Computer Architecture and Computer Organization

- Attributes of a system that are visible to the programmer.
- Have a direct impact on the logical execution of a program

Computer Architecture

Architectural attributes include:

- Instruction set, number of bits used to represent various data types, I/O mechanisms, techniques for addressing memory

Organizational attributes include:

- Hardware details transparent to the programmer, control signals, interfaces between the computer and peripherals, memory technology used

Computer Organization

- The operational units and their interconnections that realize the architectural specifications

1 Slide summary: Architecture vs Microarchitecture

Architecture – ISA – Programmers specification

Architecture

Programmer visible state (Memory & Register)

Operations (Instructions and how they work)

Data Types/Sizes

Execution Semantics (interrupts)

Transparent to programmer (Hardware details)

How to implement ISA (and tradeoffs on 3Ps)

- Pipeline depth,
- cache size
- Bus widths, ALU width,..

Microarchitecture

µarchitecture – HW designers specification

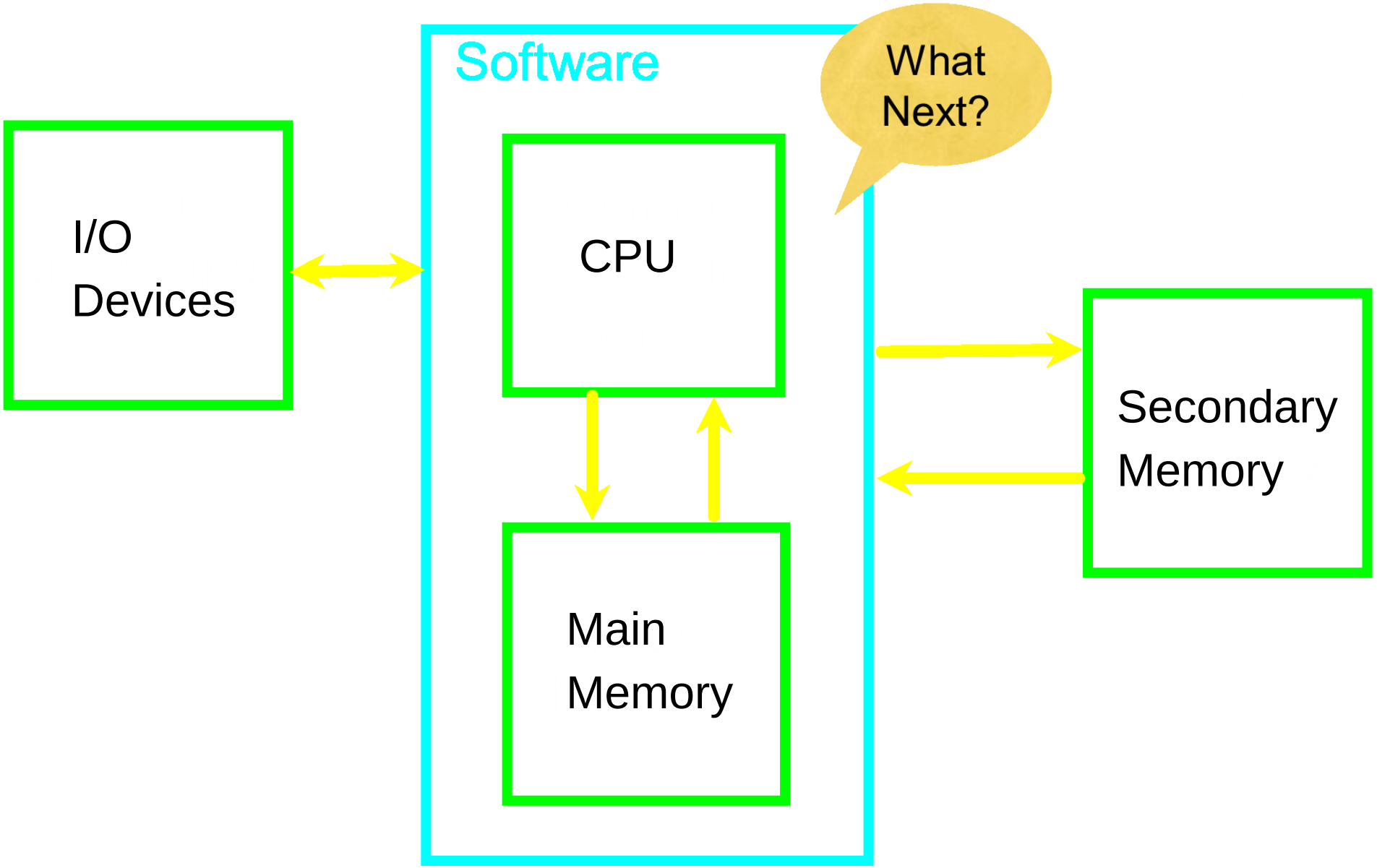
DIGRESSION

- ***How many of you have taken the Introduction to computing or any Equivalent programming course?***

- ***What is a Code? A Program?***

How does a program get executed?

How does a Program Execute?



Bit, Byte, Word and Sentences in Computer

Byte System		
Name (symbol)	Power of 10 Value (in bytes)	Power of 2 Value (in bytes)
kilobyte (kB)	10^3	2^{10}
megabyte (MB)	10^6	2^{20}
gigabyte (GB)	10^9	2^{30}
terabyte (TB)	10^{12}	2^{40}
petabyte (PB)	10^{15}	2^{50}
exabyte (EB)	10^{18}	2^{60}
zettabyte (ZB)	10^{21}	2^{70}
yottabyte (YB)	10^{24}	2^{80}

Examples and comparisons with SI prefixes	
one **kibibit**	1 Kibit = 2^{10} bit = **1024 bit**
one **kilobit**	1 kbit = 10^3 bit = **1000 bit**
one **byte**	1 B = 2^3 bit = **8 bit**
one **mebibyte**	1 MiB = 2^{20} B = **1 048 576 B**
one **megabyte**	1 MB = 10^6 B = **1 000 000 B**
one **gibibyte**	1 GiB = 2^{30} B = **1 073 741 824 B**
one **gigabyte**	1 GB = 10^9 B = **1 000 000 000 B**

Factor	Name	Symbol	Origin	Derivation
2^{10}	kibi	Ki	kilobinary: $(2^{10})^1$	kilo: $(10^3)^1$
2^{20}	mebi	Mi	megabinary: $(2^{10})^2$	mega: $(10^3)^2$
2^{30}	gibi	Gi	gigabinary: $(2^{10})^3$	giga: $(10^3)^3$
2^{40}	tebi	Ti	terabinary: $(2^{10})^4$	tera: $(10^3)^4$
2^{50}	pebi	Pi	petabinary: $(2^{10})^5$	peta: $(10^3)^5$
2^{60}	exbi	Ei	exabinary: $(2^{10})^6$	exa: $(10^3)^6$

Source: https://physics.nist.gov/cuu/Units/binary.html

https://groups.ischool.berkeley.edu/archive/how-much-info/datapowers.html
http://www.ccsf.caltech.edu/~roy/dataquan/

UNDERSTANDING PERFORMANCE

Chap 1.6 (COAHP –ed 5); Chap2: COAWS

- Which Vehicle has the best performance?

Automobile	Capacity (#persons)	Speed (Kmph)	Mileage (Kmpl)	Time to reach Mumbai (500Kms)	Throughput (persons/hr)	Fuel Efficiency (persons/ltr.)
Bike	2	50	80	10 hrs	0.2 pph	0.32 ppl
Car	5	100	30	5 hrs	1 pph	0.30 ppl
Bus	50	100	20	5 hrs	10 pph	2 ppl

- Time to do a task
  - Execution time/Response time/Latency
- Tasks performed per unit time (per day, hr, seconds)
  - Throughput
- Other Metrics:
  - Efficiency (Energy consumed to per task) etc.

Vehicle Throughput:
The rate at which the persons reach the destination in a given vehicle.

Response Time and Throughput

- Response time (Latency)
  - How long it takes to do a task.
  - Execution time; Cycles per Instruction (CPI); Avg. Memory access time (AMAT)

- Throughput
  - Total work done per unit time.
    - e.g., tasks/transactions/… per hour
    - Transactions per second (TPS); Million Instructions per second (MIPS)

- How are response time and throughput affected by
  - Replacing the processor with a faster version?
  - Adding more processors?

RELATIVE PERFORMANCE

- Define Performance = *Unit of Things done per unit time.* ***<Larger the better>***

$$Performance(x) = 1/Execution\ time(x)$$

- "X is n time faster than Y" *or* "X is n times as fast as Y" or "From Y to X the speedup is n times "

$$\text{Performance}_X / \text{Performance}_Y$$
$$= \text{Execution time}_Y / \text{Execution time}_X = n$$

- Example: time taken to run a program (*same program on different machines*)
  - *Program A* takes ***10s*** on Machine ***X***, and ***15s*** on Machine ***Y***
  - Execution Time(***Y***) / Execution Time(***X***)
    = 15s/10s = 1.5
  - So ***X*** is 1.5 times faster than ***Y***; Speedup(***X/Y***) = 1.5

How do we measure the Execution time of a Program?

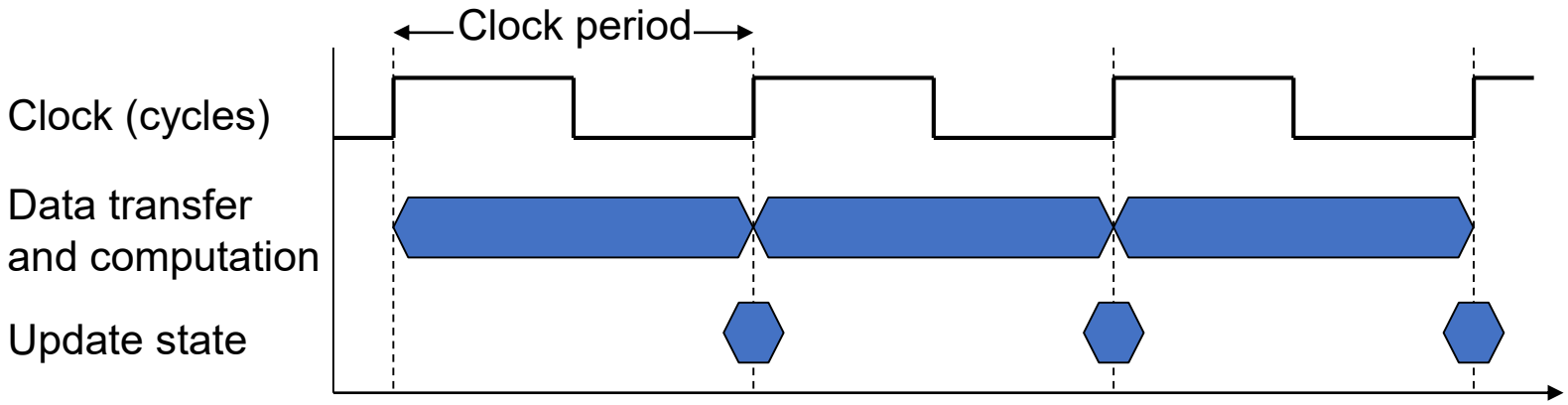
Measuring Execution Time

- **Elapsed time**
  - Total response time, including all aspects in the system.
    - Processing, I/O, OS overhead, idle time
  - Determines the overall system performance

