CS151B
Computer Systems Architecture

Week 6 Discussion

2/16/2018

Agenda

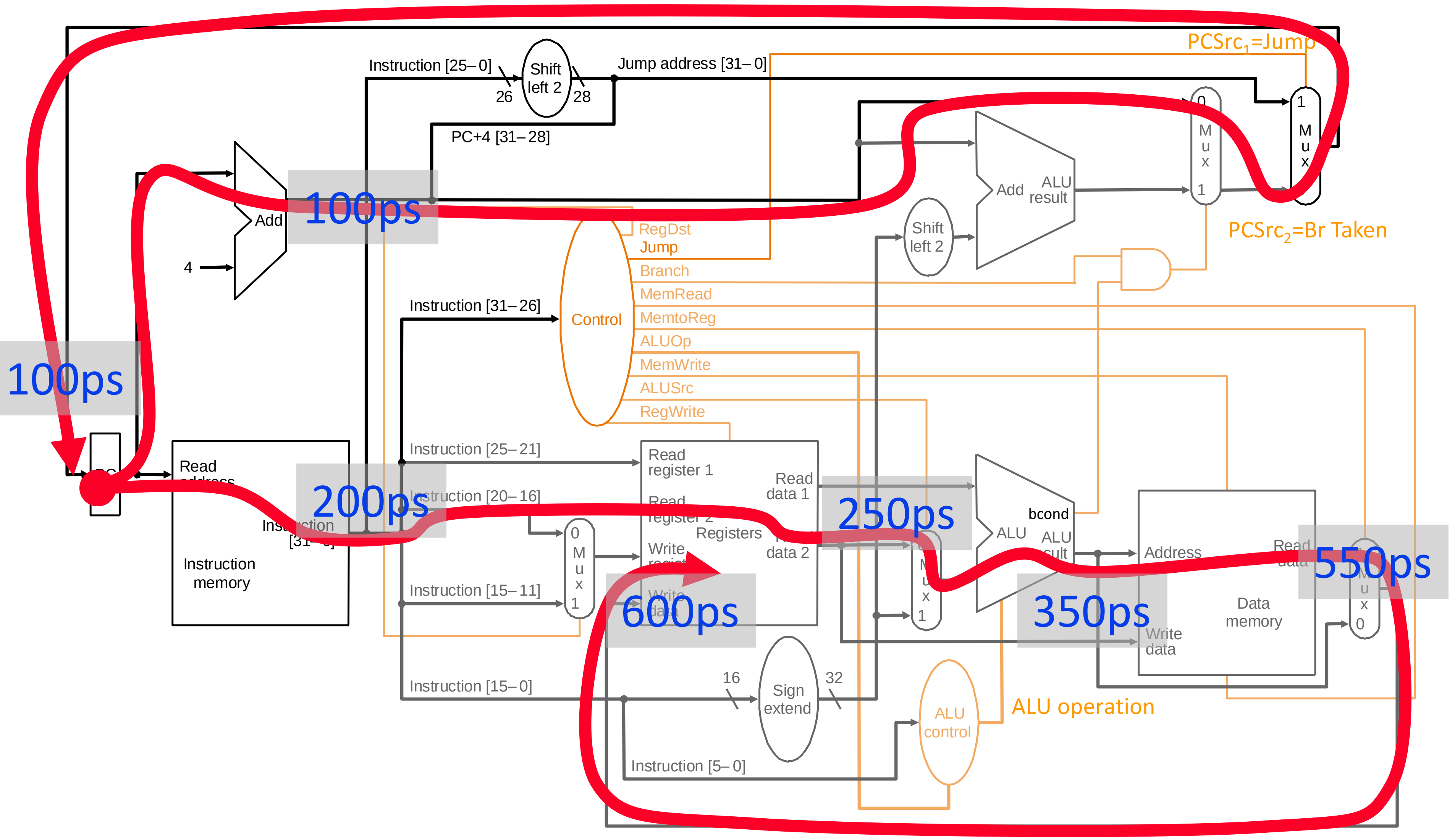
- Pipelining

Single-cycle Datapath

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

- All instructions run as slow as the slowest instruction

LW



PCSrc$_1$=Jump
Instruction [25– 0]
Shift left 2
26
28
Jump address [31– 0]
PC+4 [31– 28]
Add
4
100ps
Add
ALU result
M u x
0
1
PCSrc$_2$=Br Taken
Shift left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31– 26]
100ps
PC
Read address
Instruction [31– 0]
Instruction memory
200ps
Instruction [25– 21]
Instruction [20– 16]
Instruction [15– 11]
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
250ps
bcond
ALU
ALU result
350ps
Address
Read data
Data memory
Write data
550ps
600ps
Instruction [15– 0]
16
Sign extend
32
Instruction [5– 0]
ALU control
ALU operation

