inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture 32 – Pipeline III

2018-11-09

Instructors

Dan Garcia and Bora Nikolic

Waymo Gets Green Light for Robot Cars in California; No Humans Needed

The California Department of Motor Vehicles has granted a permit to self-driving technology development company Waymo to test fully autonomous cars on public roads with no human driver. The company now will be able to deploy up to 40 self-driving vehicles to drive day and night on urban streets, rural highways, and highways with posted speeds up to 65 miles-per-hour. Said Waymo, "We will gradually begin driverless testing on city streets in a limited territory and, over time, expand the area that we drive in as we gain confidence and experience." The company already had been testing its fleet of 100 driverless cars in California, but with drivers at the ready to assume control in the event of malfunction. As with other companies' autonomous vehicles being tested, Waymo's no-driver autonomous cars must have remote operators who can communicate with passengers and send the cars recommendations for how to proceed if their automation becomes confused.

www.sfchronicle.com/business/article/Waymo-gets-green-light-for-robot-cars-in-13349173.php

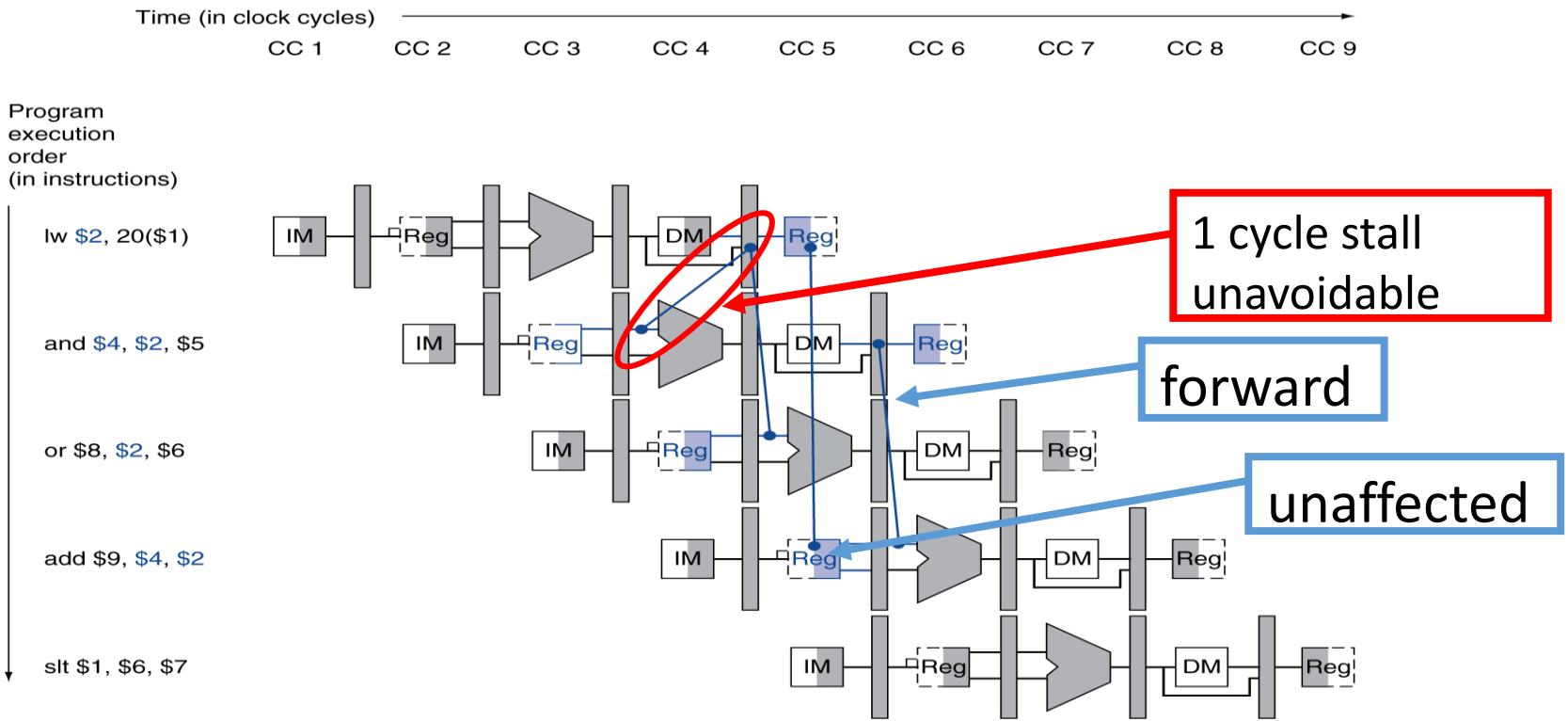
Review

- Pipelining increases throughput by overlapping execution of multiple instructions
- All pipeline stages have same duration
  - Choose partition that accommodates this constraint
- Hazards potentially limit performance
  - Maximizing performance requires programmer/compiler assistance
  - We've seen so far **Structure** & **Data** hazards so far…

