18-447
Computer Architecture
Lecture 7: Pipelining

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2014, 1/29/2014

Can We Do Better?

- What limitations do you see with the multi-cycle design?

- Limited concurrency
  - Some hardware resources are idle during different phases of instruction processing cycle
  - "Fetch" logic is idle when an instruction is being "decoded" or "executed"
  - Most of the datapath is idle when a memory access is happening

Can We Use the Idle Hardware to Improve Concurrency?

- Goal: Concurrency → throughput (more "work" completed in one cycle)

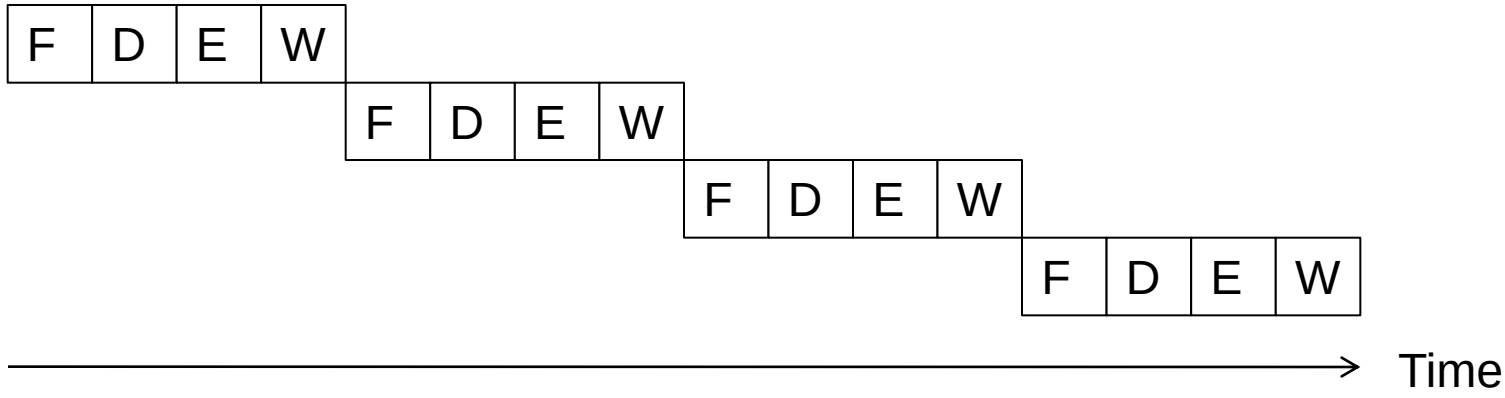
- Idea: When an instruction is using some resources in its processing phase, process other instructions on idle resources not needed by that instruction
  - E.g., when an instruction is being decoded, fetch the next instruction
  - E.g., when an instruction is being executed, decode another instruction
  - E.g., when an instruction is accessing data memory (ld/st), execute the next instruction
  - E.g., when an instruction is writing its result into the register file, access data memory for the next instruction

Pipelining: Basic Idea

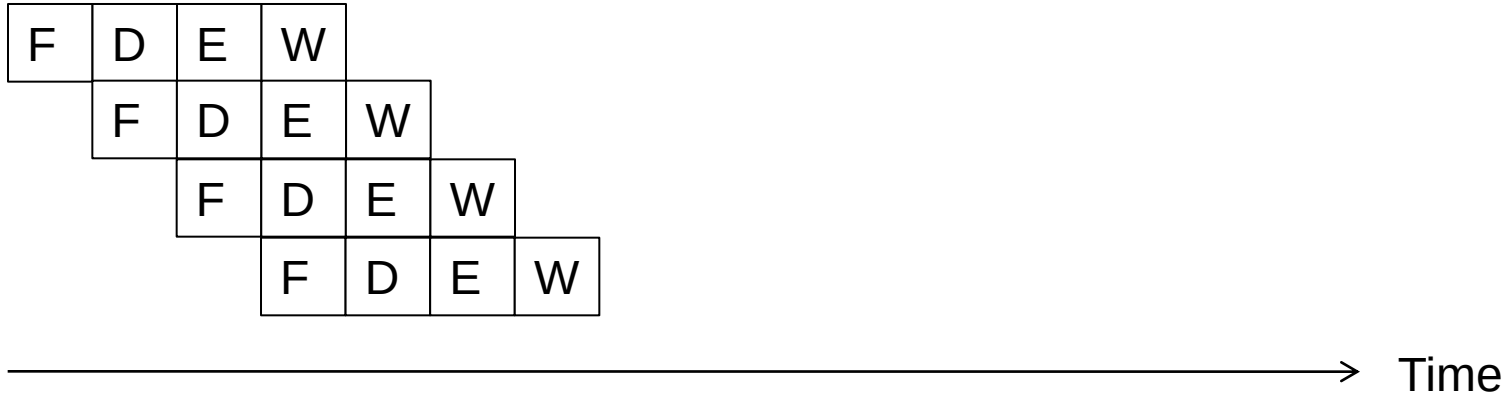
- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: "Assembly line processing" of instructions
- Idea:
  - Divide the instruction processing cycle into distinct "stages" of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a different instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages
- Benefit: Increases instruction processing throughput (1/CPI)
- Downside: Start thinking about this…

Example: Execution of Four Independent ADDs

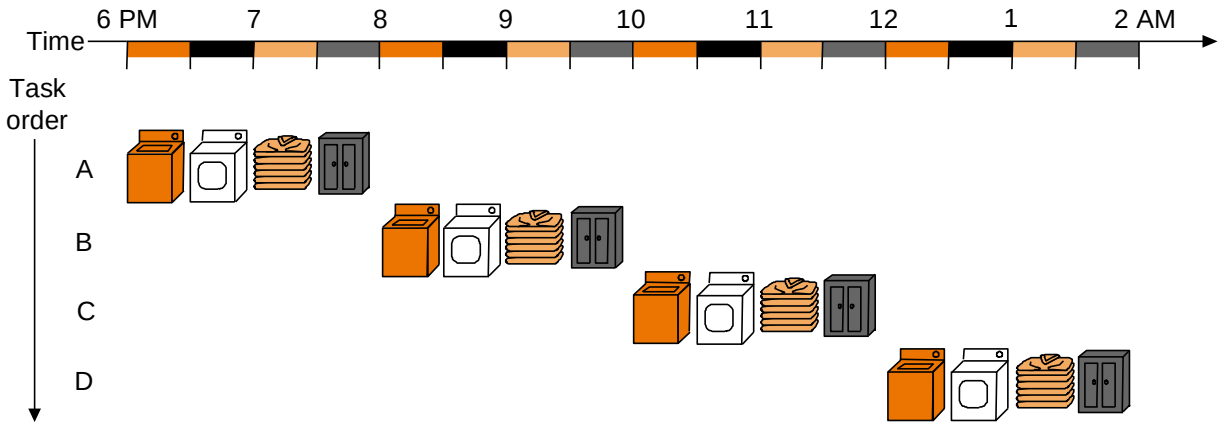
- Multi-cycle: 4 cycles per instruction



- Pipelined: 4 cycles per 4 instructions (steady state)



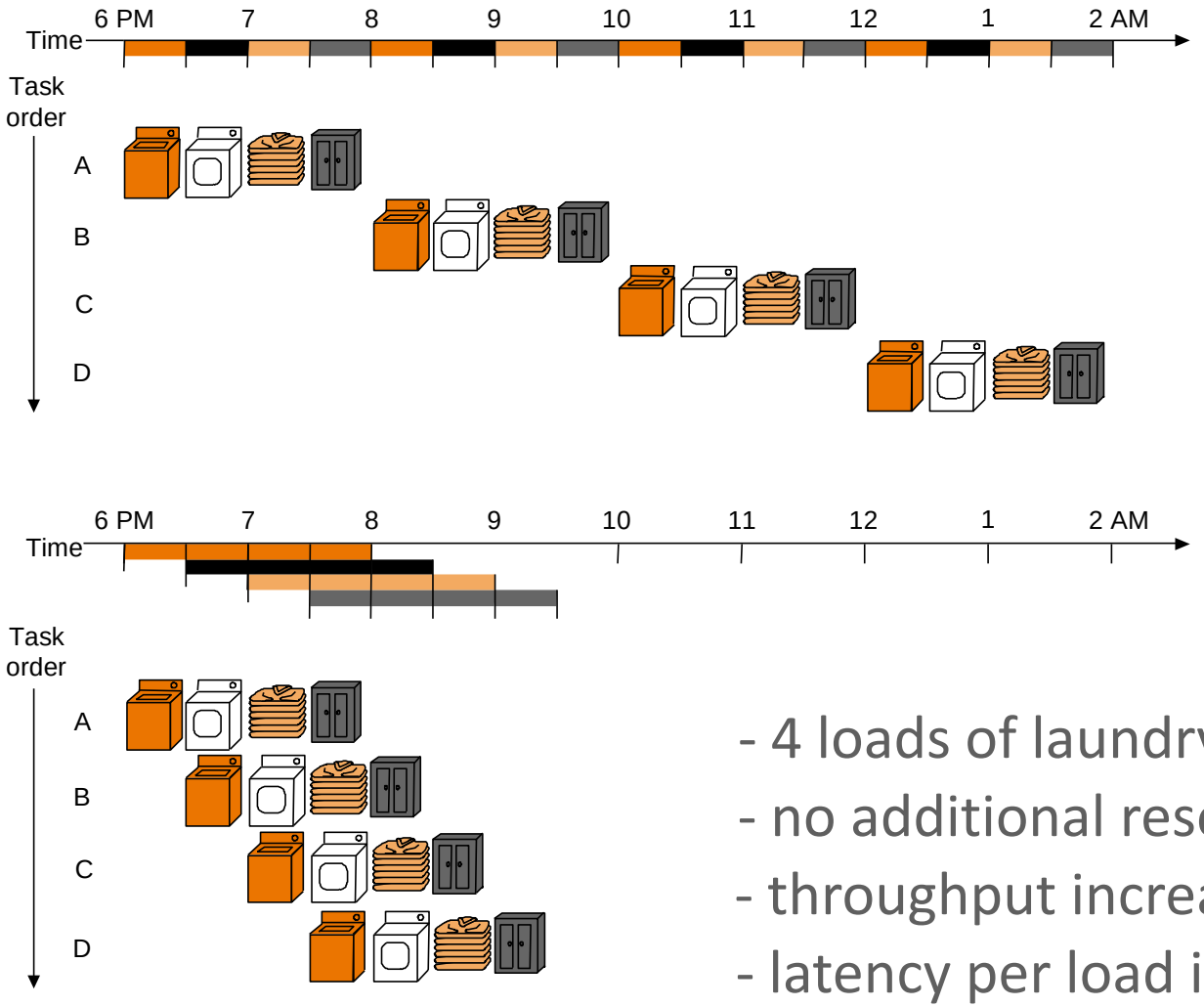
The Laundry Analogy



- "place one dirty load of clothes in the washer"
- "when the washer is finished, place the wet load in the dryer"
- "when the dryer is finished, take out the dry load and fold"
- "when folding is finished, ask your roommate (??) to put the clothes away"

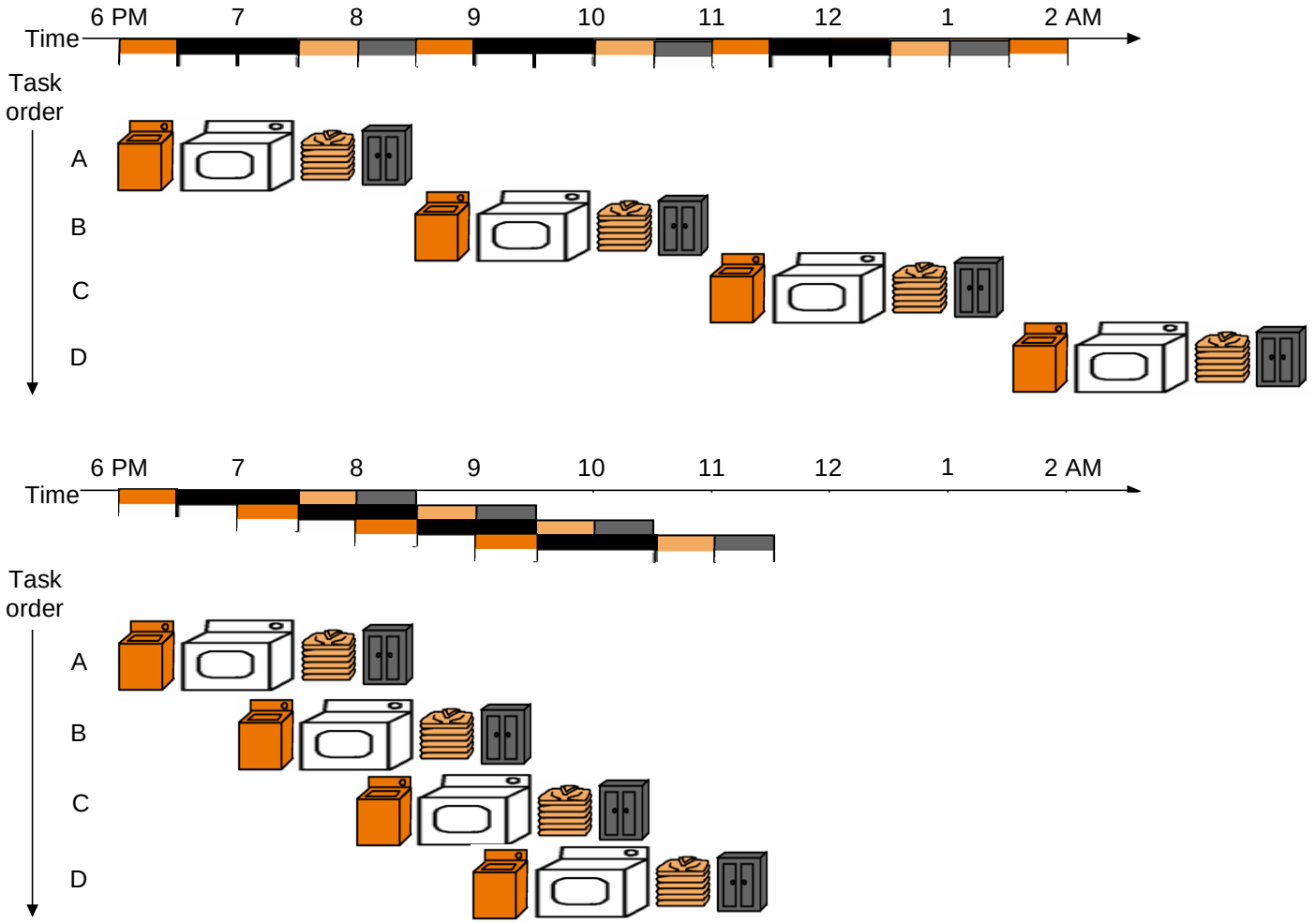
- steps to do a load are sequentially dependent
- no dependence between different loads
- different steps do not share resources

Pipelining Multiple Loads of Laundry



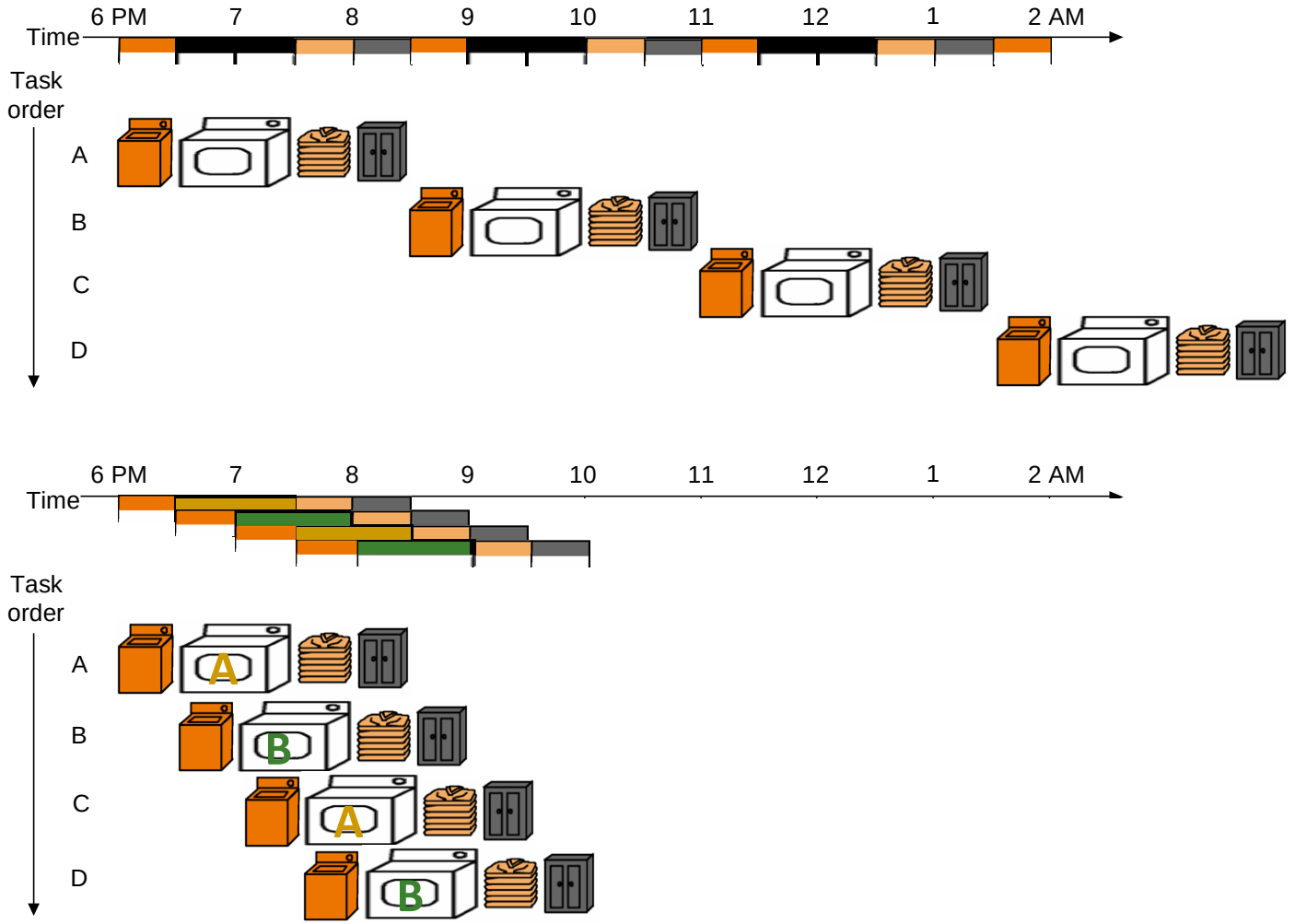
- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

