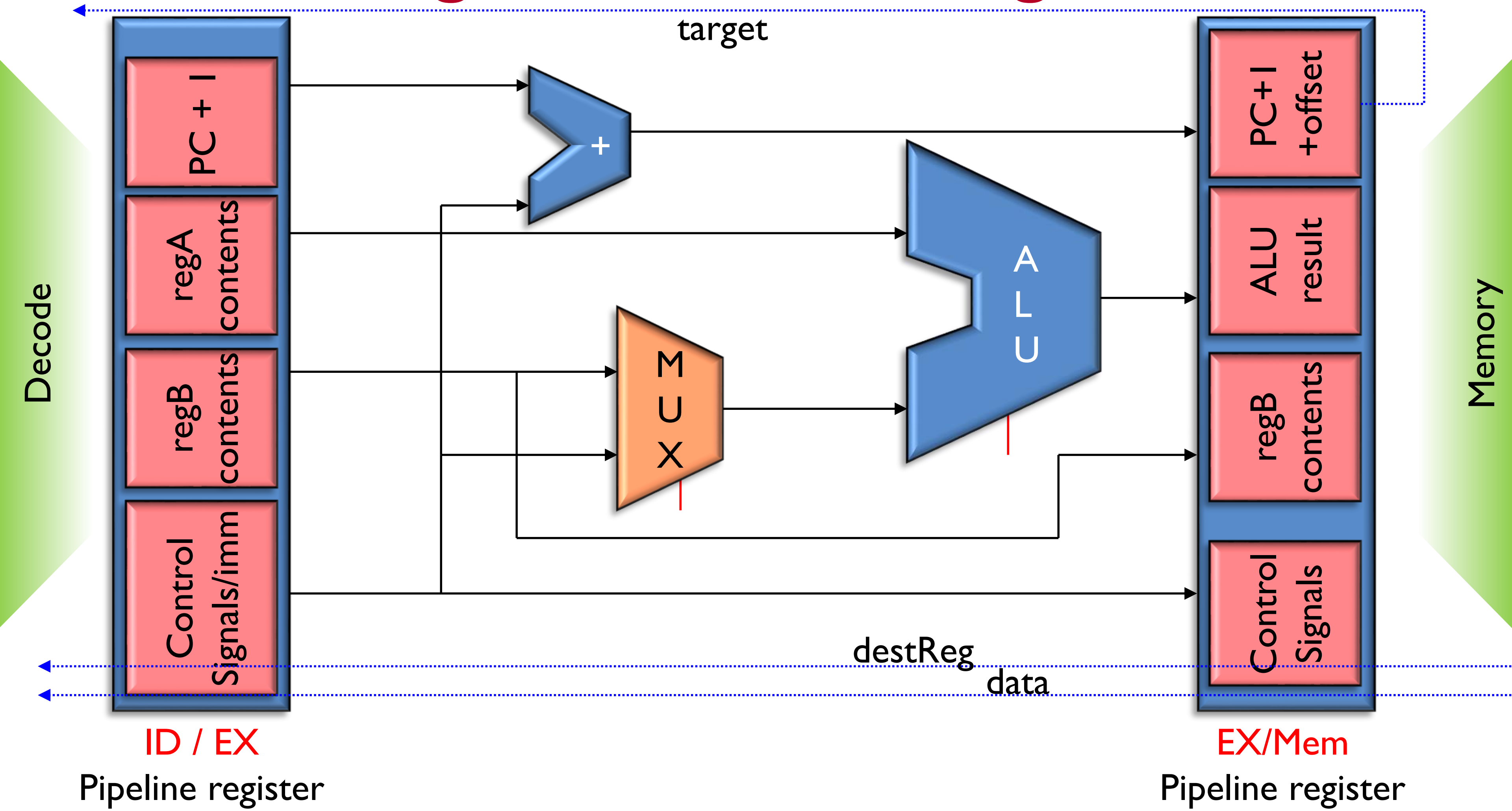
Stage 2: Decode Diagram



Stage 3: Execute

- Perform ALU operations
  - Calculate result of instruction
    - Control signals select operation
    - Contents of regA used as one input
    - Either regB or constant offset (imm from insn) used as second input
  - Calculate PC-relative branch target
    - PC+1+(constant offset)
- Write state to the pipeline register (EX/Mem)
  - ALU result, contents of regB, and PC+1+offset
  - Control signals (from insn) for opcode and destReg

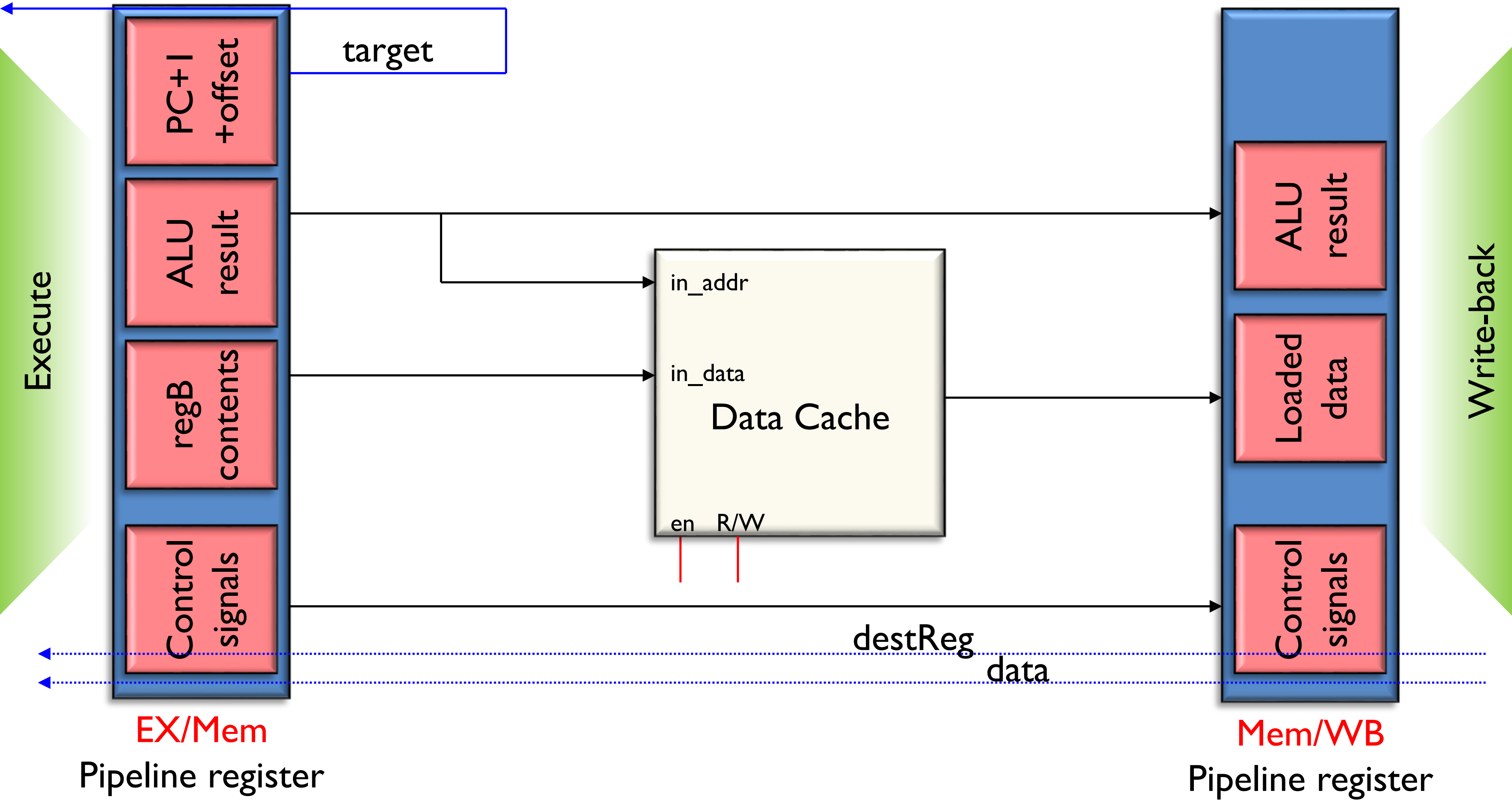
Stage 3: Execute Diagram



Stage 4: Memory

- Perform data cache access
  - ALU result contains address for LD or ST
  - Opcode bits control R/W and enable signals
- Write state to the pipeline register (Mem/WB)
  - ALU result and Loaded data
  - Control signals (from insn) for opcode and destReg