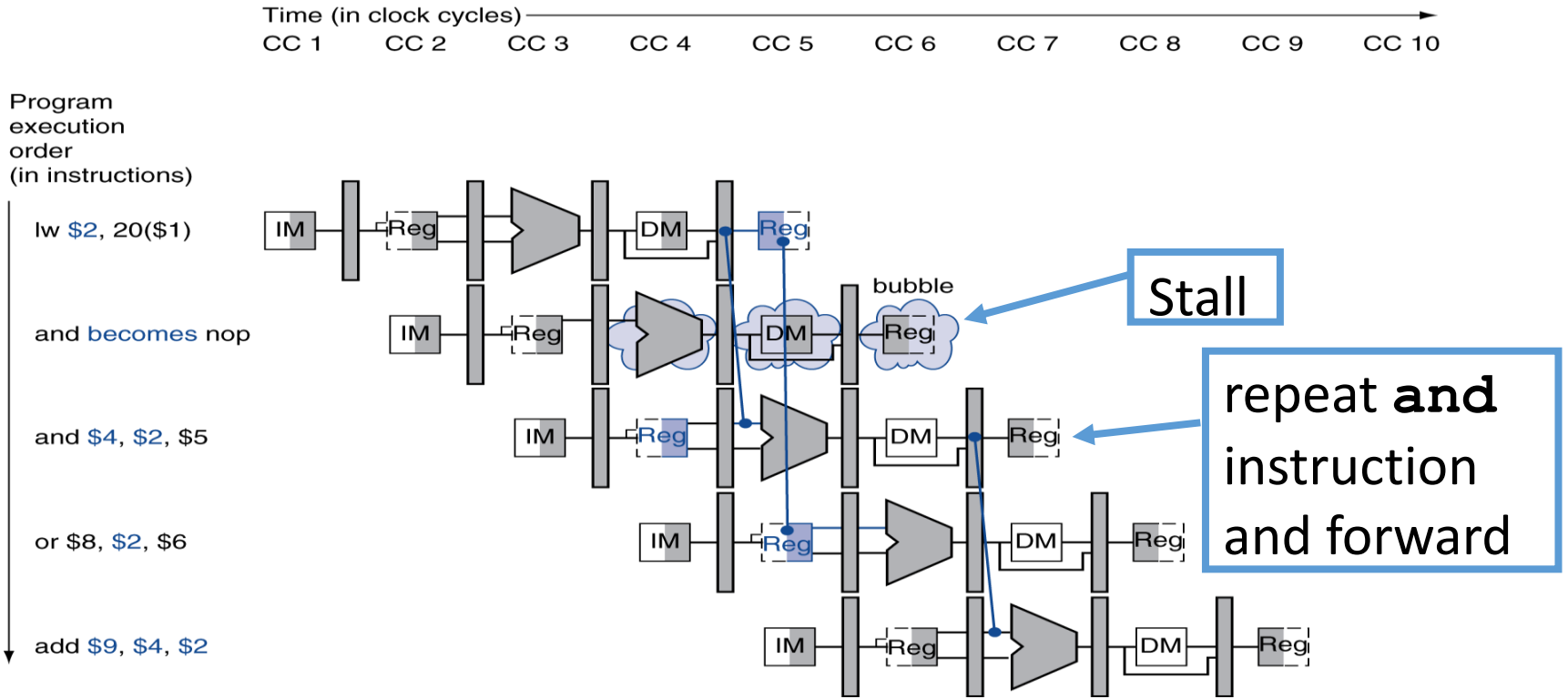
Agenda

- Hazards
  - Structural
  - Data
    - R-type instructions
    - **Load**
  - Control
- Superscalar processors

