18-747 Lecture 2: Pipelining Fundamentals

James C. Hoe

Dept of ECE, CMU

August 29, 2001

Reading Assignments: *S&L Ch 2 pp1-34*

Announcements: Office hours, MW, 4:30-5:30 PM

Textbook S&L, see Melissa HH-D204, $20 check to CMU

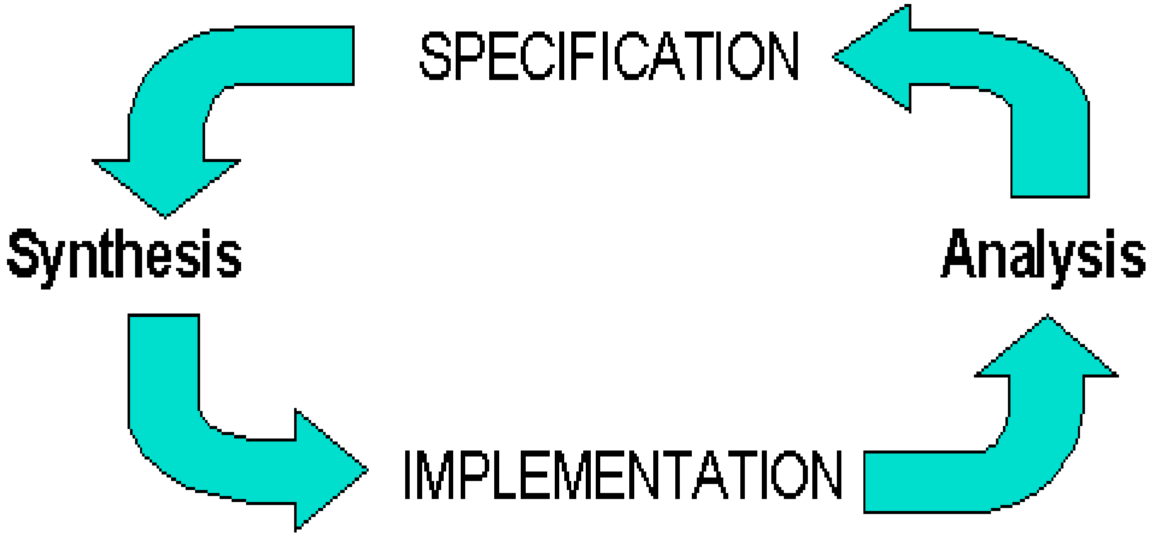
No cash, No credit cards!

Handout#0 due tomorrow noon, outside HH-D201

No recitation this week

No class on next Monday

Anatomy of Engineering Design



Specification: Behavioral description of *"What does it do?"*

Synthesis: Search for possible solutions; pick the best one.

Implementation: Structural description of *"How is it constructed?"*

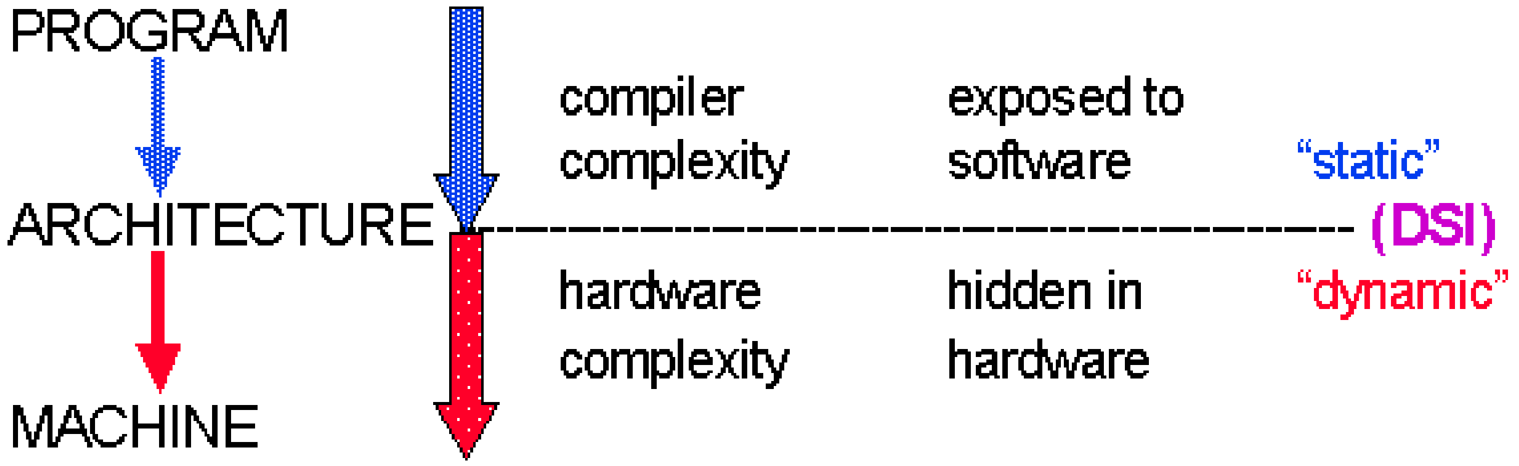
Analysis: Figure out if the design meets the specification.

"Does it do the right thing?" + *"How well does it perform?"*

Instruction Set Architecture

- ISA, the boundary between software and hardware
  - Specifies the logical machine that is visible to the programmer
  - Also, a functional spec for the processor designers
- What needs to be specified by an ISA
  - Operations
    - what to perform and what to perform next
  - Temporary Operand Storage in the CPU
    - accumulator, stacks, registers
  - Number of operands per instruction
  - Operand location
    - where and how to specify the operands
  - Type and size of operands
  - Instruction-to-Binary Encoding

Dynamic-Static Interface



DSI = ISA

= a contract between the program and the machine.

Anatomy of a Modern ISA

- Operations
  simple ALU op's, data movement, control transfer
- Temporary Operand Storage in the CPU
  Large General Purpose Register (GPR) File
- Number of operands per instruction
  triadic A $\Leftarrow$ B op C
- Operand location
  load-store architecture with register indirect addressing
- Type and size of operands
  32/64-bit integers, IEEE floats
- Instruction-to-Binary Encoding
  Fixed width, regular fields

Exceptions: Intel x86, IBM 390 (aka z900)

“Iron Law” of Processor Performance

$$\text{Processor Performance} = \frac{\text{Wall-Clock Time}}{\text{Program}}$$

$$= \boxed{\frac{\text{Instructions}}{\text{Program}}} \times \boxed{\frac{\text{Cycles}}{\text{Instruction}}} \times \boxed{\frac{\text{Time}}{\text{Cycle}}}$$

(code size) *(CPI)* *(cycle time)*

Architecture → Implementation → Realization

Compiler Designer Processor Designer Chip Designer

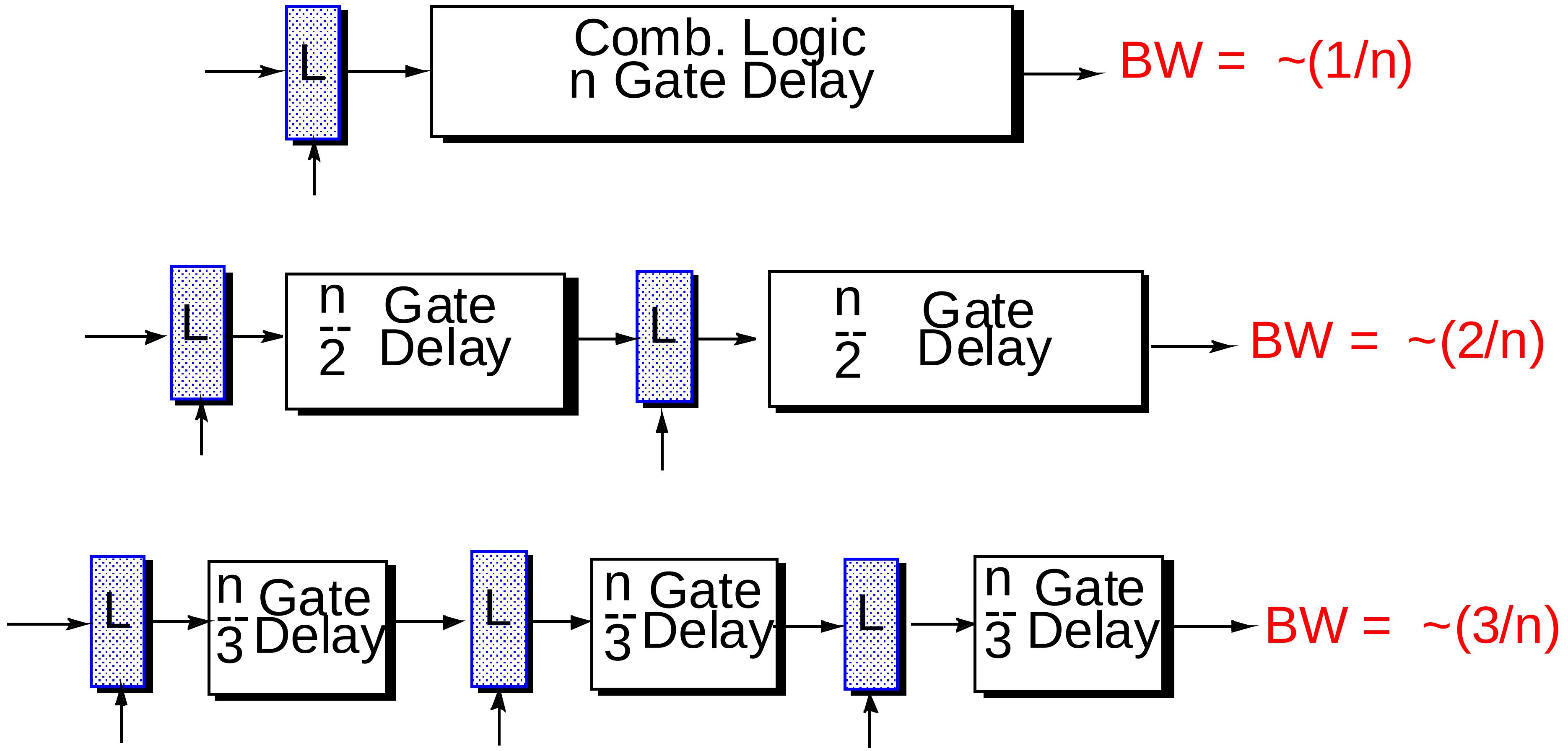
Pipelined Design

*Motivation: Increase throughput with
little increase in hardware*

- Bandwidth or Throughput = Performance
- Bandwidth (BW) = no. of tasks/unit time
- For a system that operates on one task at a time:

  BW = 1/ latency
- BW can be increased by pipelining if many operands exist which need the same operation, i.e. many repetitions of the same task are to be performed.
- Latency required for each task remains the same or may even increase slightly.