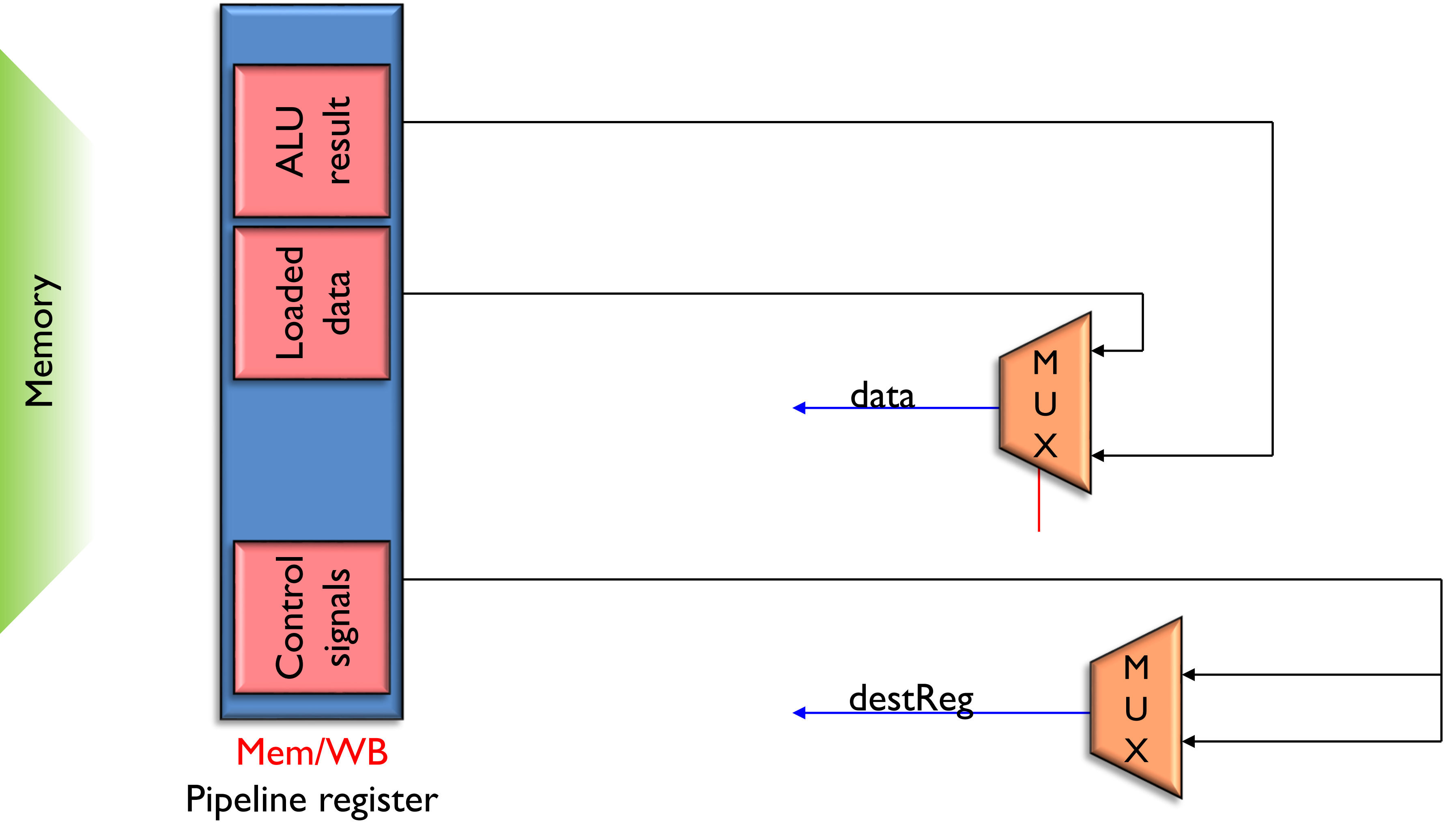
Stage 4: Memory Diagram



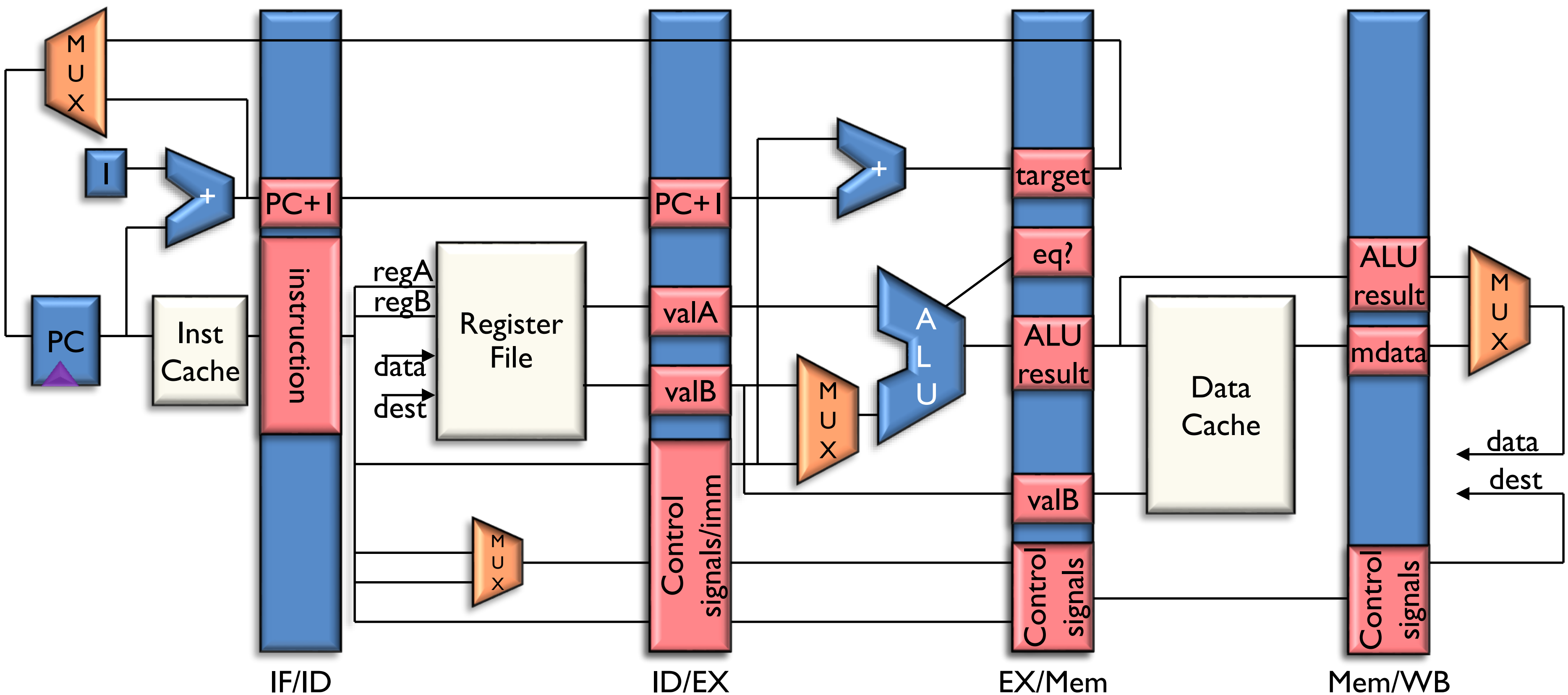
Stage 5: Write-back

- Writing result to register file (if required)
  - Write Loaded data to destReg for LD
  - Write ALU result to destReg for ALU insn
  - Opcode bits control register write enable signal

Stage 5: Write-back Diagram



Putting It All Together



Pipelining Idealism

- Uniform Sub-operations
  - Operation can partitioned into uniform-latency sub-ops
- Repetition of Identical Operations
  - Same ops performed on many different inputs
- Independent Operations
  - All ops are mutually independent

Pipeline Realism

- Uniform Sub-operations … NOT!
  - Balance pipeline stages
    - Stage quantization to yield balanced stages
    - Minimize internal fragmentation (left-over time near end of cycle)
- Repetition of Identical Operations … NOT!
  - Unifying instruction types
    - Coalescing instruction types into one “multi-function” pipe
    - Minimize external fragmentation (idle stages to match length)
- Independent Operations … NOT!
  - Resolve data and resource hazards
    - Inter-instruction dependency detection and resolution