Load Data Hazard



Stall Pipeline

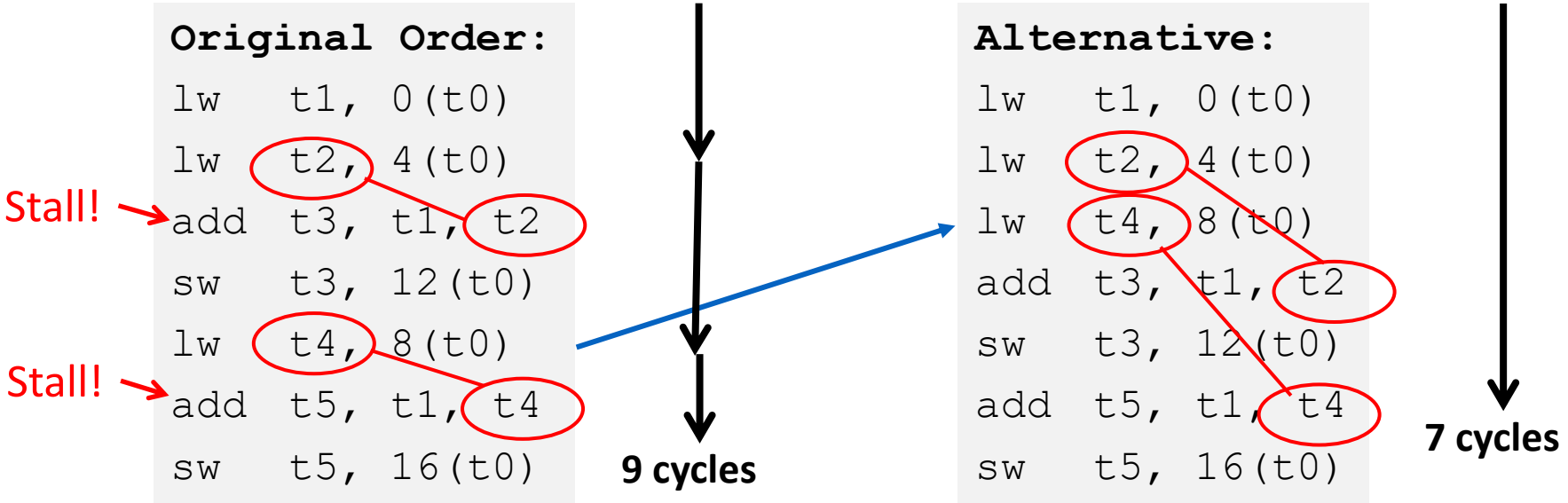


`lw` Data Hazard

- Slot after a load is called a *load delay slot*
  - If that instruction uses the result of the load, then the hardware will stall for one cycle
  - Equivalent to inserting an explicit **`nop`** in the slot
    - except the latter uses more code space
  - Performance loss
- Idea:
  - Put unrelated instruction into load delay slot
  - No performance loss!

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instr!
- RISC-V code for **`A[3]=A[0]+A[1]; A[4]=A[0]+A[2]`**



Agenda

- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - **Control**
- Superscalar processors

Control Hazards

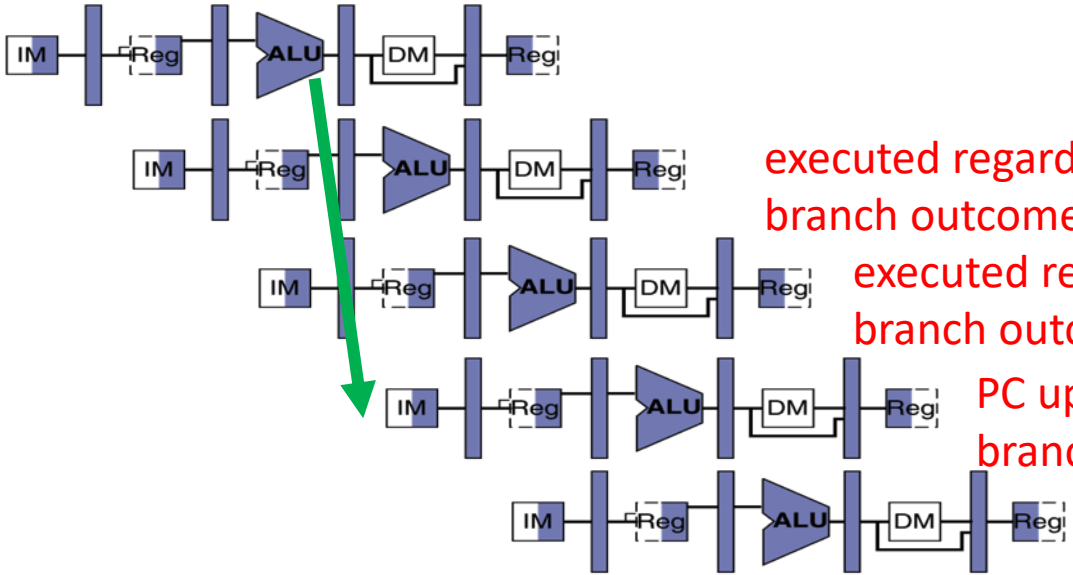
beq t0, t1, label

sub t2, s0, t5

or t6, s0, t3

xor t5, t1, s0

sw s0, 8(t3)



executed regardless of branch outcome!

executed regardless of branch outcome!!!