Pipeline Illustrated:



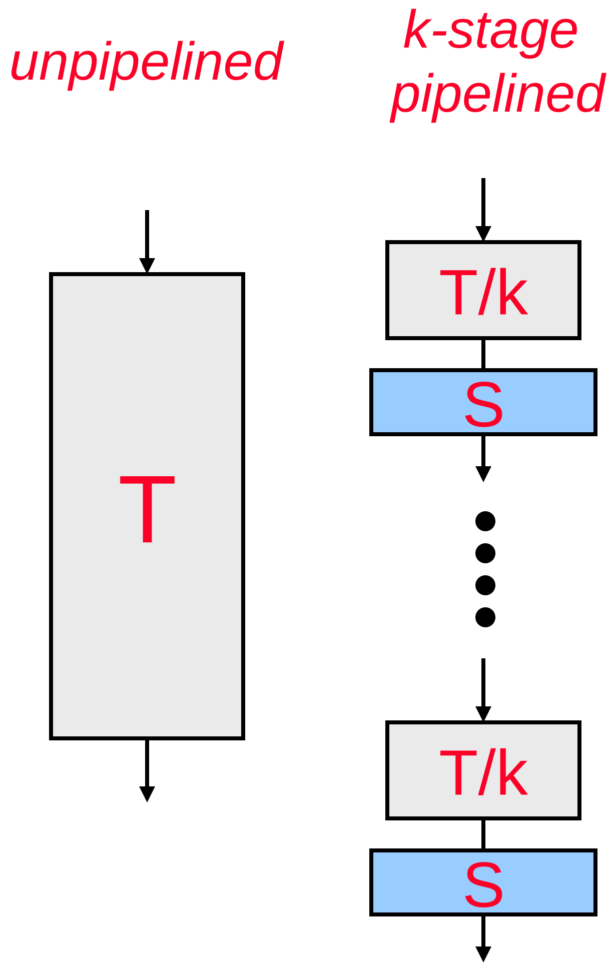
Performance Model

- Starting from an unpipelined version with propagation delay T and BW = 1/T

$P_{pipelined} = BW_{pipelined} = 1 / (T/k + S)$

where

S = delay through latch

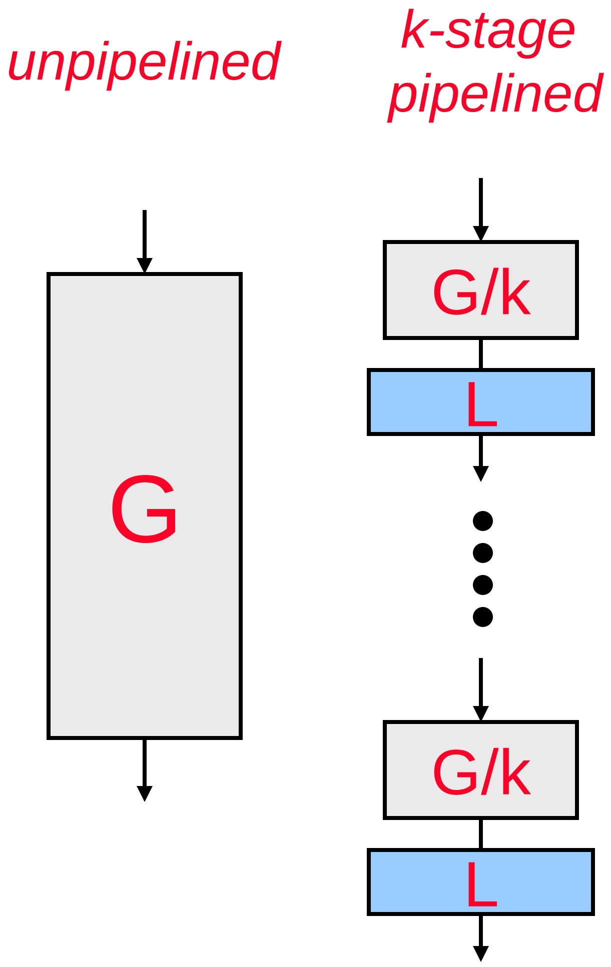


Hardware Cost Model

- Starting from an unpipelined version with hardware cost G

$$\text{Cost}_{\text{pipelined}} = kL + G$$

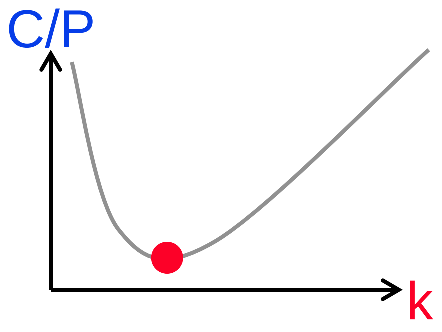
where
L = cost of adding each latch, and
k = number of stages



Cost/Performance Trade-off
[Peter M. Kogge, 1981]

Cost/Performance:

C/P = [Lk + G] / [1/(T/k + S)] = (Lk + G) (T/k + S)
= LT + GS + LSk + GT/k



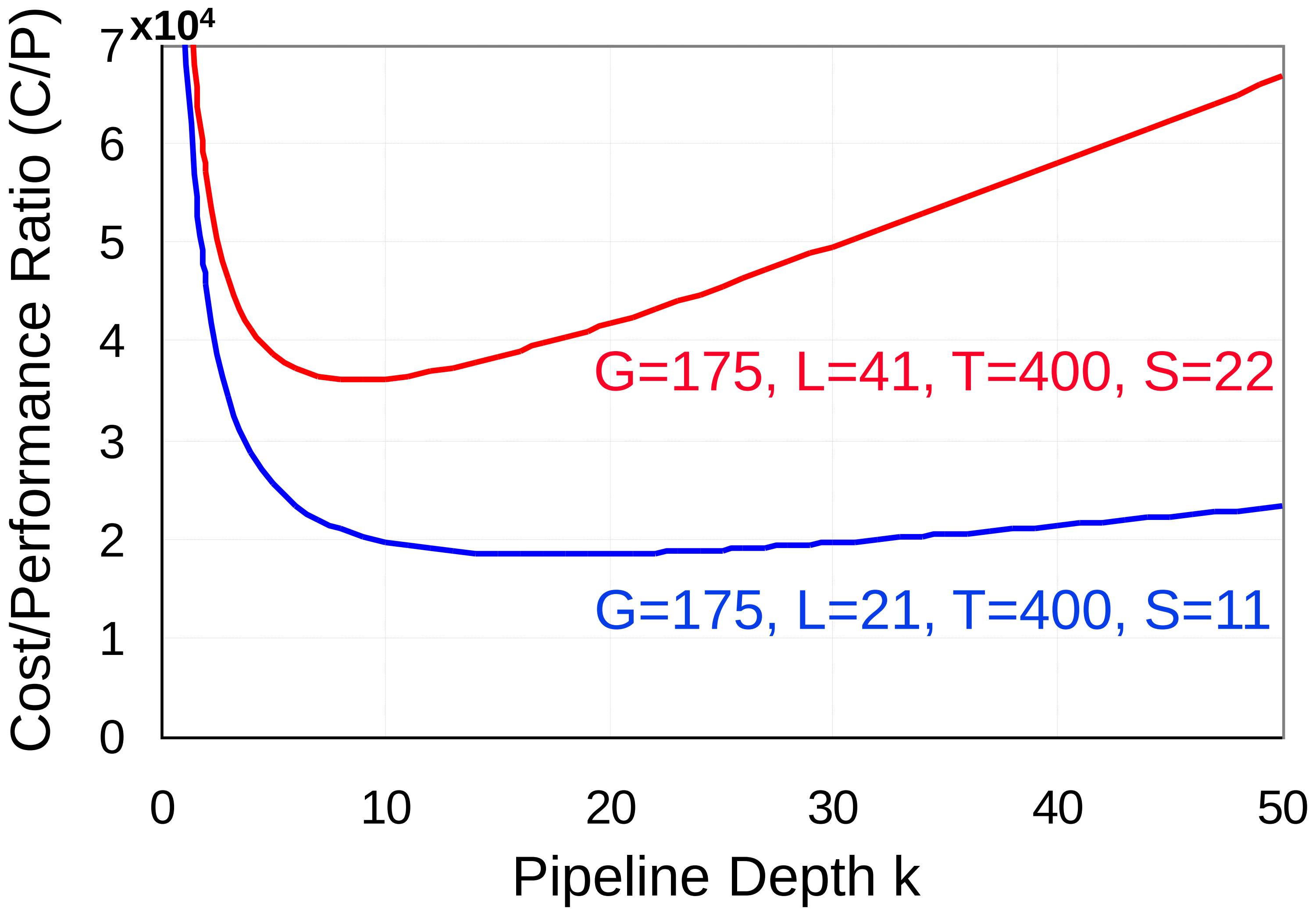
Optimal Cost/Performance: find min. C/P w.r.t. choice of k

$$\frac{d}{dk}\left(\frac{Lk+G}{\frac{1}{\frac{T}{k}+S}}\right) = 0 + 0 + LS - \frac{GT}{k^2}$$

$$LS - \frac{GT}{k^2} = 0$$

$$k_{opt} = \sqrt{\frac{GT}{LS}}$$

"Optimal" Pipeline Depth (k_{opt})



Pipelining Idealism

- *Uniform Suboperations*

  The operation to be pipelined can be evenly partitioned into uniform-latency suboperations

- *Repetition of Identical Operations*

  The same operations are to be performed repeatedly on a large number of different inputs

- *Repetition of Independent Operations*

  All the repetitions of the same operation are mutually independent, *i.e. no data dependence*

  and no resource conflicts

Good Examples: automobile assembly line

floating-point multiplier

instruction pipeline???

Instruction Pipeline Design

- Uniform Suboperations ... NOT!

  ⇒ balance pipeline stages

  - stage quantization to yield balanced stages
  - minimize internal fragmentation (some waiting stages)

- Identical operations ... NOT!

  ⇒ unifying instruction types

  - coalescing instruction types into one “multi-function” pipe
  - minimize external fragmentation (some idling stages)

- Independent operations ... NOT!