Pipelining is expensive

The Generic Instruction Pipeline

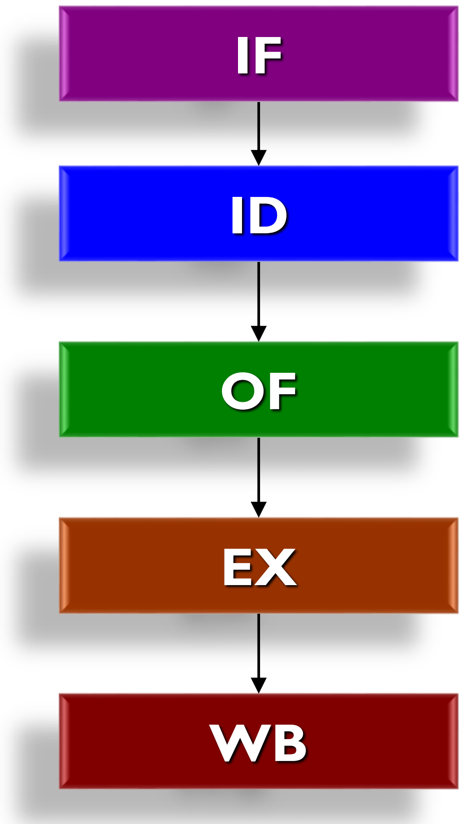
Instruction Fetch

Instruction Decode

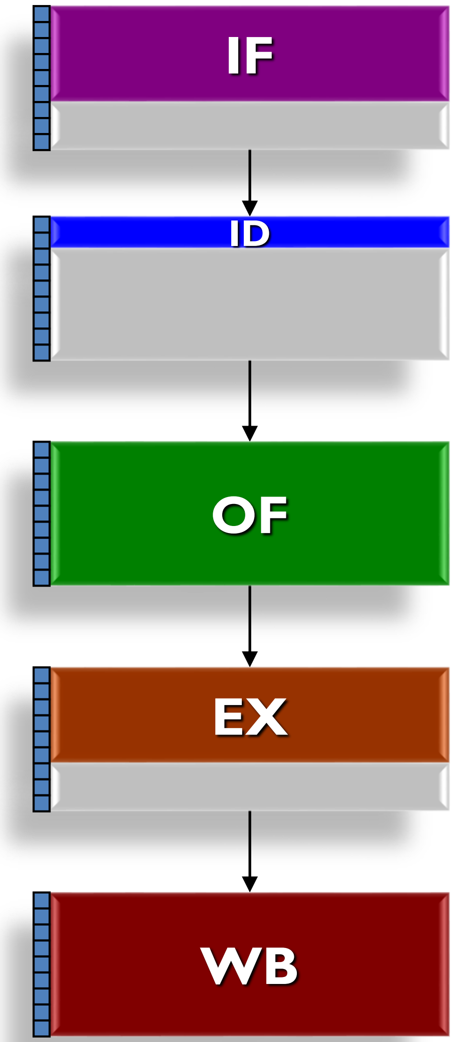
Operand Fetch

Instruction Execute

Write-back



Balancing Pipeline Stages



T_{IF}= 6 units

T_{ID}= 2 units

T_{ID}= 9 units

T_{EX}= 5 units

T_{OS}= 9 units

Without pipelining

$$T_{cyc} \approx T_{IF}+T_{ID}+T_{OF}+T_{EX}+T_{OS}$$
$$= 31$$

Pipelined

$$T_{cyc} \approx \max\{T_{IF}, T_{ID}, T_{OF}, T_{EX}, T_{OS}\}$$
$$= 9$$

Speedup = 31 / 9 = 3.44

Can we do better?

Balancing Pipeline Stages (1/2)

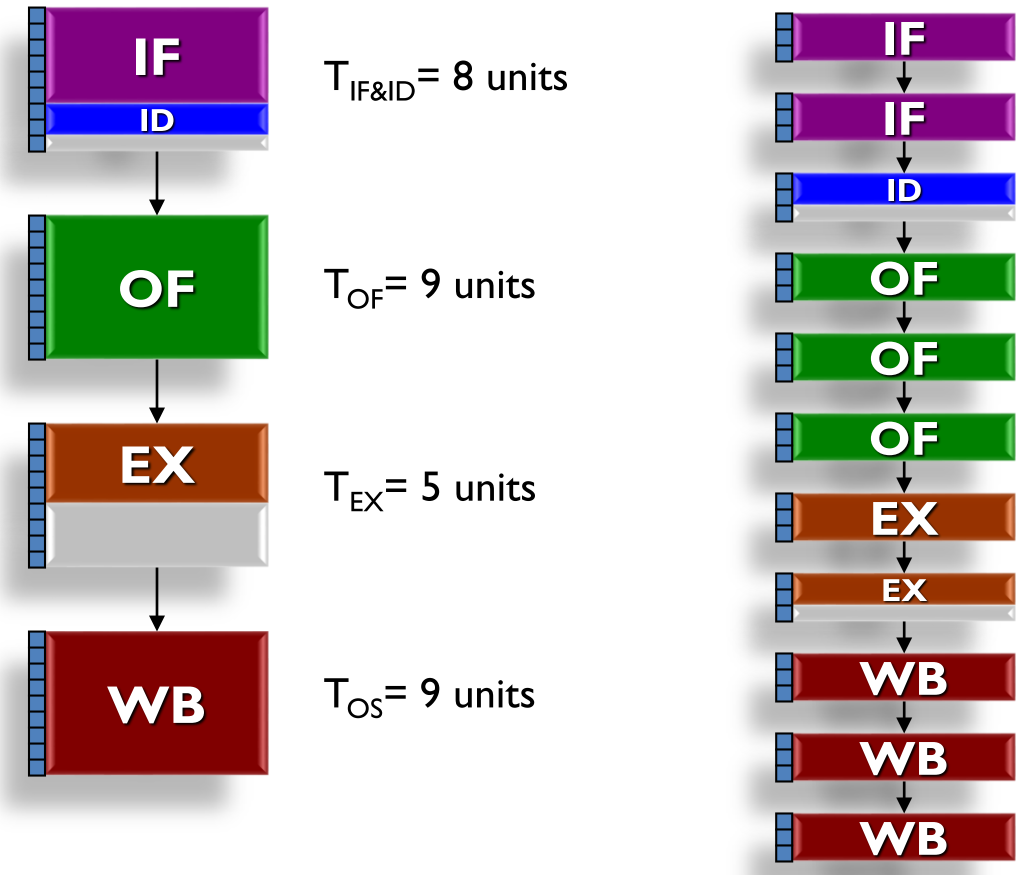
- Two methods for stage quantization
  - Divide sub-ops into smaller pieces
  - Merge multiple sub-ops into one
- Recent/Current trends
  - Deeper pipelines (more and more stages)
  - Pipelining of memory accesses
  - Multiple different pipelines/sub-pipelines

Balancing Pipeline Stages (2/2)

Coarser-Grained Machine Cycle:
4 machine cyc / instruction

Finer-Grained Machine Cycle:
11 machine cyc /instruction

stages = 4
T_{cyc}= 9 units



$T_{IF\&ID}$= 8 units

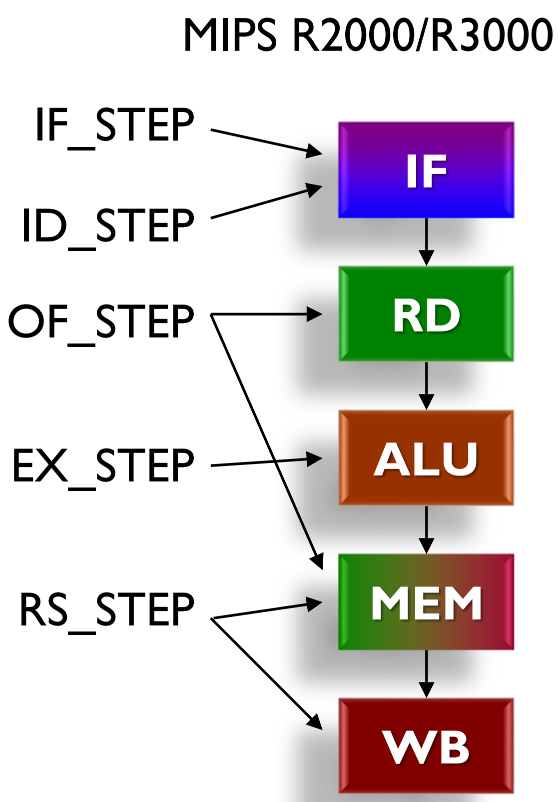
T_{OF}= 9 units

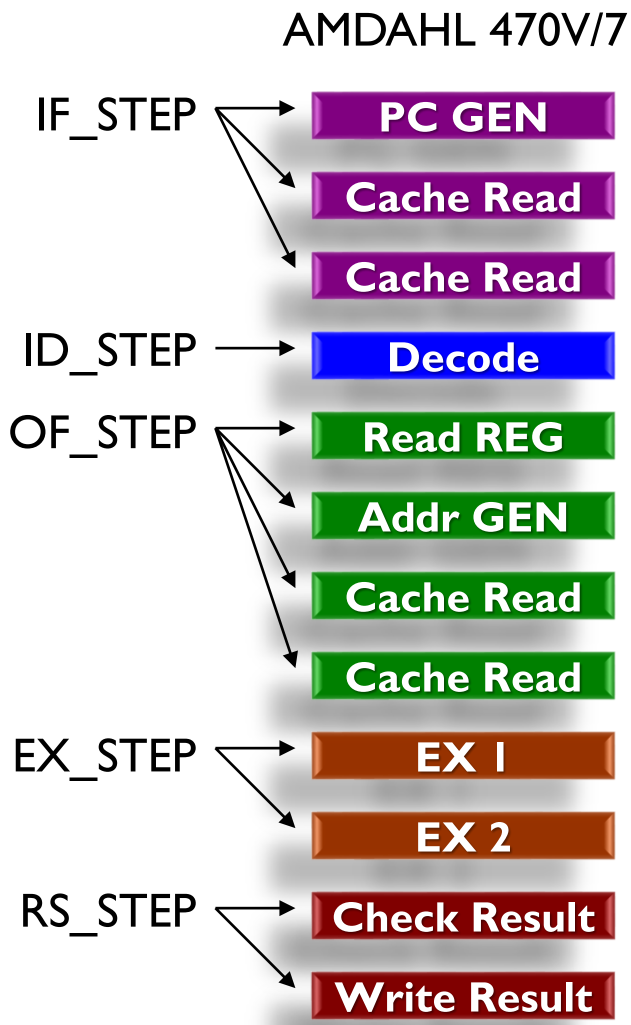
T_{EX}= 5 units

T_{OS}= 9 units

stages = 11
T_{cyc}= 3 units

Pipeline Examples





Instruction Dependencies (1/2)