PC updated reflecting branch outcome

Observation

- If branch not taken, then instructions fetched sequentially after branch are correct
- If branch or jump taken, then need to flush incorrect instructions from pipeline by converting to NOPs

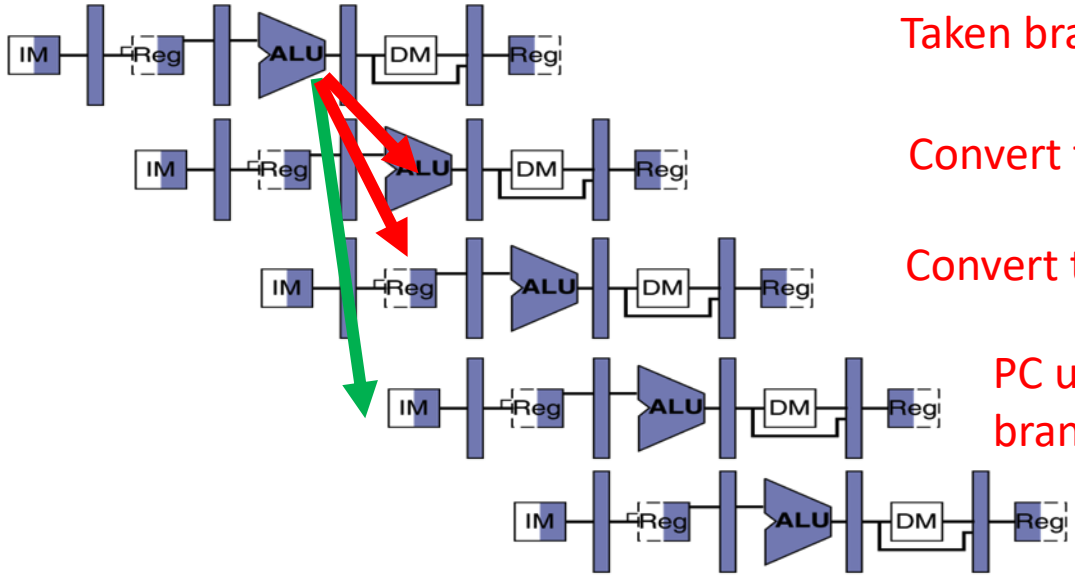
Kill Instructions after Branch if Taken

beq t0, t1, label

sub t2, s0, t5

or t6, s0, t3

label: xxxxxx



Taken branch

Convert to NOP

Convert to NOP

PC updated reflecting branch outcome

Reducing Branch Penalties

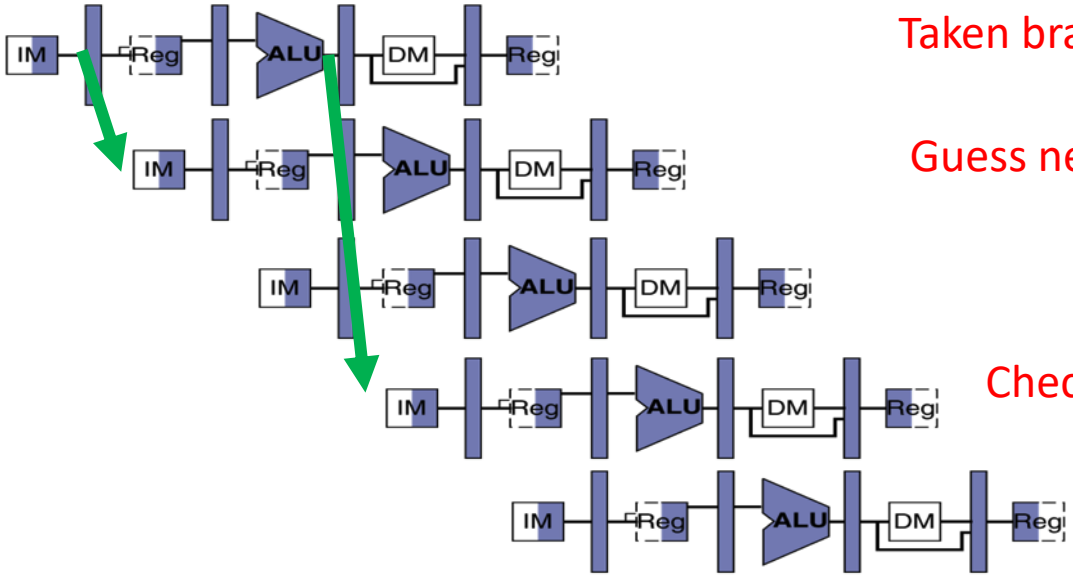
- Every taken branch in simple pipeline costs 2 dead cycles
- To improve performance, use “branch prediction” to guess which way branch will go earlier in pipeline
- Only flush pipeline if branch prediction was incorrect