  ⇒ resolve data and resource hazards

  - inter-instruction dependency detection and resolution
  - minimize performance lose

The Generic Instruction Cycle

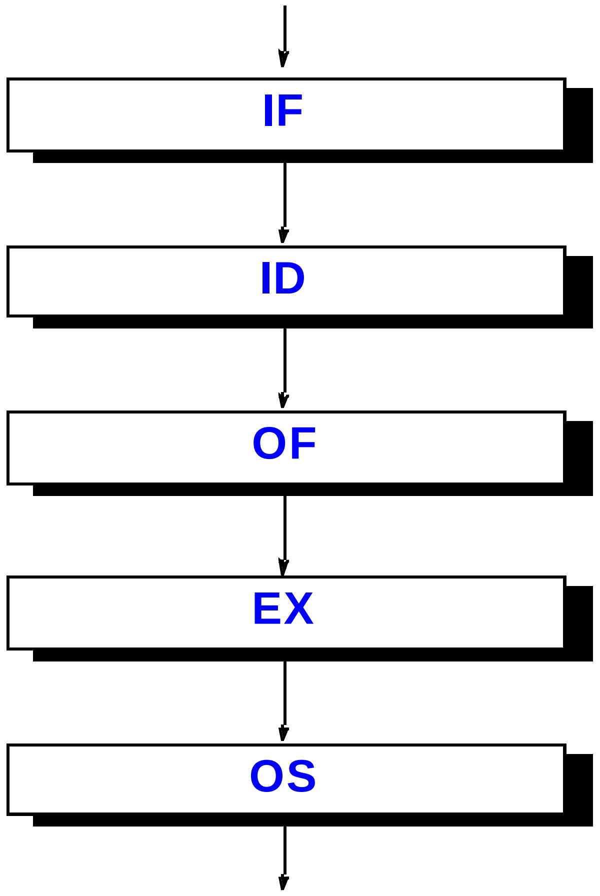
- The “computation” to be pipelined

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Operand(s) Fetch (OF)
4. Instruction Execution (EX)
5. Operand Store (OS)
6. Update Program Counter (PC)

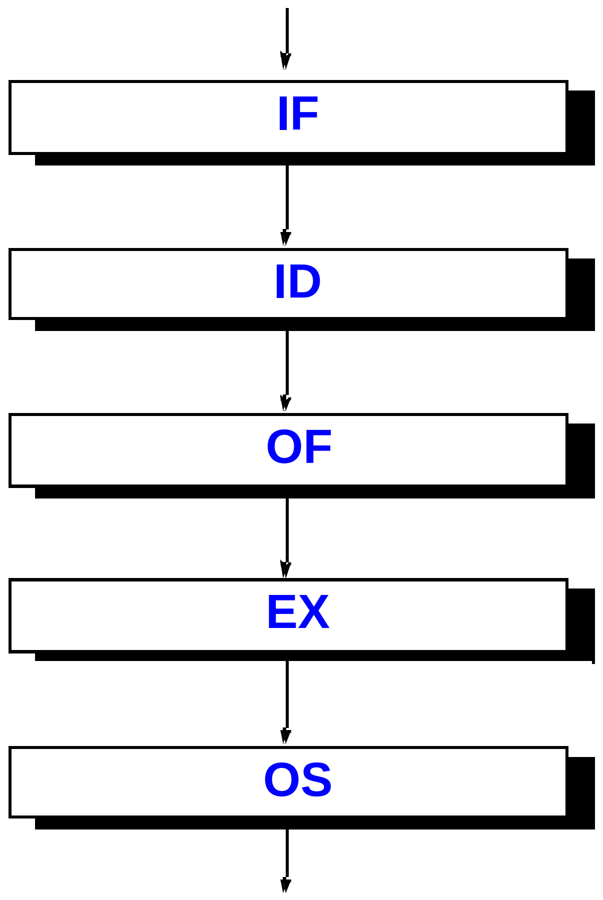
The GENERIC Instruction Pipeline (GNR)

Based on Obvious Subcomputations:

1. Instruction Fetch
2. Instruction Decode
3. Operand Fetch
4. Instruction Execute
5. Operand Store



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

Can we do better in terms of either performance or efficiency?

Balancing Pipeline Stages

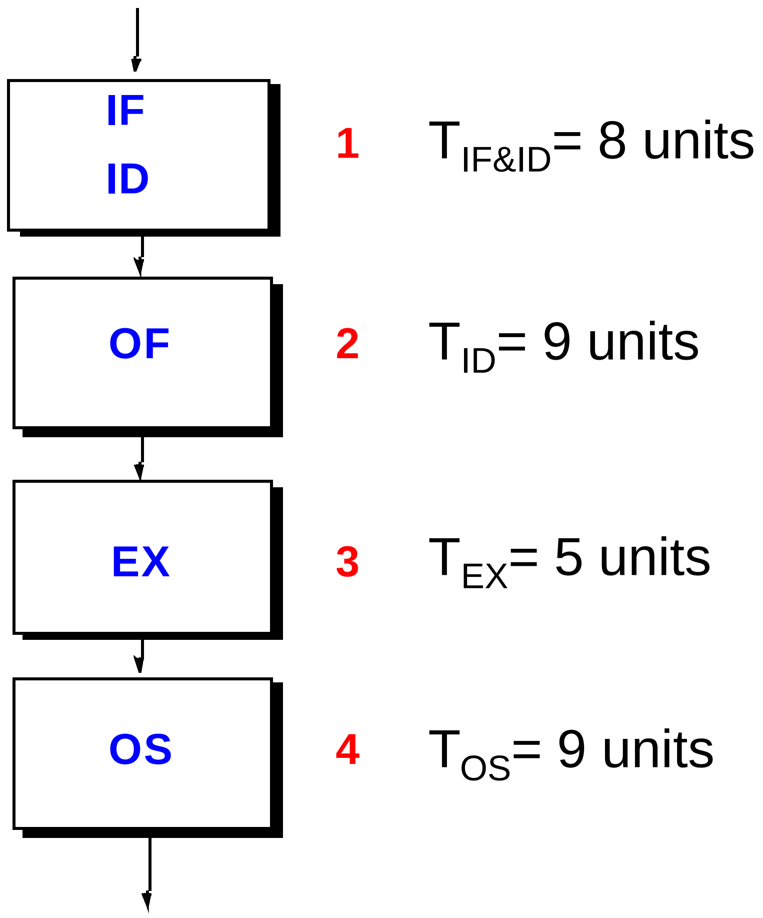
- Two Methods for Stage Quantization:
  - Merging of multiple subcomputations into one.
  - Subdividing a subcomputation into multiple subcomputations.

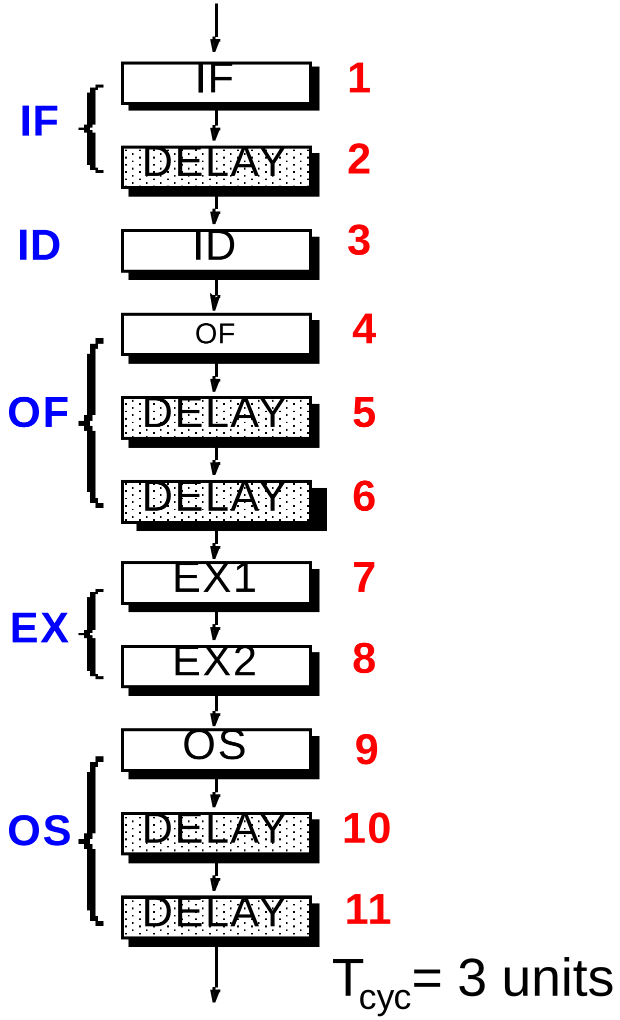
- Current Trends:
  - Deeper pipelines (more and more stages).
  - Multiplicity of different (subpipelines).
  - Pipelining of memory access (tricky).

Granularity of Pipeline Stages

Coarser-Grained Machine Cycle:
4 machine cyc / instruction cyc

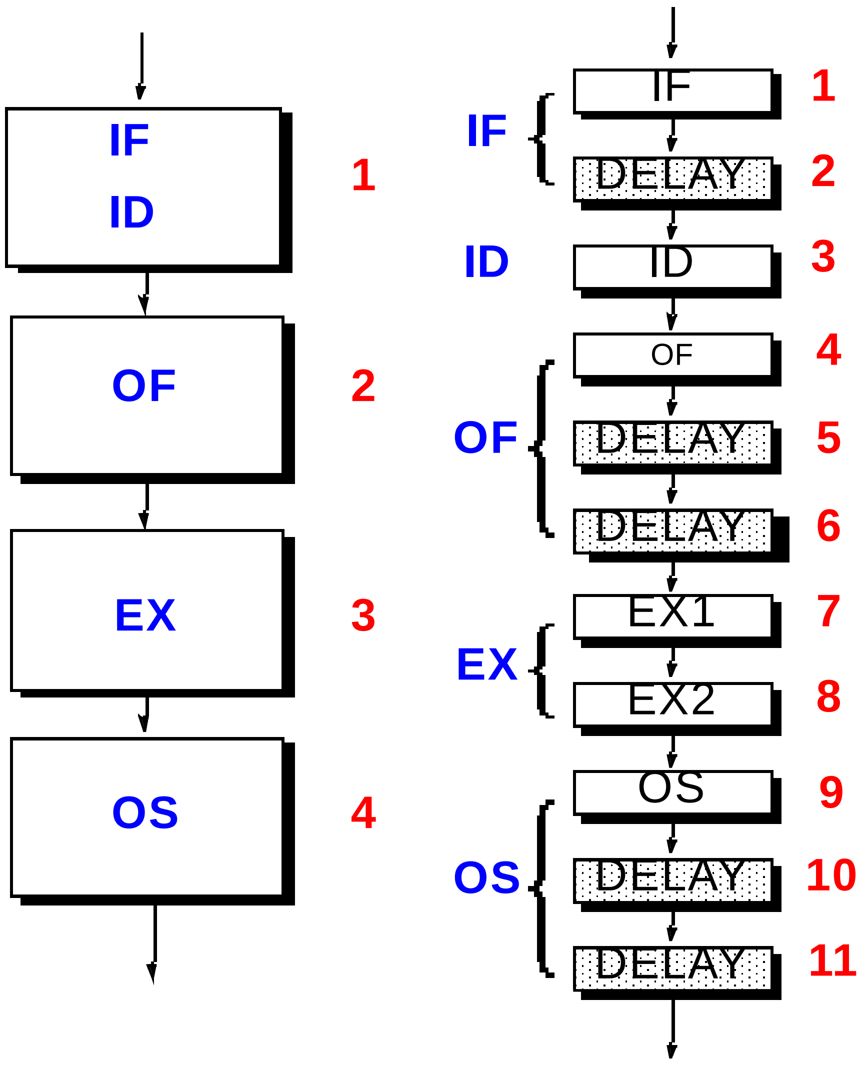
Finer-Grained Machine Cycle:
11 machine cyc /instruction cyc