- Data Dependence
  - *Read-After-Write* (*RAW*) (the only true dependence)
    - Read must wait until earlier write finishes
  - *Anti-Dependence* (*WAR*)
    - Write must wait until earlier read finishes (avoid clobbering)
  - *Output Dependence* (*WAW*)
    - Earlier write can't overwrite later write
- Control Dependence (a.k.a. Procedural Dependence)
  - Branch condition must execute before branch target
  - Instructions after branch cannot run before branch

Instruction Dependencies (1/2)

From Quicksort:

```
#   for ( ; (j < high) && (array[j] < array[low]); ++j);

        bge     j,    high,  L2
        mul     $15,  j,     4
        addu    $24,  array, $15
        lw      $25,  0($24)
        mul     $13,  low,   4
        addu    $14,  array, $13
        lw      $15,  0($14)
        bge     $25,  $15,   L2
L1:
        addu    j,    j,     1
        . . .
L2:
        addu    $11,  $11,   -1
        . . .
```

Real code has lots of dependencies

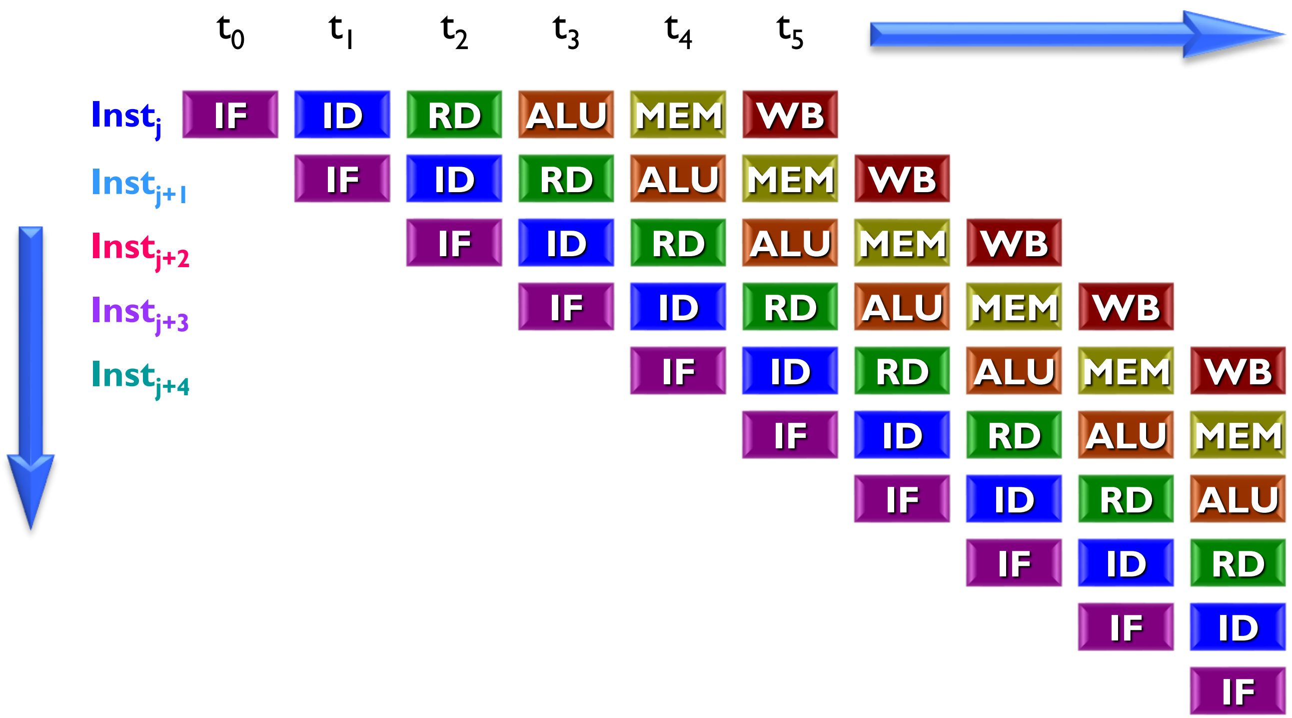
Hardware Dependency Analysis

- Processor must handle
  - Register Data Dependencies (same register)
    - RAW, WAW, WAR
  - Memory Data Dependencies (same address)
    - RAW, WAW, WAR
  - Control Dependencies

Pipeline Terminology

- ***Pipeline Hazards***
  - Potential violations of program dependencies
    - Due to multiple in-flight instructions
  - Must ensure program dependencies are not violated
- ***Hazard Resolution***
  - Static method: compiler guarantees correctness
    - By inserting No-Ops or independent insns between dependent insns
  - Dynamic method: hardware checks at runtime
    - Two basic techniques: ***Stall*** (costs perf.), ***Forward*** (costs hw)
- ***Pipeline Interlock***
  - Hardware mechanism for dynamic hazard resolution
  - Must detect and enforce dependencies at runtime