Branch Prediction

beq t0, t1, label

label: …..

…..



Taken branch

Guess next PC!

Check guess correct

Agenda

- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- **Superscalar processors**

Increasing Processor Performance

1. Clock rate
   - Limited by technology and power dissipation
2. Pipelining
   - "Overlap" instruction execution
   - Deeper pipeline: 5 => 10 => 15 stages
     - Less work per stage → shorter clock cycle
     - But more potential for hazards (CPI > 1)
3. Multi-issue "superscalar" processor

Superscalar Processor

- Multiple issue “superscalar”
  - Replicate pipeline stages ⇒ multiple pipelines
  - Start multiple instructions per clock cycle
  - CPI < 1, so use Instructions Per Cycle (IPC)
  - E.g., 4GHz 4-way multiple-issue
    - 16 BIPS, peak CPI = 0.25, peak IPC = 4
  - Dependencies reduce this in practice
- “Out-of-Order” execution
  - Reorder instructions dynamically in hardware to reduce impact of hazards
- *CS152 discusses these techniques!*

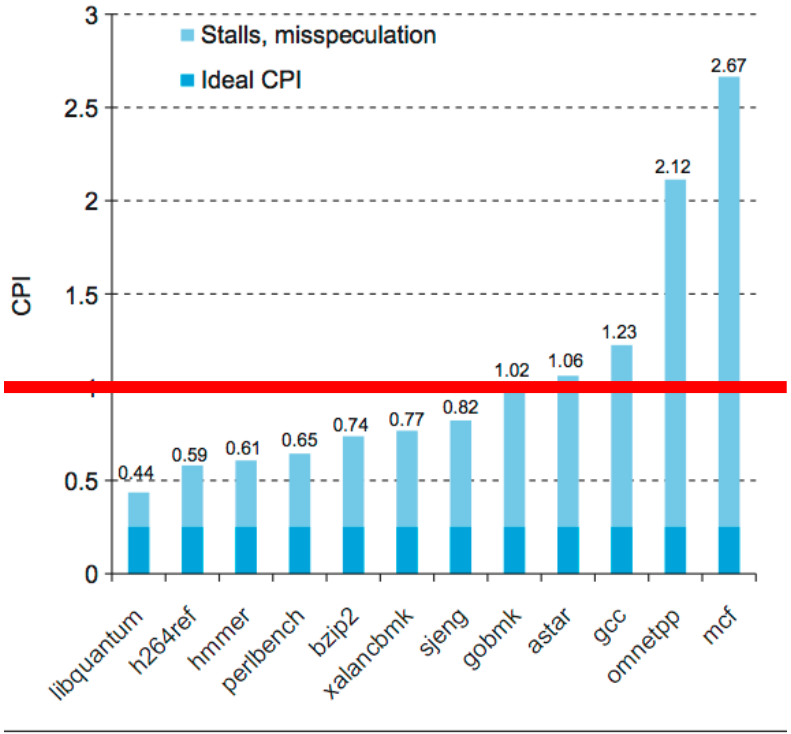
https://en.wikipedia.org/wiki/Superscalar_processor

Superscalar = Multicore?

- A superscalar processor is a CPU that implements a form of parallelism called instruction-level parallelism within a single processor. In contrast to a scalar processor that can execute at most one single instruction per clock cycle, a superscalar processor can execute more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to different execution units on the processor. It therefore allows for more throughput (the number of instructions that can be executed in a unit of time) than would otherwise be possible at a given clock rate. **Each execution unit is not a separate processor (or a core if the processor is a multi-core processor), but an execution resource within a single CPU such as an arithmetic logic unit.**
- In Flynn's taxonomy, **a single-core superscalar processor is classified as an SISD** processor (Single Instruction stream, Single Data stream), though many superscalar processors support short vector operations and so could be classified as SIMD (Single Instruction stream, Multiple Data streams). A multi-core superscalar processor is classified as an MIMD processor (Multiple Instruction streams, Multiple Data streams).