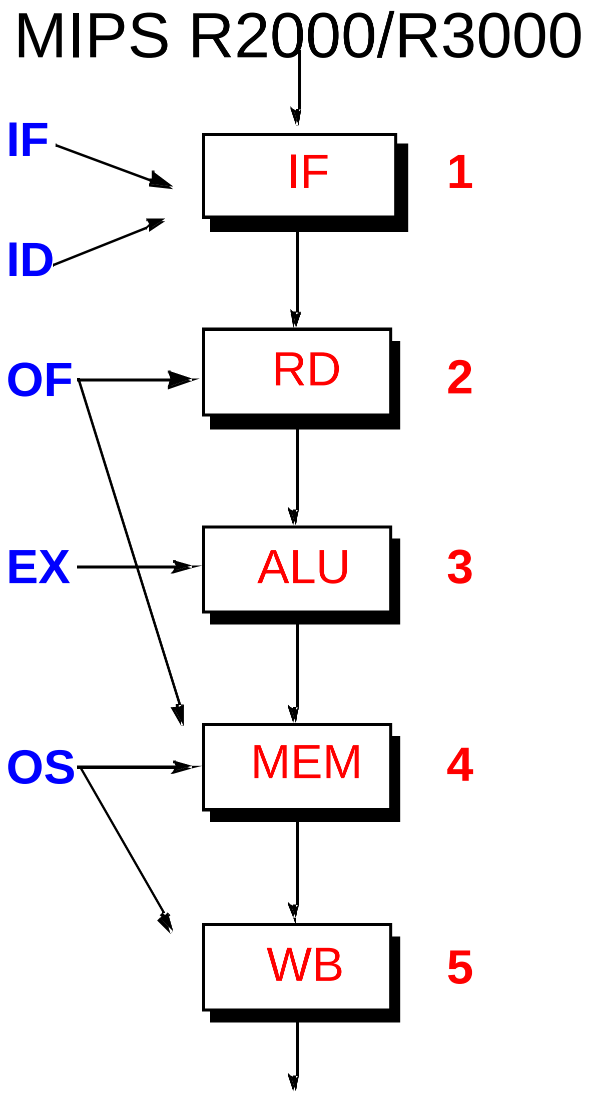


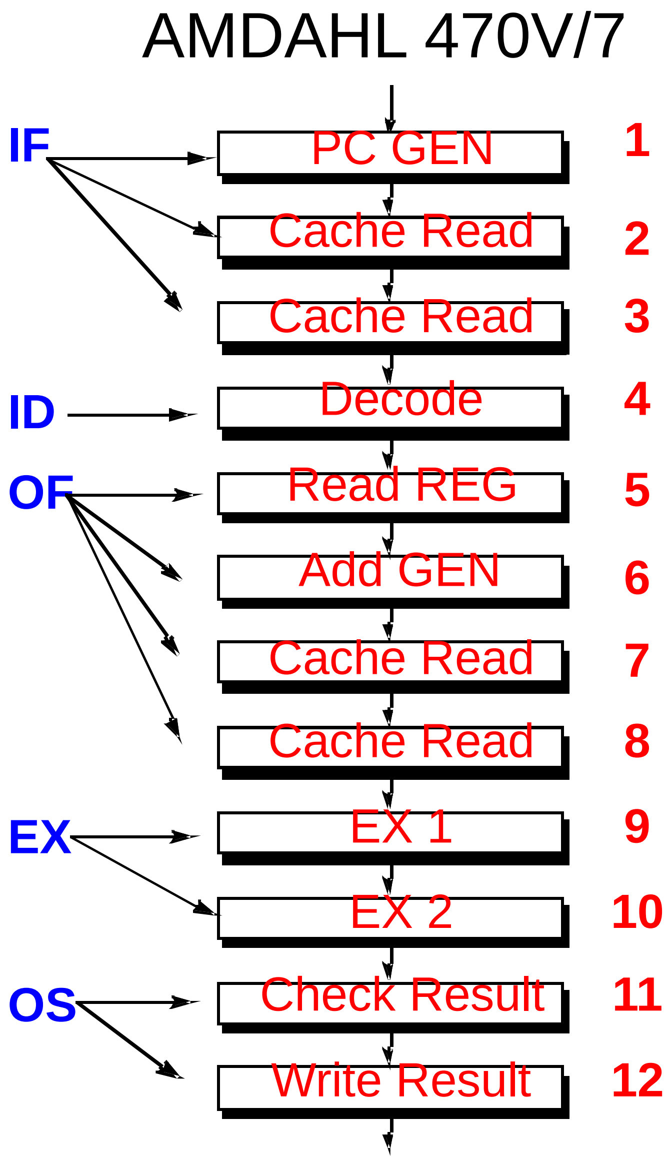
Hardware Requirements

- Logic needed for each pipeline stage
- Register file ports needed to support all the stages
- Memory accessing ports needed to support all the stages