Pipelining Multiple Loads of Laundry: In Practice



the slowest step decides throughput

Pipelining Multiple Loads of Laundry: In Practice

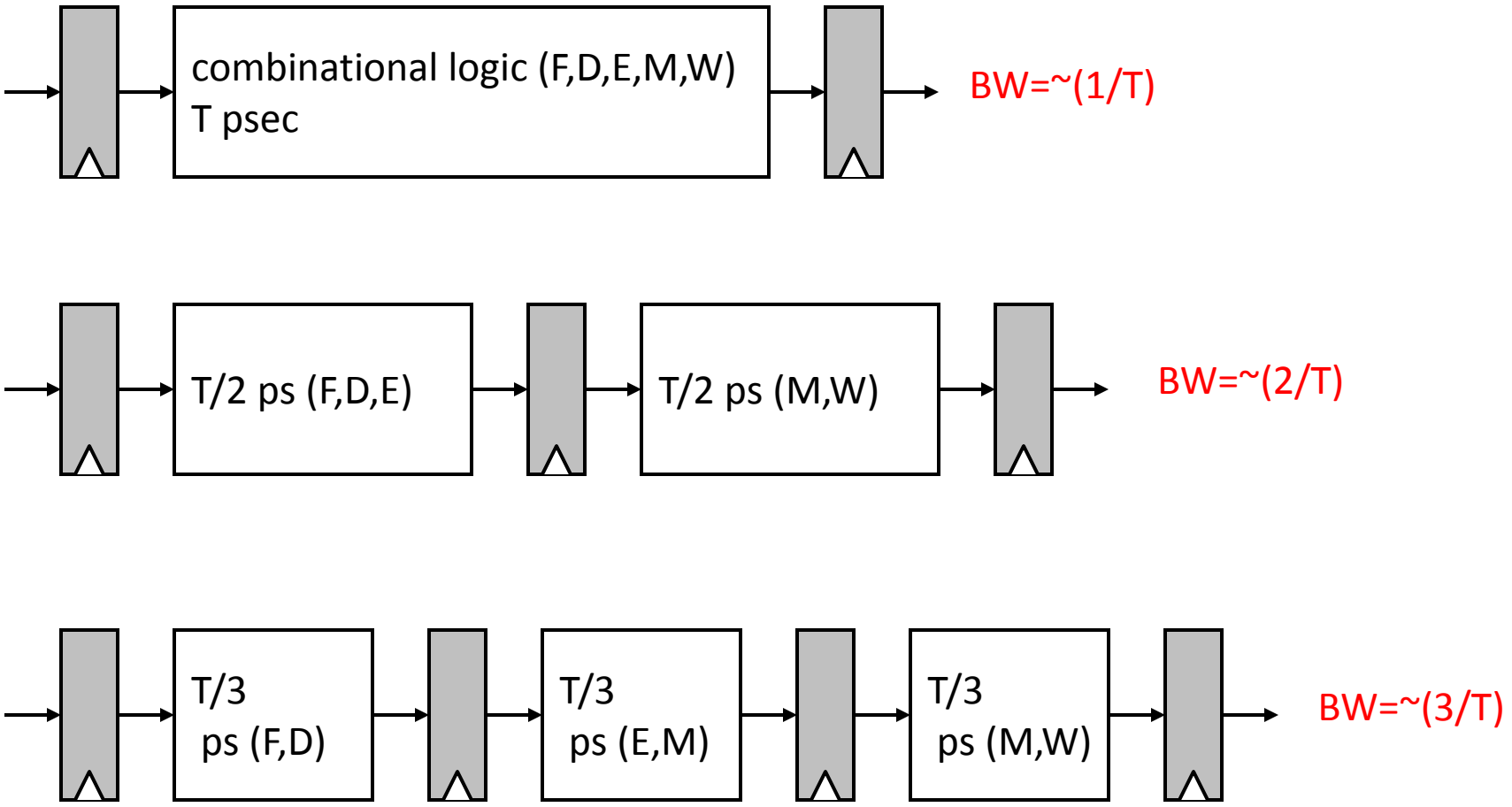


Throughput restored (2 loads per hour) using 2 dryers

An Ideal Pipeline

- Goal: Increase throughput with little increase in cost (hardware cost, in case of instruction processing)
- Repetition of identical operations
  - The same operation is repeated on a large number of different inputs
- Repetition of independent operations
  - No dependencies between repeated operations
- Uniformly partitionable suboperations
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry
  - What about the instruction processing "cycle"?

Ideal Pipelining



More Realistic Pipeline: Throughput

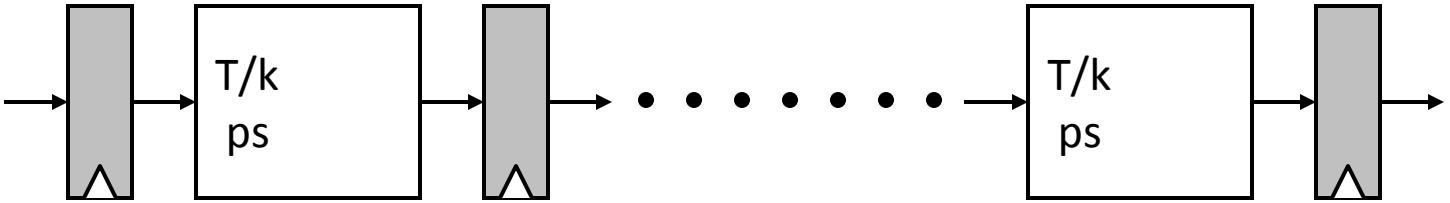
- Nonpipelined version with delay T

  BW = 1/(T+S) where S = latch delay

- k-stage pipelined version

  $BW_{k\text{-stage}} = 1 / (T/k + S)$

  $BW_{max} = 1 / (1 \text{ gate delay} + S)$



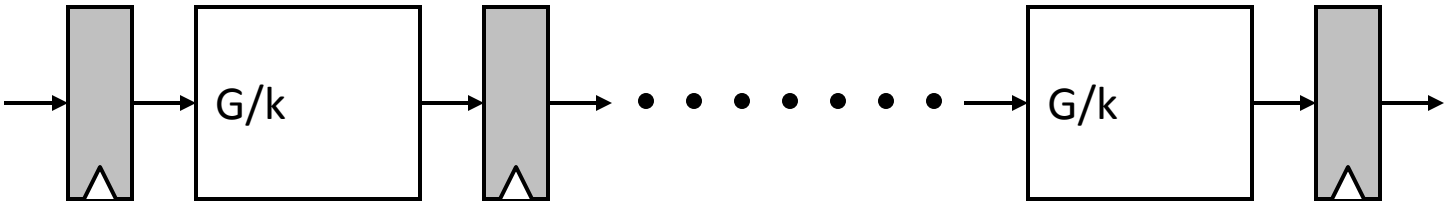
More Realistic Pipeline: Cost

- Nonpipelined version with combinational cost G

  Cost = G+L where L = latch cost

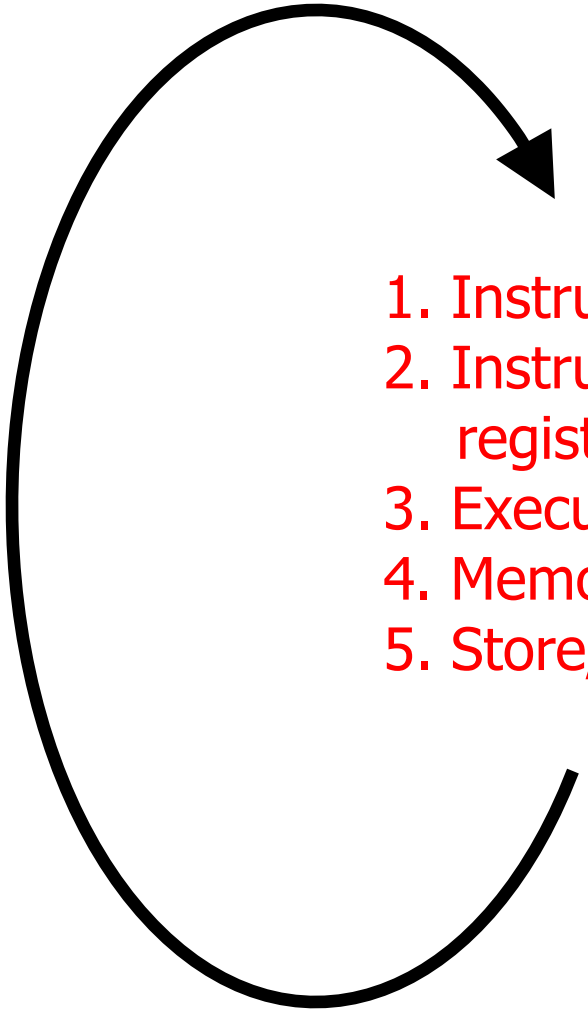
- k-stage pipelined version

  $Cost_{k\text{-}stage} = G + Lk$



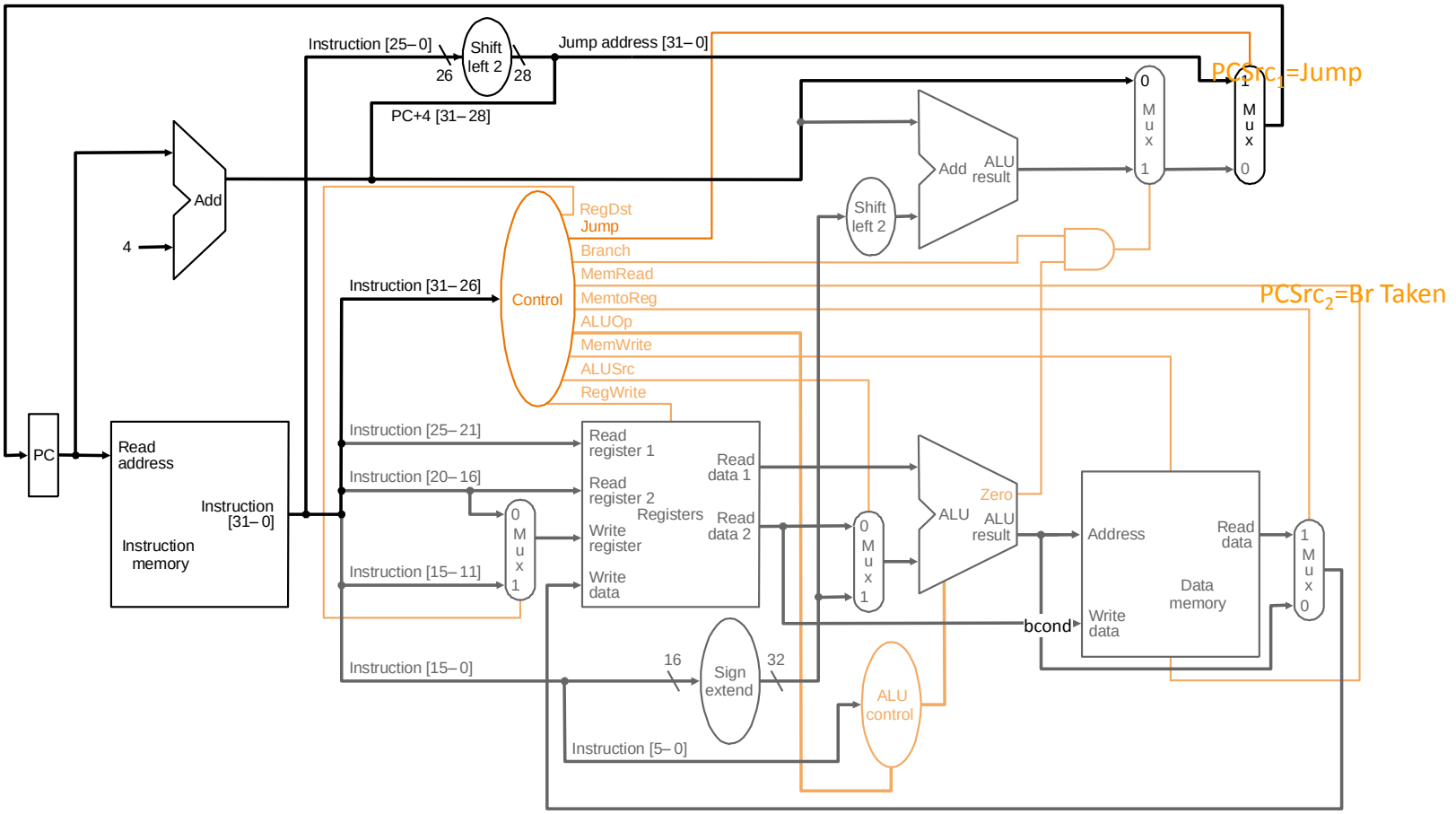
Pipelining Instruction Processing

Remember: The Instruction Processing Cycle



1. Instruction fetch (IF)
2. Instruction decode and
   register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)

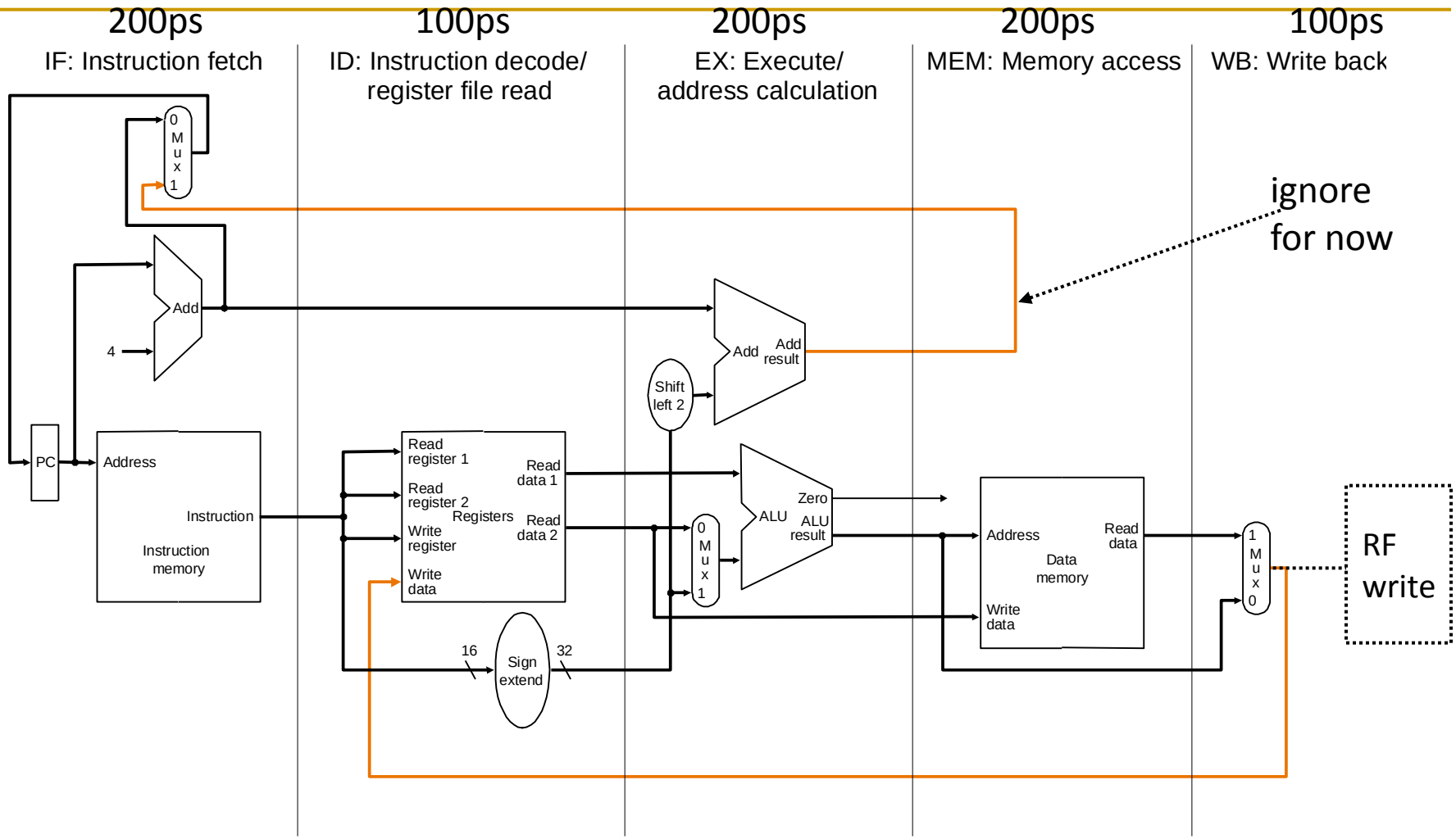
Remember the Single-Cycle Uarch



ALU operation

BW=~(1/T)

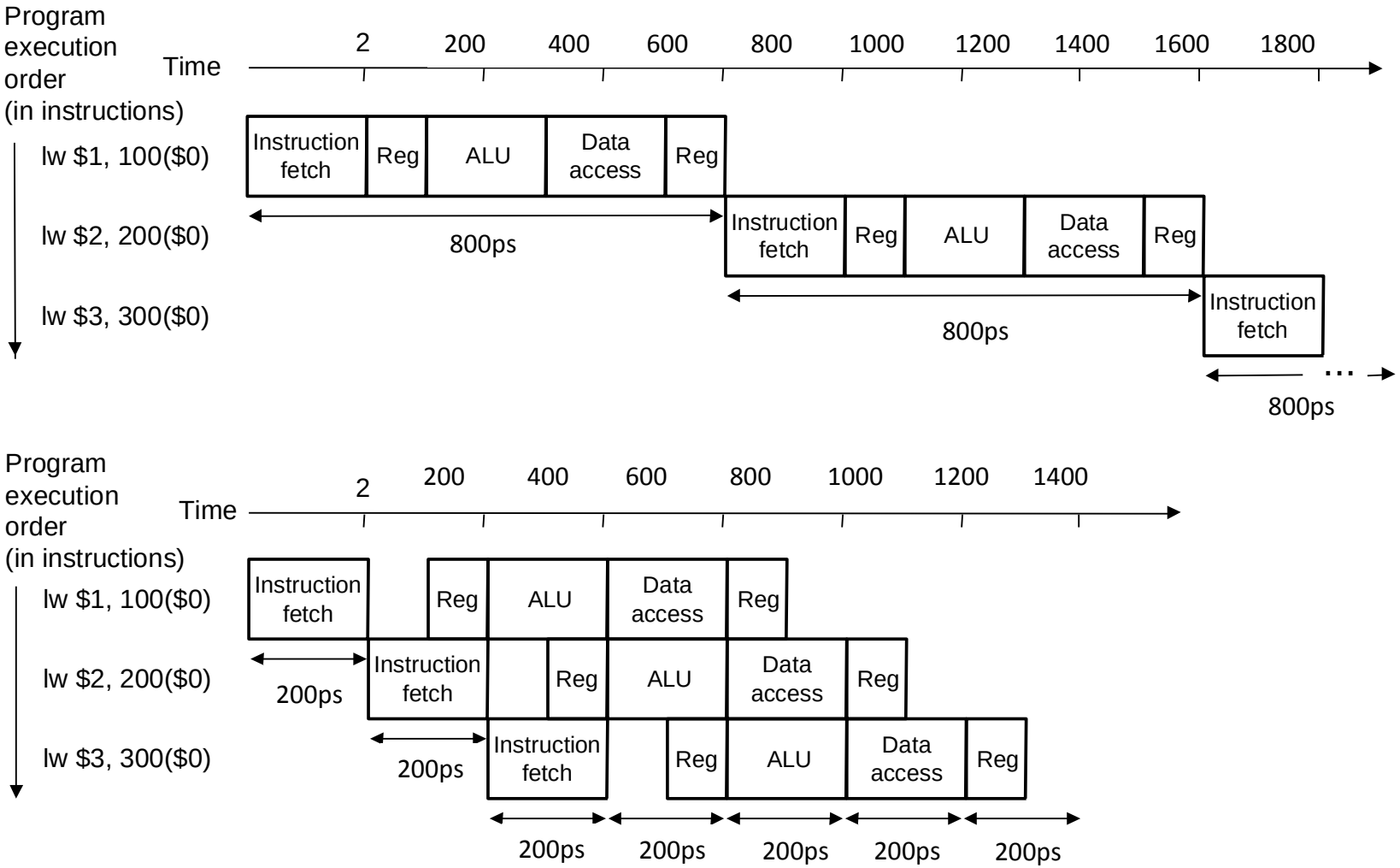
Dividing Into Stages



Is this the correct partitioning?