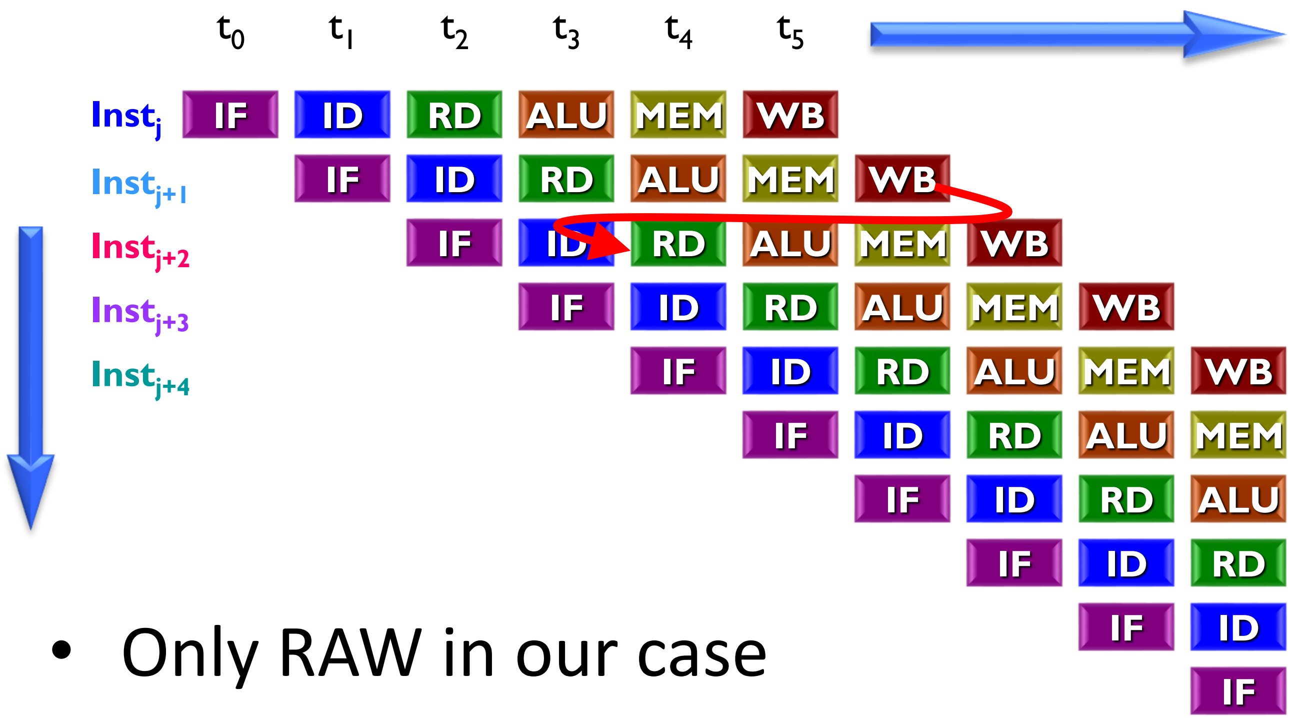
Pipeline: Steady State



Data Hazards

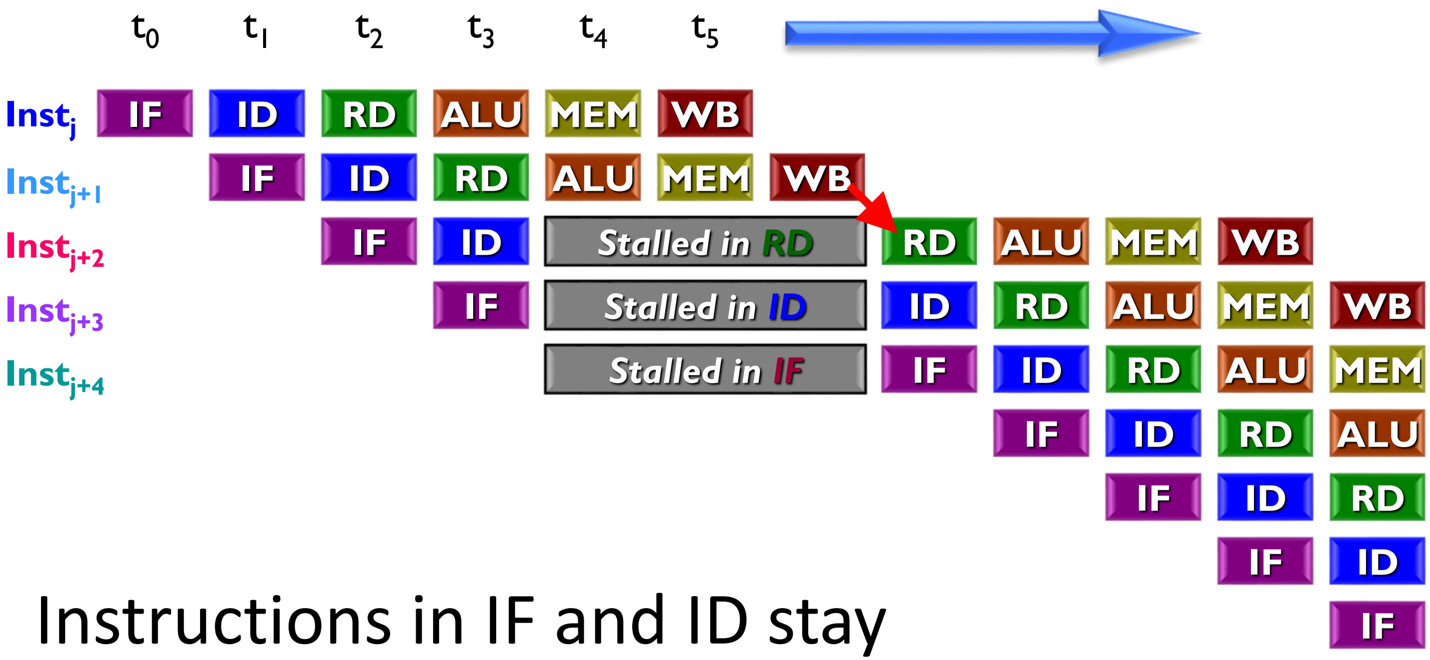
- Necessary conditions:
  - WAR: write stage earlier than read stage
    - Is this possible in IF-ID-RD-EX-MEM-WB?
  - WAW: write stage earlier than write stage
    - Is this possible in IF-ID-RD-EX-MEM-WB?
  - RAW: read stage earlier than write stage
    - Is this possible in IF-ID-RD-EX-MEM-WB?
- If conditions not met, no need to resolve
- Check for both register and memory

Pipeline: Data Hazard



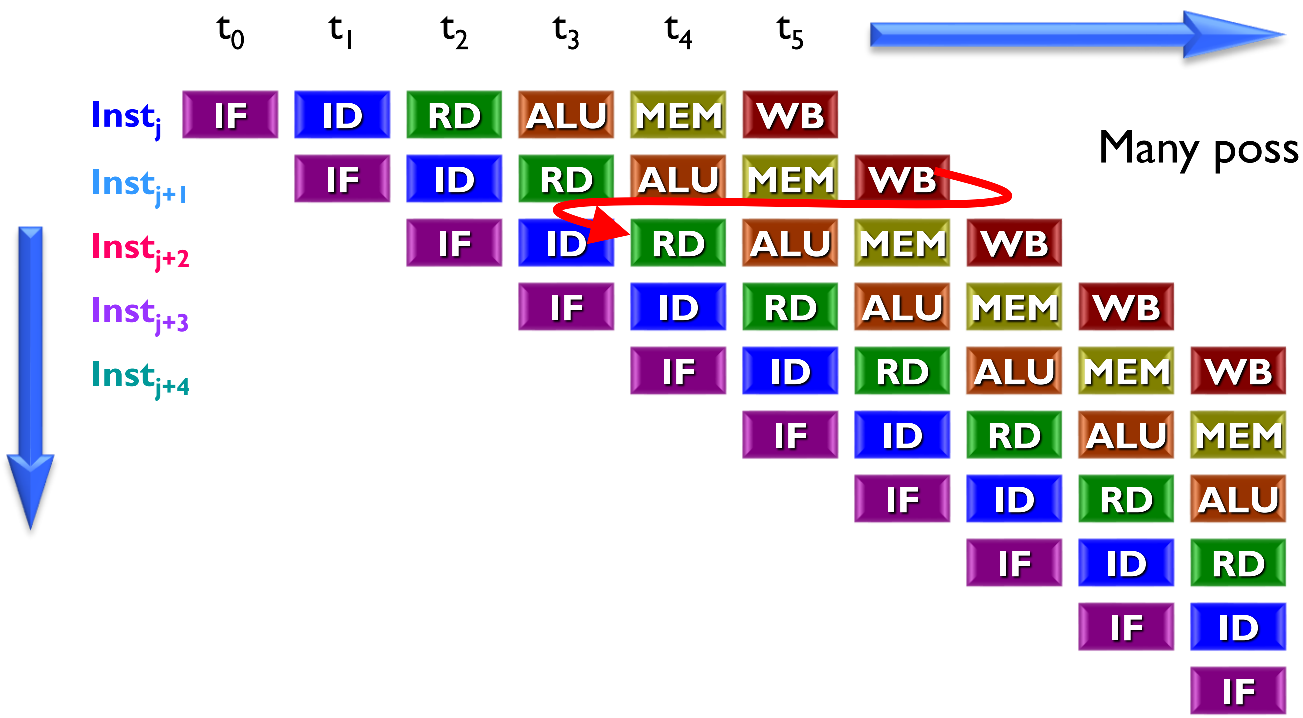
- Only RAW in our case
- How to detect?
  - Compare read register specifiers for newer instructions with write register specifiers for older instructions

Option 1: Stall on Data Hazard



- Instructions in IF and ID stay
- IF/ID pipeline latch not updated
- Send no-op down pipeline (called a bubble)

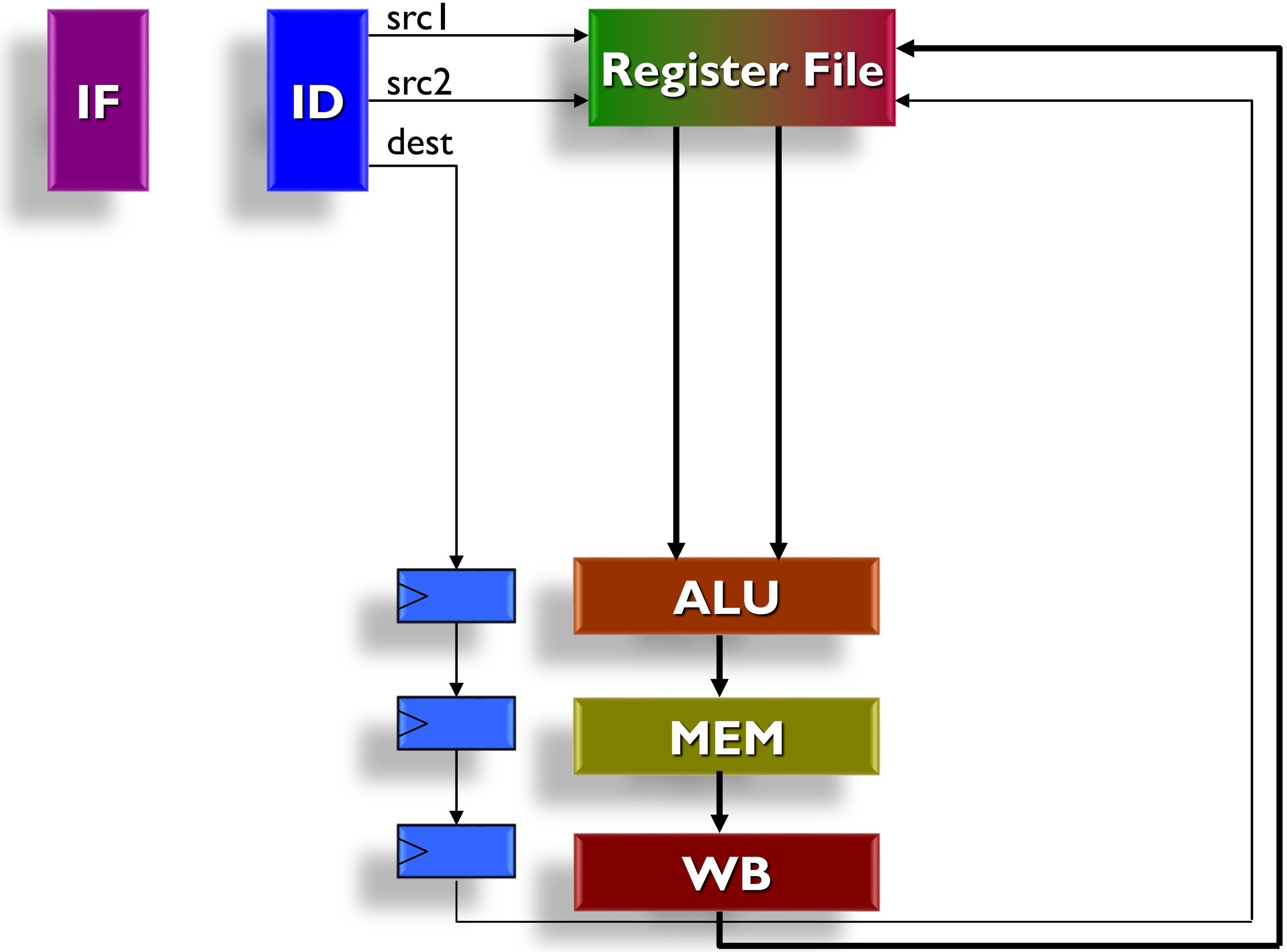
Option 2: Forwarding Paths (1/3)