- **CPU time**
  - Time spent processing a given job
    - Discounts I/O time; other jobs' share of elapsed time
  - CPU time comprises of two components (i) user CPU time and (ii) system CPU time
    - ***User CPU time***: Time spent in the user program (all the processing in the user space).
    - ***System CPU time***: Time spent in the system (OS) for doing tasks on behalf of user program.
  - Different programs are affected differently by CPU and system performance

CPU Clocking

- Operation of digital hardware is governed by a constant-rate clock



- Clock period: duration of a clock cycle
  - e.g., 250ps = 0.25ns = 250×10^{-12}s
- Clock frequency (rate): cycles per second
  - e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

https://www.intel.in/content/www/in/en/gaming/resources/cpu-clock-speed.html

CPU Time – More fine-grained view

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

- Performance can be improved (Lower the CPU Time) by:
  - Reducing number of CPU clock cycles (good algorithm or hardware design)
  - Increasing clock rate (good technology) or Reducing the Clock Cycle Time

Hardware Designer's Perspective

Let's say "Computer A": with 2GHz clock, takes 10s CPU time

- Goal: Designing a Faster "Computer B" that would take only 6s CPU time.
  - Target for 6s CPU time
  - Consider that Computer B causes 20% increase in clock cycles. (i.e. 1.2 × clock cycles(A))

- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$

$$= 10\text{s} \times 2\text{GHz} = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$$

Instruction Count, CPI and MIPS

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- Instruction Count (IC) for a program
  - *Is determined by program, ISA and compiler*
- Average cycles per instruction (CPI)
  - *Is determined by CPU hardware*

$$MIPS = Instruction\ Count/(\ Execution\ Time \times 10^6)$$

- If *different instructions* have *different CPI*
  - *Average CPI is affected by instruction mix*

CPI in More Detail

- If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

Influence on CPI – Compiler and Programmers Choice

- Consider compiled code sequences using instructions in classes A, B, C
- On two machines with Clock Rate of 1Ghz

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

- Sequence 1: IC = 5
  - Clock Cycles
    = 2×1 + 1×2 + 2×3
    = 10
  - Avg. CPI = 10/5 = 2.0

Ex Time = (5)*(2)*(10^-9)s = 10ns

MIPS = (1x10^9)/(2)*(10^-6) = 500 MIPS

- Sequence 2: IC = 6
  - Clock Cycles
    = 4×1 + 1×2 + 1×3
    = 9
  - Avg. CPI = 9/6 = 1.5

Ex Time = (6)*(1.5)*(10^-9)s = 9ns

MIPS = (1x10^9)/(1.5)*(10^-6) = 667MIPS

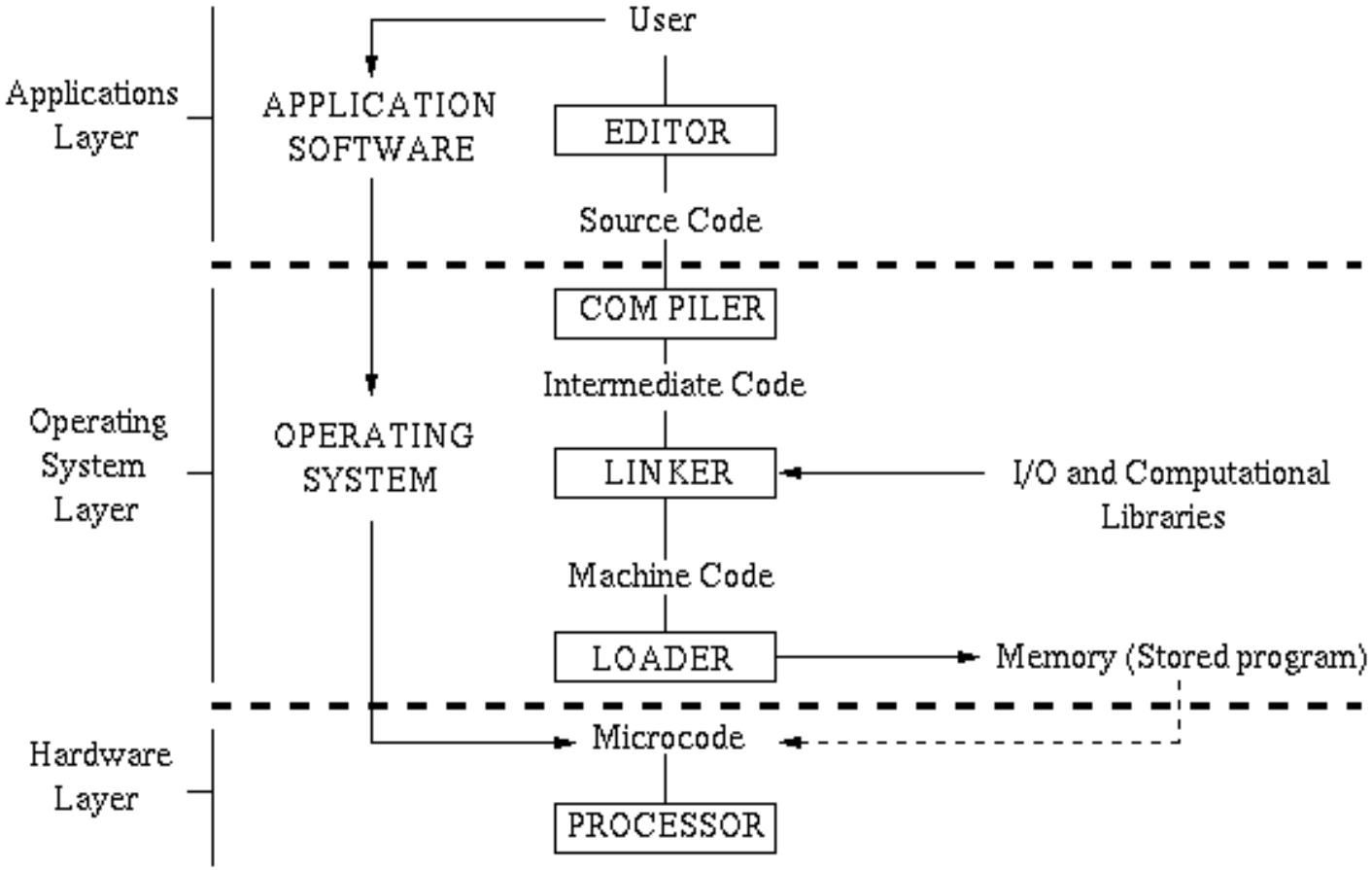
The BIG Picture

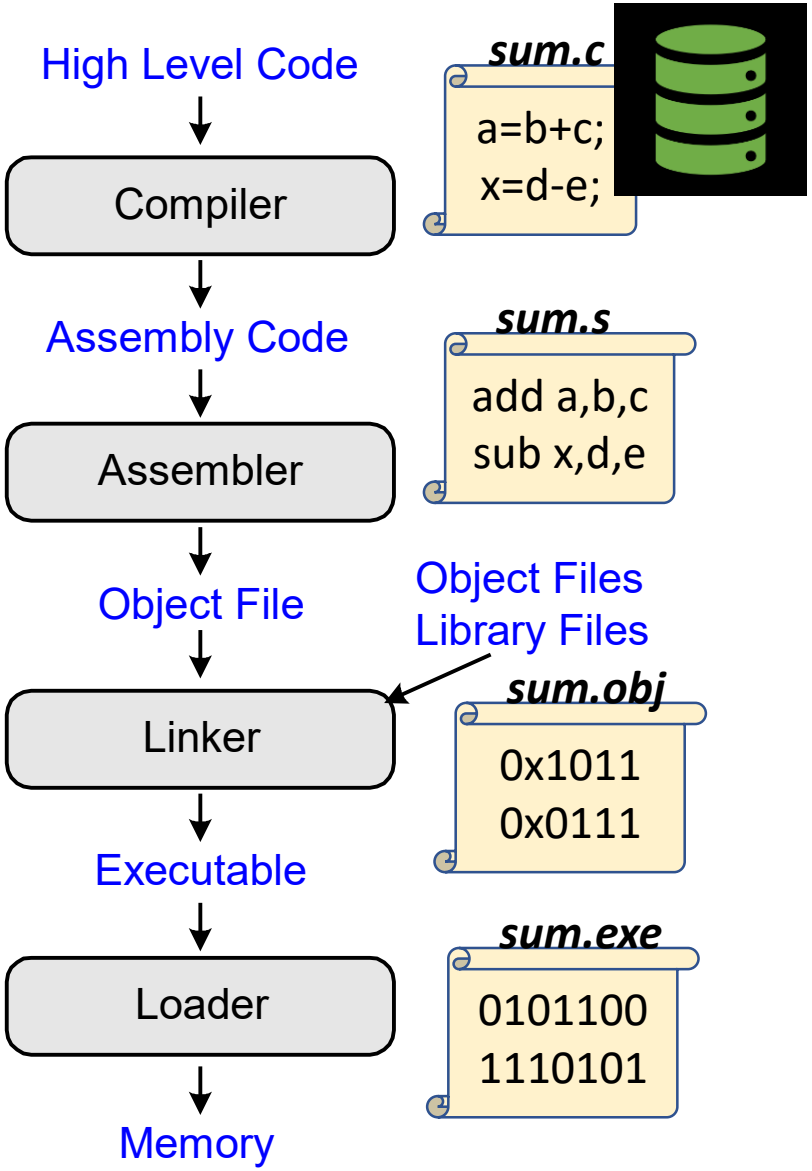
$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

$$\text{CPU Time} = IC \times CPI \times T_c$$

- Performance depends on different aspects:
  - Algorithm: affects Instruction count (IC), possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c
  - Computer Organization: affects CPI, T_c