Why not 4 or 6 stages? Why not different boundaries?

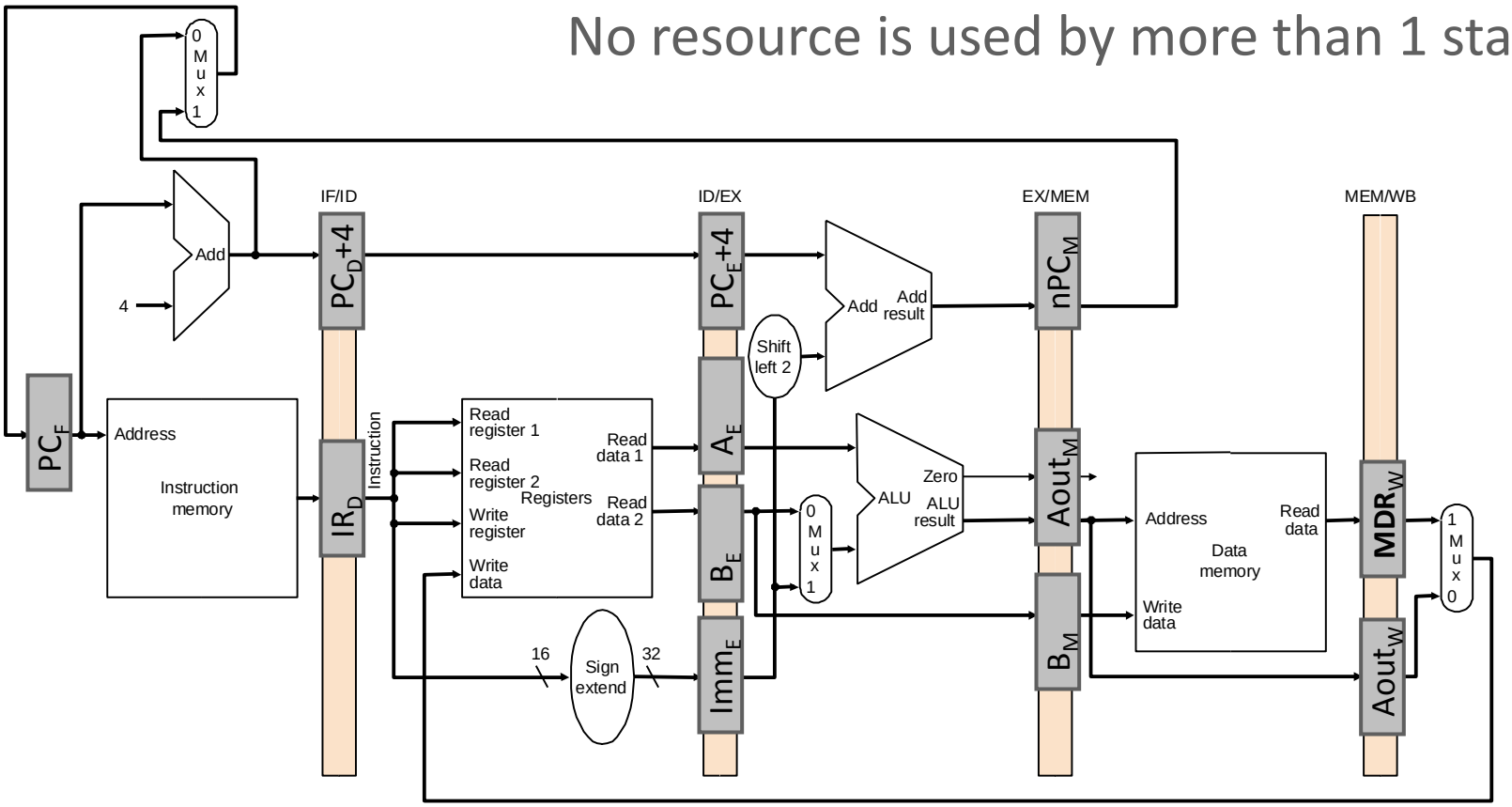
Instruction Pipeline Throughput

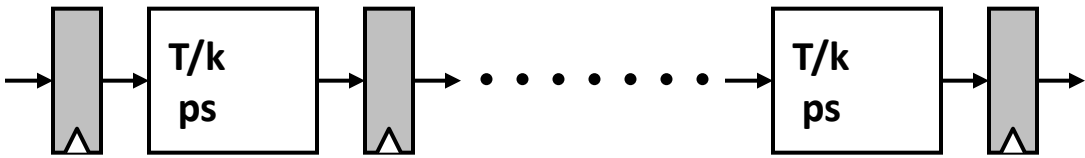


5-stage speedup is 4, not 5 as predicted by the ideal model. Why?

Enabling Pipelined Processing: Pipeline Registers

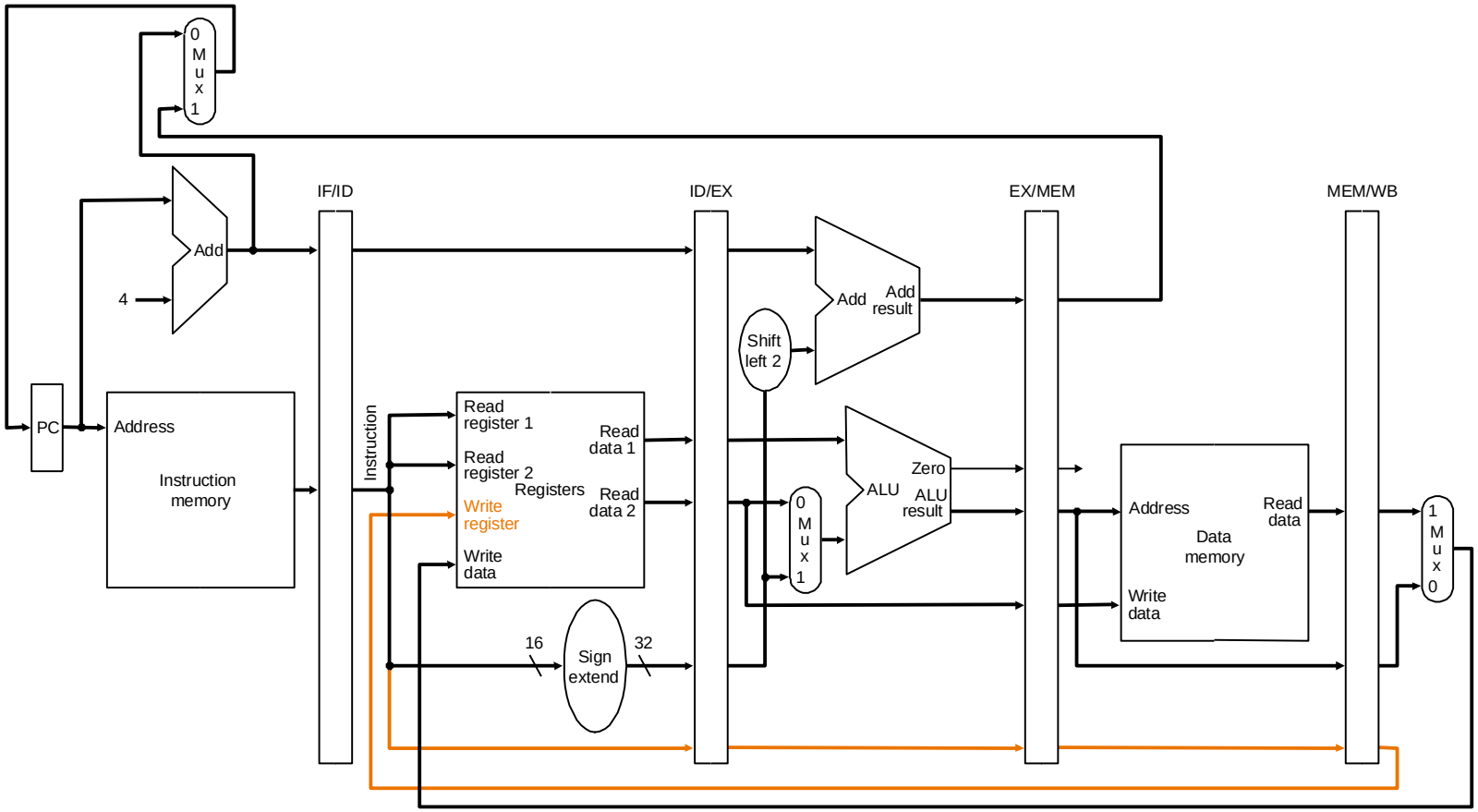
No resource is used by more than 1 stage!



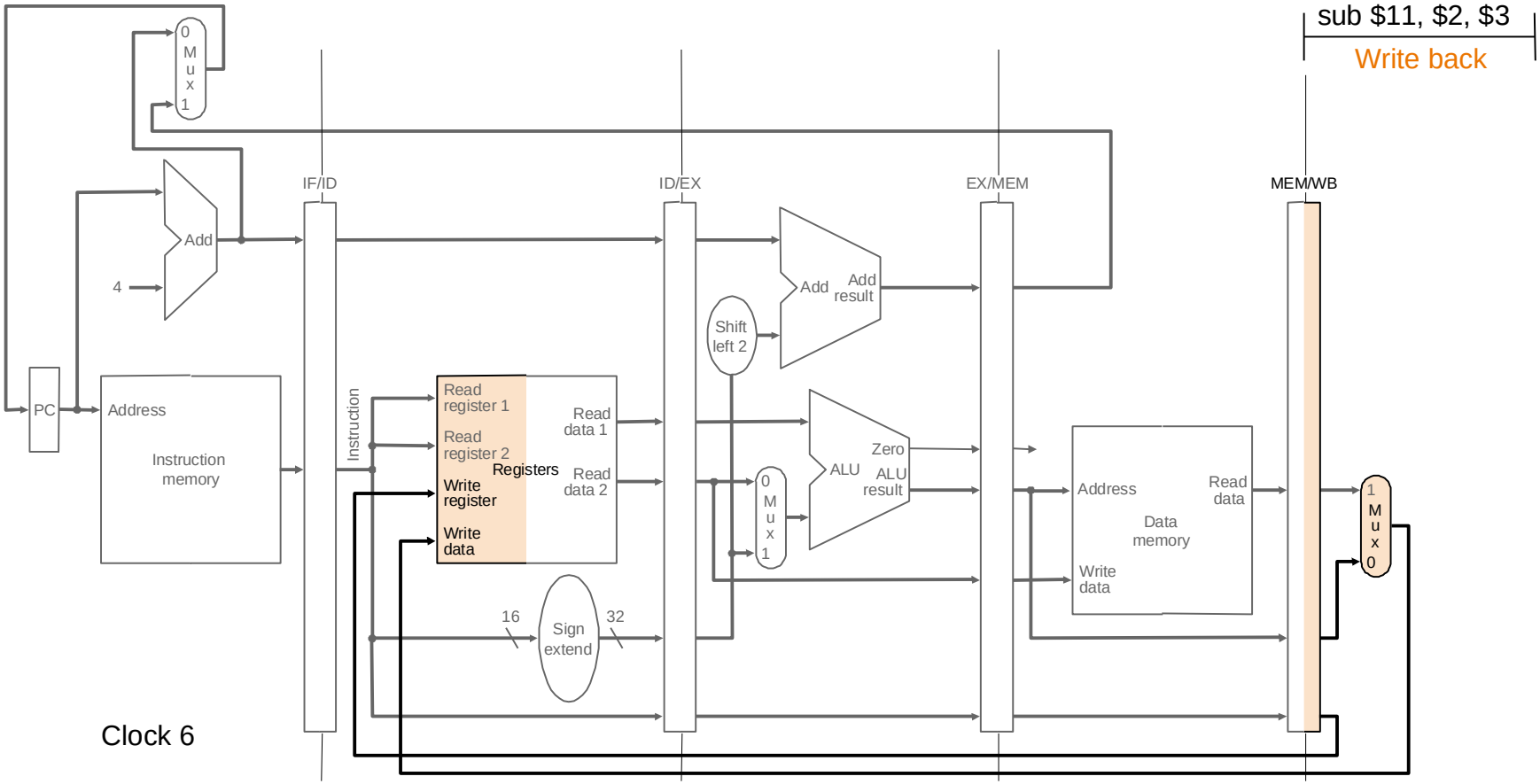


Pipelined Operation Example

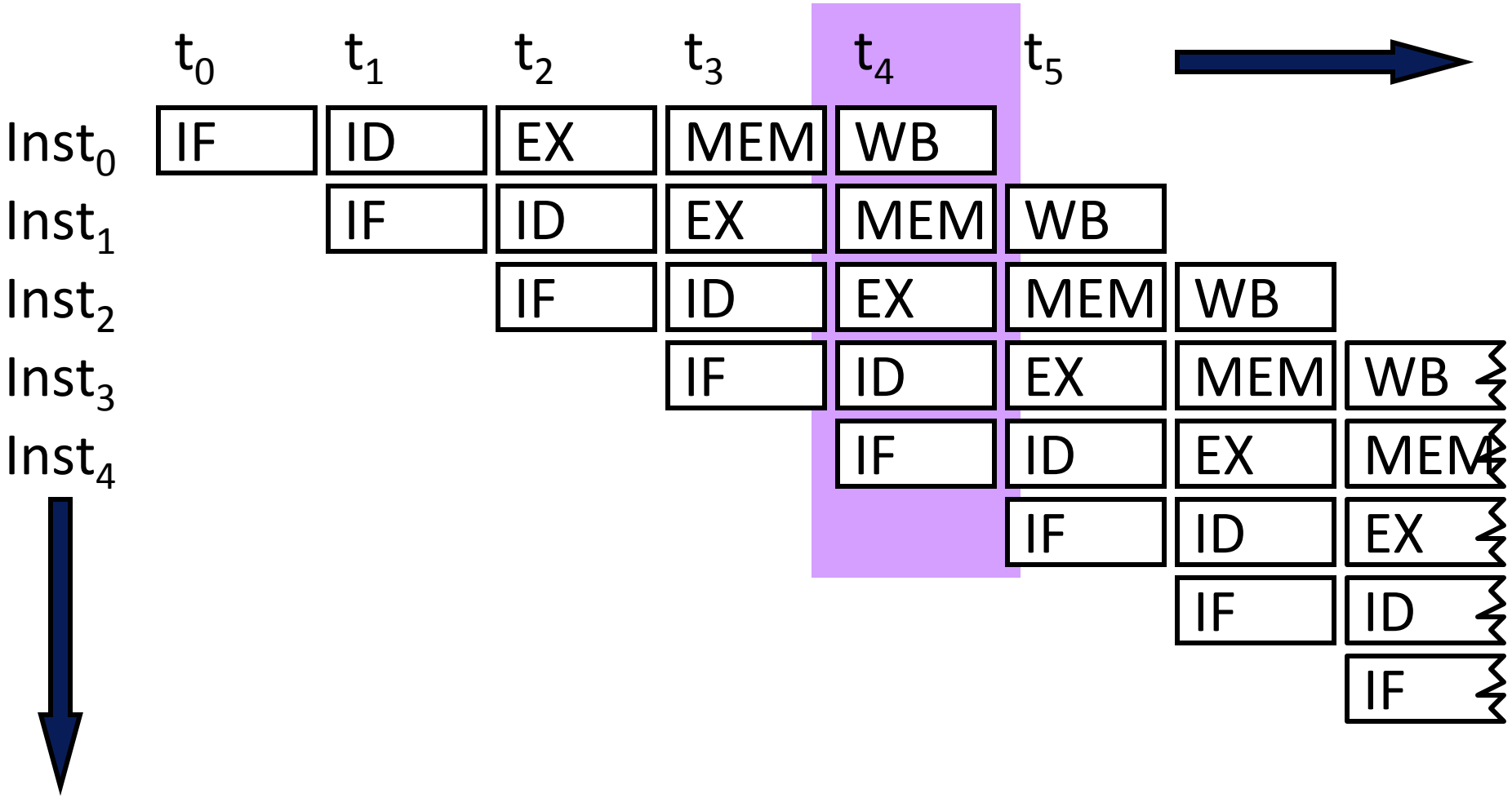
All instruction classes must follow the same path and timing through the pipeline stages. Any performance impact?