Pipeline Examples



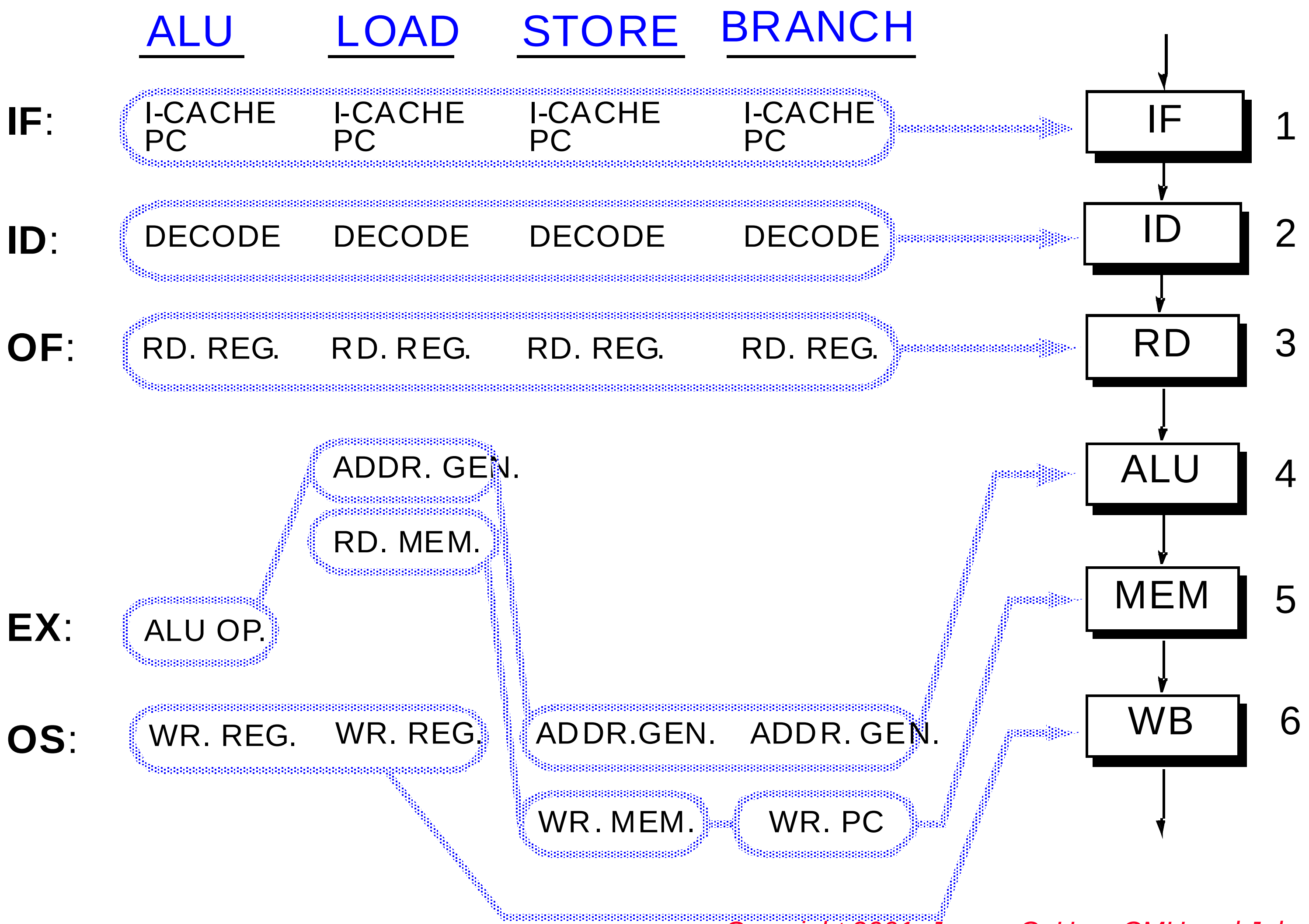


Unifying Instruction Types

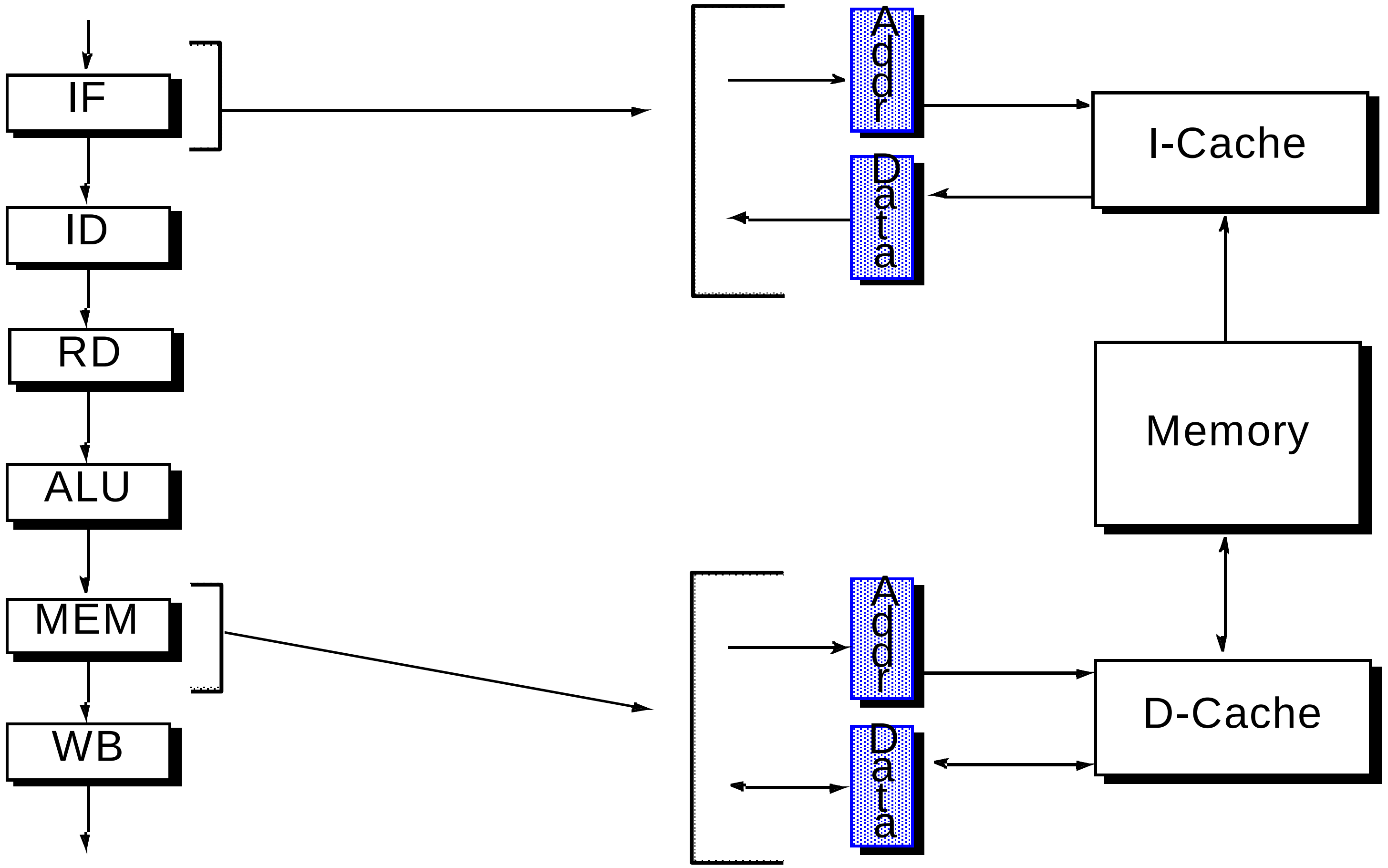
- Procedure:
  1. Analyze the sequence of register transfers required by each instruction type.
  2. Find commonality across instruction types and merge them to share the same pipeline stage.
  3. If there exists flexibility, shift or reorder some register transfers to facilitate further merging.

Coalescing Resource Requirements

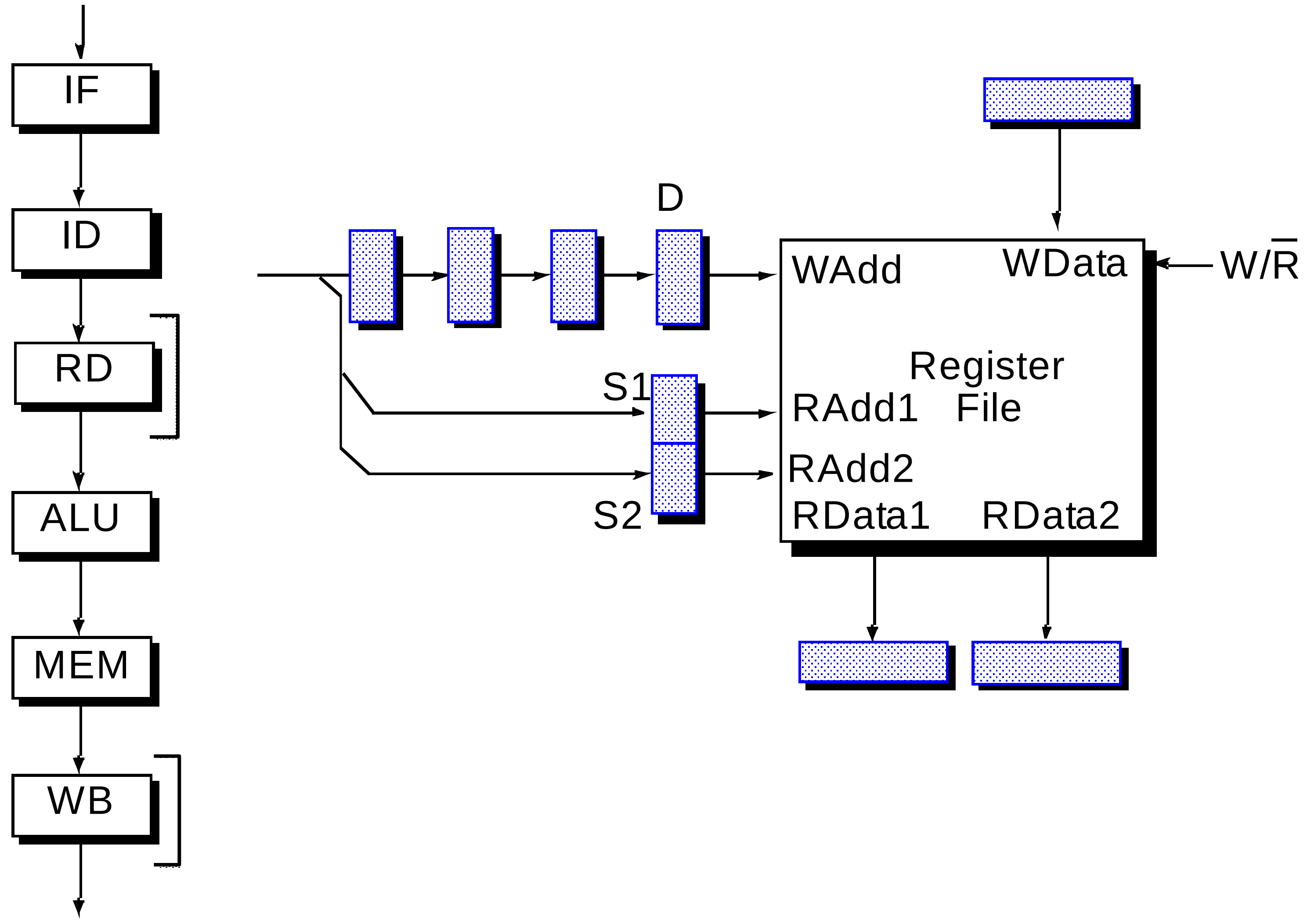
The 6-stage TYPICAL (TYP) pipeline:



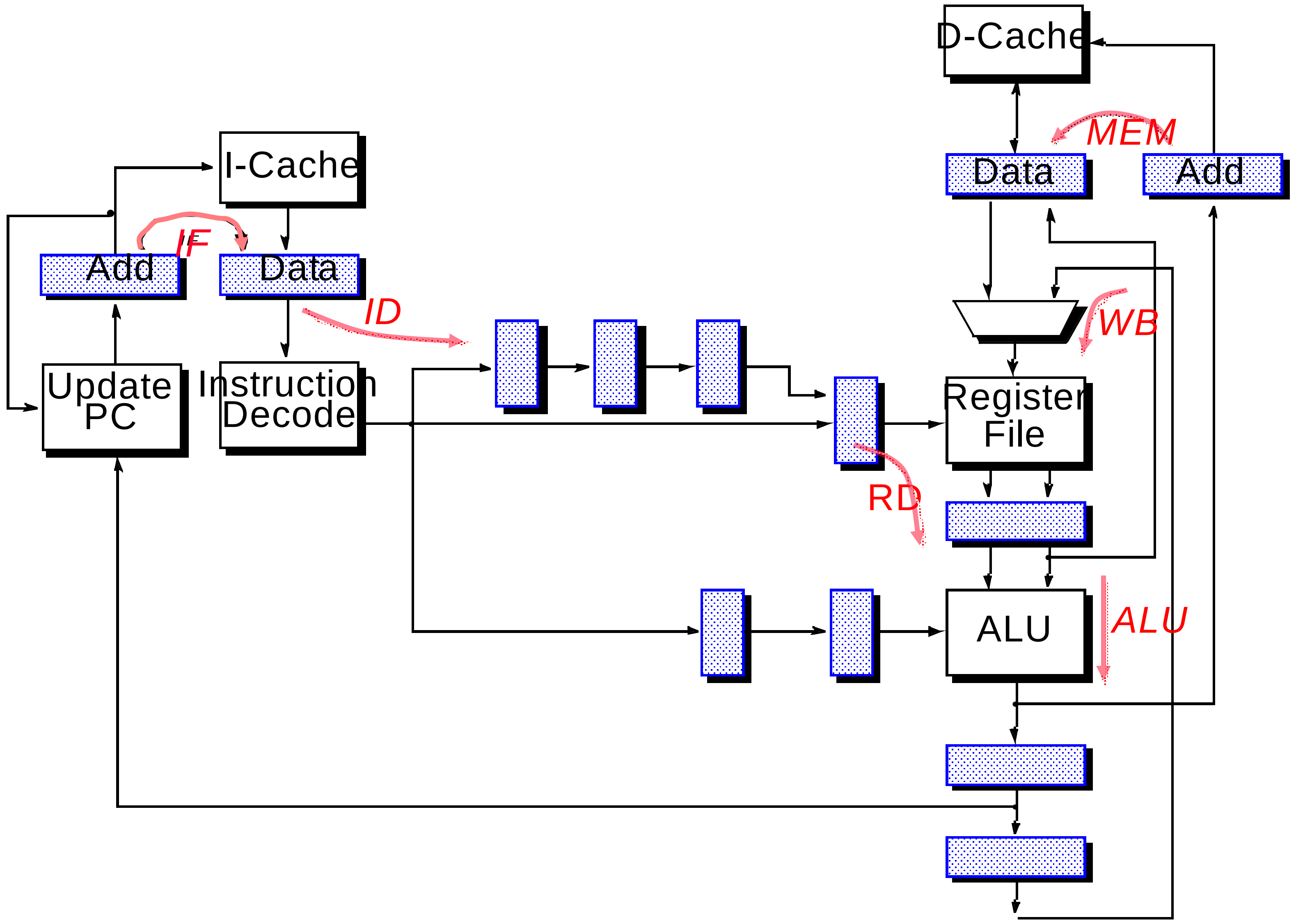
Interface to Memory Subsystem



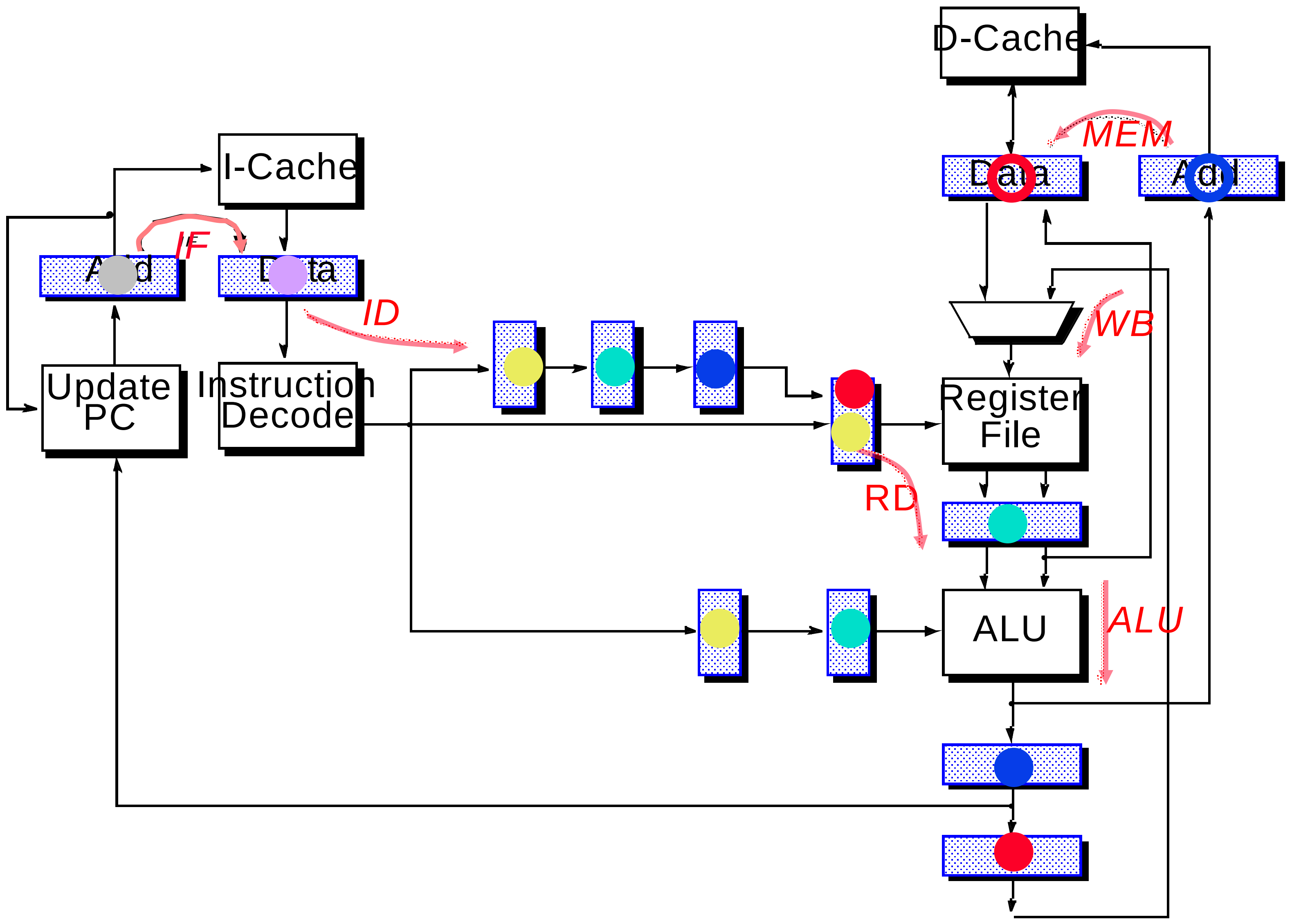
Pipeline Interface to Register File:



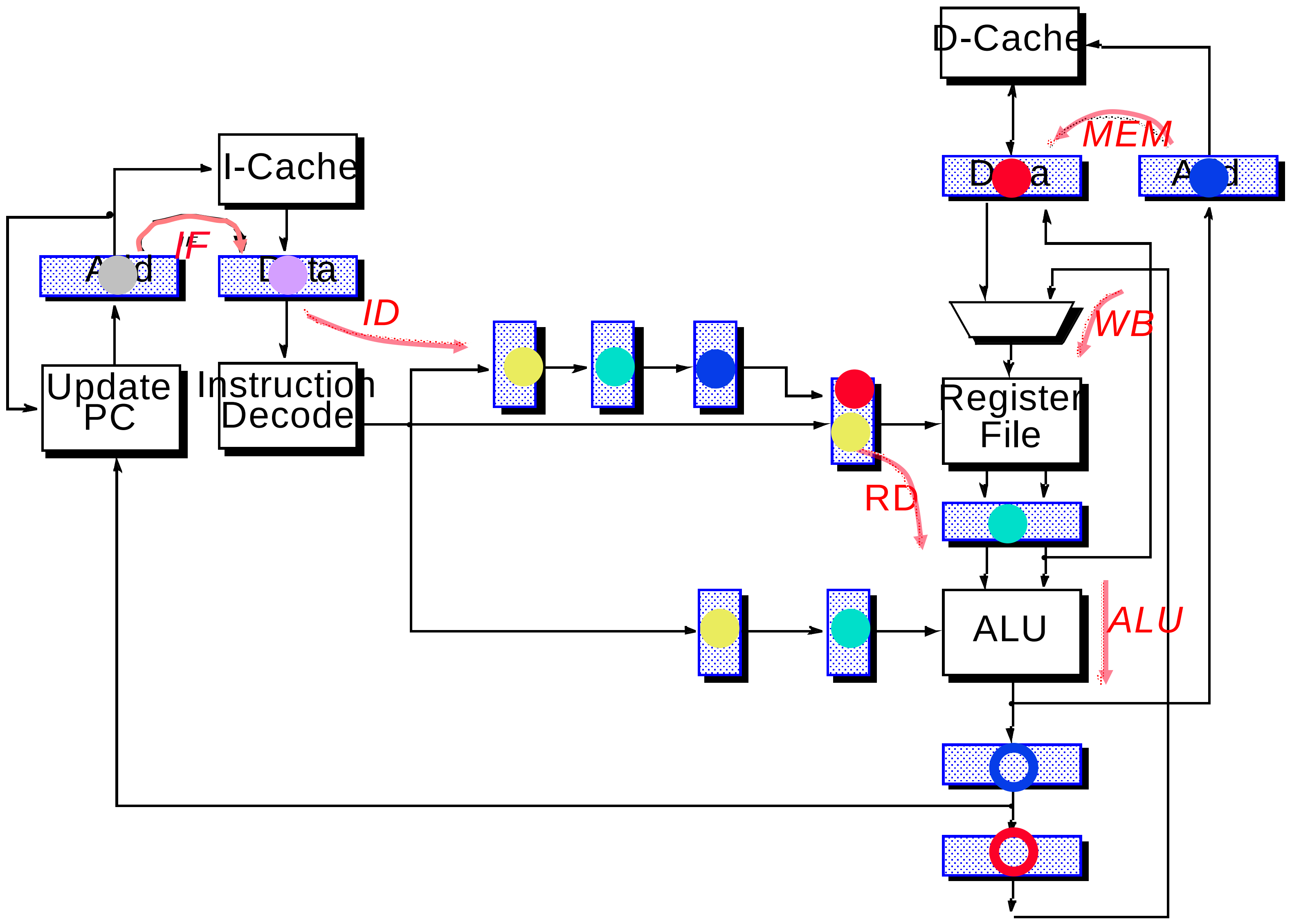
6-stage TYP Pipeline



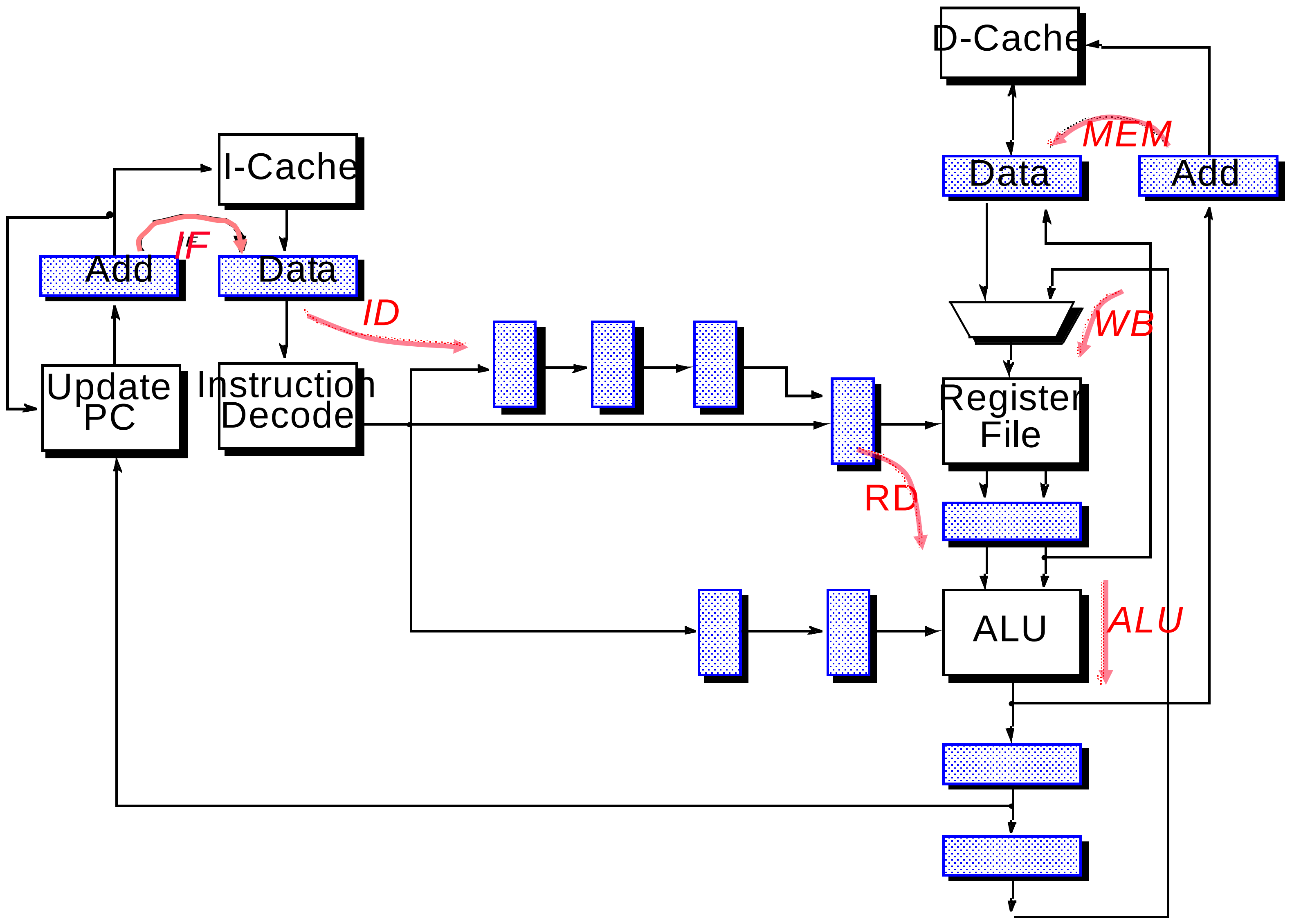
ALU Instruction Flow Path



Load Instruction Flow Path

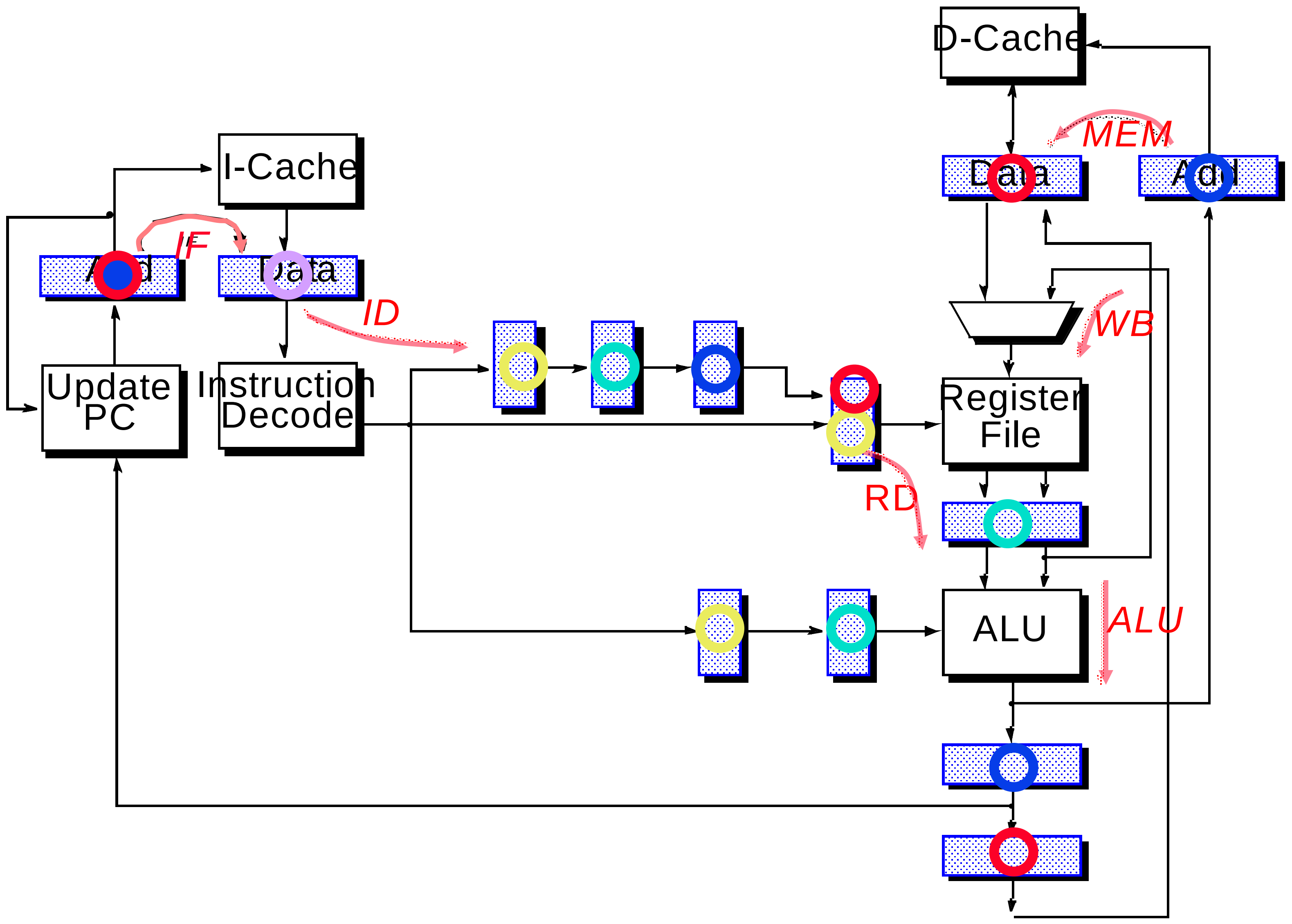


Store Instruction Flow Path



What is wrong in this figure (from p2-31 S&L)?

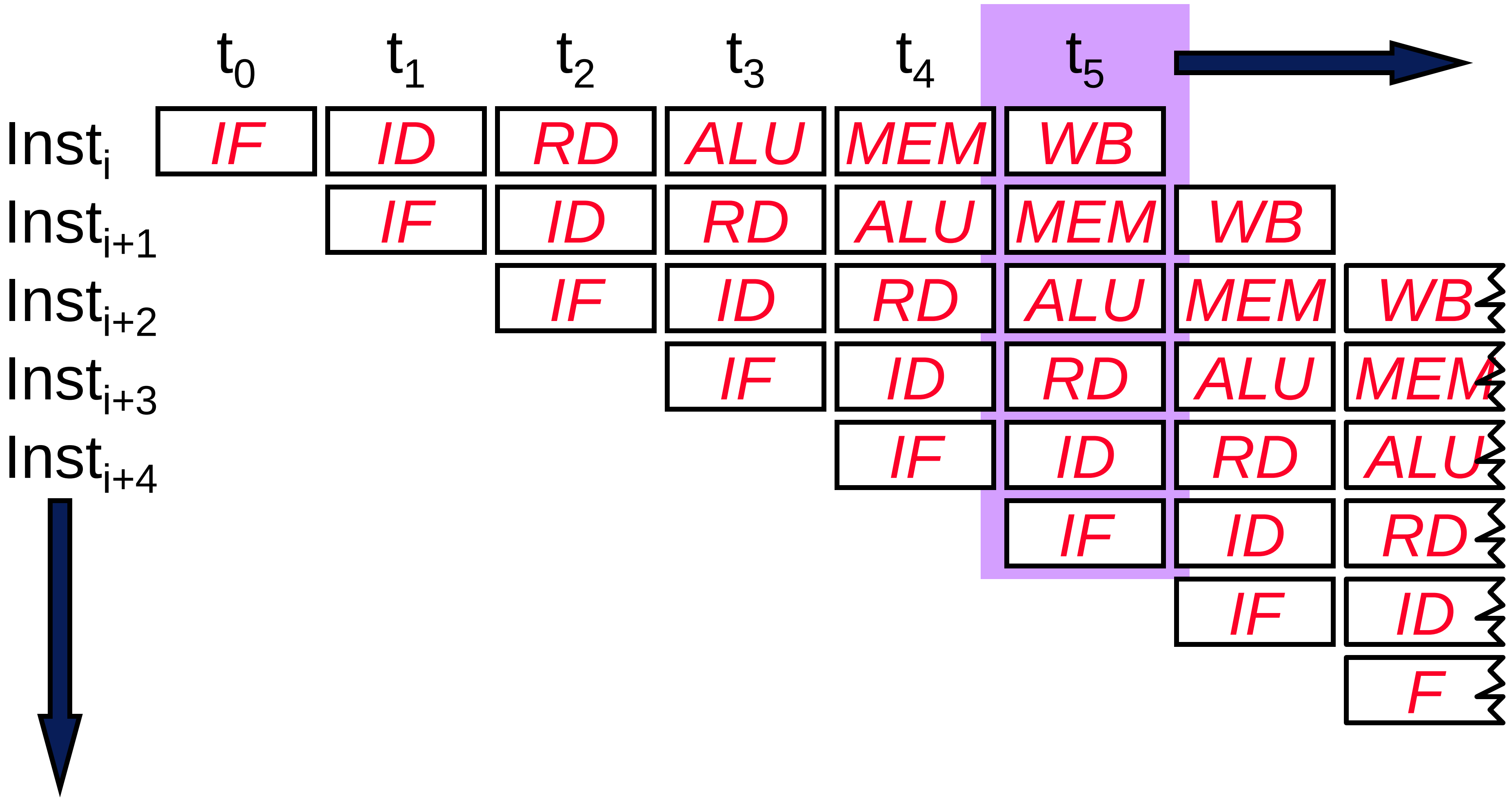
Branch Instruction Flow Path



Pipeline Resource Diagram

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}
ID		I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}
RD			I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
ALU				I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
MEM					I_1	I_2	I_3	I_4	I_5	I_6	I_7
WB						I_1	I_2	I_3	I_4	I_5	I_6

Pipelining: Steady State



Instruction Dependencies

- Data Dependence
  - True dependence (RAW)

    Instruction must wait for all required input operands
  - Anti-Dependence (WAR)