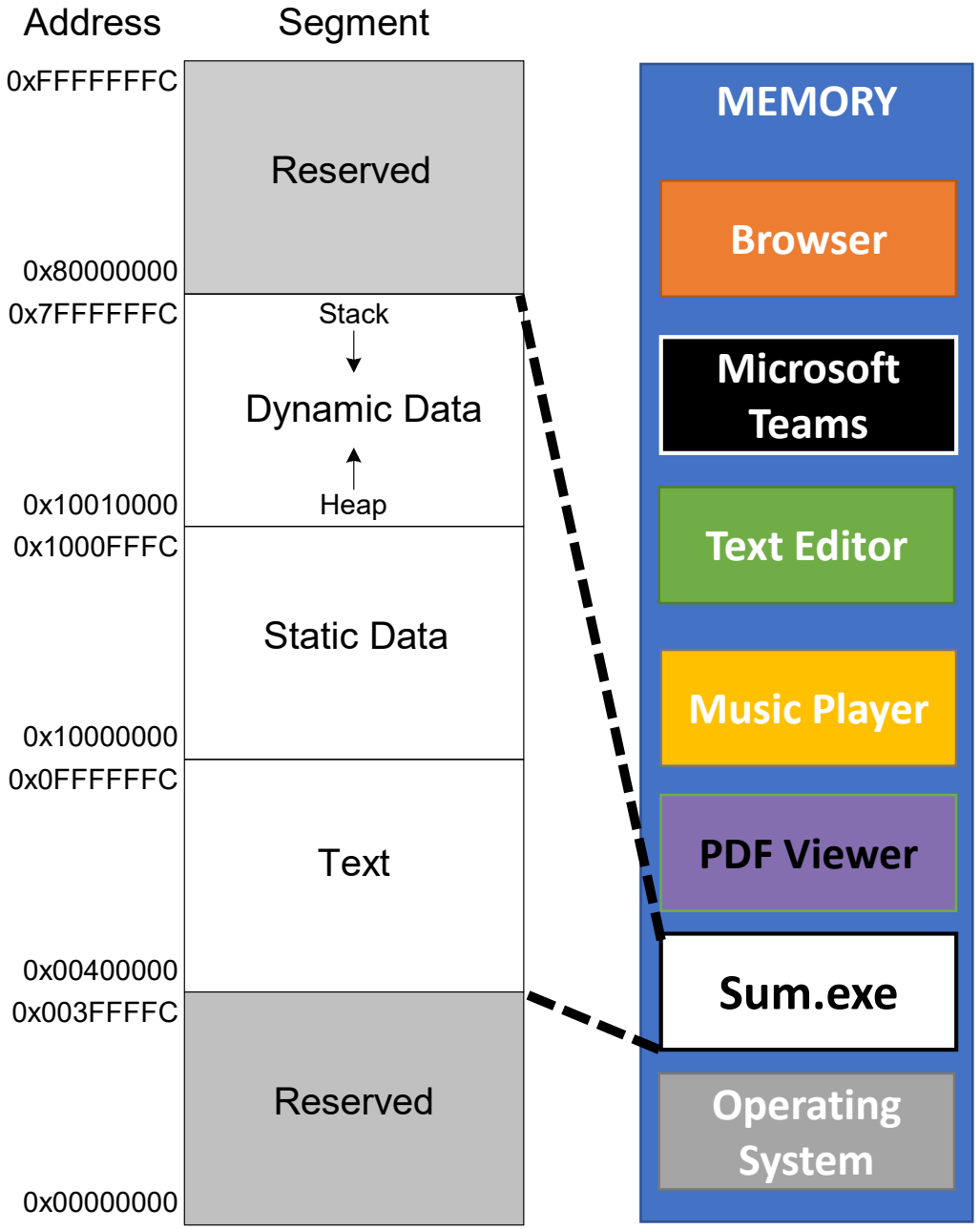
HW-SW INTERFACE





Memory Map/Layout of a Program

- Instructions (also called *text*)
  - Set of Instructions to be executed.
- Data
  - Global/static: allocated before program begins
  - Dynamic: allocated within program
  - Stack: Intermediate/temporary store
- Everything is stored in Memory
  - How big is the memory?
  - What is the address range?
  - How to access data in this memory?



Example Program: C Code to Assembly Map

```c
int x, y, z;  // global variables
int sum(int a, int b) {
  return (a + b);
}
int main(void) {
  x = 2;
  y = 3;
  z = sum(x, y);
  return z;
}
```

Symbol	Address
X	0x10000000
Y	0x10000004
Z	0x10000008
Main	0x00400000
sum	0x0040002C

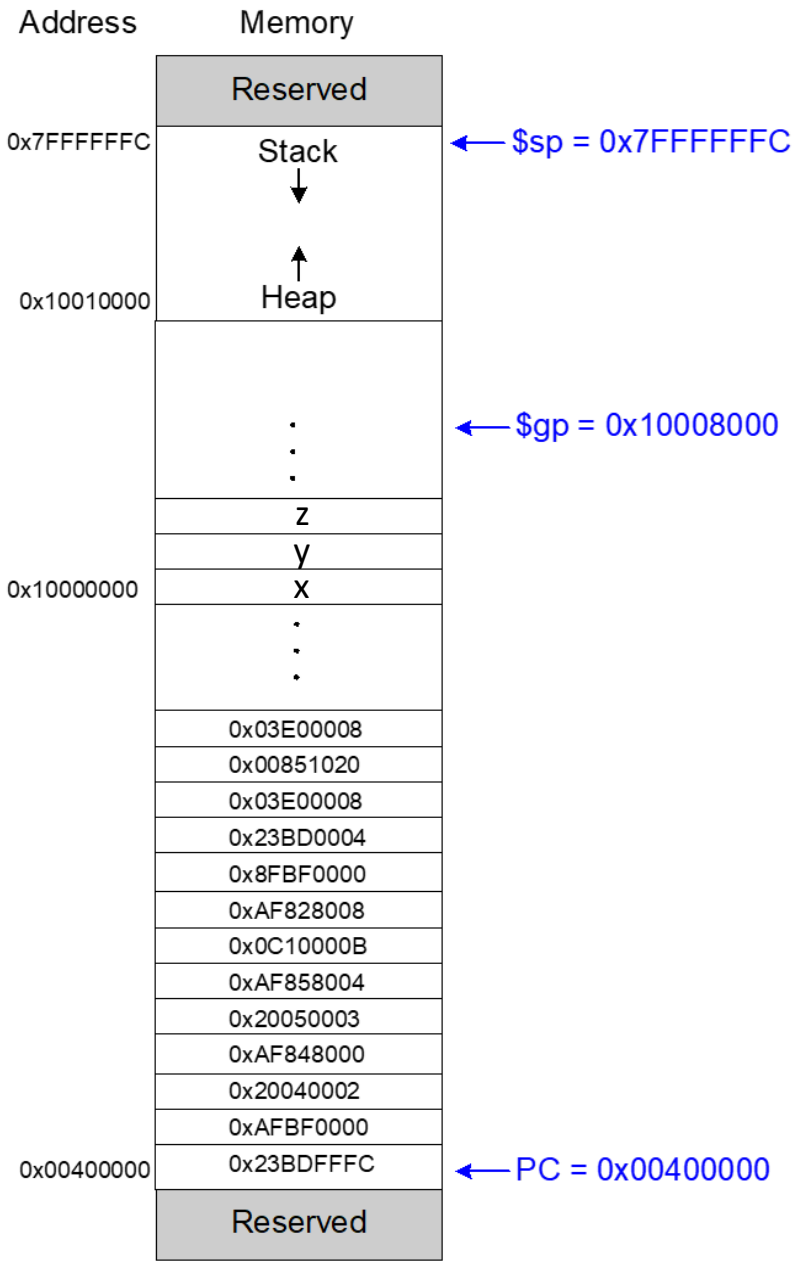
```asm
.data
x:
y:
z:
.text
main:
  addi $sp, $sp, -4    # stack frame
  sw   $ra, 0($sp)     # store $ra
  addi $a0, $0, 2      # $a0 = 2
  sw   $a0, x          # x = 2
  addi $a1, $0, 3      # $a1 = 3
  sw   $a1, y          # y = 3
  jal  sum             # call sum
  sw   $v0, z          # z = sum()
  lw   $ra, 0($sp)     # restore $ra
  addi $sp, $sp, 4     # restore $sp
  jr   $ra             # return to OS
sum:
  add  $v0, $a0, $a1   # $v0 = a + b
  jr   $ra             # return
```

Example Program: Executable

Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)

Text segment	Address	Instruction	
	0x00400000	0x23BDFFFC	addi $sp, $sp, -4
	0x00400004	0xAFBF0000	sw $ra, 0 ($sp)
	0x00400008	0x20040002	addi $a0, $0, 2
	0x0040000C	0xAF848000	sw $a0, 0x8000 ($gp)
	0x00400010	0x20050003	addi $a1, $0, 3
	0x00400014	0xAF858004	sw $a1, 0x8004 ($gp)
	0x00400018	0x0C10000B	jal 0x0040002C
	0x0040001C	0xAF828008	sw $v0, 0x8008 ($gp)
	0x00400020	0x8FBF0000	lw $ra, 0 ($sp)
	0x00400024	0x23BD0004	addi $sp, $sp, -4
	0x00400028	0x03E00008	jr $ra
	0x0040002C	0x00851020	add $v0, $a0, $a1
	0x00400030	0x03E00008	jr $ra

Data segment	Address	Data
	0x10000000	x
	0x10000004	y
	0x10000008	z

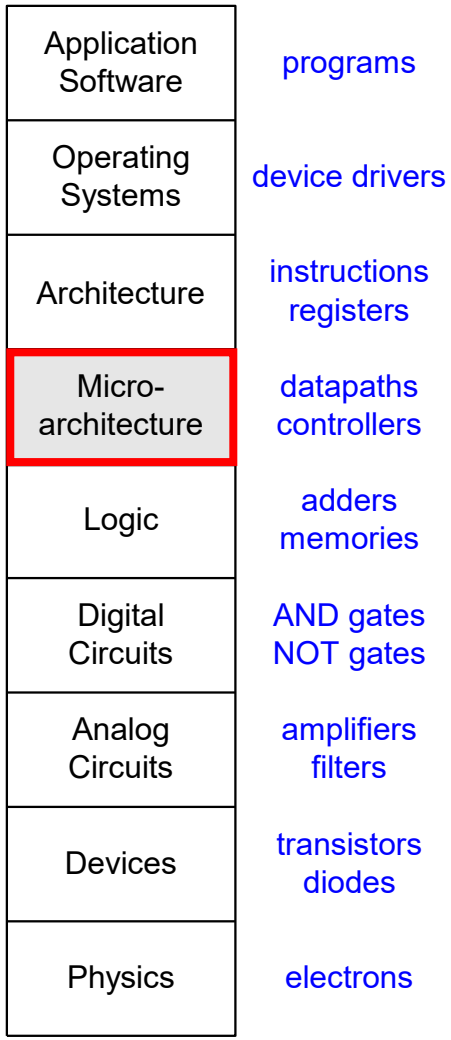


PROCESSOR DESIGN – A MICROARCHITECTURAL CONCEPT

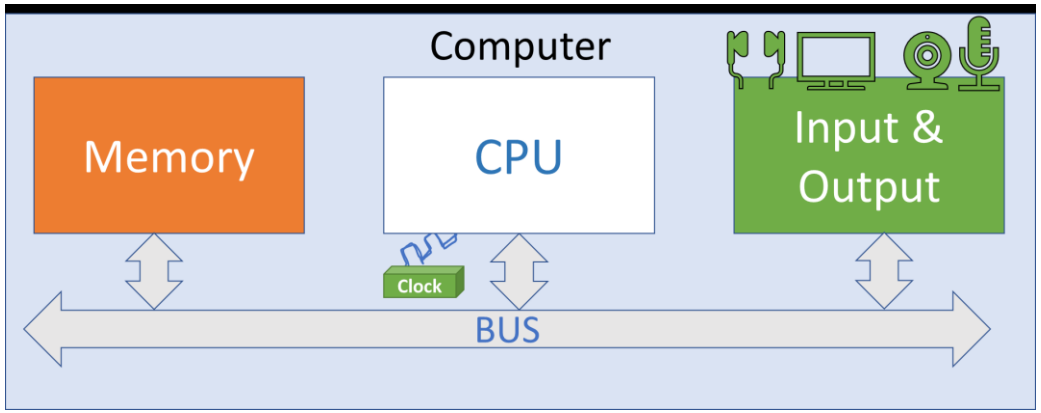
- **Microarchitecture:** *how to implement an architecture in hardware*

- Processor:
  - **Datapath:** functional blocks
  - **Control:** control signals

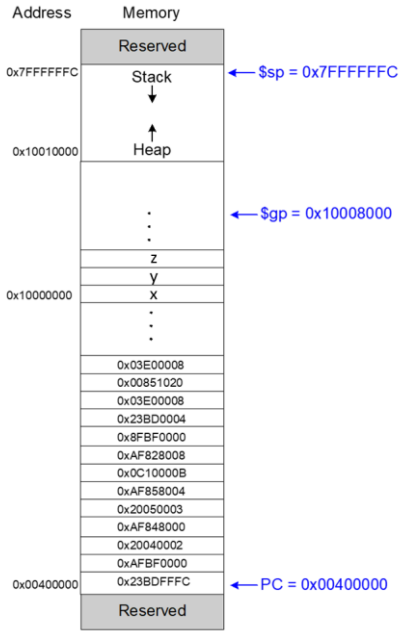
- Multiple implementations for a single architecture:
  - **Single-cycle:** Each instruction executes in a single cycle
  - **Multicycle:** Each instruction is broken into series of shorter steps and executed over multiple cycles.
  - **Pipelined:** Each instruction broken up into series of steps & multiple (different) instructions can execute at once.