Pipelined Operation Example



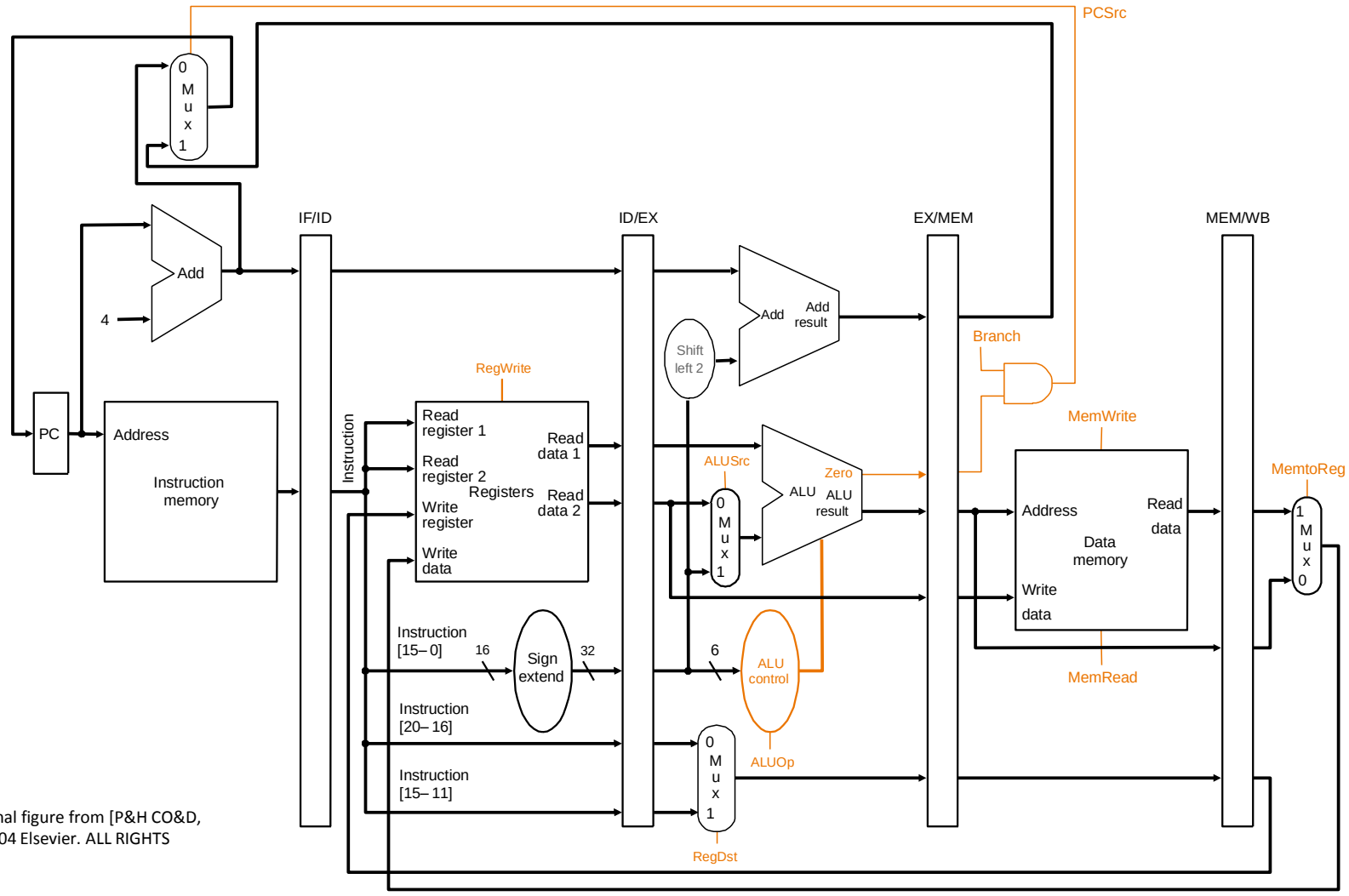
Illustrating Pipeline Operation: Operation View



Illustrating Pipeline Operation: Resource View

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}
ID		I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
EX			I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
MEM				I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
WB					I_0	I_1	I_2	I_3	I_4	I_5	I_6

Control Points in a Pipeline



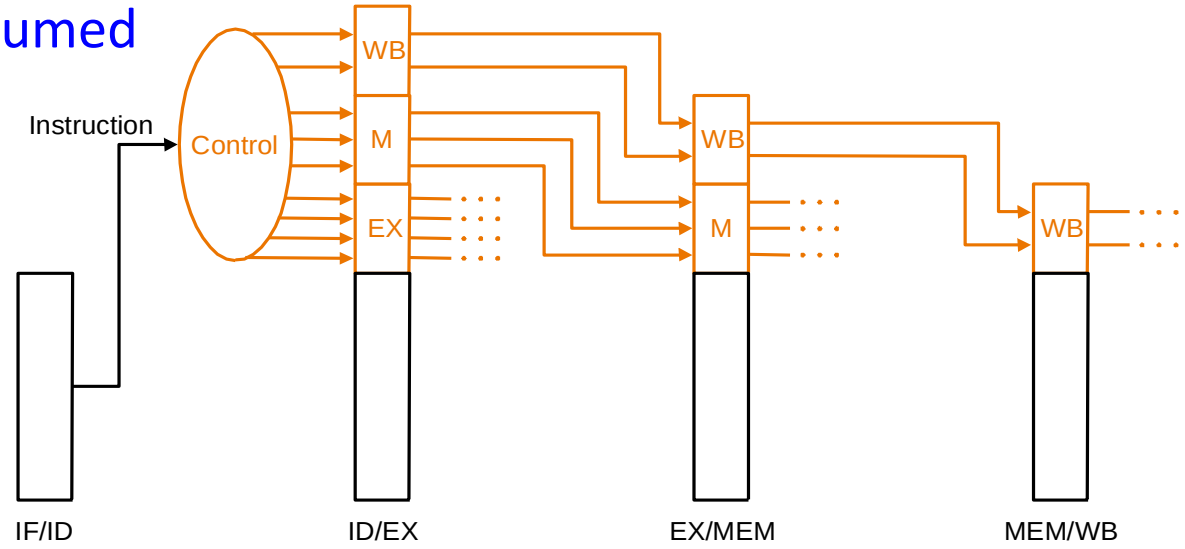
Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Identical set of control points as the single-cycle datapath!!

Control Signals in a Pipeline

- For a given instruction
  - same control signals as single-cycle, but
  - control signals required at different cycles, depending on stage

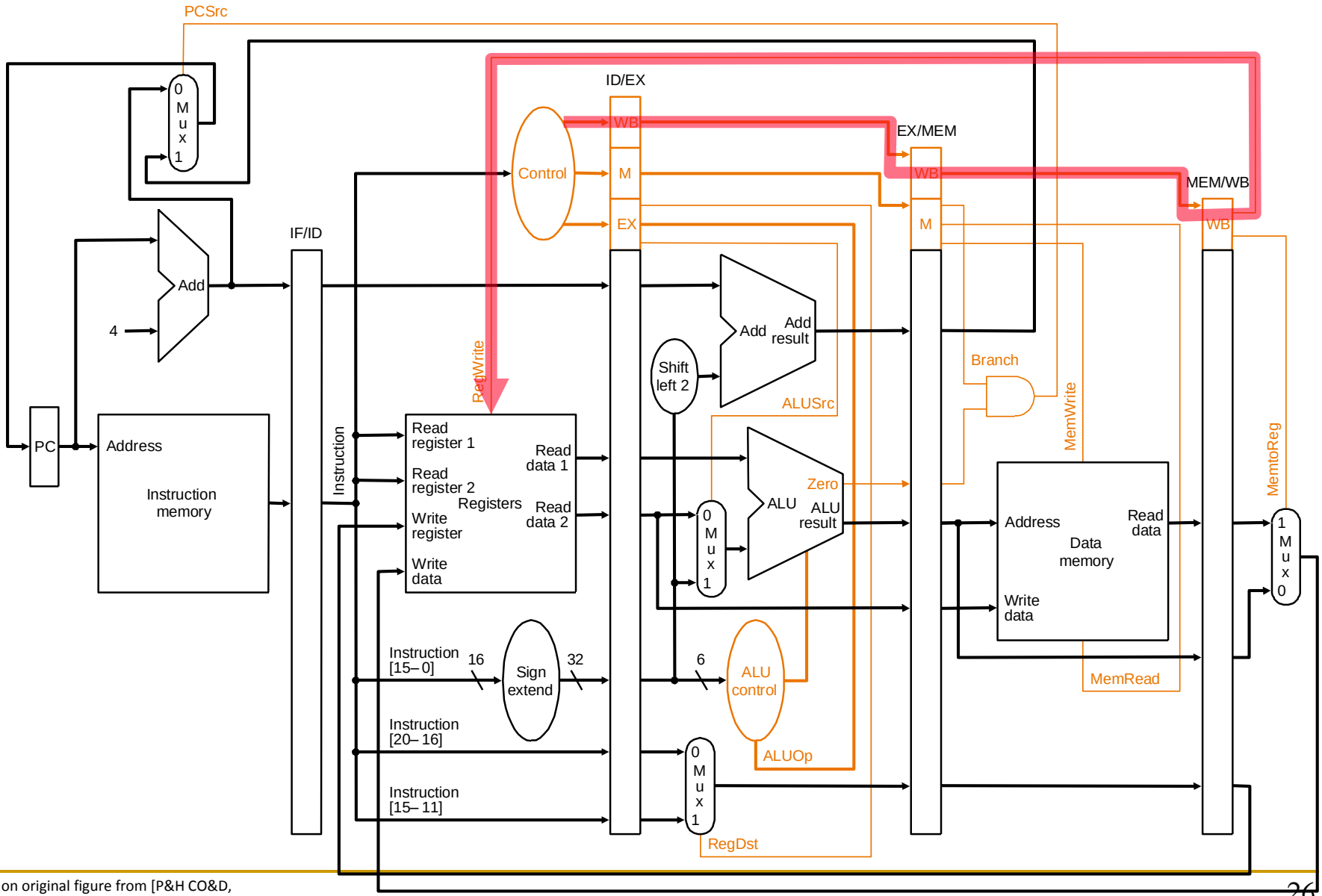
⇒ decode once using the same logic as single-cycle and buffer control signals until consumed



⇒ or carry relevant "instruction word/field" down the pipeline and decode locally within each or in a previous stage

Which one is better?

Pipelined Control Signals



An Ideal Pipeline

- Goal: Increase throughput with little increase in cost (hardware cost, in case of instruction processing)
- Repetition of identical operations
  - The same operation is repeated on a large number of different inputs
- Repetition of independent operations
  - No dependencies between repeated operations
- Uniformly partitionable suboperations
  - Processing an be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry
  - What about the instruction processing "cycle"?

Instruction Pipeline: Not An Ideal Pipeline

- Identical operations ... NOT!

  ⇒ different instructions do not need all stages

  - Forcing different instructions to go through the same multi-function pipe

  → external fragmentation (some pipe stages idle for some instructions)

- Uniform suboperations ... NOT!

  ⇒ difficult to balance the different pipeline stages

  - Not all pipeline stages do the same amount of work

  → internal fragmentation (some pipe stages are too-fast but take the same clock cycle time)

- Independent operations ... NOT!

  ⇒ instructions are not independent of each other

  - Need to detect and resolve inter-instruction dependencies to ensure the pipeline operates correctly

  → Pipeline is not always moving (it stalls)

Issues in Pipeline Design

- Balancing work in pipeline stages
  - How many stages and what is done in each stage
- Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow
  - Handling dependences
    - Data
    - Control
  - Handling resource contention
  - Handling long-latency (multi-cycle) operations
- Handling exceptions, interrupts
- Advanced: Improving pipeline throughput
  - Minimizing stalls

Causes of Pipeline *Stalls*

- Resource contention
- Dependences (between instructions)
  - Data
  - Control
- Long-latency (multi-cycle) operations

Dependences and Their Types

- Also called "dependency" or *less desirably* "hazard"
- Dependencies dictate ordering requirements between instructions
- Two types
  - Data dependence
  - Control dependence
- Resource contention is sometimes called resource dependence
  - However, this is not fundamental to (dictated by) program semantics, so we will treat it separately

Handling Resource Contention

- Happens when instructions in two pipeline stages need the same resource

- Solution 1: Eliminate the cause of contention
  - Duplicate the resource or increase its throughput
    - E.g., use separate instruction and data memories (caches)
    - E.g., use multiple ports for memory structures

- Solution 2: Detect the resource contention and stall one of the contending stages
  - Which stage do you stall?
  - Example: What if you had a single read and write port for the register file?

Data Dependences

- Types of data dependences
  - Flow dependence (true data dependence – read after write)
  - Output dependence (write after write)
  - Anti dependence (write after read)

- Which ones cause stalls in a pipelined machine?
  - For all of them, we need to ensure semantics of the program are correct
  - Flow dependences always need to be obeyed because they constitute true dependence on a value
  - Anti and output dependences exist due to limited number of architectural registers
    - They are dependence on a name, not a value
    - We will later see what we can do about them

Data Dependence Types

Flow dependence

$r_3 \leftarrow r_1$ op r_2 Read-after-Write

$r_5 \leftarrow r_3$ op r_4 (RAW)

Anti dependence

$r_3 \leftarrow r_1$ op r_2 Write-after-Read

$r_1 \leftarrow r_4$ op r_5 (WAR)

Output-dependence

$r_3 \leftarrow r_1$ op r_2 Write-after-Write

$r_5 \leftarrow r_3$ op r_4 (WAW)

$r_3 \leftarrow r_6$ op r_7

How to Handle Data Dependences

- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order
- Flow dependences are more interesting
- Five fundamental ways of handling flow dependences

Readings for Next Few Lectures

- P&H Chapter 4.9-4.11

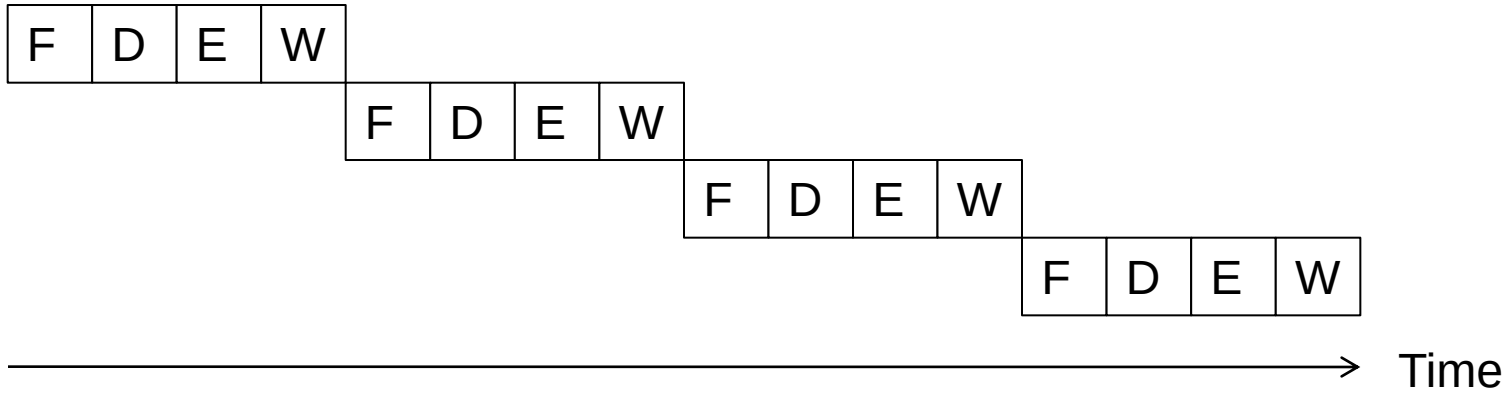
- Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, 1995
  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts

Review: Pipelining: Basic Idea

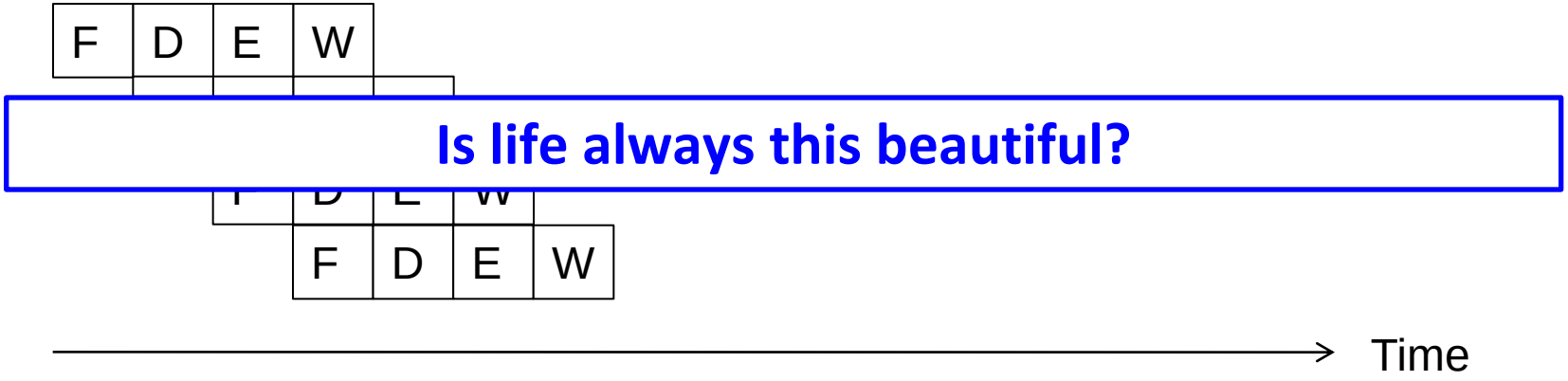
- Idea:
  - Divide the instruction processing cycle into distinct "stages" of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a different instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages
- Benefit: Increases instruction processing throughput (1/CPI)
- Downside: ???

Review: Execution of Four Independent ADDs

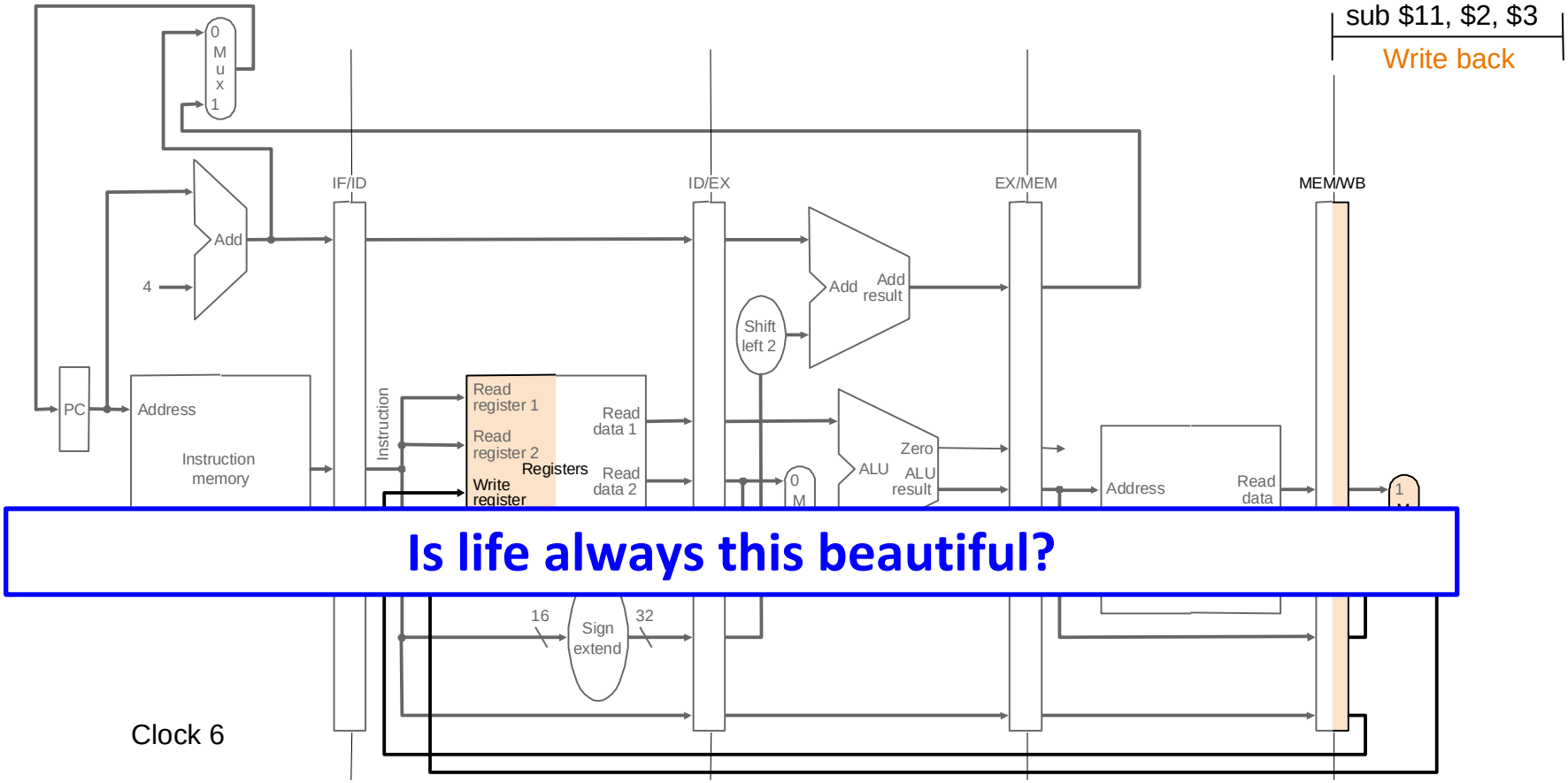
- Multi-cycle: 4 cycles per instruction



- Pipelined: 4 cycles per 4 instructions (steady state)



Review: Pipelined Operation Example



Is life always this beautiful?