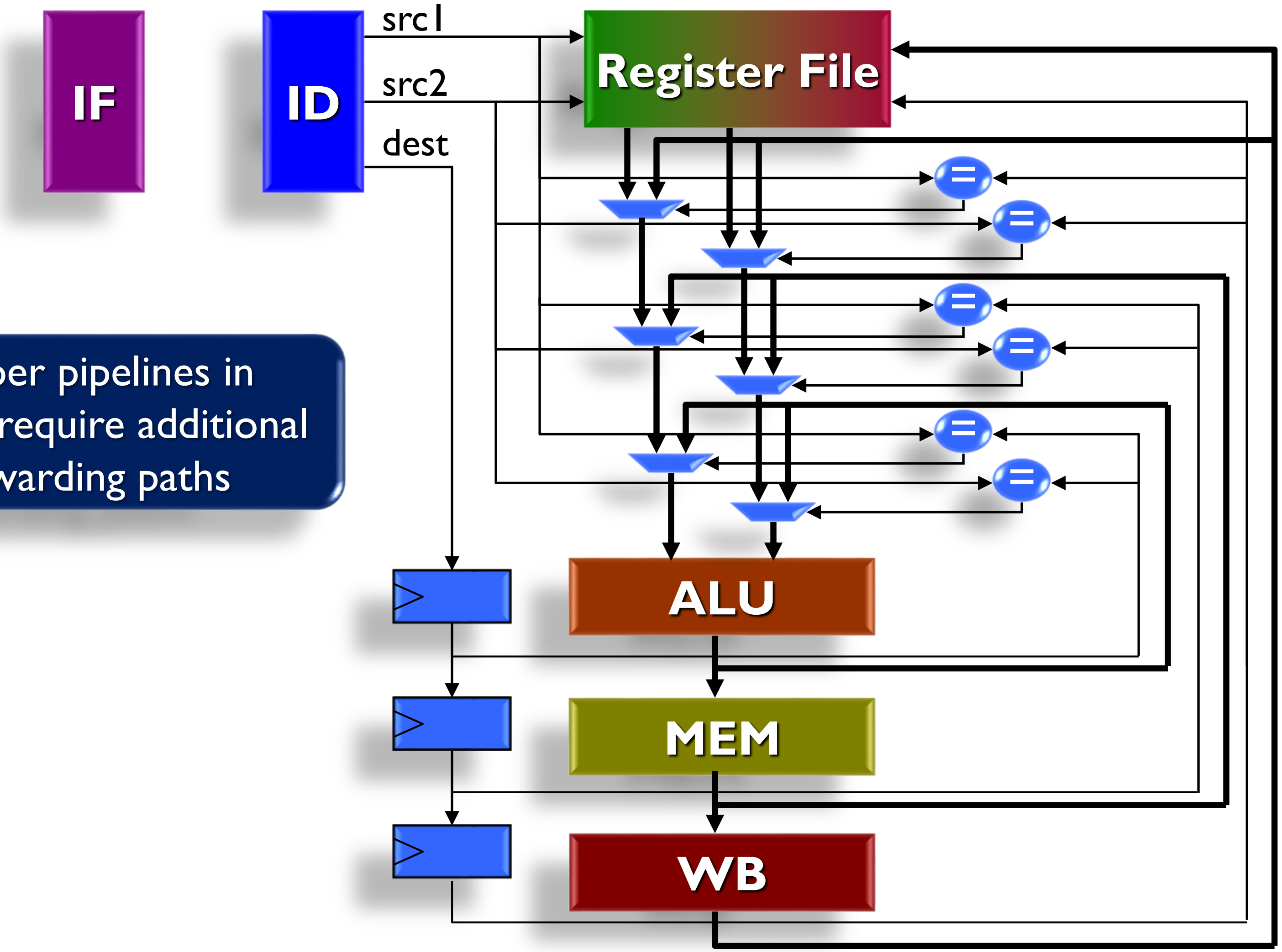
Many possible paths

Requires stalling even with forwarding paths

Option 2: Forwarding Paths (2/3)

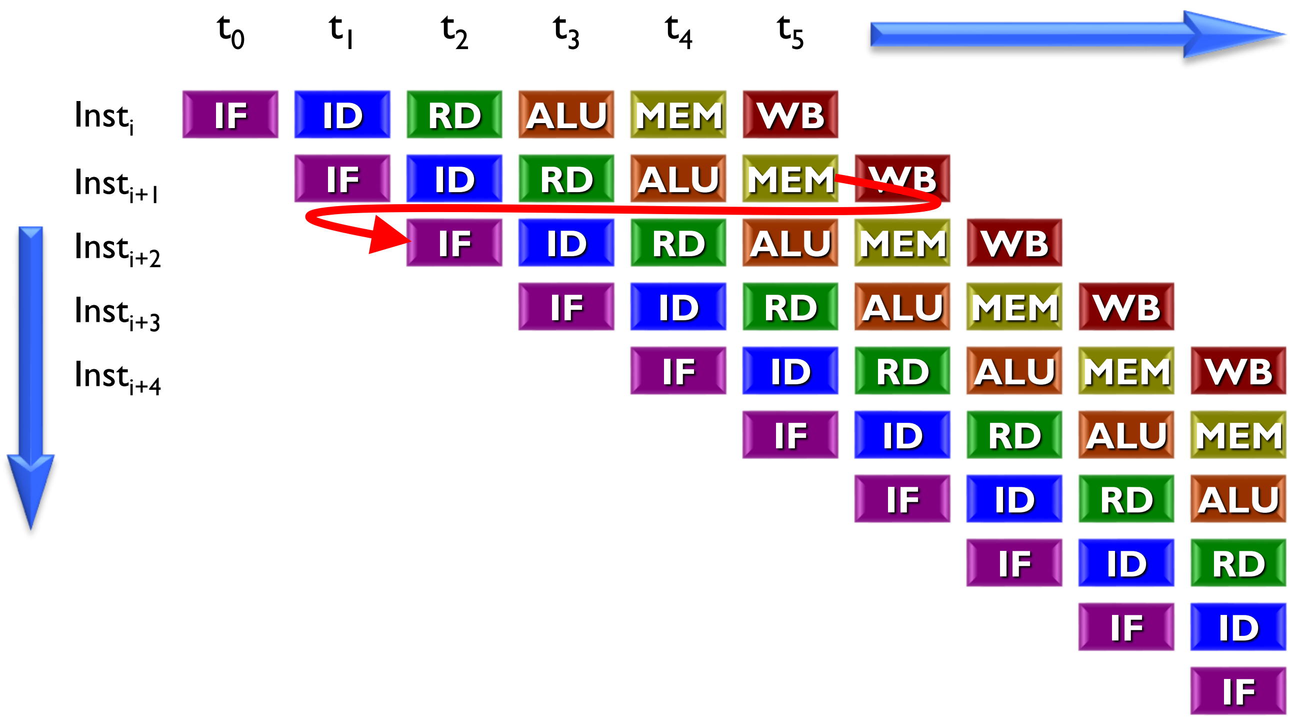


Option 2: Forwarding Paths (3/3)