Single-cycle Datapath

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

- All instructions run as slow as the slowest instruction

- 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

Pipelining: Basic Idea

- Idea
  - Divide the instruction processing cycle into distinct stages of processing
  - Ensure that 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

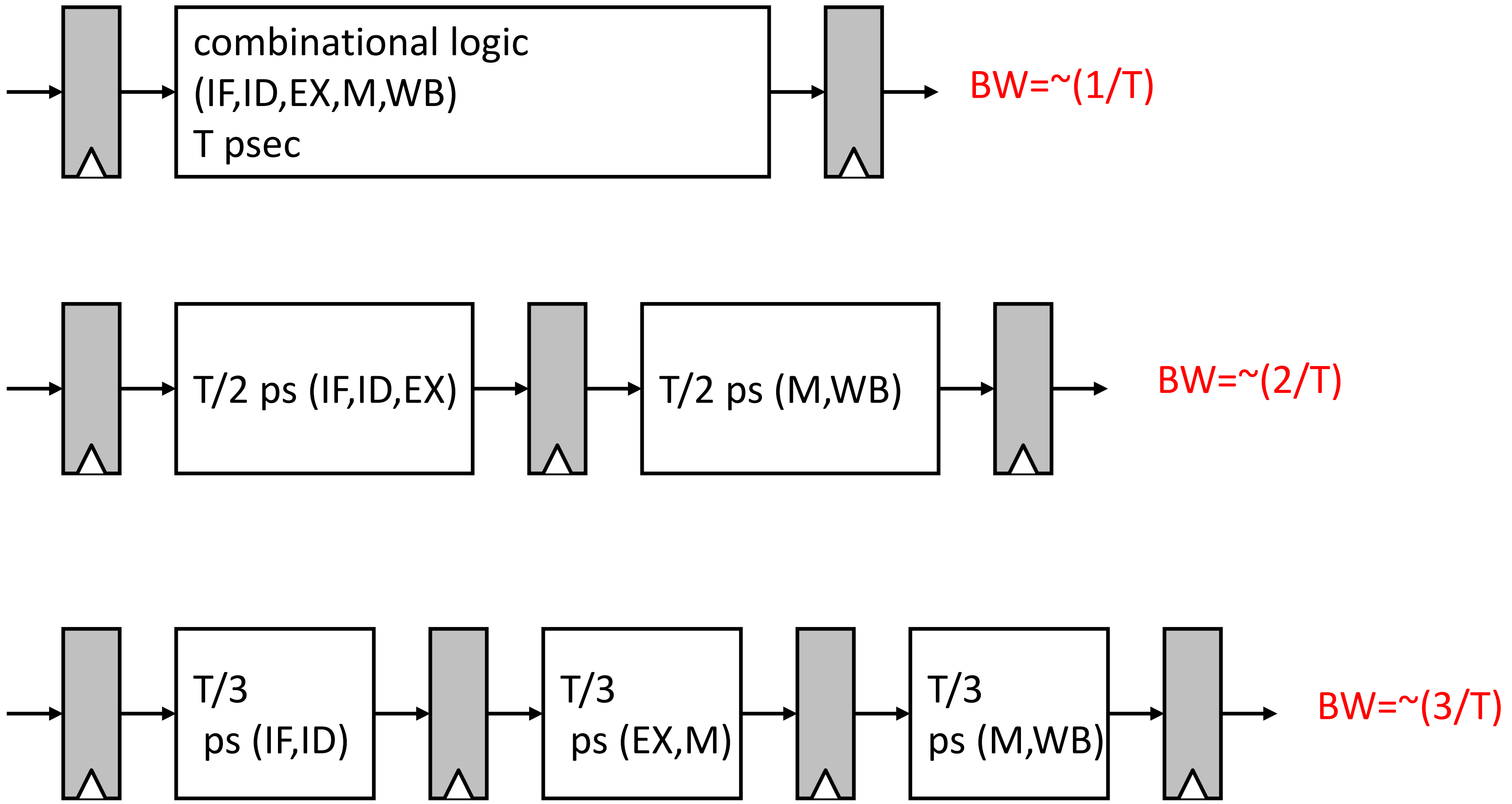
- Benefit: increases instruction processing throughput
- Downside?