Review: Instruction Pipeline: Not An Ideal Pipeline

- Identical operations ... NOT!

  ⇒ different instructions do not need all stages

  - Forcing different instructions to go through the same multi-function pipe

  → external fragmentation (some pipe stages idle for some instructions)

- Uniform suboperations ... NOT!

  ⇒ difficult to balance the different pipeline stages

  - Not all pipeline stages do the same amount of work

  → internal fragmentation (some pipe stages are too-fast but take the same clock cycle time)

- Independent operations ... NOT!

  ⇒ instructions are not independent of each other

  - Need to detect and resolve inter-instruction dependencies to ensure the pipeline operates correctly

  → Pipeline is not always moving (it stalls)

Review: Fundamental Issues in Pipeline Design

- Balancing work in pipeline stages
  - How many stages and what is done in each stage
- Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow
  - Handling dependences
    - Data
    - Control
  - Handling resource contention
  - Handling long-latency (multi-cycle) operations
- Handling exceptions, interrupts
- Advanced: Improving pipeline throughput
  - Minimizing stalls

Review: Data Dependences

- Types of data dependences
  - Flow dependence (true data dependence – read after write)
  - Output dependence (write after write)
  - Anti dependence (write after read)

- Which ones cause stalls in a pipelined machine?
  - For all of them, we need to ensure semantics of the program is correct
  - Flow dependences always need to be obeyed because they constitute true dependence on a value
  - Anti and output dependences exist due to limited number of architectural registers
    - They are dependence on a name, not a value
    - We will later see what we can do about them

Data Dependence Types

Flow dependence

$r_3 \leftarrow r_1$ op r_2 Read-after-Write

$r_5 \leftarrow r_3$ op r_4 (RAW)

Anti dependence

$r_3 \leftarrow r_1$ op r_2 Write-after-Read

$r_1 \leftarrow r_4$ op r_5 (WAR)

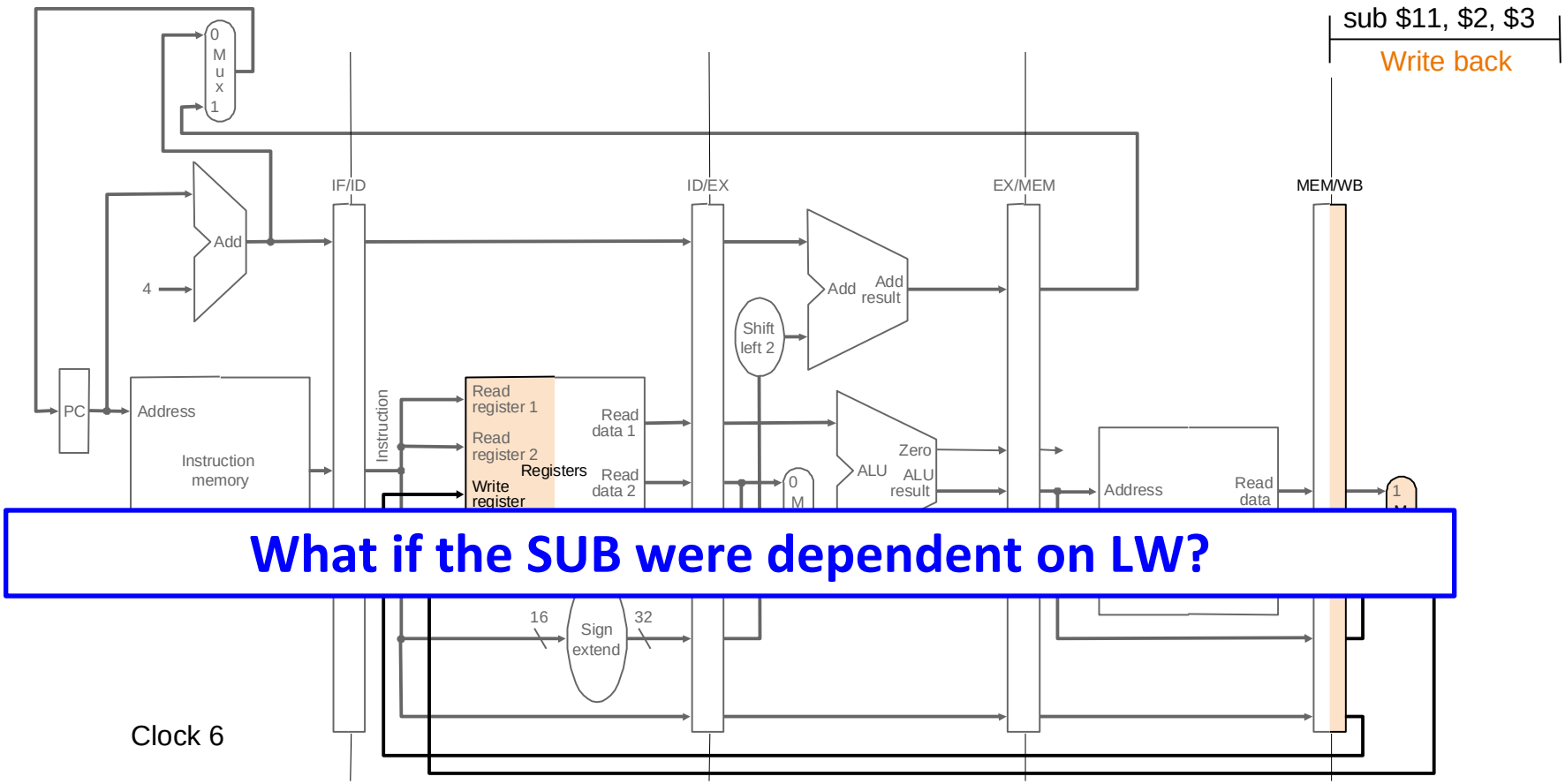
Output-dependence

$r_3 \leftarrow r_1$ op r_2 Write-after-Write

$r_5 \leftarrow r_3$ op r_4 (WAW)

$r_3 \leftarrow r_6$ op r_7

Pipelined Operation Example



What if the SUB were dependent on LW?

How to Handle Data Dependences

- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order
- Flow dependences are more interesting
- Five fundamental ways of handling flow dependences
  - Detect and wait until value is available in register file
  - Detect and forward/bypass data to dependent instruction
  - Detect and eliminate the dependence at the software level
    - No need for the hardware to detect dependence
  - Predict the needed value(s), execute "speculatively", and verify
  - Do something else (fine-grained multithreading)
    - No need to detect

Interlocking

- Detection of dependence between instructions in a pipelined processor to guarantee correct execution

- Software based interlocking

  vs.

- Hardware based interlocking

- MIPS acronym?

Approaches to Dependence Detection (I)

- Scoreboarding
  - Each register in register file has a Valid bit associated with it
  - An instruction that is writing to the register resets the Valid bit
  - An instruction in Decode stage checks if all its source and destination registers are Valid
    - Yes: No need to stall… No dependence
    - No: Stall the instruction

- Advantage:
  - Simple. 1 bit per register

- Disadvantage:
  - Need to stall for all types of dependences, not only flow dep.

Not Stalling on Anti and Output Dependences

- What changes would you make to the scoreboard to enable this?

Approaches to Dependence Detection (II)

- Combinational dependence check logic
  - Special logic that checks if any instruction in later stages is supposed to write to any source register of the instruction that is being decoded
  - Yes: stall the instruction/pipeline
  - No: no need to stall… no flow dependence

- Advantage:
  - No need to stall on anti and output dependences

- Disadvantage:
  - Logic is more complex than a scoreboard
  - Logic becomes more complex as we make the pipeline deeper and wider (flash-forward: think superscalar execution)

Once You Detect the Dependence in Hardware

- What do you do afterwards?

- Observation: Dependence between two instructions is detected before the communicated data value becomes available

- Option 1: Stall the dependent instruction right away
- Option 2: Stall the dependent instruction only when necessary → data forwarding/bypassing
- Option 3: …

Data Forwarding/Bypassing

- Problem: A consumer (dependent) instruction has to wait in decode stage until the producer instruction writes its value in the register file
- Goal: We do not want to stall the pipeline unnecessarily
- Observation: The data value needed by the consumer instruction can be supplied directly from a later stage in the pipeline (instead of only from the register file)
- Idea: Add additional dependence check logic and data forwarding paths (buses) to supply the producer's value to the consumer right after the value is available
- Benefit: Consumer can move in the pipeline until the point the value can be supplied → less stalling

A Special Case of Data Dependence

- Control dependence
  - Data dependence on the Instruction Pointer / Program Counter

Control Dependence

- Question: What should the fetch PC be in the next cycle?
- Answer: The address of the next instruction
  - All instructions are control dependent on previous ones. Why?

- If the fetched instruction is a non-control-flow instruction:
  - Next Fetch PC is the address of the next-sequential instruction
  - Easy to determine if we know the size of the fetched instruction

- If the instruction that is fetched is a control-flow instruction:
  - How do we determine the next Fetch PC?

- In fact, how do we know whether or not the fetched instruction is a control-flow instruction?

Data Dependence Handling:
More Depth & Implementation

Remember: Data Dependence Types

Flow dependence

$r_3 \leftarrow r_1$ op r_2 Read-after-Write

$r_5 \leftarrow r_3$ op r_4 (RAW)

Anti dependence

$r_3 \leftarrow r_1$ op r_2 Write-after-Read

$r_1 \leftarrow r_4$ op r_5 (WAR)

Output-dependence

$r_3 \leftarrow r_1$ op r_2 Write-after-Write

$r_5 \leftarrow r_3$ op r_4 (WAW)

$r_3 \leftarrow r_6$ op r_7

How to Handle Data Dependences

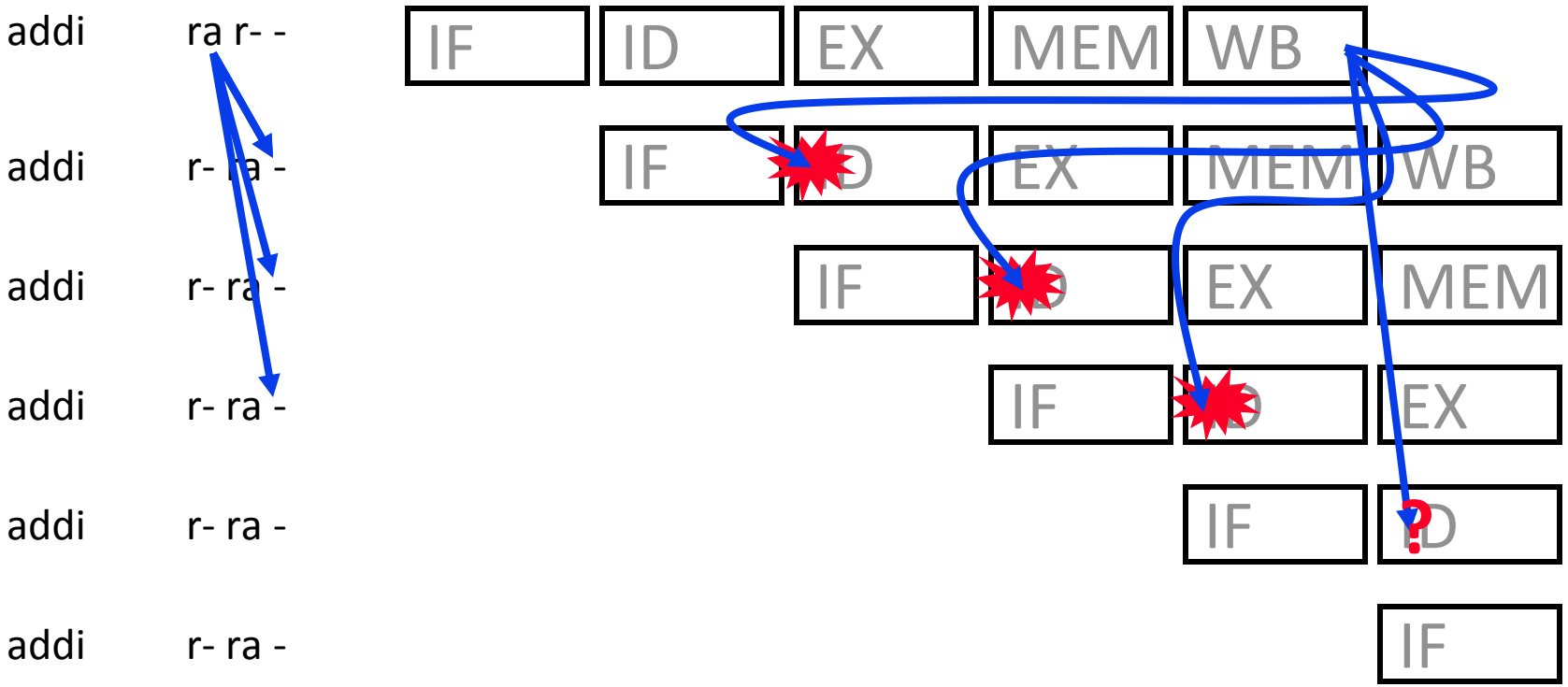
- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order

- Flow dependences are more interesting

- Five fundamental ways of handling flow dependences
  - Detect and wait until value is available in register file
  - Detect and forward/bypass data to dependent instruction
  - Detect and eliminate the dependence at the software level
    - No need for the hardware to detect dependence
  - Predict the needed value(s), execute "speculatively", and verify
  - Do something else (fine-grained multithreading)
    - No need to detect

RAW Dependence Handling

- Following flow dependences lead to conflicts in the 5-stage pipeline

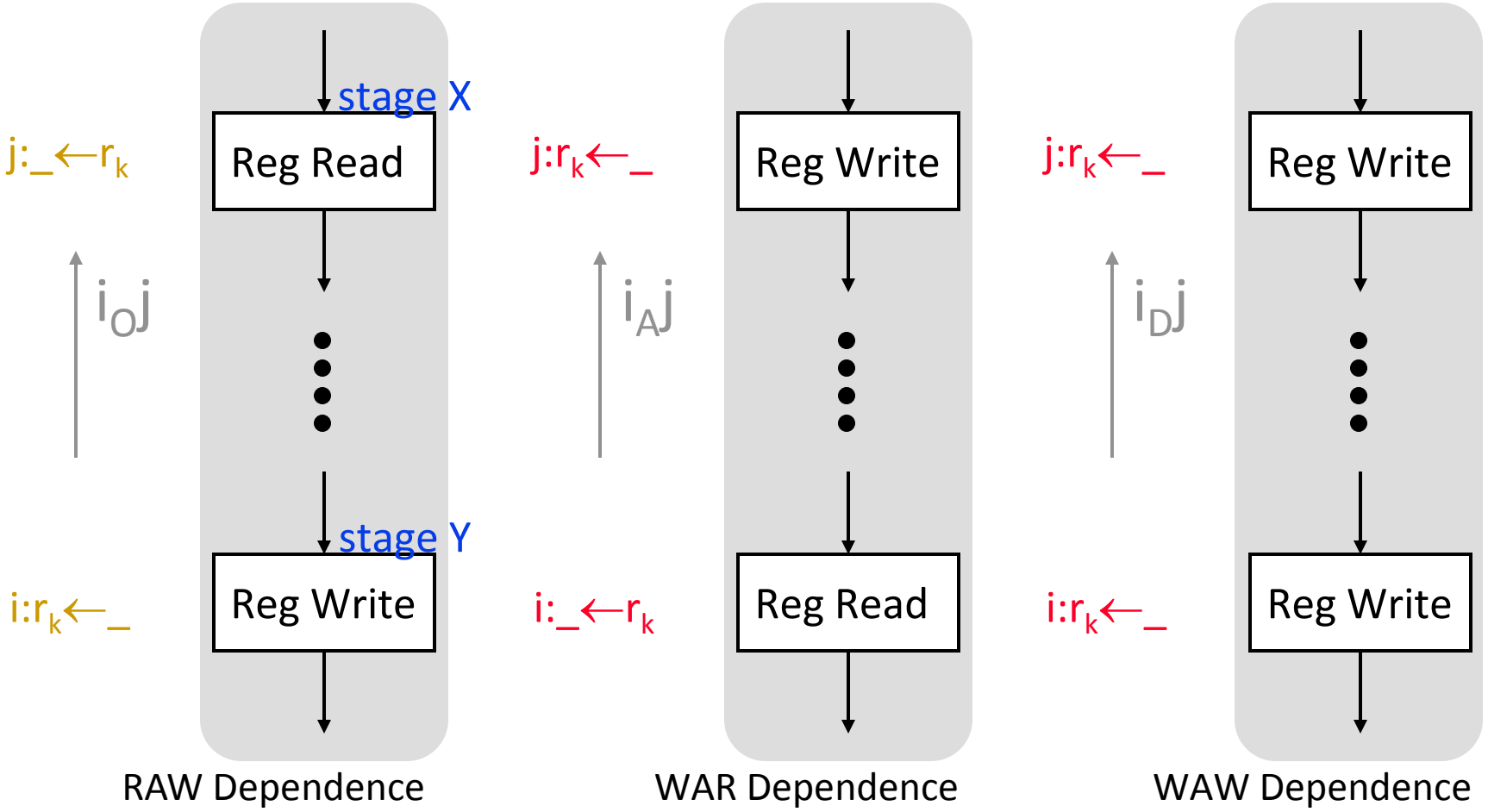


Register Data Dependence Analysis

	R/I-Type	LW	SW	Br	J	Jr
IF						
ID	read RF	read RF	read RF	read RF		read RF
EX						
MEM						
WB	write RF	write RF				

- For a given pipeline, when is there a potential conflict between 2 data dependent instructions?
  - dependence type: RAW, WAR, WAW?
  - instruction types involved?
  - distance between the two instructions?