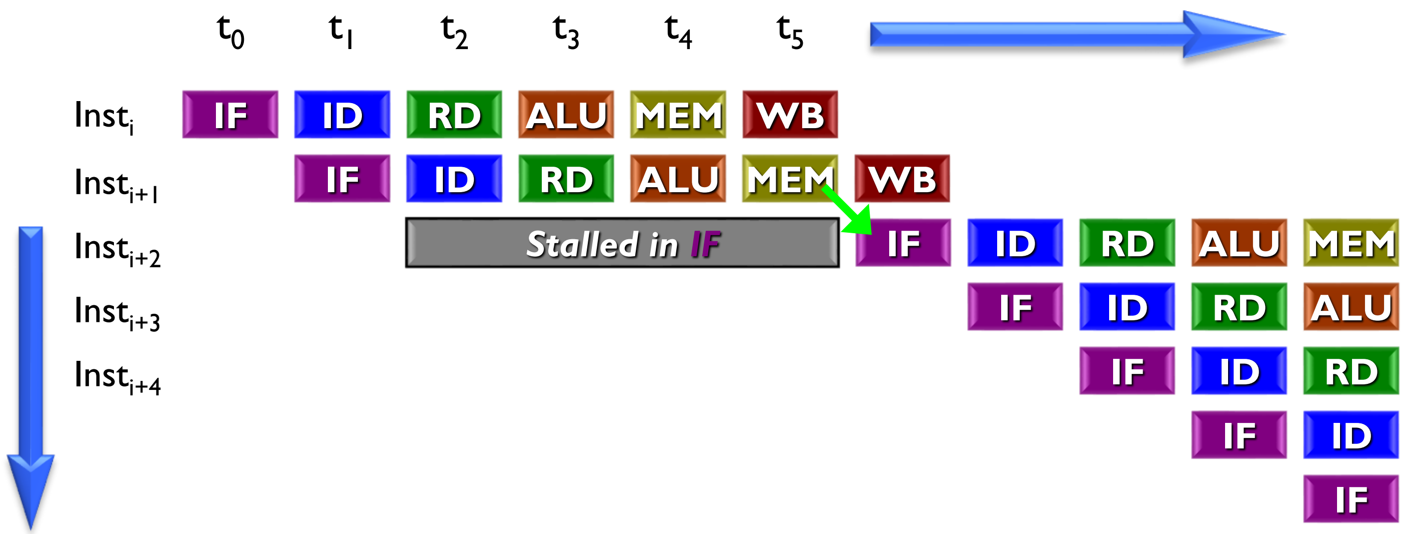
Deeper pipelines in general require additional forwarding paths

Pipeline: Control Hazard



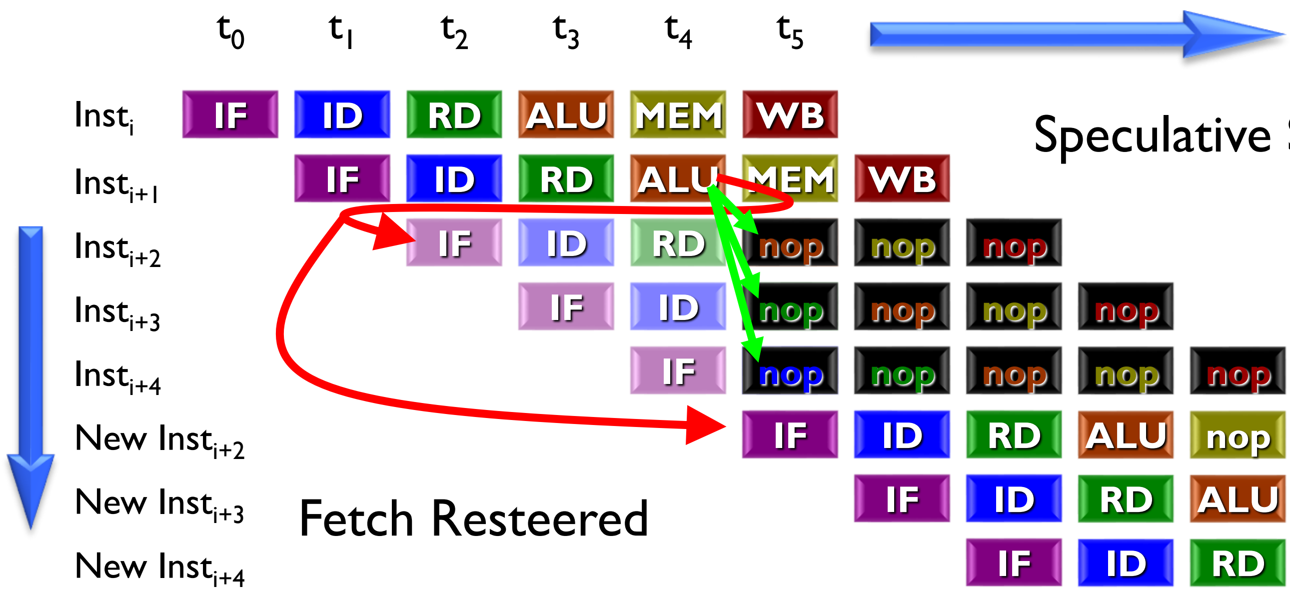
- ***Note***: The target of Inst_{i+1} is available at the end of the ALU stage, but it takes one more cycle (MEM) to be written to the PC register

Option 1: Stall on Control Hazard



- Stop fetching until branch outcome is known
  - Send no-ops down the pipe
- Easy to implement
- Performs poorly
  - ~1 of 6 instructions are branches
  - Each branch takes 4 cycles
  - CPI = 1 + 4 x 1/6 = 1.67 (lower bound)

Option 2: Prediction for Control Hazards



- Predict branch not taken
- Send sequential instructions down pipeline
- Must stop memory and RF writes
- Kill instructions later if incorrect; we would know at the end of ALU
- Fetch from branch target

Option 3: Delay Slots for Control Hazards

- Another option: delayed branches
  - # of delay slots (*ds*) : stages between IF and where the branch is resolved
    - 3 in our example
  - Always execute following *ds* instructions
  - Put useful instruction there, otherwise no-op

- Losing popularity
  - Just a stopgap (one cycle, one instruction)
  - Superscalar processors (later)
    - Delay slot just gets in the way (special case)

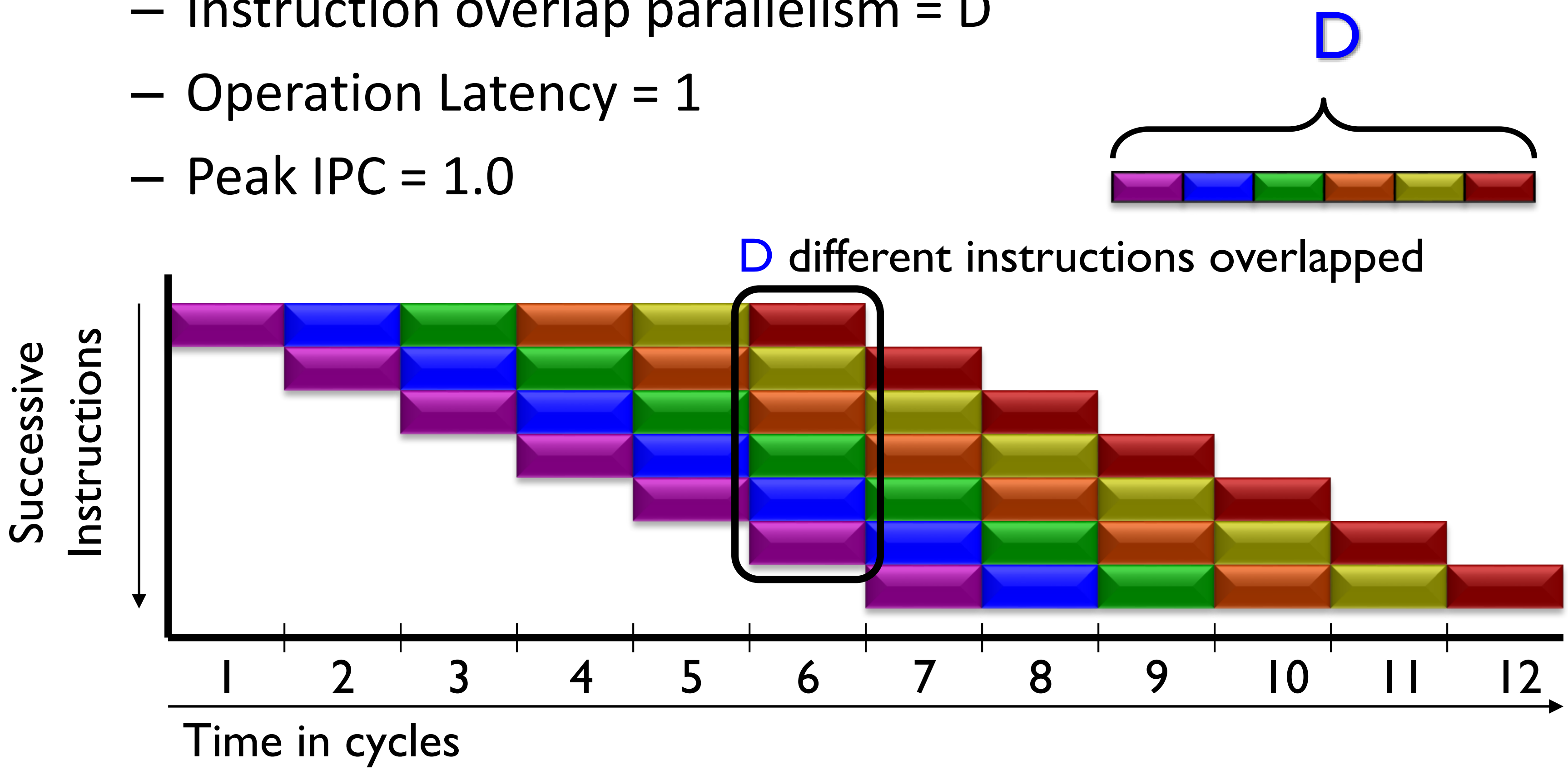
Legacy from old RISC ISAs

Going Beyond Scalar

- Scalar pipeline limited to CPI $\geq$ 1.0
  - Can never run more than 1 insn per cycle
- "Superscalar" can achieve CPI $\leq$ 1.0 (i.e., IPC $\geq$ 1.0)
  - *<u>Superscalar</u>* means executing multiple insns in parallel