An Ideal Pipeline

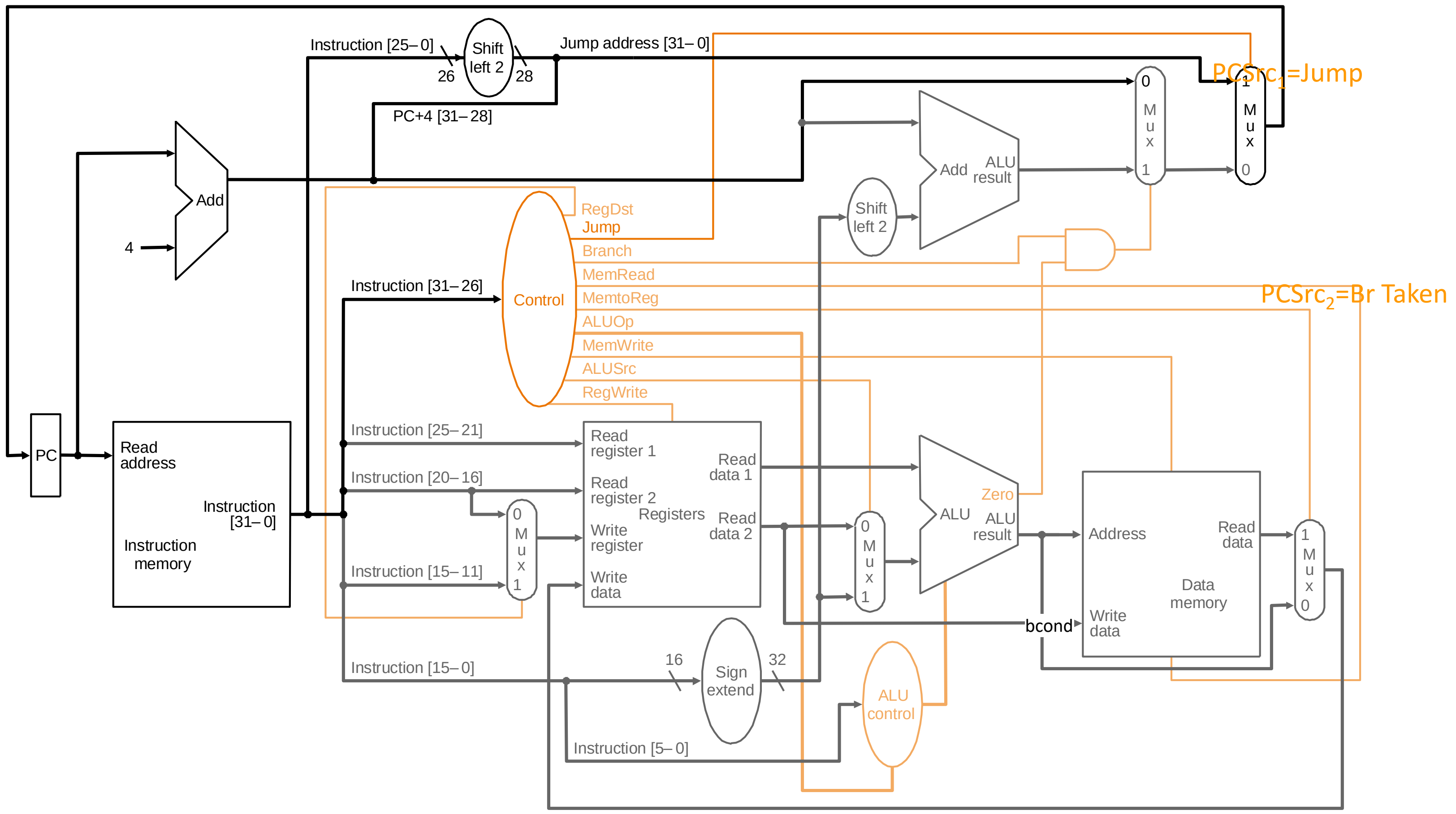
- Goal: **Increase throughput with little increase in cost** (hardware cost, in case of instruction processing)