Safe and Unsafe Movement of Pipeline



dist(i,j) $\leq$ dist(X,Y) $\Rightarrow$ Unsafe to keep j moving

dist(i,j) > dist(X,Y) $\Rightarrow$ Safe

RAW Dependence Analysis Example

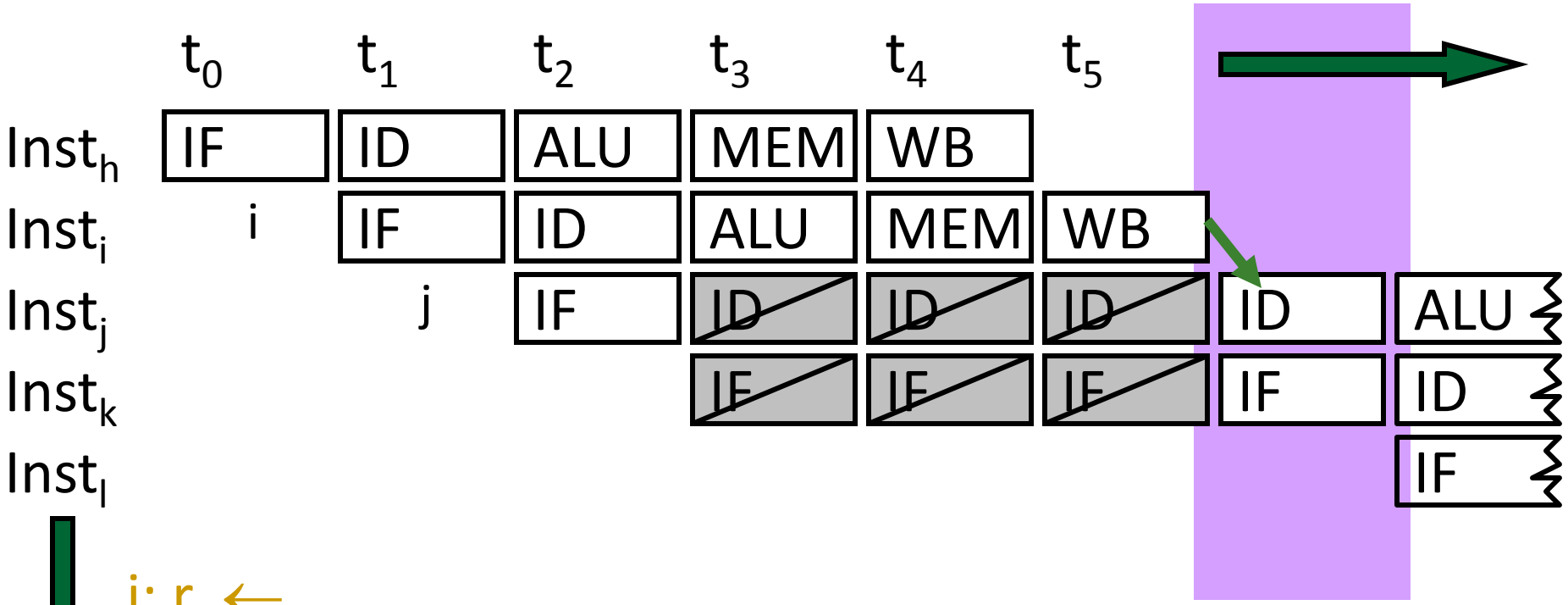
	R/I-Type	LW	SW	Br	J	Jr
IF						
ID	read RF	read RF	read RF	read RF		read RF
EX						
MEM						
WB	write RF	write RF				

- Instructions I_A and I_B (where I_A comes before I_B) have RAW dependence iff
  - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
  - $dist(I_A, I_B) \leq dist(ID, WB) = 3$

What about WAW and WAR dependence?

What about memory data dependence?

Pipeline Stall: Resolving Data Dependence



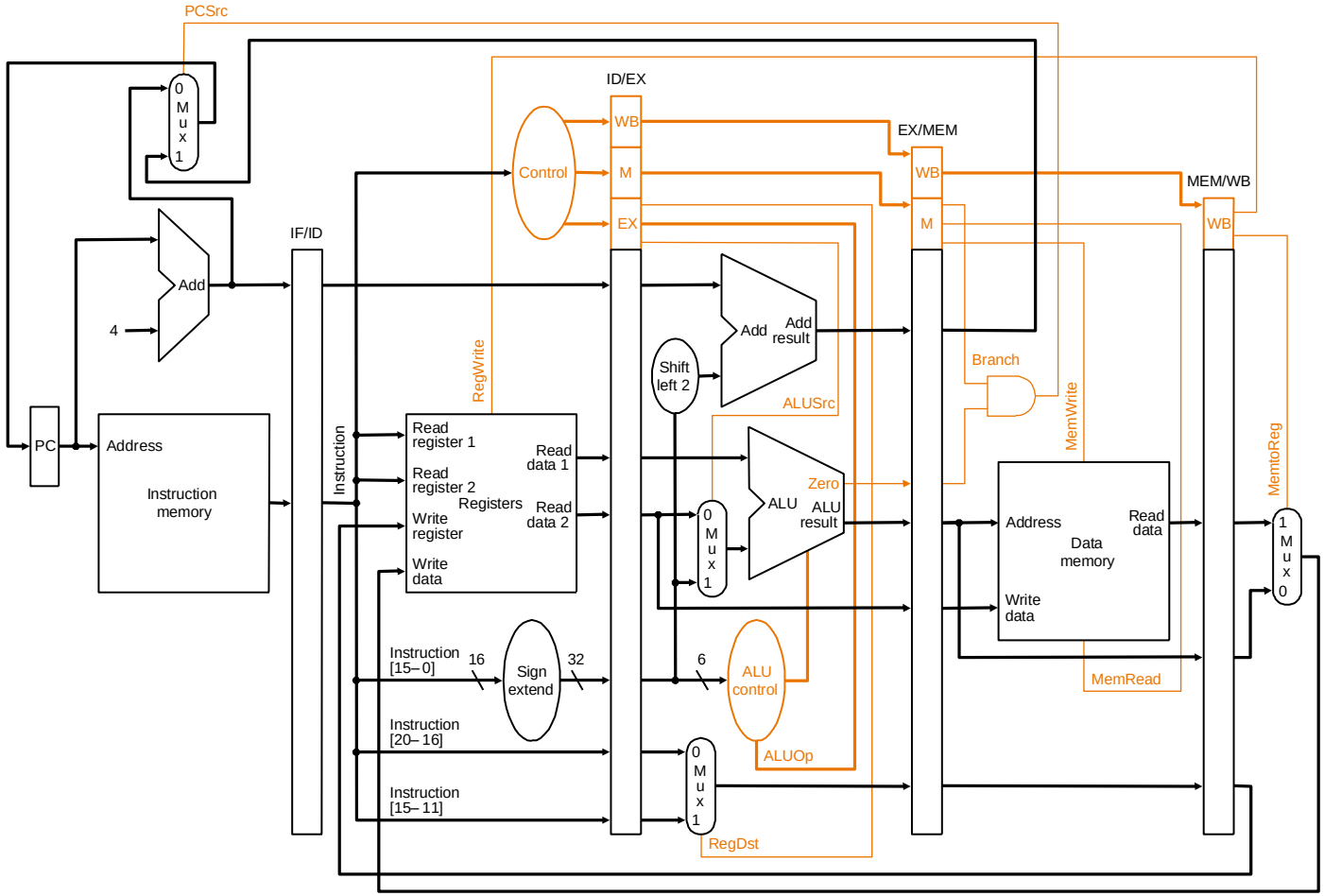
i: $r_x \leftarrow \_$
bubble
bubble
bubble
j: $\_ \leftarrow r_x$

dist(i,j)=4

Stall==make the dependent instruction wait until its source data value is available

1. stop all up-stream stages
2. drain all down-stream stages

How to Implement Stalling



- Stall
  - disable **PC** and **IR** latching; ensure stalled instruction stays in its stage
  - Insert “invalid” instructions/nops into the stage following the stalled one

Stall Conditions

- Instructions I_A and I_B (where I_A comes before I_B) have RAW dependence iff
  - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
  - $dist(I_A, I_B) \leq dist(ID, WB) = 3$

- In other words, must stall when I_B in ID stage wants to read a register to be written by I_A in EX, MEM or WB stage

Stall Conditions

- Helper functions
  - rs(I) returns the rs field of I
  - use_rs(I) returns true if I requires **RF**[rs] and rs!=r0
- Stall when
  - (rs(IR_{ID})==$dest_{EX}$) && use_rs(IR_{ID}) && $RegWrite_{EX}$ or
  - (rs(IR_{ID})==$dest_{MEM}$) && use_rs(IR_{ID}) && $RegWrite_{MEM}$ or
  - (rs(IR_{ID})==$dest_{WB}$) && use_rs(IR_{ID}) && $RegWrite_{WB}$ or
  - (rt(IR_{ID})==$dest_{EX}$) && use_rt(IR_{ID}) && $RegWrite_{EX}$ or
  - (rt(IR_{ID})==$dest_{MEM}$) && use_rt(IR_{ID}) && $RegWrite_{MEM}$ or
  - (rt(IR_{ID})==$dest_{WB}$) && use_rt(IR_{ID}) && $RegWrite_{WB}$
- It is crucial that the EX, MEM and WB stages continue to advance normally during stall cycles

Impact of Stall on Performance

- Each stall cycle corresponds to 1 lost ALU cycle

- For a program with N instructions and S stall cycles,
  Average CPI=(N+S)/N

- S depends on
  - frequency of RAW dependences
  - exact distance between the dependent instructions
  - distance between dependences

    suppose i_1,i_2 and i_3 all depend on i_0, once i_1's dependence is resolved, i_2 and i_3 must be okay too

Sample Assembly (P&H)

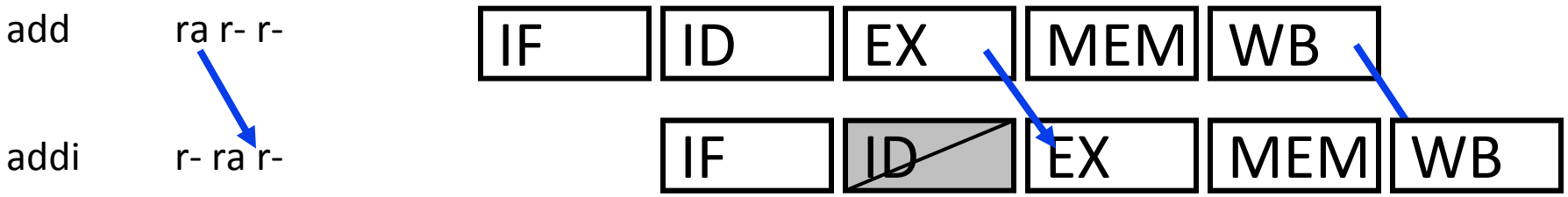
- for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { ..... }

```
                addi    $s1, $s0, -1
for2tst:        slti    $t0, $s1, 0
                bne     $t0, $zero, exit2
                sll     $t1, $s1, 2
                add     $t2, $a0, $t1
                lw      $t3, 0($t2)
                lw      $t4, 4($t2)
                slt     $t0, $t4, $t3
                beq     $t0, $zero, exit2
                .........
                addi    $s1, $s1, -1
                j       for2tst
exit2:
```

3 stalls
3 stalls
3 stalls
3 stalls
3 stalls
3 stalls

Data Forwarding (or Data Bypassing)

- It is intuitive to think of RF as state
  - "add rx ry rz" literally means get values from RF[ry] and RF[rz] respectively and put result in RF[rx]
- But, RF is just a part of a computing abstraction
  - "add rx ry rz" means 1. get the results of the last instructions to define the values of RF[ry] and RF[rz], respectively, and 2. until another instruction redefines RF[rx], younger instructions that refers to RF[rx] should use this instruction's result
- What matters is to maintain the correct "dataflow" between operations, thus



Resolving RAW Dependence with Forwarding

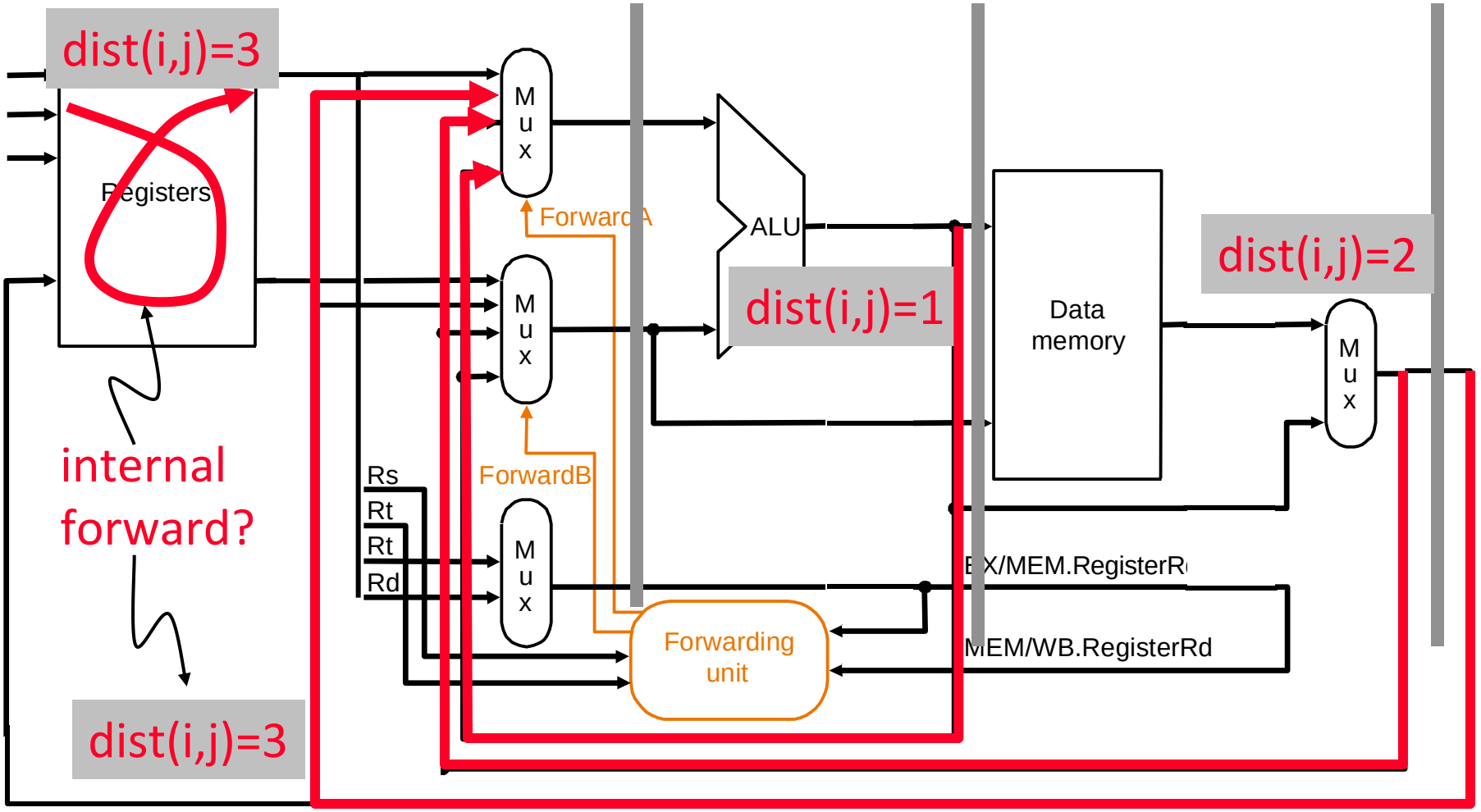
- Instructions I_A and I_B (where I_A comes before I_B) have RAW dependence iff
  - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
  - $dist(I_A, I_B) \leq dist(ID, WB) = 3$

- In other words, if I_B in ID stage reads a register written by I_A in EX, MEM or WB stage, then the operand required by I_B is not yet in RF

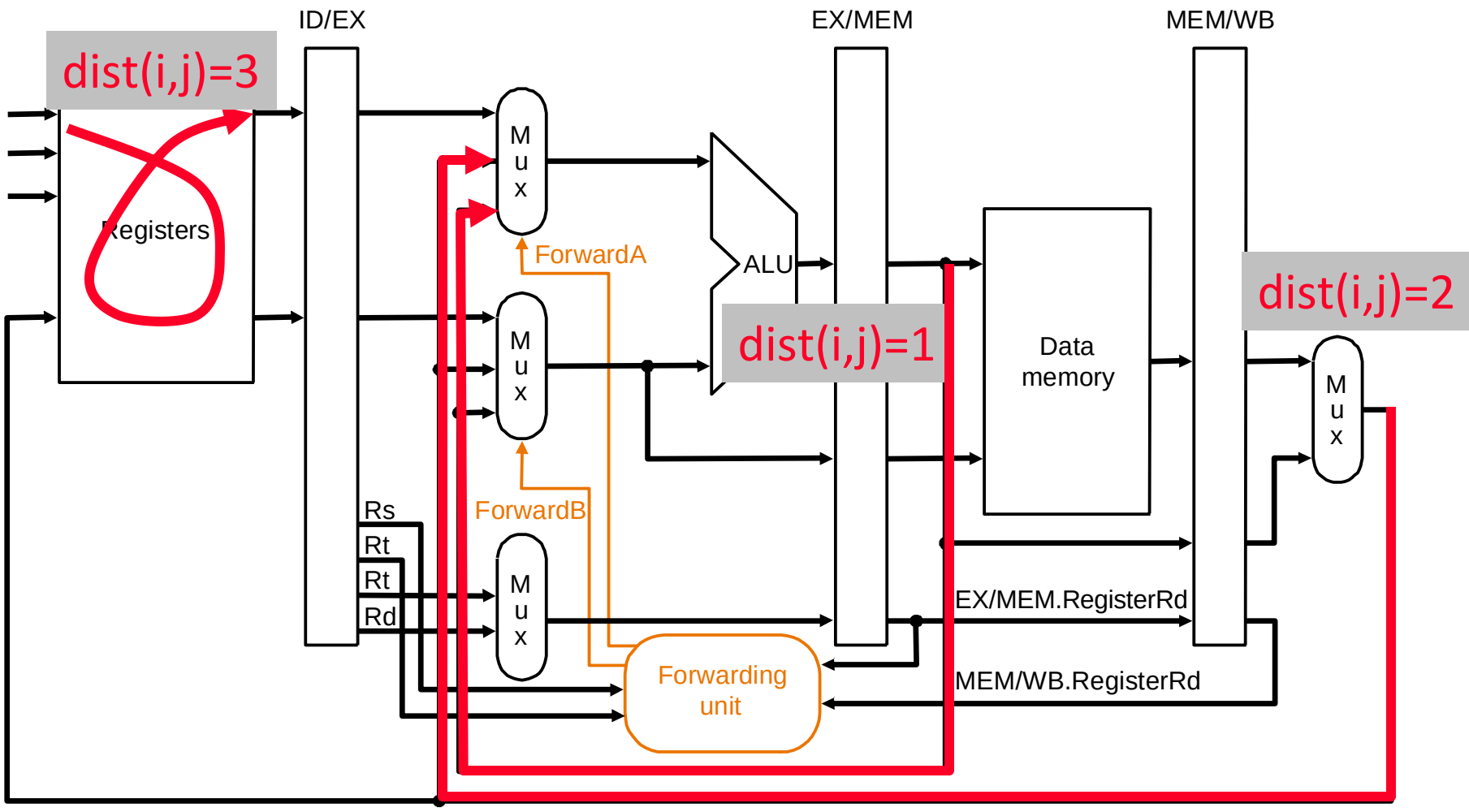
  $\Rightarrow$ retrieve operand from datapath instead of the RF

  $\Rightarrow$ retrieve operand from the youngest definition if multiple definitions are outstanding

Data Forwarding Paths (v1)



Data Forwarding Paths (v2)



b. With forwarding

Assumes RF forwards internally

Data Forwarding Logic (for v2)

if (rs_{EX}!=0) && (rs_{EX}==$dest_{MEM}$) && $RegWrite_{MEM}$ then
    forward operand from MEM stage // dist=1
else if (rs_{EX}!=0) && (rs_{EX}==$dest_{WB}$) && $RegWrite_{WB}$ then
    forward operand from WB stage // dist=2
else
    use A_{EX} (operand from register file) // dist >= 3

Ordering matters!! Must check youngest match first

Why doesn't use_rs( ) appear in the forwarding logic?

What does the above not take into account?