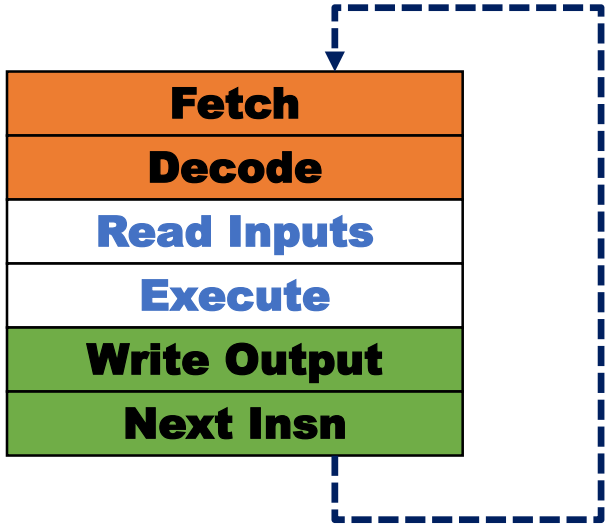


Instruction Execution Model



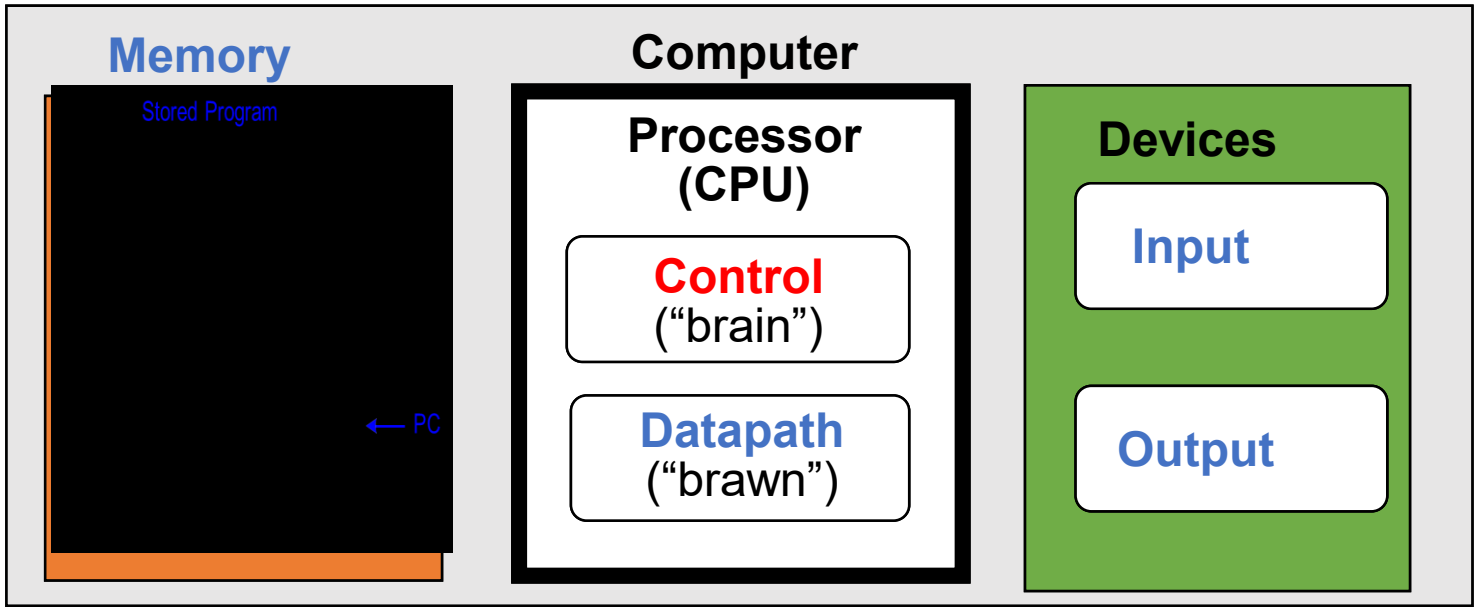
Fetch → **Decode** → **Execute** "cycle"





- **Program is just "data in memory"**
  - Makes computers programmable ("universal")
- The computer is just finite state machine
  - **Registers** (few of them, but fast)
  - **Memory** (lots of memory, but slower)
  - **Program counter** (next instruction to execute)
- A computer executes **instructions**
  - **Fetches** next instruction from memory
  - **Decodes** it (figure out what it does)
  - **Reads** its **inputs** (registers & memory)
  - **Executes** it (adds, multiply, etc.)
  - Write its **outputs** (registers & memory)
  - **Next insn** (adjust the program counter)

DATA PATH AND CONTROL PATH



Fetch → **Decode** → **Execute** "cycle"

Datapath: consists of all the elements in a processor that are dedicated to storing, retrieving, and processing data such as register files, memory, and the ALU

Datapath performs computation

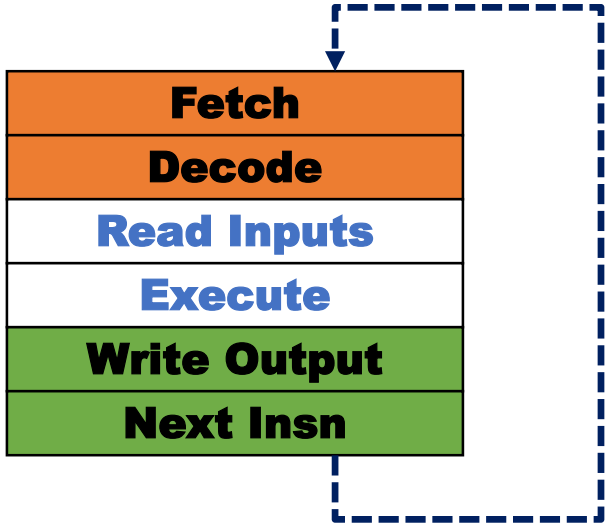
ISA specific: can implement every instruction (single-cycle: in one pass!)

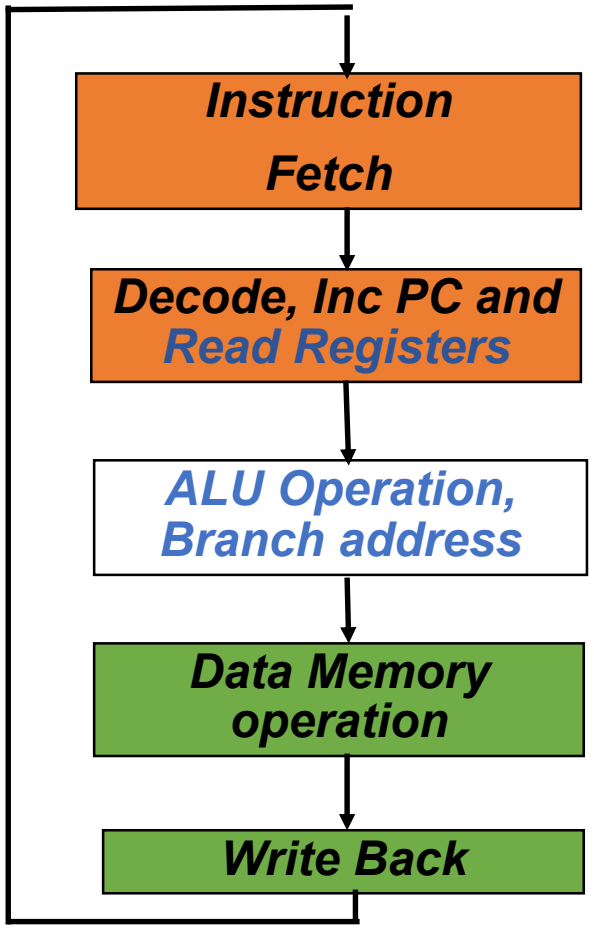
Control path: control unit that generates the appropriate signals to control the movement of data and execution of instructions in the data path.

determines which computation is performed

Routes data through the data path (which regs, which ALU operation)

Typical Instruction execution Steps





Steps involved in MIPS Instruction Execution

Step1: Instruction Fetch

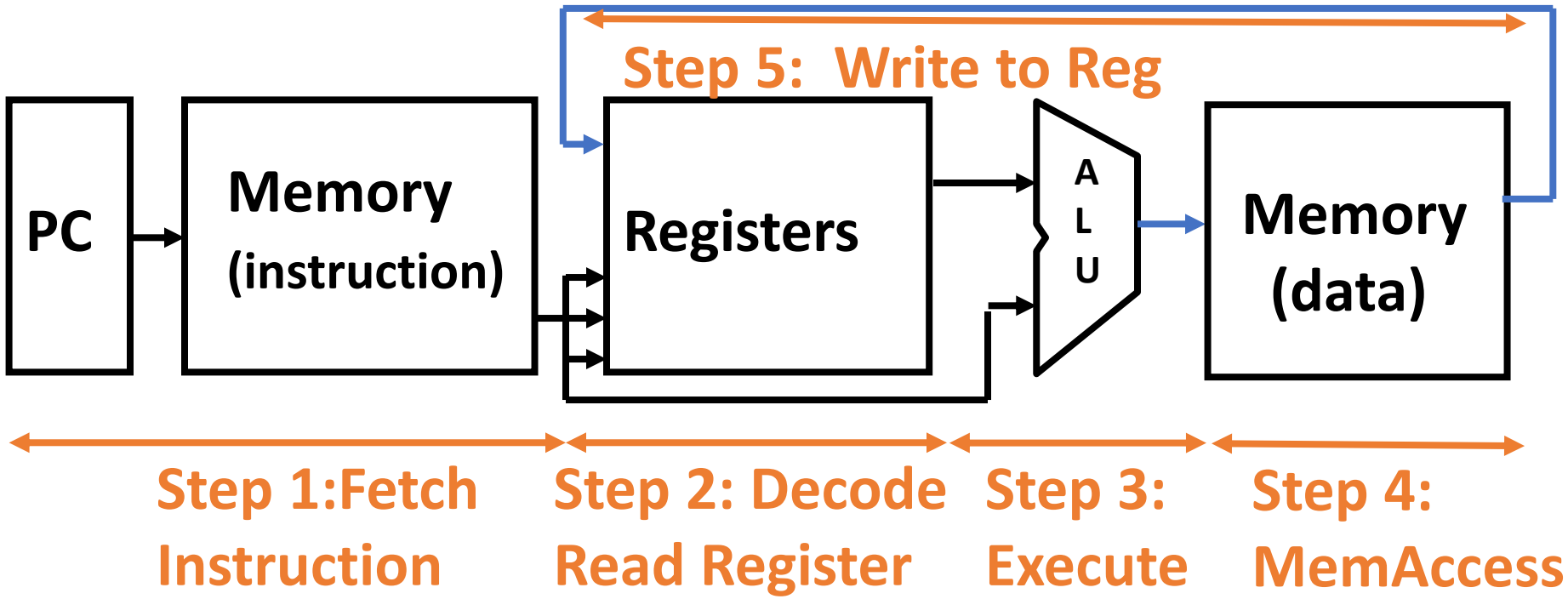
Step2: Instruction Decode + Read Register + *Inc PC*