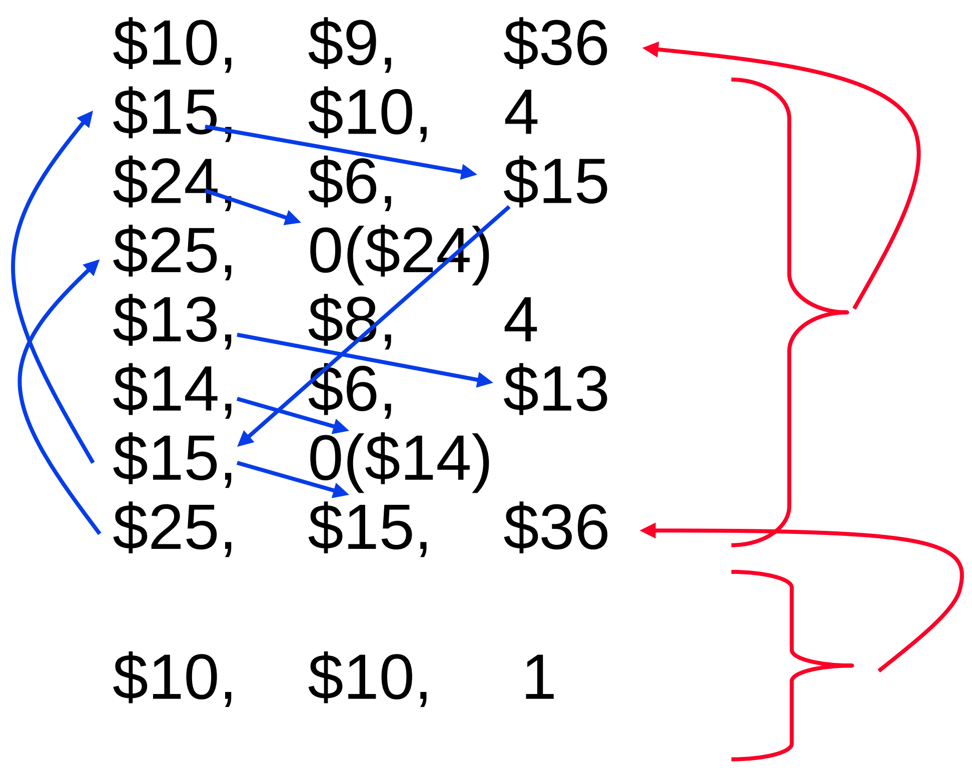
    Later write must not clobber a still-pending earlier read
  - Output dependence (WAW)

    Earlier write must not clobber an already-finished later write

- Control Dependence (aka Procedural Dependence)
  - Conditional branches cause uncertainty to instruction sequencing
  - Instructions following a conditional branch depends on the resolution of the branch instruction

(more exact definition later)

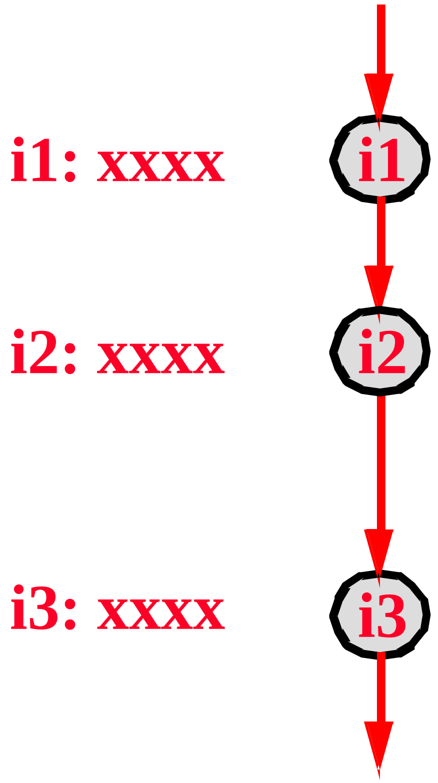
Example: Quick Sort on MIPS R2000



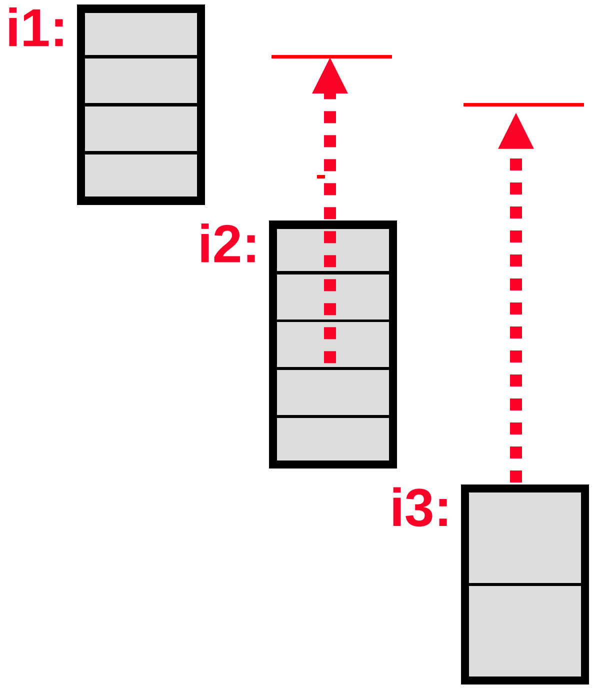
```
        bge     $10,  $9,    $36
        mul     $15,  $10,   4
        addu    $24,  $6,    $15
        lw      $25,  0($24)
        mul     $13,  $8,    4
        addu    $14,  $6,    $13
        lw      $15,  0($14)
        bge     $25,  $15,   $36
$35:
        addu    $10,  $10,   1
        . . .
$36:
        addu    $11,  $11,   -1
        . . .

#       for (;(j<high)&&(array[j]<array[low]);++j);
#       $10 = j; $9 = high; $6 = array; $8 = low
```

Instruction Dependences and Pipeline Hazards

Sequential Code Semantics

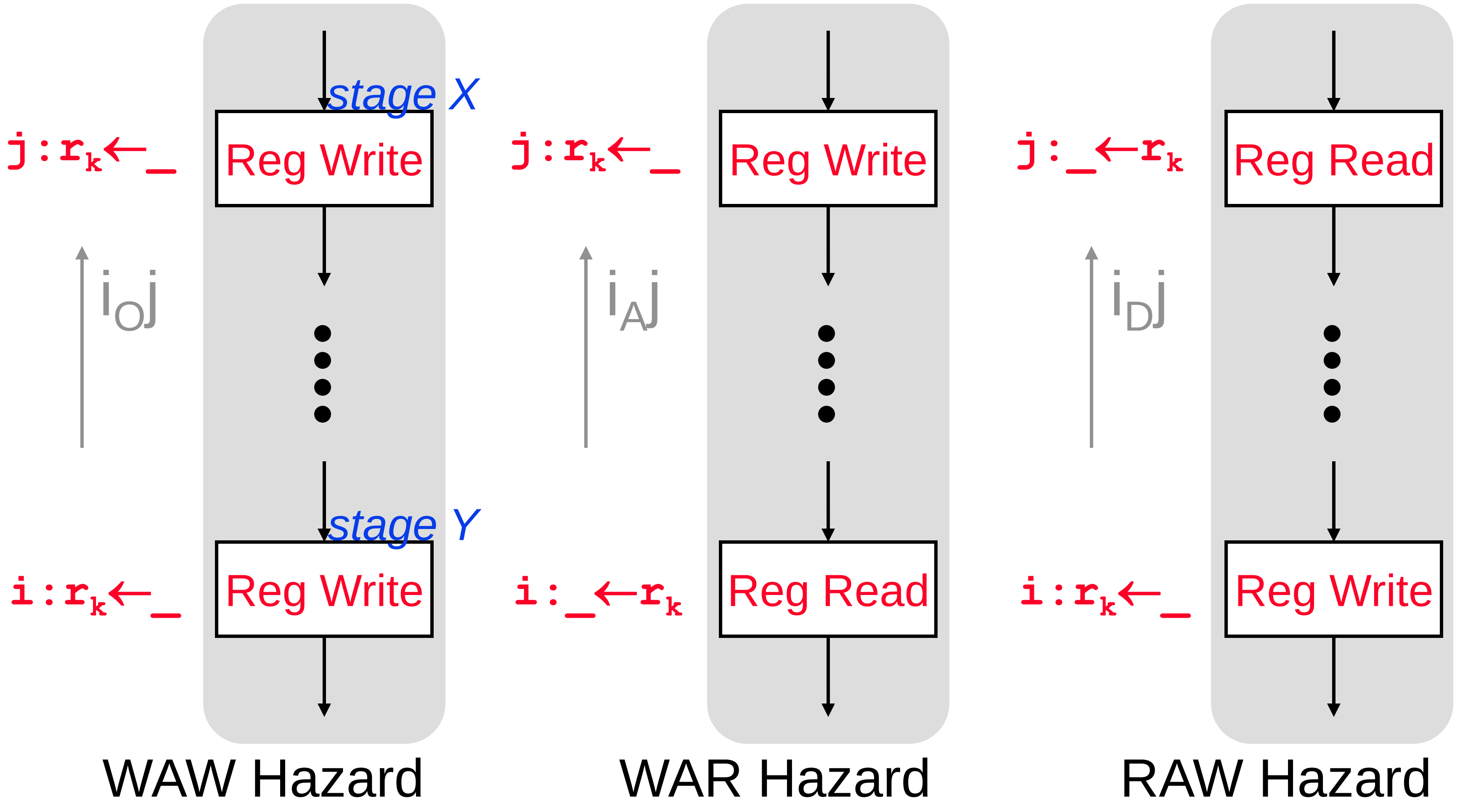


A true dependence between two instructions may only involve one subcomputation of each instruction.



The implied sequential precedences are overspecifications. It is sufficient but not necessary to ensure program correctness.

Necessary Conditions for Data Hazards



dist(i,j) ≤ *dist*(X,Y) ⇒ Hazard!!

dist(i,j) > *dist*(X,Y) ⇒ Safe

Hazards due to Memory Data Dependences

Pipe Stage	ALU Inst.	Load inst.	Store inst.	Branch inst.
1. IF	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4
2. ID	decode	decode	decode	decode
3. RD	read reg.	read reg.	read reg.	read reg.
4. ALU	ALU op.	addr. gen.	addr. gen.	addr. gen. cond. gen.
5. MEM	-----	read mem.	write mem.	PC<-br. addr.
6. WB	write reg.	write reg.	-----	-----

Hazards due to Register Data Dependences

Pipe Stage	ALU Inst.	Load inst.	Store inst.	Branch inst.
1. IF	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4
2. ID	decode	decode	decode	decode
3. RD	read reg.	read reg.	read reg.	read reg.
4. ALU	ALU op.	addr. gen.	addr. gen.	addr. gen. cond. gen.
5. MEM	-----	read mem.	write mem.	PC<-br. addr.
6. WB	write reg.	write reg.	-----	-----

Hazards due to Control Dependences

Pipe Stage	ALU Inst.	Load inst.	Store inst.	Branch inst.
1. IF	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4
2. ID	decode	decode	decode	decode
3. RD	read reg.	read reg.	read reg.	read reg.
4. ALU	ALU op.	addr. gen.	addr. gen.	addr. gen. cond. gen.
5. MEM	-----	read mem.	write mem.	PC<-br. addr.
6. WB	write reg.	write reg.	-----	-----