Data Forwarding (Dependence Analysis)

	R/I-Type	LW	SW	Br	J	Jr
IF						
ID						use
EX	use produce	use	use	use		
MEM		produce	(use)			
WB						

- Even with data-forwarding, RAW dependence on an immediately preceding LW instruction requires a stall

Sample Assembly, Revisited (P&H)

- for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { ..... }

```
                    addi    $s1, $s0, -1
for2tst:            slti    $t0, $s1, 0
                    bne     $t0, $zero, exit2
                    sll     $t1, $s1, 2
                    add     $t2, $a0, $t1
                    lw      $t3, 0($t2)
                    lw      $t4, 4($t2)
                    nop
                    slt     $t0, $t4, $t3
                    beq     $t0, $zero, exit2
                    .........
                    addi    $s1, $s1, -1
                    j       for2tst
exit2:
```

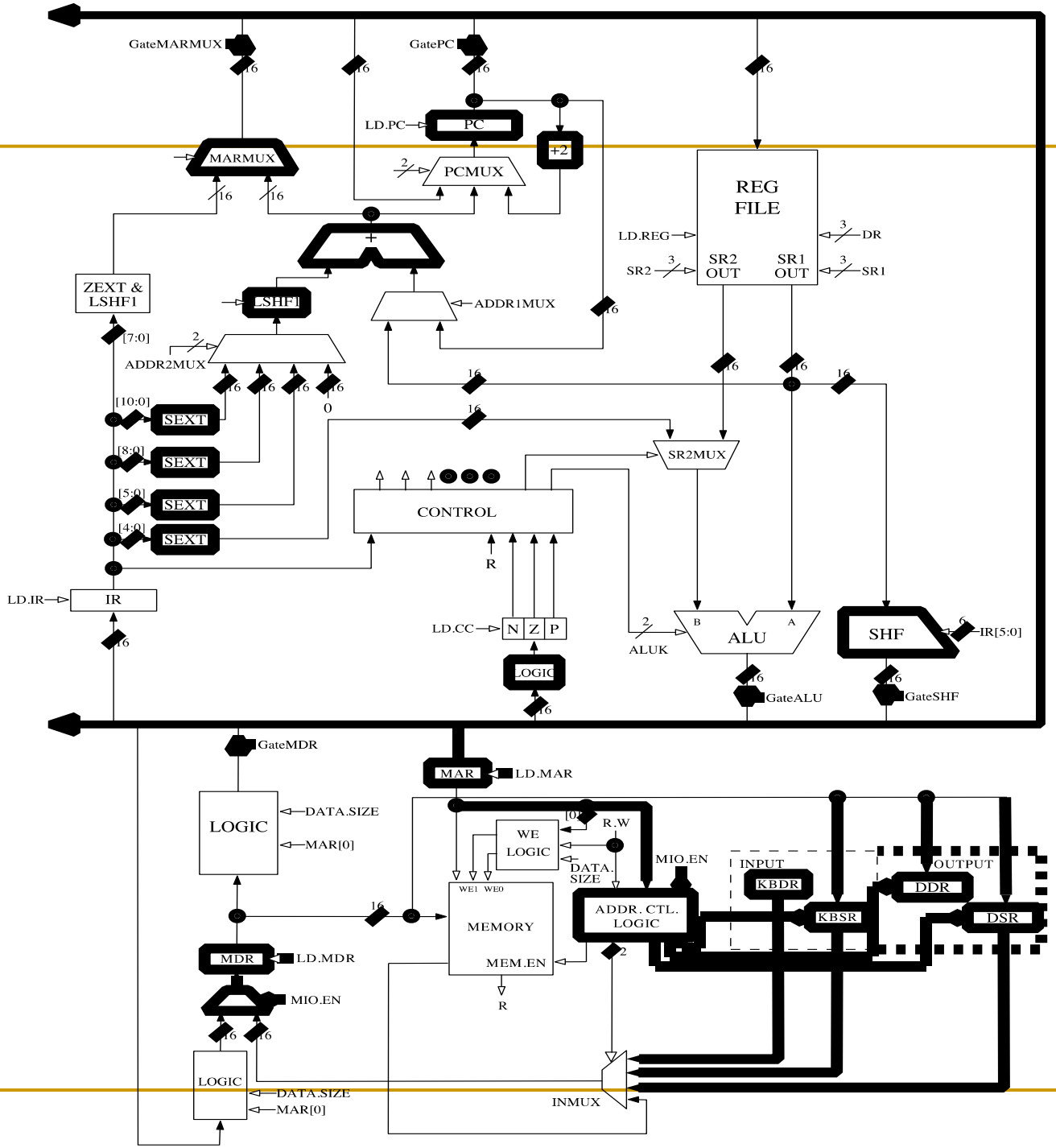
Pipelining the LC-3b

Pipelining the LC-3b

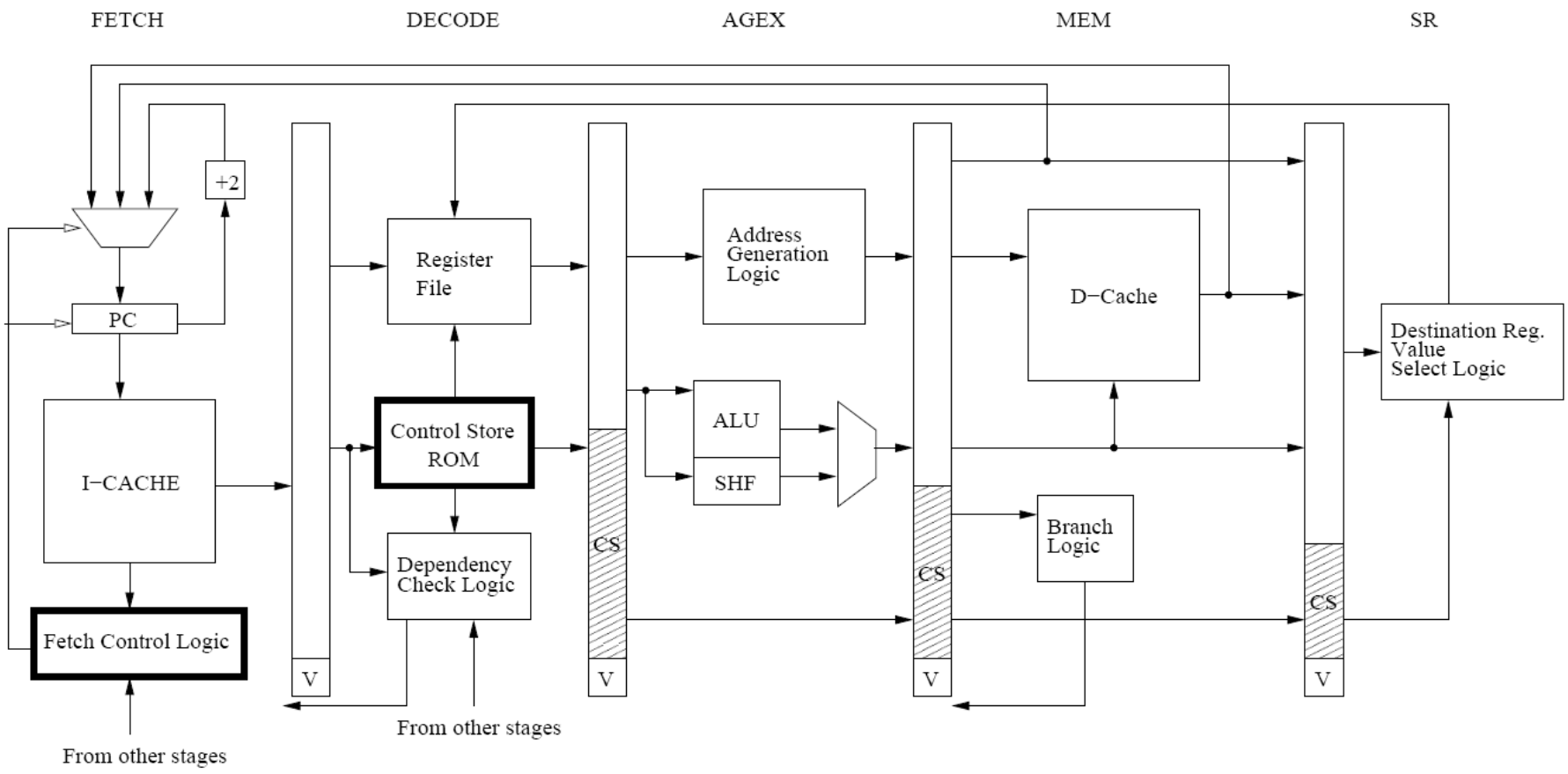
- Let's remember the single-bus datapath

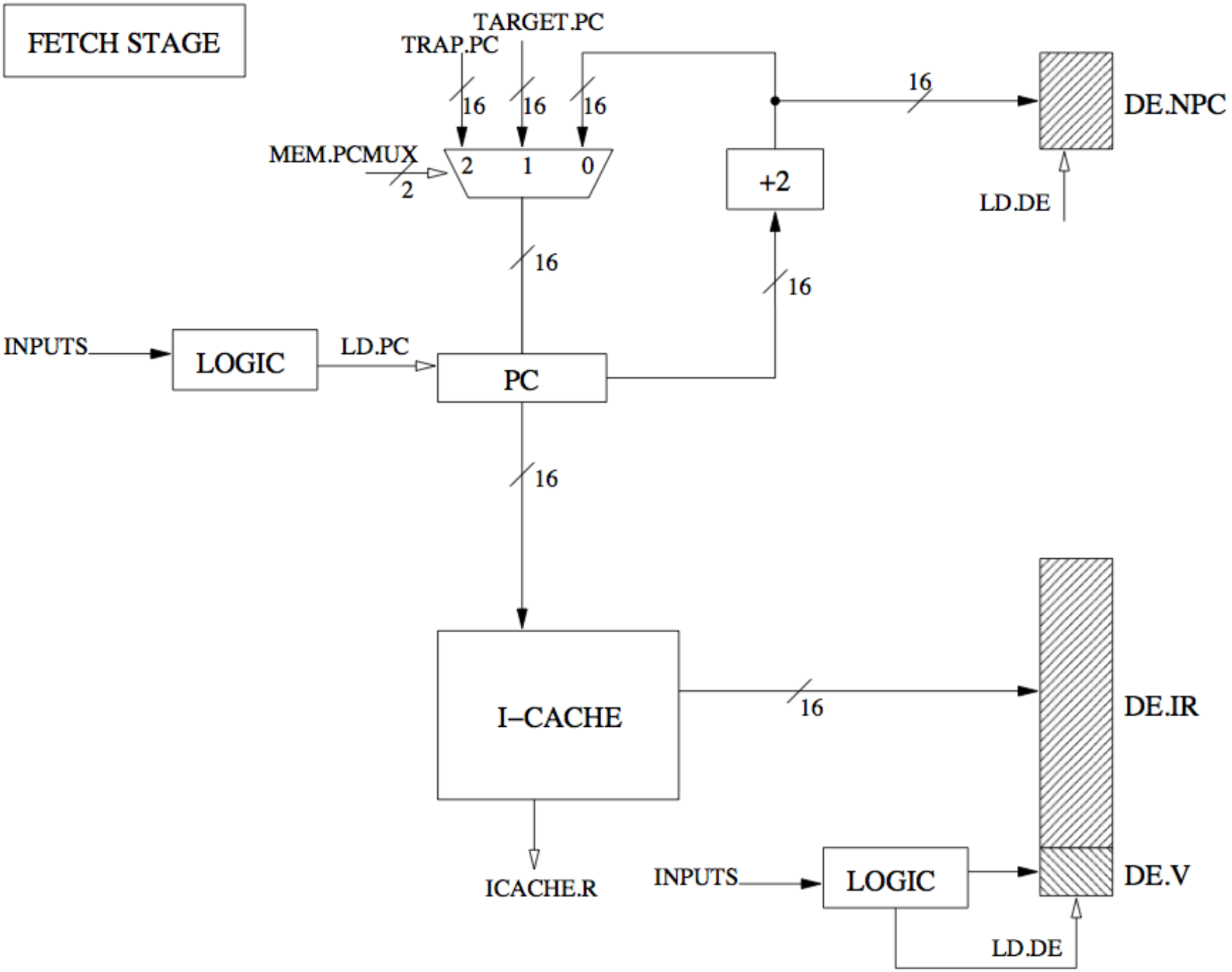
- We'll divide it into 5 stages
  - Fetch
  - Decode/RF Access
  - Address Generation/Execute
  - Memory
  - Store Result

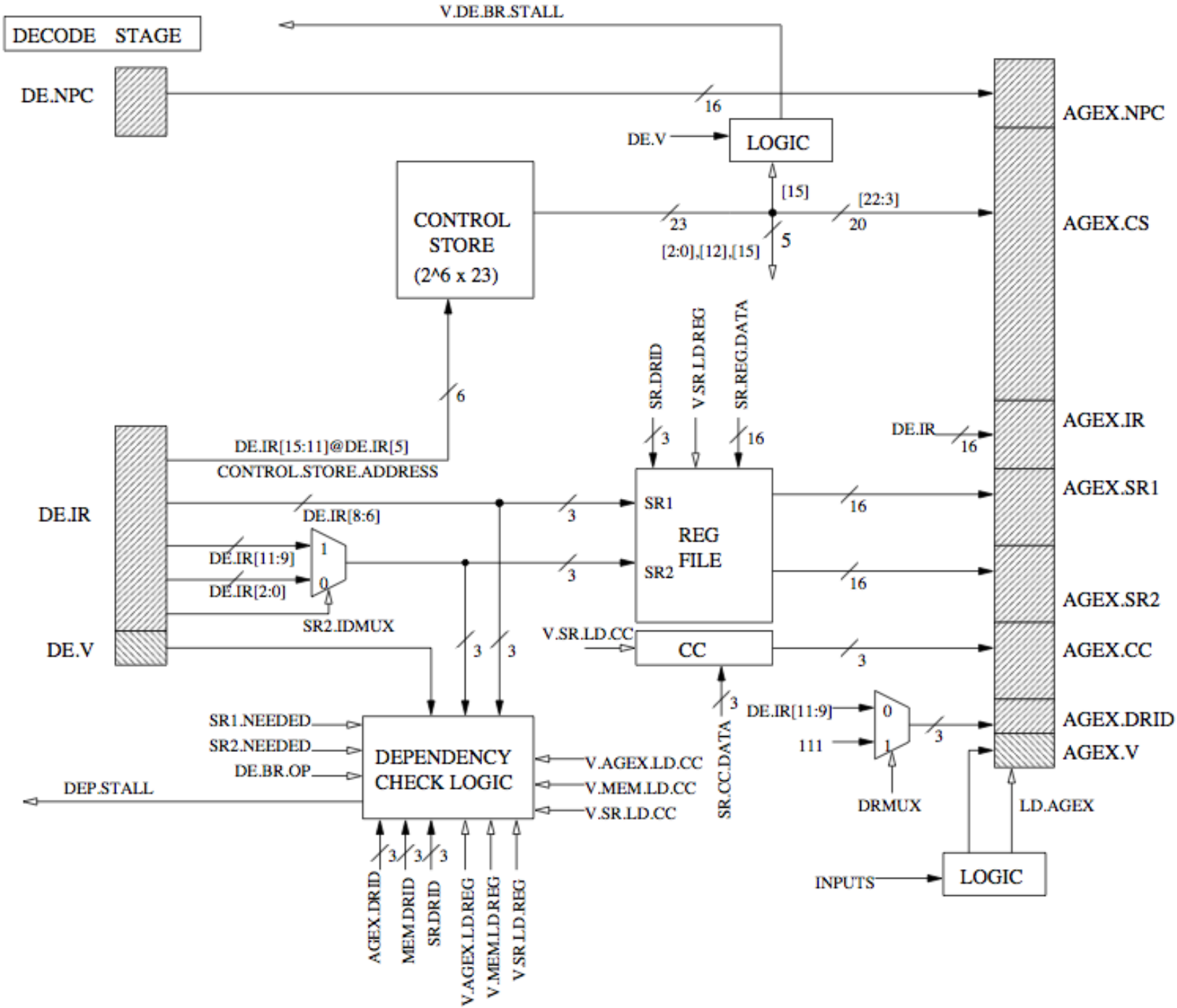
- Conservative handling of data and control dependences
  - Stall on branch
  - Stall on flow dependence

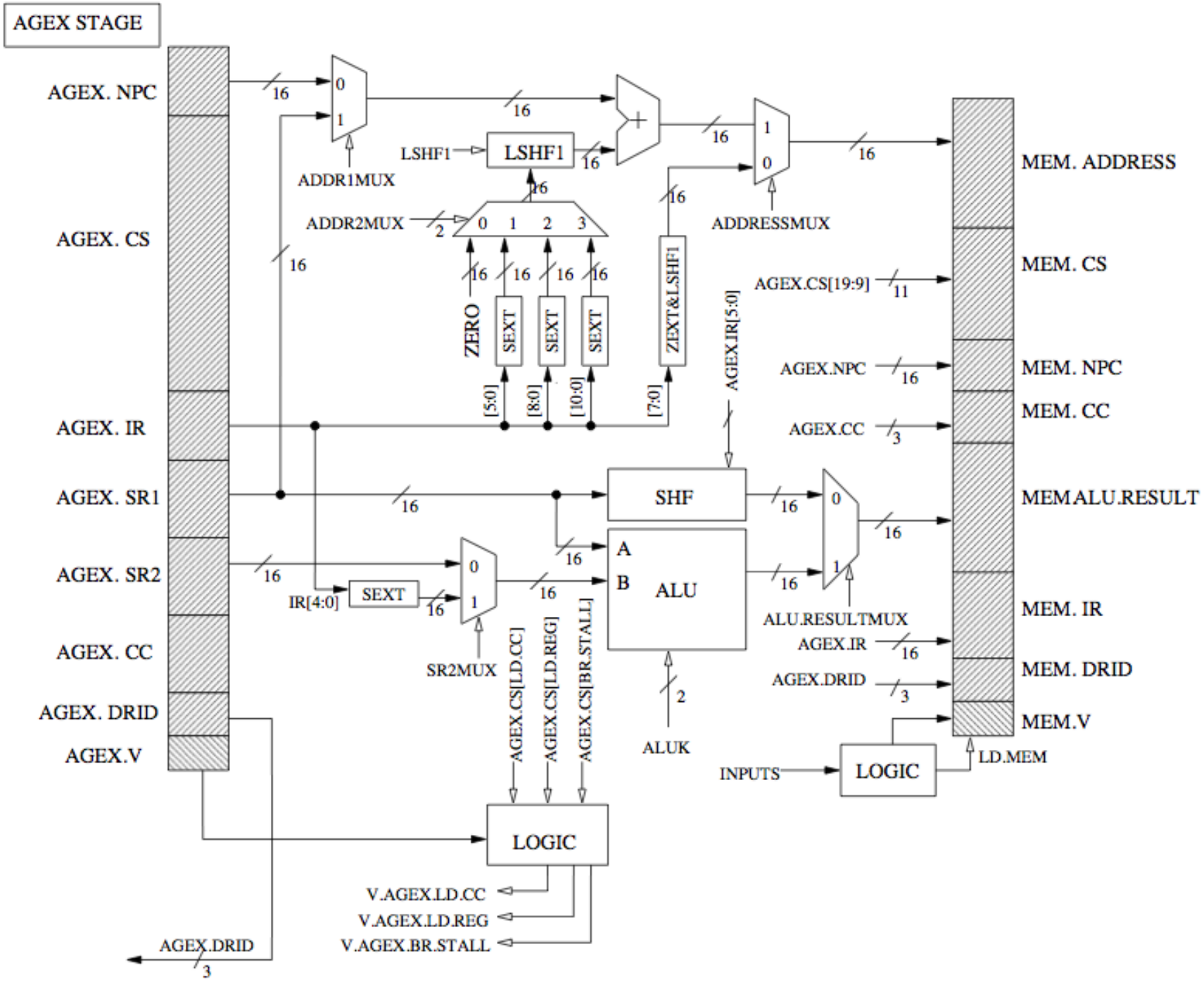

GateMARMUX
GatePC
LD.PC
PC
+2
MARMUX
PCMUX
REG FILE
LD.REG
DR
SR2
SR2 OUT
SR1 OUT
SR1
ZEXT & LSHF1
LSHF1
ADDR1MUX
[7:0]
ADDR2MUX
[10:0]
SEXT
[8:0]
SEXT
[5:0]
SEXT
[4:0]
SEXT
0
SR2MUX
CONTROL
R
LD.IR
IR
LD.CC
N
Z
P
LOGIC
ALUK
B
A
ALU
SHF
IR[5:0]
GateALU
GateSHF
GateMDR
MAR
LD.MAR
LOGIC
DATA.SIZE
MAR[0]
WE LOGIC
R.W
DATA. SIZE
MIO.EN
INPUT
OUTPUT
KBDR
DDR
KBSR
DSR
WE1 WE0
MEMORY
MEM.EN
ADDR. CTL. LOGIC
MDR
LD.MDR
MIO.EN
LOGIC
DATA.SIZE
MAR[0]
INMUX