Step3: Execute Instruction (ALU Operation)

Step4: Data Memory Operation

Step5: Write back (to register)

High Level View of 5 Steps MIPS Instruction Execution

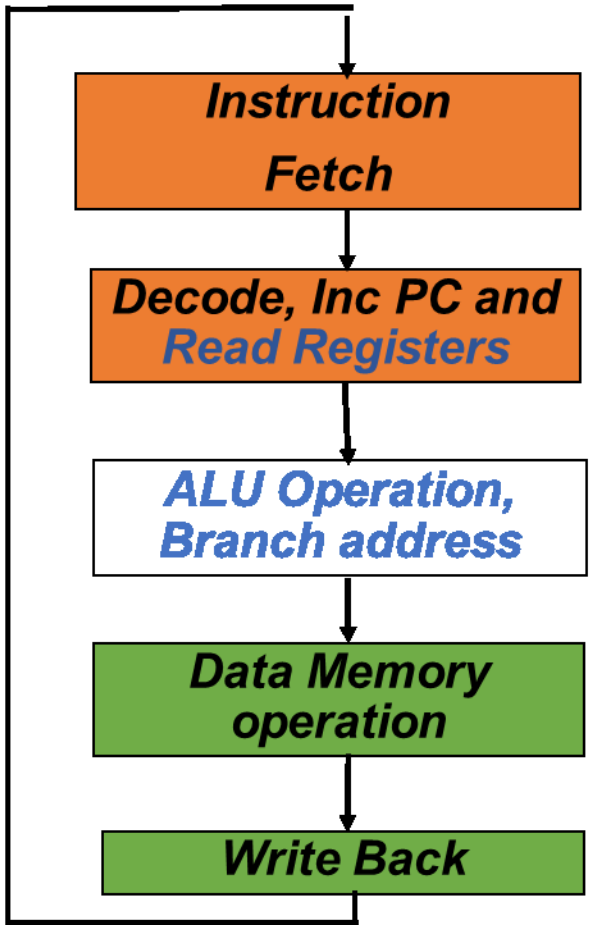



```
1000:  Start: add $t0, $t1, $t2
1004:         beq $t0, $t1, Start
1008:         addi $t2, $t2, 2
1012:         lw $t1, 20($s0)
1016:         beq $t0, $t1, Redo
1020:         sw $t1, 4($t0)
```

SINGLE-CYCLE MACHINE: APPRAISAL

- All instructions complete in one clock cycle (CPI = 1)
- Some instructions take more steps than others
  - lw is most expensive (5 steps), vs. sw (4 steps)
  - R-type instructions take 4 for R-type
  - 3 steps for `beq`

- Clock cycle must cover longest instruction ⇒ inefficient
  - suppose `mul` is added?
  - 32-shift/add steps ⇒ would delay every other instruction



EXAMPLE QUESTIONS

- Assume 2ns for instruction/data memory, 1ns for decode/register read, 2ns for ALU and 1 ns for register write.

Single-cycle datapath clock period = ?

I-type	IF	ID	EX	Mem	WB	Total
R-type						
LW (I)						
SW (I)						
BR (I)						

- Assume memory read takes 6ns, memory write takes 10ns, register read/write takes 2ns, ALU operation takes 2ns and rest 1ns. Compute the best possible clock frequency for this single cycle MIPS processor.

Clock Frequency = ?

MULTICYCLE IMPLEMENTATION

- *Change: Each **step** will take one clock cycle (not each instruction) [CPI > 1]*

⇒ shorter clock cycle: cycle time constrained by longest *step*, not longest insn.

- simpler instructions take fewer cycles

⇒ higher overall performance

- complex control: finite state machine

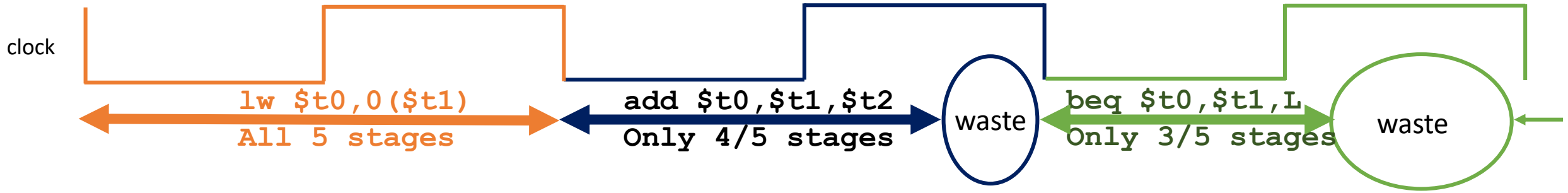
- Datapath changes
  - one memory: both instructions and data (because can access on separate steps)
  - one ALU (eliminate extra adders)
  - extra "invisible" registers to capture intermediate (per-step) datapath results
- Controller changes
  - controller must fire control lines in correct *sequence and correct time*

  ⇒controller must remember current execution step, advance to next step

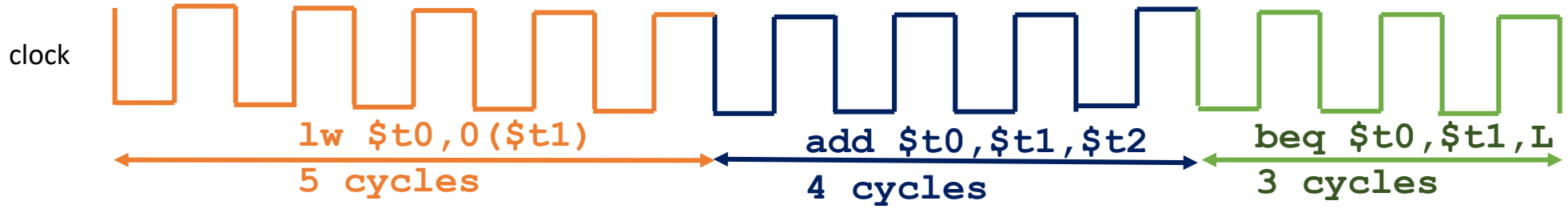
CLOCKING: SINGLE-CYCLE VS. MULTICYCLE PROCESSOR

I1: lw $t0, 0($t1)
I2: add $t0, $t1, $t2
I3: beq $t0, $t1, Lab

Single-cycle Implementation



Multicycle Implementation



- **Multicycle Implementation ➔ less waste & higher performance (*most often*)**

Summary: Single Cycle vs Multicycle Processor Design

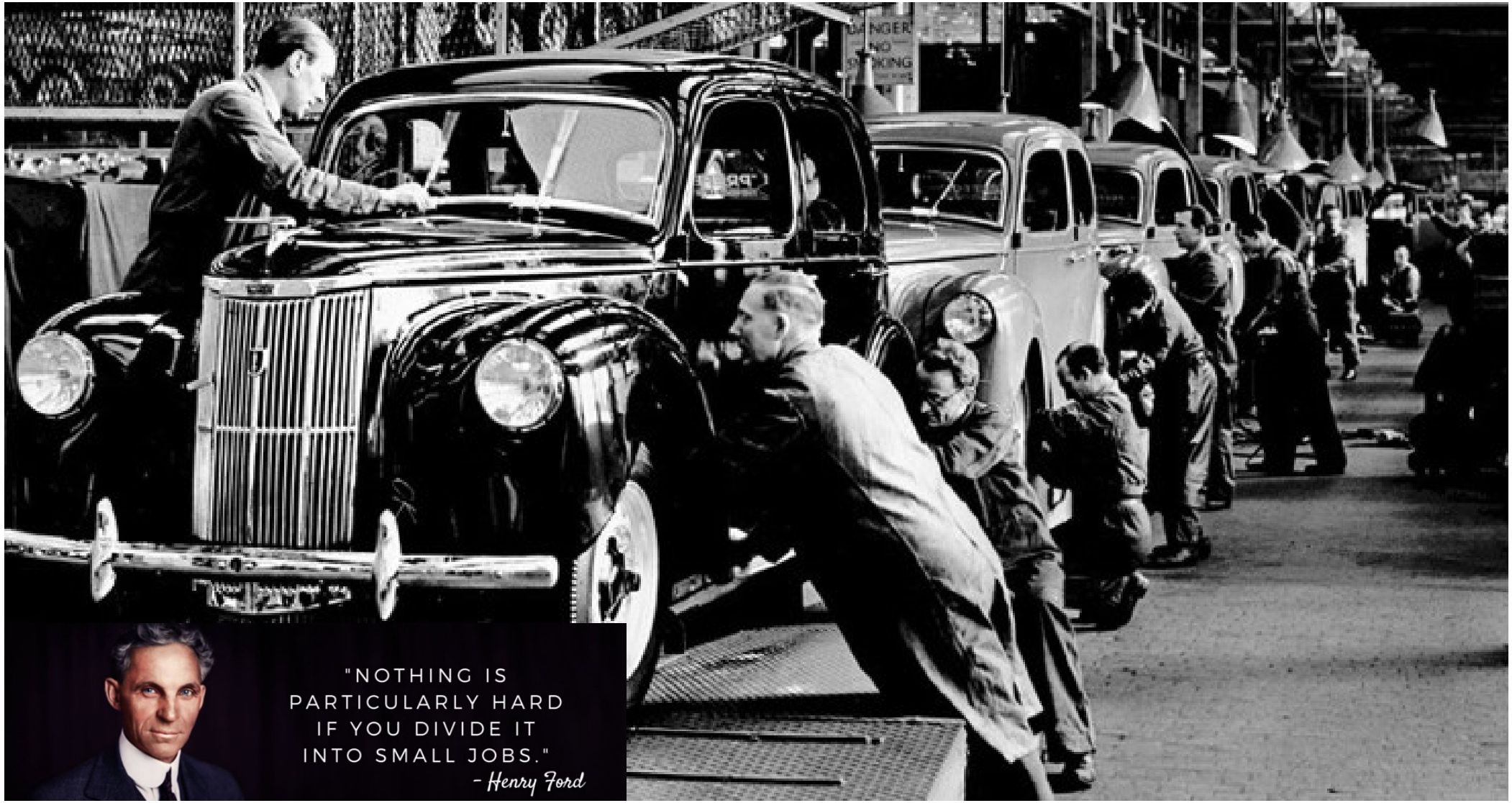
Single Cycle Design

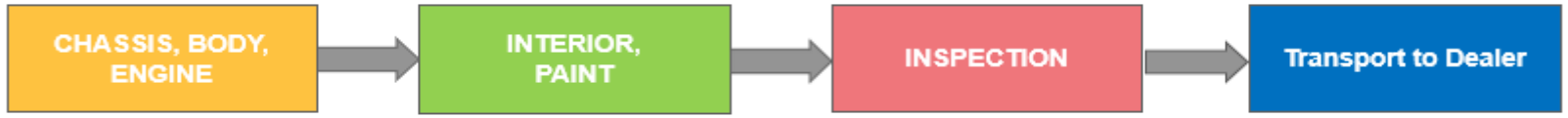
- All instructions execute in single cycle.
  - CPI = 1
  - Instruction execution may still be carried out in multiple states (5/6 stages) all within one cycle.
- Clock cycle is determined by the longest execution time of the instruction.
- Different Instruction types need different amount of time
  - *Results in wastage of portion of cpu cycles*
- Control unit is simple and can generate all control signals for entire instruction at once.