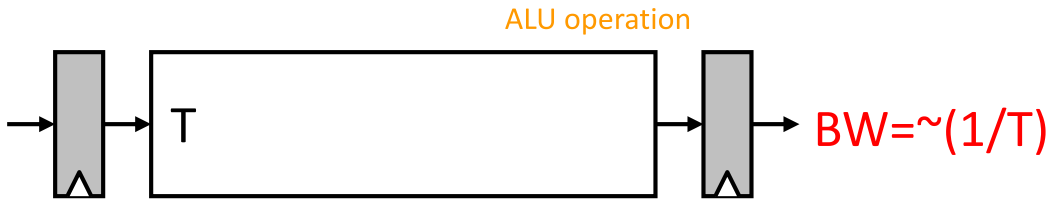
- Repetition of **identical operations**
  - The same operation is repeated on a large number of different inputs (e.g., all laundry loads go through the same steps)
- Repetition of **independent operations**
  - No dependencies between repeated operations
- **Uniformly partitionable suboperations**
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)

Ideal Pipelining

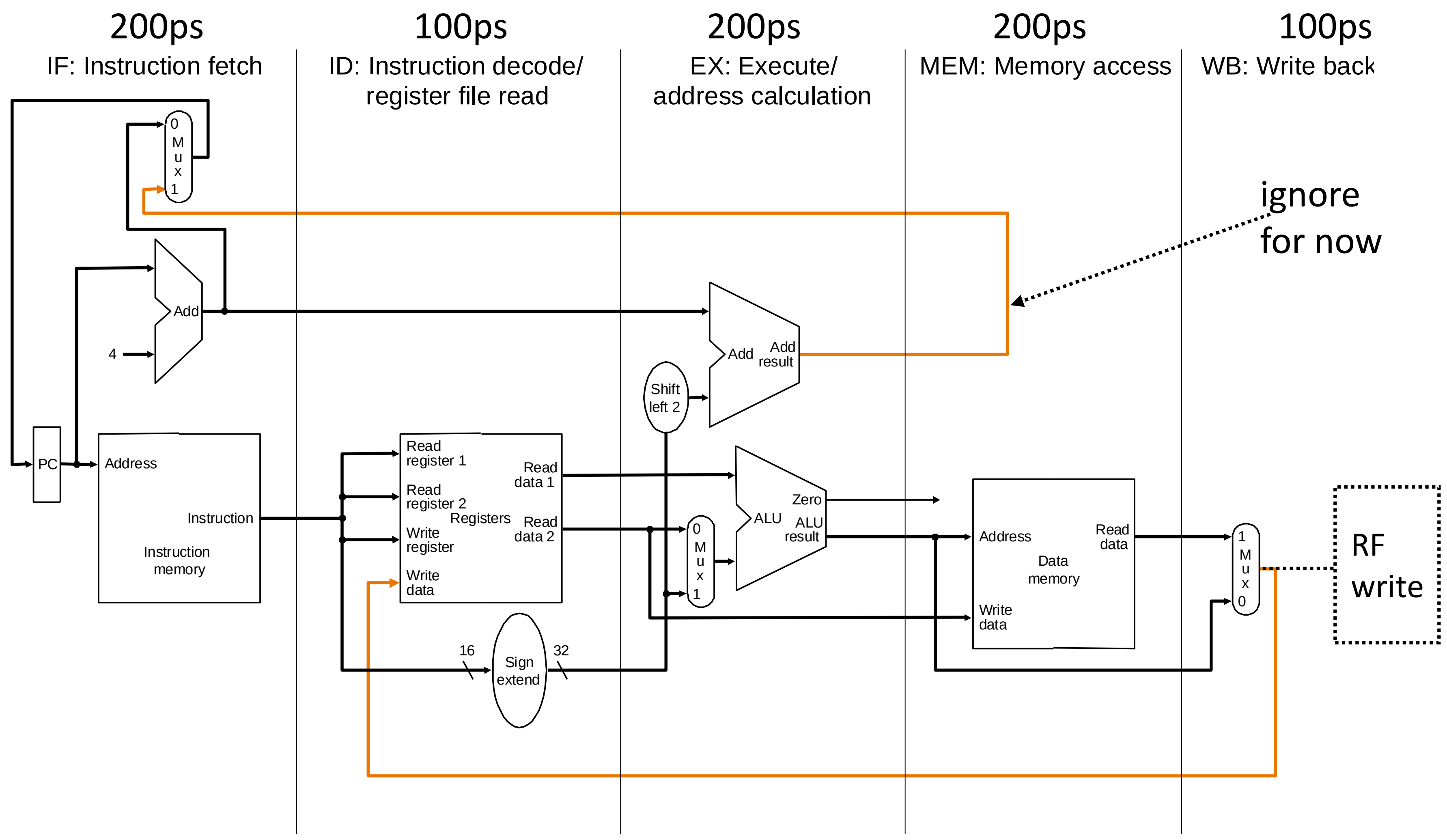