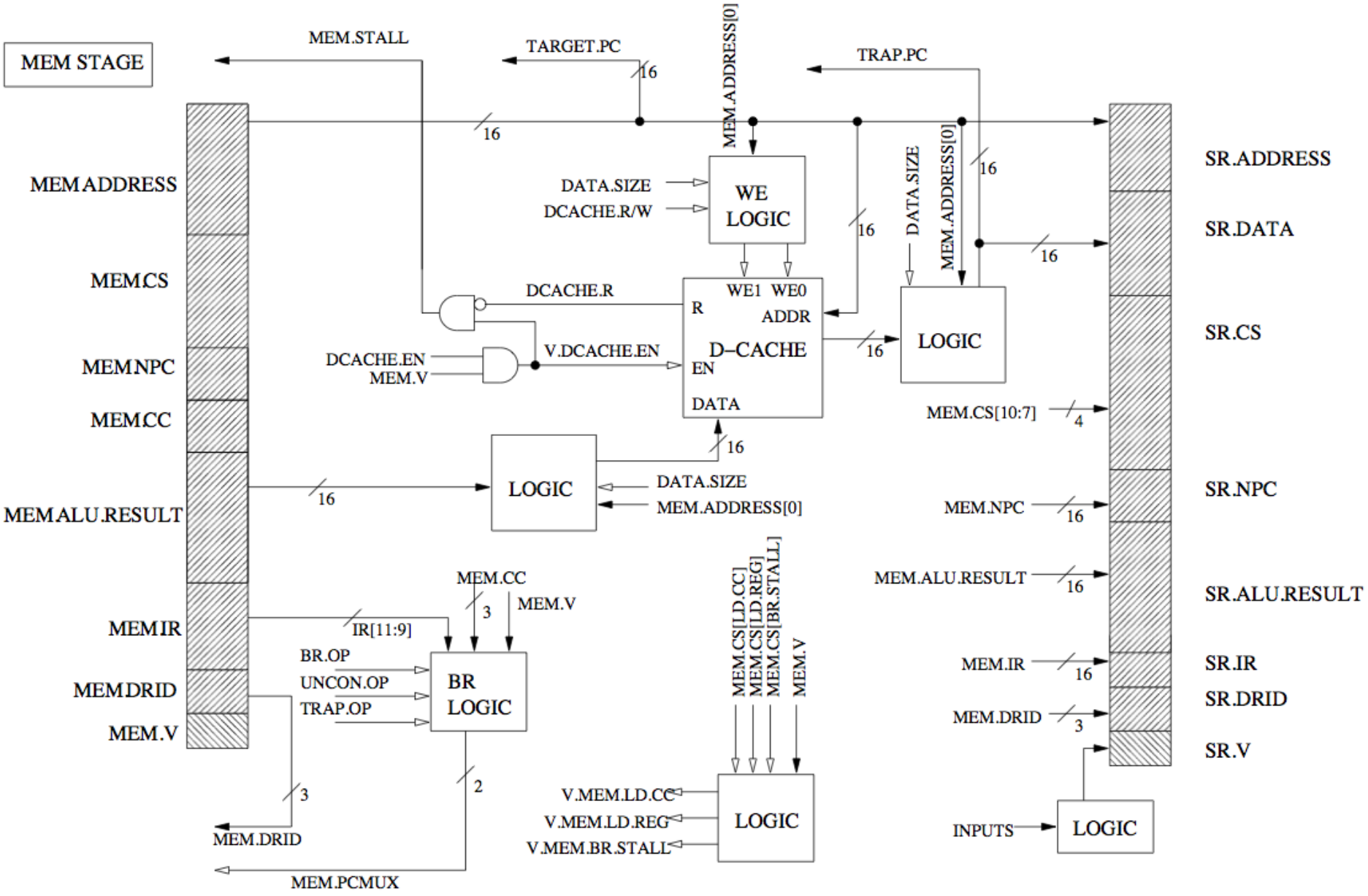
An Example LC-3b Pipeline

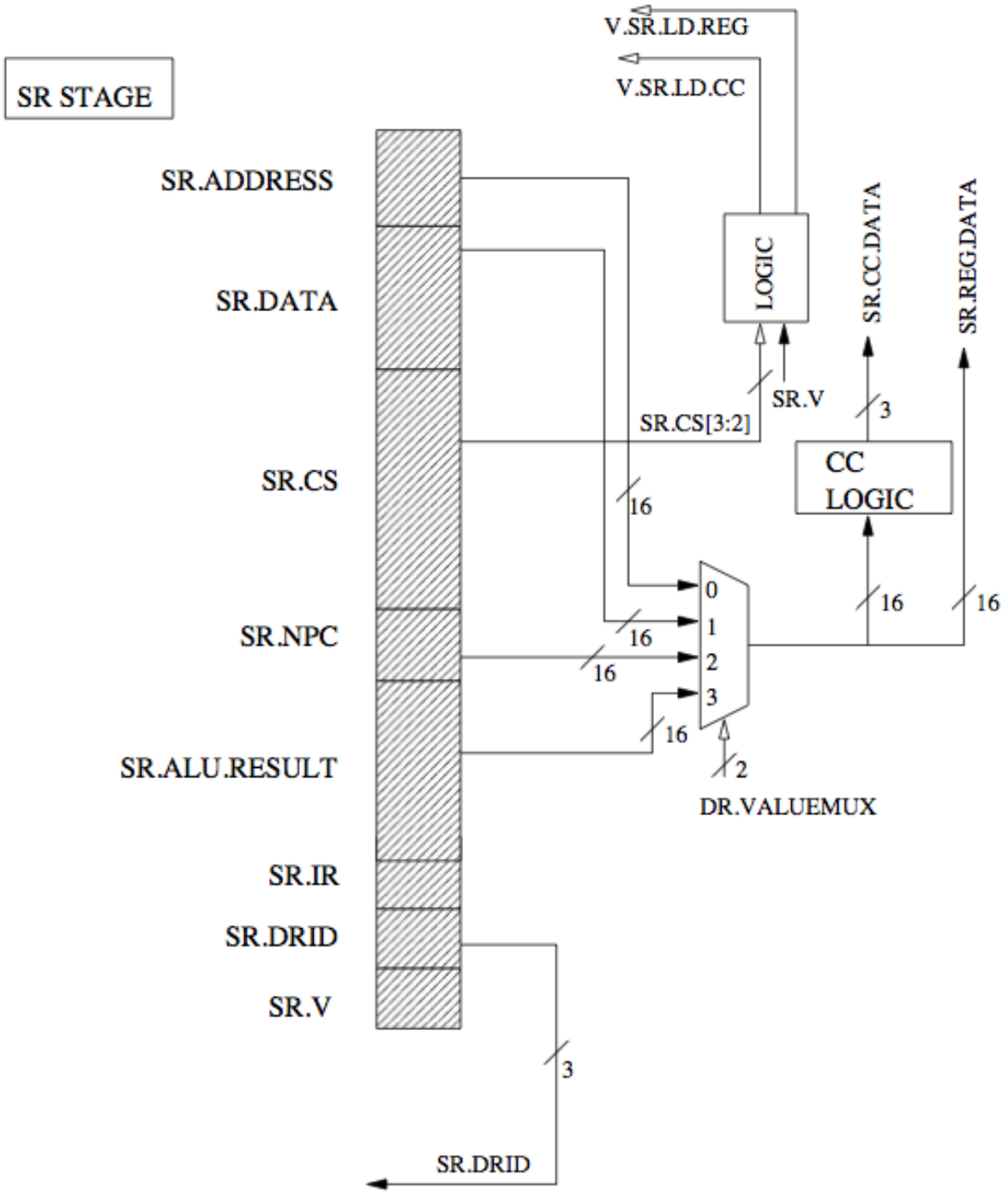



FETCH STAGE
TARGET.PC
TRAP.PC
16
16
16
16
MEM.PCMUX
2
2
1
0
+2
16
DE.NPC
LD.DE
16
16
INPUTS
LOGIC
LD.PC
PC
16
I–CACHE
16
DE.IR
ICACHE.R
INPUTS
LOGIC
DE.V
LD.DE


DECODE STAGE
V.DE.BR.STALL
DE.NPC
16
AGEX.NPC
DE.V
LOGIC
[15]
CONTROL STORE (2^6 x 23)
23
[22:3]
20
AGEX.CS
[2:0],[12],[15]
5
6
SR.DRID
V.SR.LD.REG
SR.REG.DATA
3
16
DE.IR
16
AGEX.IR
DE.IR[15:11]@DE.IR[5]
CONTROL.STORE.ADDRESS
DE.IR
DE.IR[8:6]
3
SR1
REG FILE
SR2
16
AGEX.SR1
DE.IR[11:9]
1
0
DE.IR[2:0]
3
16
AGEX.SR2
SR2.IDMUX
DE.V
V.SR.LD.CC
CC
3
AGEX.CC
3
3
3
SR1.NEEDED
SR2.NEEDED
DE.BR.OP
DEPENDENCY CHECK LOGIC
V.AGEX.LD.CC
V.MEM.LD.CC
V.SR.LD.CC
SR.CC.DATA
DE.IR[11:9]
0
1
111
3
AGEX.DRID
AGEX.V
DEP.STALL
DRMUX
LD.AGEX
INPUTS
LOGIC
AGEX.DRID
MEM.DRID
SR.DRID
V.AGEX.LD.REG
V.MEM.LD.REG
V.SR.LD.REG


AGEX STAGE
AGEX. NPC
AGEX. CS
AGEX. IR
AGEX. SR1
AGEX. SR2
AGEX. CC
AGEX. DRID
AGEX.V
16
0
1
ADDR1MUX
LSHF1
LSHF1
16
ADDR2MUX
2
0 1 2 3
ZERO
SEXT
SEXT
SEXT
[5:0]
[8:0]
[10:0]
ZEXT&LSHF1
[7:0]
ADDRESSMUX
MEM. ADDRESS
MEM. CS
AGEX.CS[19:9]
11
AGEX.IR[5:0]
AGEX.NPC
MEM. NPC
AGEX.CC
3
MEM. CC
SHF
MEM.ALU.RESULT
A
B
ALU
IR[4:0]
SEXT
SR2MUX
ALU.RESULTMUX
AGEX.IR
MEM. IR
AGEX.DRID
MEM. DRID
ALUK
2
MEM.V
INPUTS
LOGIC
LD.MEM
AGEX.CS[LD.CC]
AGEX.CS[LD.REG]
AGEX.CS[BR.STALL]
LOGIC
V.AGEX.LD.CC
V.AGEX.LD.REG
V.AGEX.BR.STALL
AGEX.DRID
3


MEM STAGE
MEM.STALL
TARGET.PC
16
MEM.ADDRESS[0]
TRAP.PC
MEMADDRESS
MEMCS
MEMNPC
MEMCC
MEMALU.RESULT
MEMIR
MEMDRID
MEM.V
DATA.SIZE
DCACHE.R/W
WE
LOGIC
DCACHE.R
WE1 WE0
R
ADDR
D–CACHE
EN
DATA
DCACHE.EN
MEM.V
V.DCACHE.EN
LOGIC
DATA.SIZE
MEM.ADDRESS[0]
MEM.CC
3
MEM.V
IR[11:9]
BR.OP
UNCON.OP
TRAP.OP
BR
LOGIC
MEM.DRID
2
MEM.PCMUX
MEM.CS[LD.CC]
MEM.CS[LD.REG]
MEM.CS[BR.STALL]
MEM.V
V.MEM.LD.CC
V.MEM.LD.REG
V.MEM.BR.STALL
LOGIC
SR.ADDRESS
SR.DATA
SR.CS
SR.NPC
SR.ALU.RESULT
SR.IR
SR.DRID
SR.V
MEM.CS[10:7]
4
MEM.NPC
MEM.ALU.RESULT
MEM.IR
MEM.DRID
INPUTS
LOGIC


SR STAGE
V.SR.LD.REG
V.SR.LD.CC
SR.ADDRESS
SR.DATA
SR.CS
SR.NPC
SR.ALU.RESULT
SR.IR
SR.DRID
SR.V
LOGIC
SR.CC.DATA
SR.REG.DATA
SR.V
SR.CS[3:2]
3
CC
LOGIC
16
16
16
16
16
0
1
2
3
2
DR.VALUEMUX
3
SR.DRID

Control of the LC-3b Pipeline

- Three types of control signals

- Datapath Control Signals
  - Control signals that control the operation of the datapath

- Control Store Signals
  - Control signals (microinstructions) stored in control store to be used in pipelined datapath (can be propagated to stages later than decode)

- Stall Signals
  - Ensure the pipeline operates correctly in the presence of dependencies

Stage	Signal Name	Signal Values	
FETCH	MEM.PCMUX/2:††	PC+2	;select pc+2
		TARGET.PC	;select MEM.TARGET.PC (branch target)
		TRAP.PC	;select MEM.TRAP.PC
	LD.PC/1:†	NO(0), LOAD(1)	
	LD.DE/1:†	NO(0), LOAD(1)	
DECODE	DRMUX/1:	11.9	;destination IR[11:9]
		R7	;destination R7
	SR1.NEEDED/1:	NO(0), YES(1)	;asserted if instruction needs SR1
	SR2.NEEDED/1:	NO(0), YES(1)	;asserted if instruction needs SR2
	DE.BR.OP/1:	NO(0), BR(1)	;BR Opcode
	SR2.IDMUX/1:†	2.0	;source IR[2:0]
		11.9	;source IR[11:9]
	LD.AGEX/1:†	NO(0), LOAD(1)	
	V.AGEX.LD.CC/1:††	NO(0), LOAD(1)	
	V.MEM.LD.CC/1:††	NO(0), LOAD(1)	
	V.SR.LD.CC/1:††	NO(0), LOAD(1)	
	V.AGEX.LD.REG/1:††	NO(0), LOAD(1)	
	V.MEM.LD.REG/1:††	NO(0), LOAD(1)	
	V.SR.LD.REG/1:††	NO(0), LOAD(1)	
AGEX	ADDR1MUX/1:	NPC	;select value from AGEX.NPC
		BaseR	;select value from AGEX.SR1(BaseR)
	ADDR2MUX/2:	ZERO	;select the value zero
		offset6	;select SEXT[IR[5:0]]
		PCoffset9	;select SEXT[IR[8:0]]
		PCoffset11	;select SEXT[IR[10:0]]
	LSHF1/1:	NO(0), 1bit Left shift(1)	
	ADDRESSMUX/1:	7.0	;select LSHF(ZEXT[IR[7:0]],1)
		ADDER	;select output of address adder
	SR2MUX/1:	SR2	;select from AGEX.SR2
		4.0	;IR[4:0]
	ALUK/2:	ADD(00), AND(01)	
		XOR(10), PASSB(11)	
	ALU.RESULTMUX/1:	SHIFTER	;select output of the shifter
		ALU	;select tput out the ALU
	LD.MEM/1:†	NO(0), LOAD(1)	
MEM	DCACHE.EN/1:	NO(0), YES(1)	;asserted if the instruction accesses memory
	DCACHE.RW/1:	RD(0), WR(1)	
	DATA.SIZE/1:	BYTE(0), WORD(1)	
	BR.OP/1:	NO(0), BR(1)	;BR
	UNCON.OP/1:	NO(0), Uncond.BR(1)	;JMP,RET, JSR, JSRR
	TRAP.OP/1:	NO(0), Trap(1)	;TRAP
SR	DR.VALUEMUX/2:	ADDRESS	;select value from SR.ADDRESS
		DATA	;select value from SR.DATA
		NPC	;select value from SR.NPC
		ALU	;select value from SR.ALU.RESULT
	LD.REG/1:	NO(0), LOAD(1)	
	LD.CC/1:	NO(0), LOAD(1)	

Table 1: Data Path Control Signals
†: The control signal is generated by logic in that stage
††: The control signal is generated by logic in another stage

Control Store in a Pipelined Machine

Number	Signal Name	Stages
0	SR1.NEEDED	DECODE
1	SR2.NEEDED	DECODE
2	DRMUX	DECODE
3	ADDR1MUX	AGEX
4	ADDR2MUX1	AGEX
5	ADDR2MUX0	AGEX
6	LSHF1	AGEX
7	ADDRESSMUX	AGEX
8	SR2MUX	AGEX
9	ALUK1	AGEX
10	ALUK0	AGEX
11	ALU.RESULTMUX	AGEX
12	BR.OP	DECODE, MEM
13	UNCON.OP	MEM
14	TRAP.OP	MEM
15	BR.STALL	DECODE, AGEX, MEM
16	DCACHE.EN	MEM
17	DCACHE.RW	MEM
18	DATA.SIZE	MEM
19	DR.VALUEMUX1	SR
20	DR.VALUEMUX0	SR
21	LD.REG	AGEX, MEM, SR
22	LD.CC	AGEX, MEM, SR

Table 2: Control Store ROM Signals

Stall Signals

- Pipeline stall: Pipeline does not move because an operation in a stage cannot complete
- Stall Signals: Ensure the pipeline operates correctly in the presence of such an operation
- Why could an operation in a stage not complete?

Signal Name	Generated in	
ICACHE.R/1:	FETCH	NO, READY
DEP.STALL/1:	DEC	NO, STALL
V.DE.BR.STALL/1:	DEC	NO, STALL
V.AGEX.BR.STALL/1:	AGEX	NO, STALL
MEM.STALL/1:	MEM	NO, STALL
V.MEM.BR.STALL/1:	MEM	NO, STALL

Table 3: STALL Signals