Multi-Cycle Design

- All instructions execute in 3-5 cycles
  - 3 cycles: `beq`
  - 4 cycles: R-type, `sw`
  - 5 cycles: `lw`
- Clock cycle is determined by the critical steps in the instruction execution.
- Allows to reuse some of the hardware blocks.
- Control unit is complex, needs to track current state (FSM) to generate appropriate control signals.
- Demands extra memory elements to keep track of intermediate state.

Henry Ford's Big Idea: Assembly Line (Pipelined Assembly)





Pipelining Instruction Execution

Similar to the use of an assembly line in a manufacturing plant/Laundry

New inputs are accepted at one end before previously accepted inputs appear as outputs at the other end

To apply this concept to instruction execution:

We must recognize that an instruction has a number of stages.

Stage$_{(i)}$ is the Producer and Stage$_{(i+1)}$ is the Consumer

Stages of Execution in Pipelined MIPS

5 stage instruction pipeline

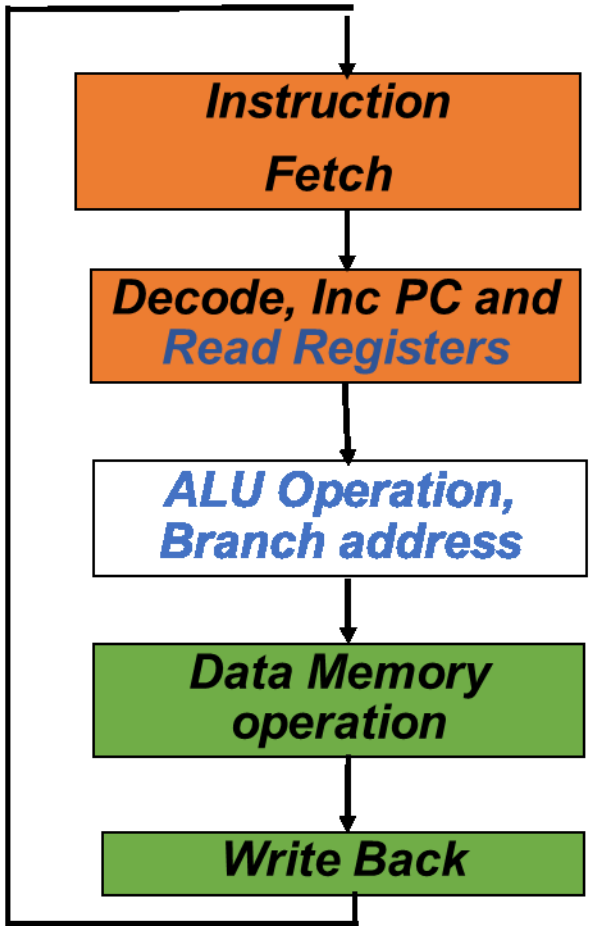
1) I-fetch: Fetch Instruction, Increment PC

2) Decode: Instruction, Read Registers

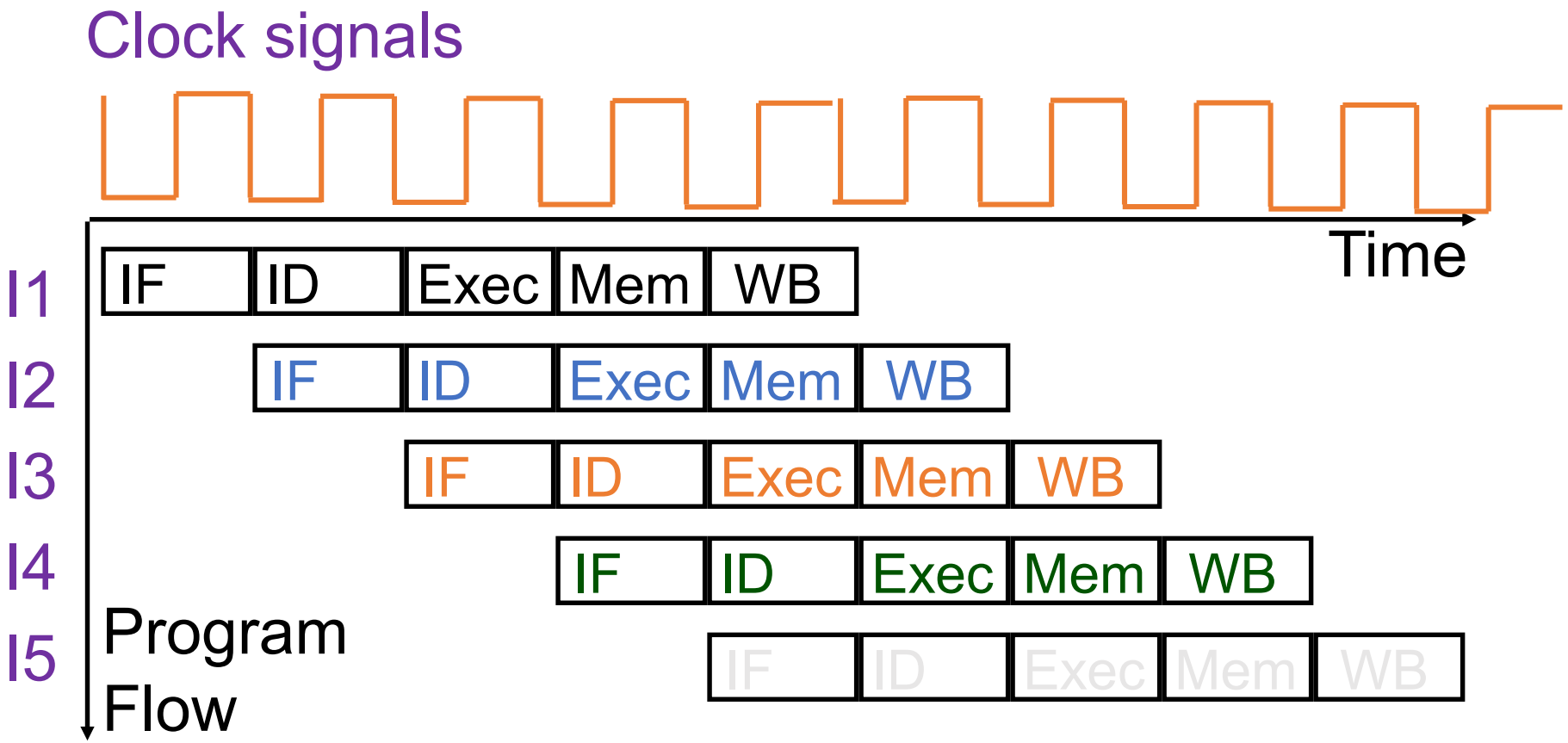
3) Execute:
   Mem-reference: Calculate Address
   R-format: Perform ALU Operation

4) Memory:
   Load: Read Data from Data Memory
   Store: Write Data to Data Memory

5) Write Back: Write Data to Register



Pipeline Diagram



- To simplify pipeline, every instruction takes same number of steps, called stages
- One clock cycle per stage

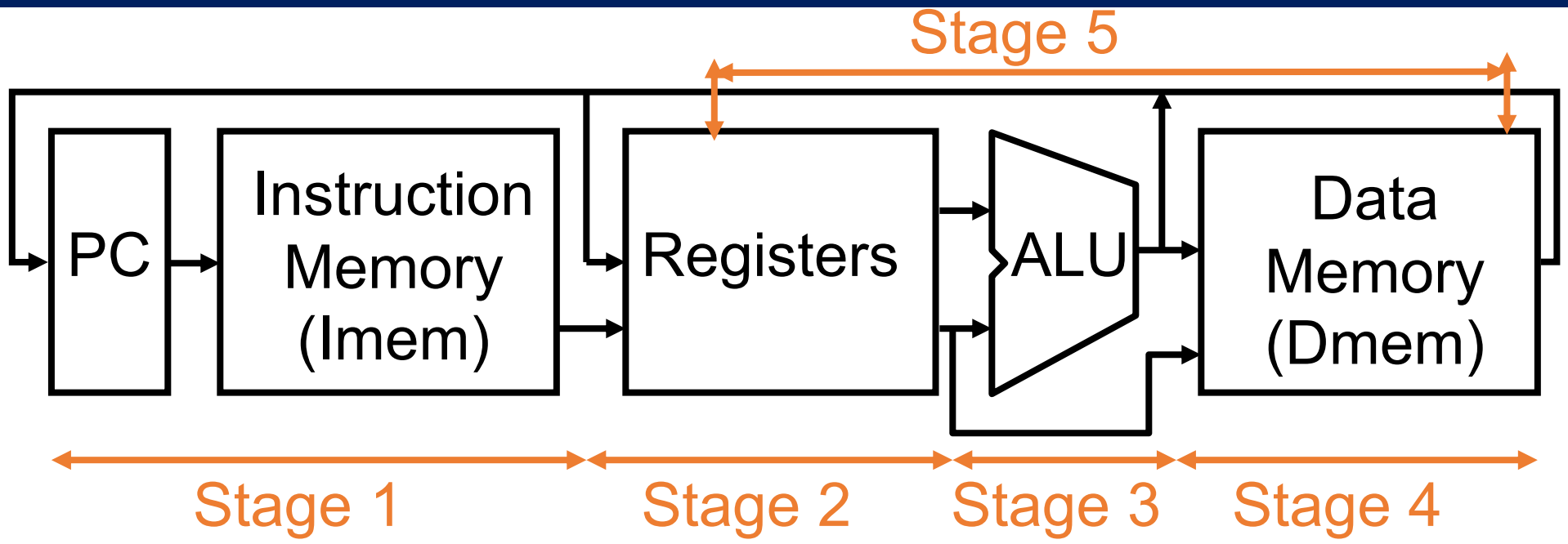
Advantage of Pipelining

- We keep all parts of the data path, busy all the time
- Let us assume that all the 5 stages do the same amount of **work**
  - Without pipelining, every T seconds, an instruction completes its execution
  - With pipelining, every T/5 seconds, a new instruction completes its execution
    - Throughput vs. Latency of Instruction Execution?
- CPI for a single cycle processor = 1; Multi-cycle processor = *k (3,4,or 5)*
- CPI for an ideal pipeline(no hazards)?
  - Assume we have *n* instructions, and *k* stages
  - The first instruction enters the pipeline in cycle 1
  - It leaves the pipeline in cycle k
  - The rest of the **(n-1) instructions** leave in the next (n-1) consecutive cycles