Remember the Single-Cycle Datapath

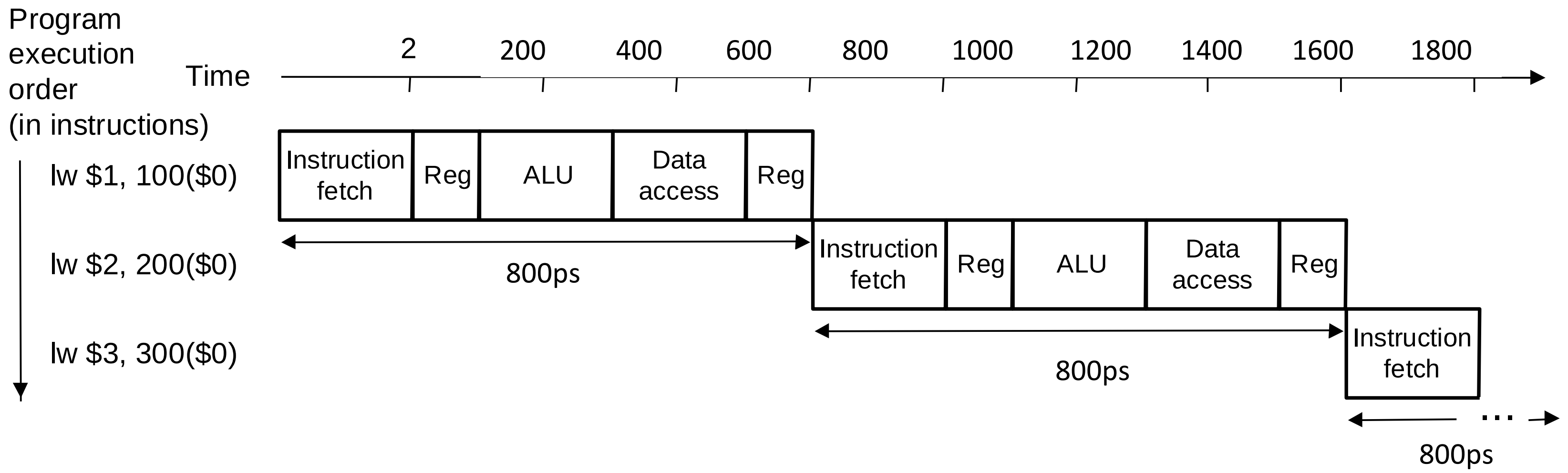


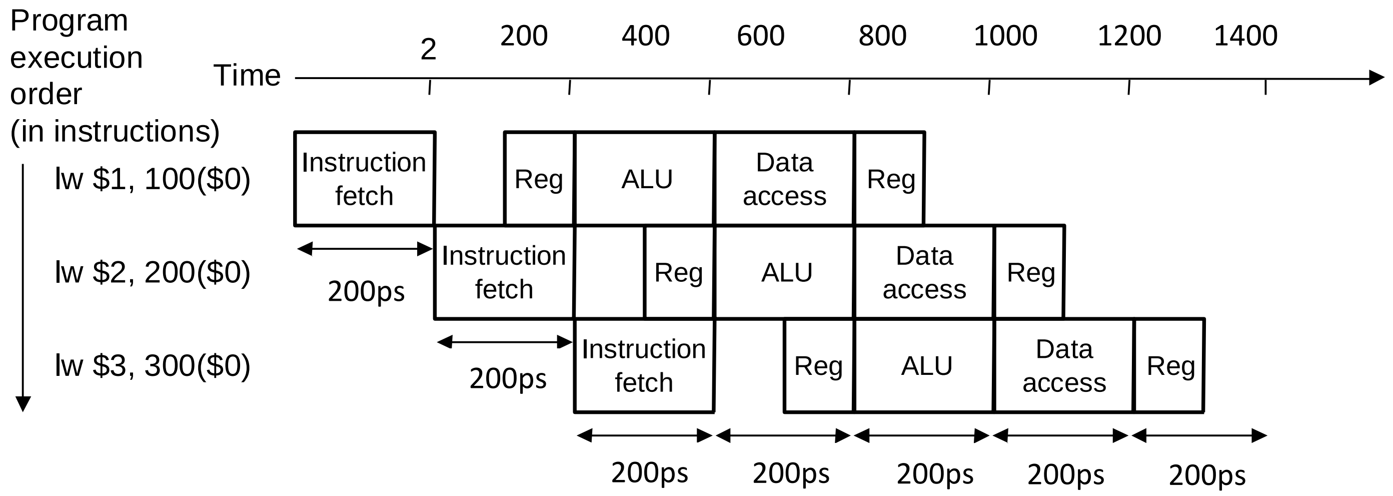


Dividing Into Stages



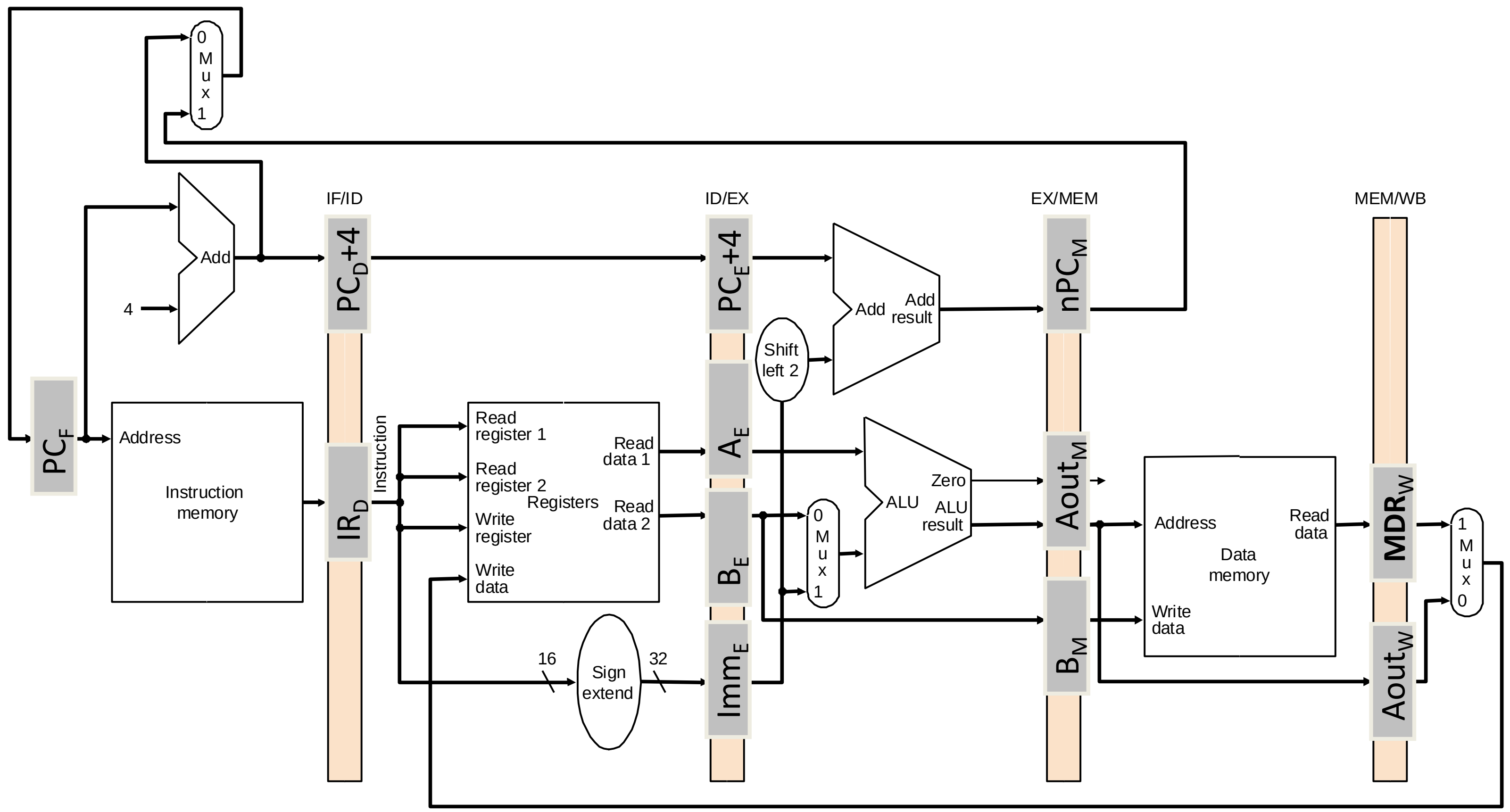
Instruction Pipeline Throughput