Benchmark: CPI of Intel Core i7

CPI = 1



CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.

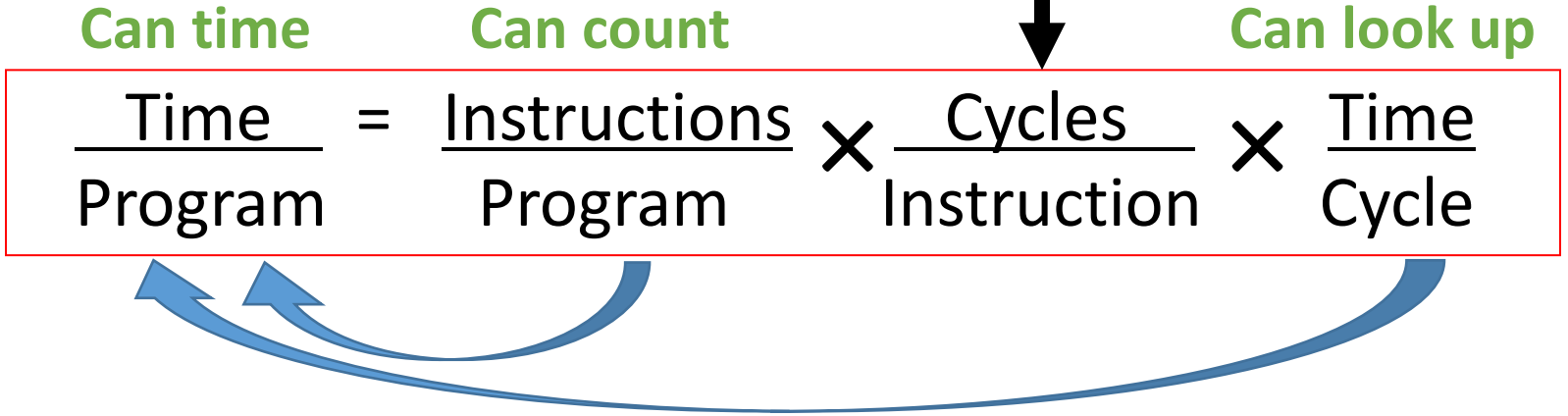
P&H p. 350

Pipelining and ISA Design

- RISC-V ISA designed for pipelining
  - All instructions are 32-bits
    - Easy to fetch and decode in one cycle
    - Versus x86: 1- to 15-byte instructions
  - Few and regular instruction formats
    - Decode and read registers in one step
  - Load/store addressing
    - Calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

“Iron Law” of Processor Performance

CPI = Cycles Per Instruction



$$\textbf{CPI} = \frac{\text{Cycles}}{\text{Instruction}} = \frac{\text{Time}}{\text{Program}} \div \left(\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Time}}{\text{Cycle}} \right)$$

Calculating CPI Another Way

- First calculate CPI for each individual instruction (**`add`**, **`sub`**, **`and`**, etc.)
- Next calculate frequency of each individual instruction
- Finally multiply these two for each instruction and add them up to get final CPI (the weighted sum)

Example (RISC processor)

Op	$Freq_i$	CPI_i	Prod	(% Time)
ALU	50%	1	.5	(23%)
Load	20%	5	1.0	(45%)
Store	10%	3	.3	(14%)
Branch	20%	2	.4	(18%)
			2.2	

Instruction Mix

(Where time spent)

- What if Branch instructions twice as fast?

Peer Instruction: yellkey mouth

1) A single core, superscalar processor is considered SISD

2) In RISC-V, the slot after a branch is called a branch delay slot (same as a load delay slot, always executed)

3) To improve Overall CPI, (if you could make one twice as fast), one should choose the sub-component (ALU, Load, Store, Branch) with the Highest CPI.

123
FFF
FFT
FTF
FTT
TFF
TFT
TTF
TTT

In Conclusion

- Pipelining increases throughput by overlapping execution of multiple instructions
- All pipeline stages have same duration
  - Choose partition that accommodates this constraint
- Hazards potentially limit performance
  - Maximizing performance requires programmer/compiler assistance
- Superscalar processors use multiple execution units for additional instruction level parallelism
  - Performance benefit highly code dependent