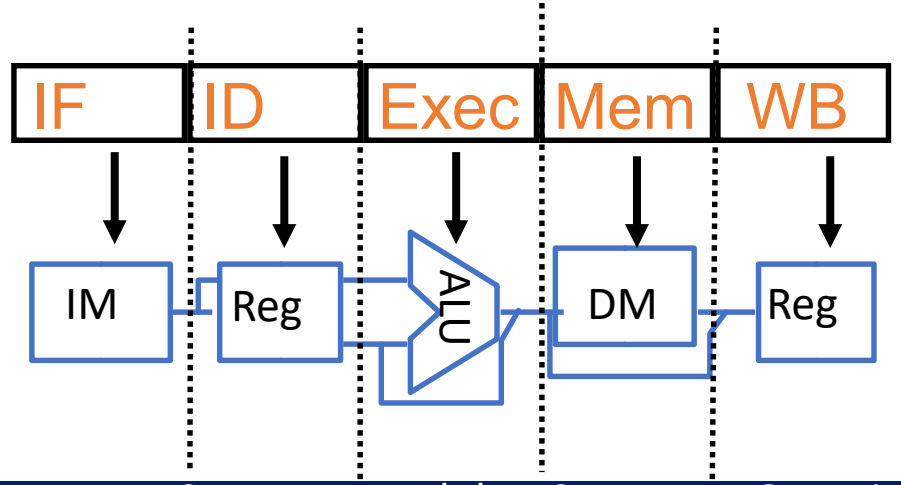
$$CPI_{pipelined} = \frac{n + k - 1}{n}$$

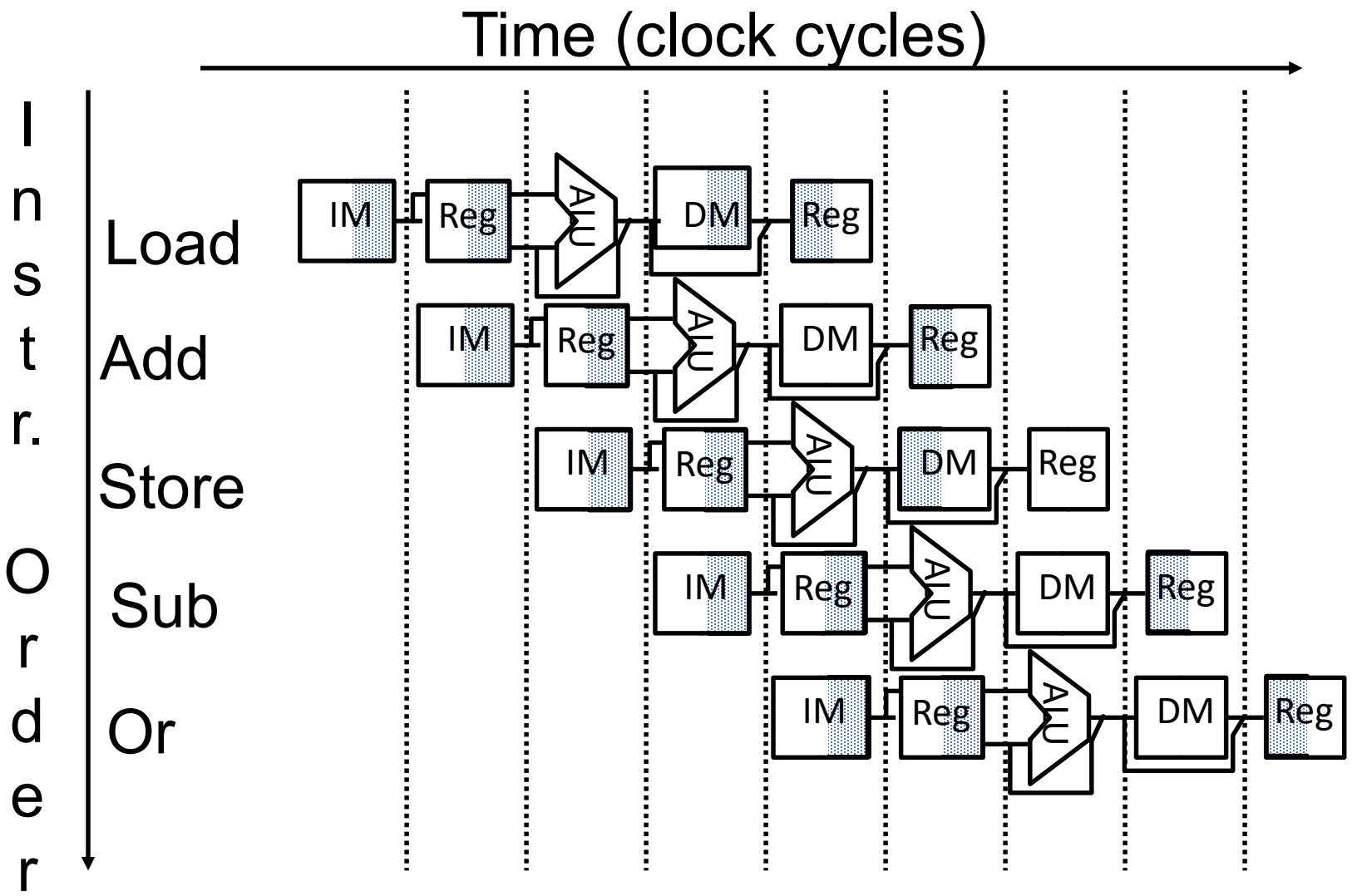
Time per stage → t_{max} / k

Review: Single-cycle Datapath for MIPS



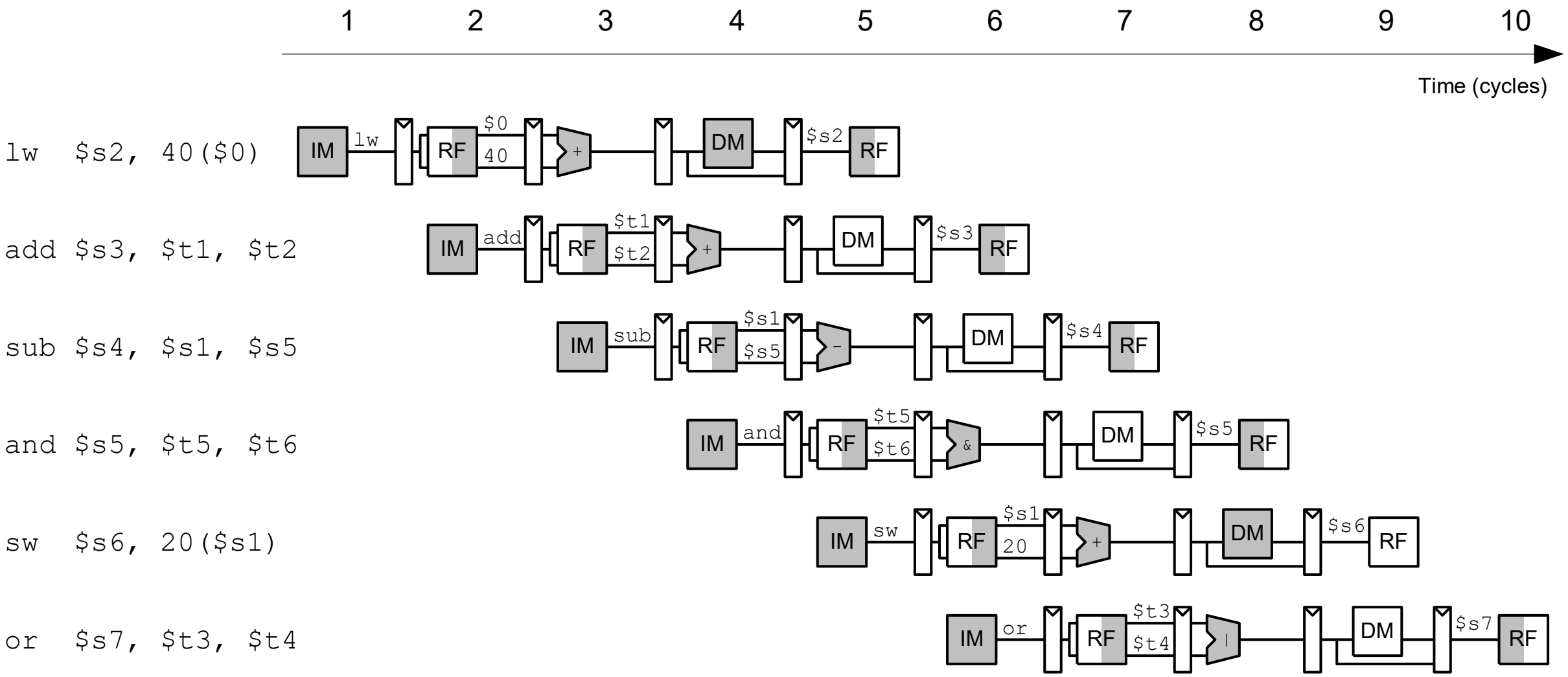
- Use datapath figure to represent pipeline





(right half highlighted means read, left half write)

PIPELINING


1
2
3
4
5
6
7
8
9
10
Time (cycles)
lw $s2, 40($0)
add $s3, $t1, $t2
sub $s4, $s1, $s5
and $s5, $t5, $t6
sw $s6, 20($s1)
or $s7, $t3, $t4
IM
RF
DM
lw
add
sub
and
sw
or
$0
40
$t1
$t2
$s1
$s5
$t5
$t6
20
$t3
$t4
$s2
$s3
$s4
$s5
$s6
$s7
+
-
&
|

PIPELINE DIAGRAM

- **Pipeline diagram**: shorthand for what we just saw
  - Across: cycles
  - Down: insns
  - assuming no *stalls* (discussed later)

	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,8($5)`		F	D	X	M	W			
`sw $6,4($7)`			F	D	X	M	W		

EXAMPLE PIPELINE PERF. CALCULATION

- Single-cycle
  - Clock period = 50ns, CPI = 1
  - Performance = 50ns/insn
- 5-stage pipelined
  - Clock period = **12ns** approx. (50ns / 5 stages) + overheads
  - \+ CPI = **1** (each insn takes 5 cycles, but 1 completes each cycle)
    - \+ Performance = **12ns/insn**
  - – Well actually … CPI = 1 + some penalty for pipelining (next)
    - CPI = **1.5** (on average insn completes every 1.5 cycles)
    - Performance = **18ns/insn**
    - Much higher performance than single-cycle

Q1: Why Is Pipeline Clock Period …

- … > (delay thru datapath) / (number of pipeline stages)?
  - Three reasons:
    - Registers add delay
    - Pipeline stages have different delays, clock period is **max** delay
    - Extra datapaths for pipelining (bypassing paths)
  - These factors have implications for ideal number pipeline stages
    - Diminishing clock frequency gains for longer (deeper) pipelines

PIPELINE SPEEDUP

- If all stages are balanced
  - i.e., all take the same time

$$\text{Speedup}_{(\text{pipelined})} = \frac{Time\ between\ Instructions_{\ nonpipelined}}{Number\ of\ pipeline\ stages}$$

- If not balanced, speedup would be less.