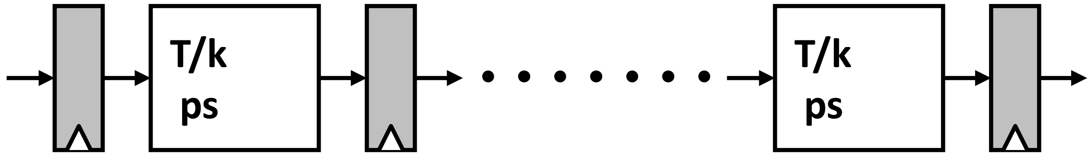




5-stage speedup is 4, not 5 as predicted by the ideal model. Why?

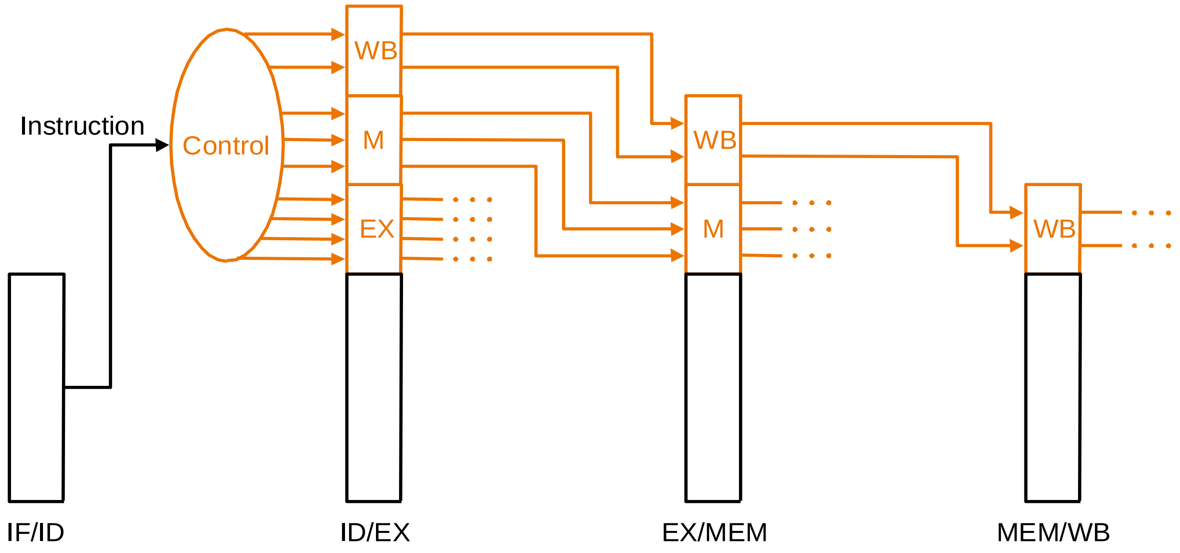
Enabling Pipelined Processing: Pipeline Registers