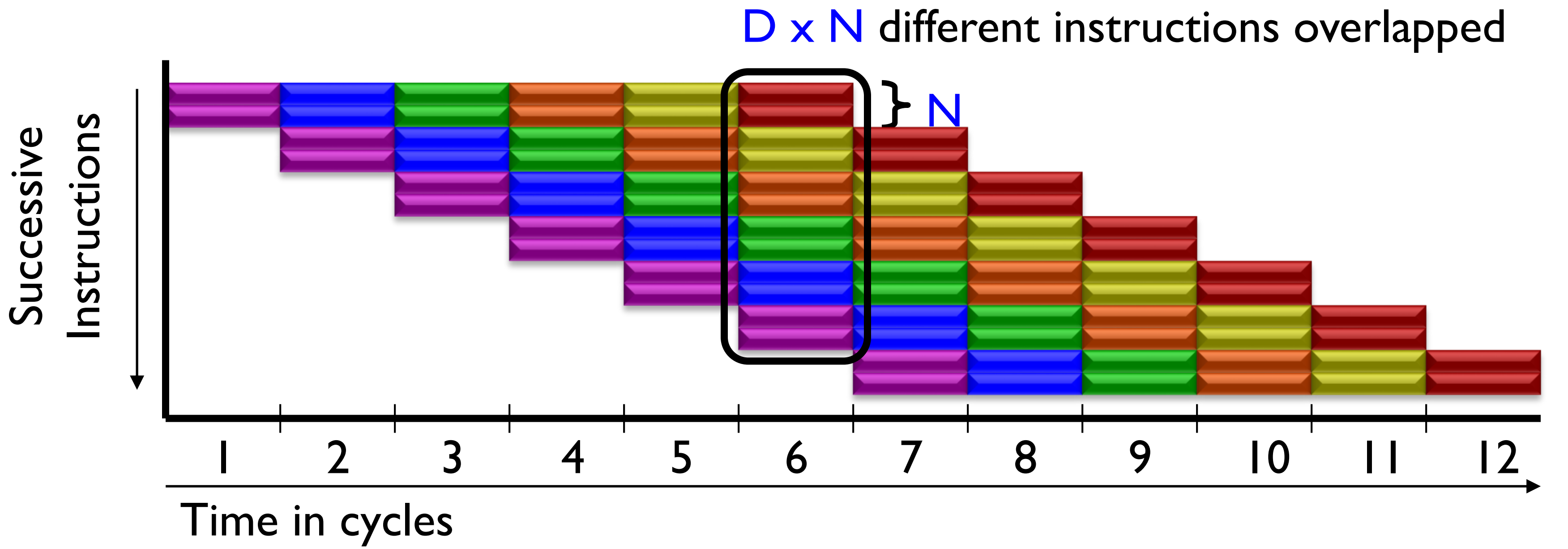
Architectures for Instruction Parallelism

- Scalar pipeline (baseline)
  - Instruction overlap parallelism = D
  - Operation Latency = 1
  - Peak IPC = 1.0

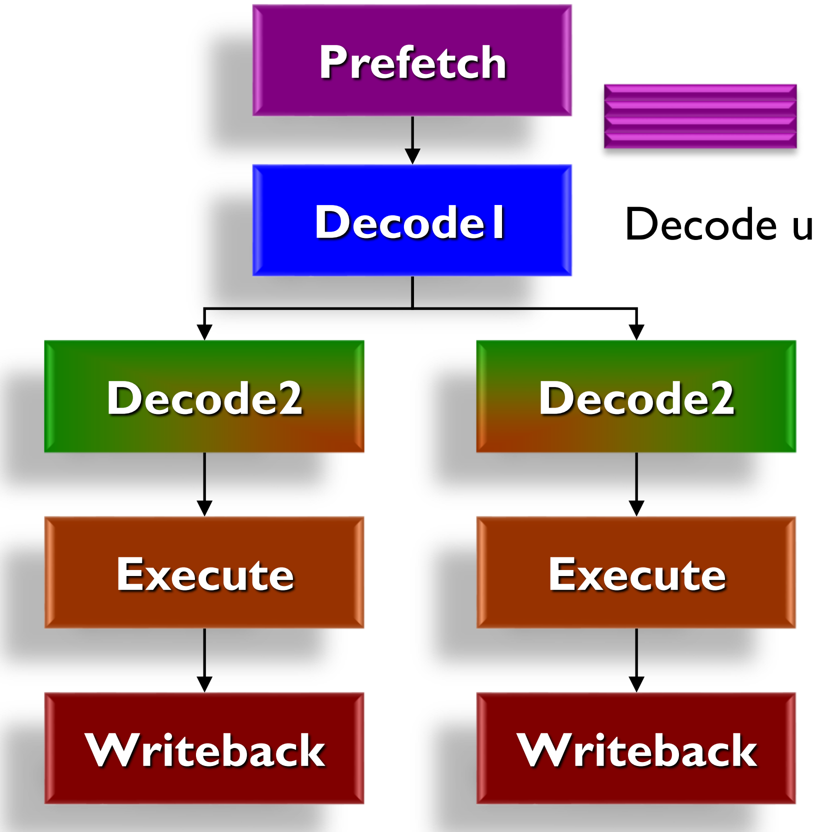


Superscalar Machine

- Superscalar (pipelined) Execution
  - Instruction parallelism = D x N
  - Operation Latency = 1
  - Peak IPC = N per cycle



Superscalar Example: Pentium



4× 32-byte buffers

Decode up to 2 insts

Read operands, Addr comp

Asymmetric pipes

both	u-pipe	v-pipe
mov, lea, simple ALU, push/pop test/cmp	shift rotate some FP	jmp, jcc, call, fxch

Pentium Hazards & Stalls

- “Pairing Rules” (when can’t two insns exec?)
  - Read/flow dependence
    - mov eax, 8
    - mov [ebp], eax
  - Output dependence
    - mov eax, 8
    - mov eax, [ebp]
  - Partial register stalls
    - mov al, 1
    - mov ah, 0
  - Function unit rules
    - Some instructions can never be paired
      - MUL, DIV, PUSHA, MOVS, some FP

Limitations of In-Order Pipelines

- If the machine parallelism is increased
  - … dependencies reduce performance
  - CPI of in-order pipelines degrades sharply
    - As N approaches avg. distance between dependent instructions
    - Forwarding is no longer effective
  - Must stall often

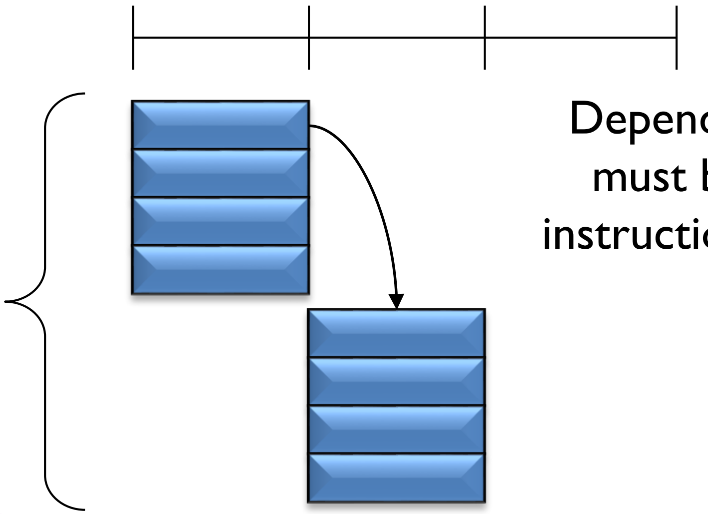
In-order pipelines are rarely full

The In-Order N-Instruction Limit

- On average, parent-child separation is about 5 insn
  - (Franklin and Sohi ’92)

Ex. Superscalar degree N = 4

Any dependency between these instructions will cause a stall



Dependent insn must be N = 4 instructions away

Average of 5 means there are many cases when the separation is < 4… each of these limits parallelism

Reasonable in-order superscalar is effectively N=2