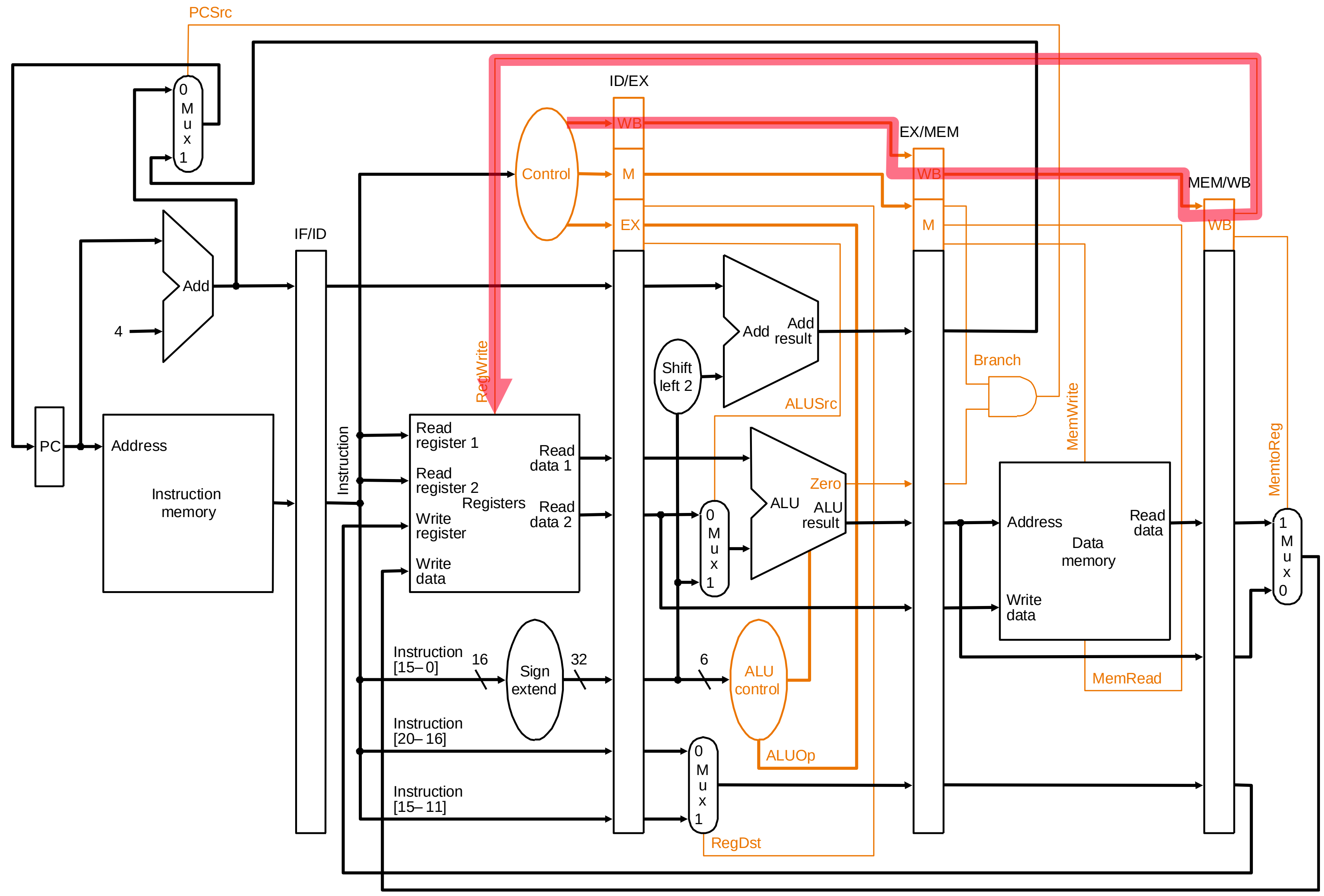
Control Signals in a Pipeline

- For a given instruction
  - same control signals as single-cycle, but
  - control signals required at different cycles, depending on stage
  - Option 1: decode once using the same logic as single-cycle and buffer signals until consumed



  - Option 2: 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



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*

- Stall: A condition when the pipeline stops moving

- Resource contention

- Dependences (between instructions)
  - Data
  - Control

- Long-latency (multi-cycle) operations

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

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 RAW

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)

Impact of Stall on Performance

- Each stall cycle corresponds to **one lost cycle** in which no instruction can be completed

- 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

Sample Assembly (P&H)

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

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

Reducing Stalls with Data Forwarding

- Forward the value to the dependent instruction as soon as it is available

- 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 **communication 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,

    2. until another instruction redefines RF[rx], younger instructions that refer to RF[rx] should use this instruction's result

Resolving RAW Dependence with Forwarding

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

- In other words, if IB in ID stage reads a register written by IA in EX, MEM or WB stage, then the operand required by IB 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 (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 (P&H)

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

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

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:
```

Slide credits

- Onur Mutlu