Note: Speedup is due to increased throughput:

- Latency (time for each instruction) does not decrease
- In fact, it might increase…

Science is always wrong. It never solves a problem without creating ten more.

George Bernard Shaw

Dependences and Hazards

- **Dependence**: relationship between two instructions.
  - **Data**: two instructions use same storage location
  - **Control**: one instruction affects whether another executes or not
  - Not a bad thing, programs would be boring without them
  - Enforced by making older instruction go before younger one
    - Happens naturally in single-/multi-cycle designs

- **Hazard**: dependence & possibility of wrong instruction order
  - Effects of wrong instruction order must not be externally visible
    - **Stall**: for order by keeping younger instruction in same stage
  - Hazards are a bad thing: stalls reduce performance

Data Dependencies

- **Flow dependence**
  - READ after WRITE (RAW) (***True/Value dependence***)
  - Read from a register after a previous instruction wrote into the register

- **Anti Dependence**
  - WRITE after READ (WAR)
  - Write to a register after a previous instruction read from the register

- **Output dependence**
  - WRITE after WRITE (WAW)
  - Write to a register after a previous instruction wrote into the register

Pipelining Hazards

- Hazards prevent next instruction from executing during its designated clock cycle, thus limiting the speedup
  - Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - Control hazards: conditional branches & other instructions may stall the pipeline delaying later instructions (must check detergent level before washing next load)
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (matching socks in later load)

- **Structural Hazard**
  - When there is a resource conflict or some functional unit is not accessible.

```
load t0, 0(t1)
add t2 t3 t4
sub t5 t6 t7
and t8 t9 t0
```

- *Consider Memory unit has* ***single Read port***

Instr	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
Lw	IF	ID	EX	MEM	WB			
Add	-	IF	ID	EX	MEM	WB		
Sub			IF	ID	EX	MEM	WB	
And				IF	ID	EX	MEM	WB

- **Structural Hazard**
  - When there is a resource conflict or some functional unit is not accessible.

```
load t0, 0(t1)
add t2 t3 t4
sub t5 t6 t7
and t8 t9 t0
```

- *Consider Memory unit has* ***single Read port***
- **Stall the pipeline →** *Results in bubbles across stages*

Instr	Cycle 1	Cycle2	Cycle 3	Cycle4	Cycle5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
Lw	IF	ID	EX	MEM	WB				
Add	-	IF	ID	EX	MEM	WB			
Sub			IF	ID	EX	MEM	WB		
STALL				Bubble	Bubble	Bubble	Bubble	Bubble	
And				STALL	IF	ID	EX	MEM	WB

DATA HAZARD CLASSIFICATION

- **Flow dependence --- RAW Hazard**
  - READ after WRITE (RAW) (***True/Value dependence***)
  - [Req] Read from a register after a previous instruction wrote into the register
  - RAW Hazard occurs if read by (I_{n+1}) occurs before write by (I_n) – "*read too soo*

```
I1: ld t0, 100(t1)
I2: add t2, t2, t0
I3: sw t2 100(t1)
I4: or t6 t7 t0
I5: xor t8 t9 t0
```

- **Output dependence – WAW Hazard**
  - WRITE after WRITE (WAW)
  - [Req] Write to a register after a previous instruction wrote into the register
  - WAW Hazard occurs if write by (I_{n+1}) occurs before write by (I_n) – "*written out of order*"

- **Anti Dependence – WAR HAZARD**
  - WRITE after READ (WAR)
  - [Req] Write to a register after a previous instruction read from the register
  - WAR Hazard occurs if write by (I_{n+1}) occurs before write by (I_n) – "*write too soon*"

- **Data Hazard**
  - When there is a data dependency i.e. read/write access to operands in specific order.

I1: add t0, t1 t2
I2: sub t2 t3 t0
I3: and t4 t5 t0
I4: or t6 t7 t0
I5: xor t8 t9 t0

- *T0 register is **written in I1** and read by others*
- *T0 register will be updated at end of Cycle 5.*
- *T0 register will be read by I2, I3, I4 in cycles 2,3,5.*

Instr	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
Add	IF (I1)	ID (I1)	EX (I1)	MEM (I1)	WB (I1)			
Sub	-	IF (I2)	IDs (I2)	EX (I2)	MEM (I2)	WB (I2)		
And			IF (I3)	IDs (I3)	EX (I3)	MEM (I3)	WB (I3)	
OR				IF (I4)	IDs (I4)	EX (I4)	MEM (I4)	WB (I4)
XOR					IF (I5)	ID (I5)	EX (I5)	MEM (I5)

- **Data Hazard**
  - When there is a data dependency i.e. read/write access to operands in specific order.

I1: add t0, t1 t2
I2: sub t2 t3 t0
I3: and t4 t5 t0
I4: or t6 t7 t0
I5: xor t8 t9 t0

- *T0 register is **written in I1** and read by others*
- *T0 register will be updated at end of Cycle 5.*
- *T0 register will be read by I2, I3, I4 in cycles 2,3,5.*

Instr	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
Add	IF (I1)	ID (I1)	EX (I1)	MEM (I1)	WB (I1)			
sub		IF (I2)	**Bubble**	**Bubble**	**Bubble**			
STALL				Bubble	Bubble	Bubble	Bubble	
STALL				Bubble	Bubble	Bubble	Bubble	Bubble
STALL					Bubble	Bubble	Bubble	Bubble
sub					IF (I3)	ID (I2)	EX (I2)	MEM (I2)

CODE ORDERING (RECAP: INSTRUCTIONS: MORE COMPLEX CODE)

High-level code

```
a = b + c - d;
```

MIPS assembly code

```
add t, b, c   # t = b + c
sub a, t, d   # a = t - d
```

- Some sequences are better than others…
- *the second sequence needs one more temporary variables (t2), while it (can perform first two instructions in parallel)*

More complex code is handled by multiple MIPS instructions.

```
a = b + c + d + e;
```

```
add t, b, c   # t = b + c
add t, t, d   # t = t + d
add a, t, e   # a = t + e
```

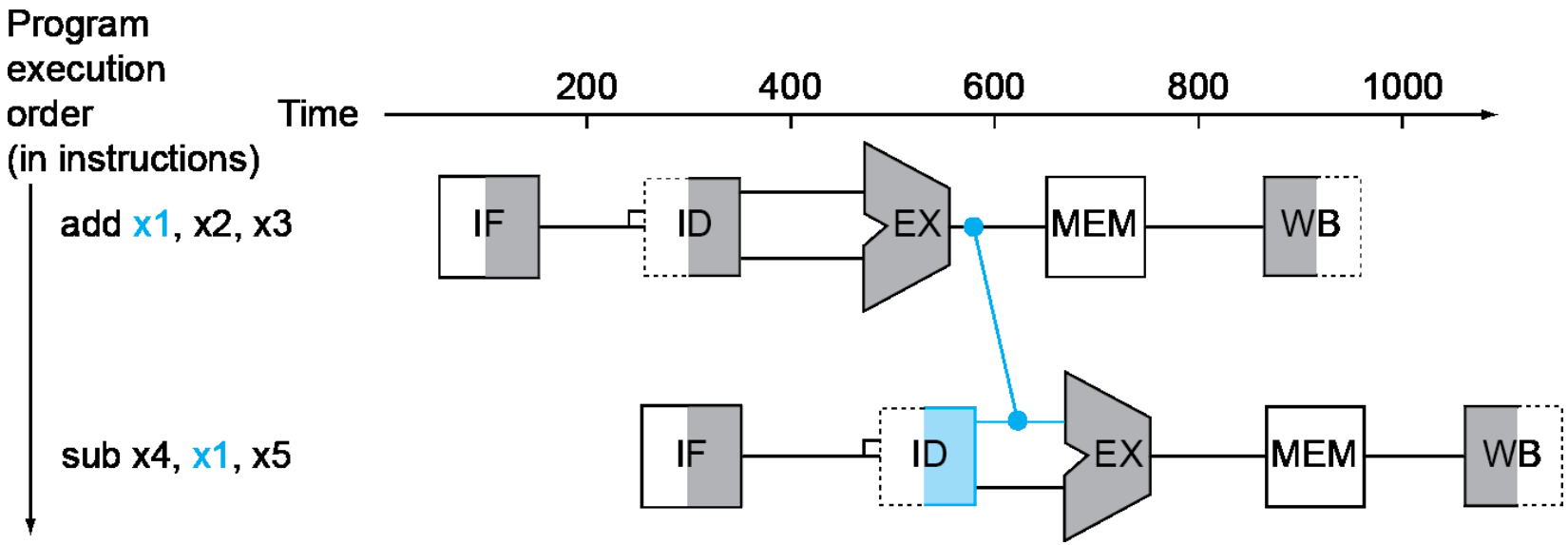
```
add t, b, c   # t = b + c
add t2, d, e  # t2 = d + e
add a, t, t2  # a = t + t2
```

- A single line of C code is converted into multiple lines of assembly code
- There can be multiple ways to execute the same HLI

Some sequences are → ***better suited fore pipelined execution.***

ADDRESSING DATA HAZARDS: FORWARDING (AKA BYPASSING)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



CONTROL HAZARDS

- What is the fundamental operating principle of pipelining?
  - ***Fetch and Execute new Instruction on every Cycle***.

- However pipeline can't always fetch/execute correct instruction.
  - We have already seen the Structural and Data Hazards.

Instr	Cycle 1	Cycle2	Cycle 3	Cycle4	Cycle5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
Lw	IF	ID	EX	MEM	WB				
Add	-	IF	ID	EX	MEM	WB			
Sub			IF	ID	EX	MEM	WB		
STALL				Bubble	Bubble	Bubble	Bubble	Bubble	
And				STALL	IF	ID	EX	MEM	WB

- What is control dependence?
  - To determine which instruction to execute next...

Branch Types

Instruction Type	Branch decision at IF/ID time	# of possible next fetch addresses?	Next fetch requires	When is next fetch address resolved?
Conditional Branch Example: ***beq $s1 $s2 loop*** ***bne $s1 $s2 loop***	Unknown	2 IF(TRUE) go to loop; ELSE do {*blah blah*}	Instruction and Data	Execution stage (register dependent)
Unconditional Example: ***J 2500***	Always taken	1	Instruction	Decode stage (PC + offset)
Call Example: ***Jal 2500***	Always taken	1	Instruction	Decode stage (PC + offset)
Return Example: ***Jr $ra***	Always taken	Many	Instruction and Data	Execution stage (register dependent)

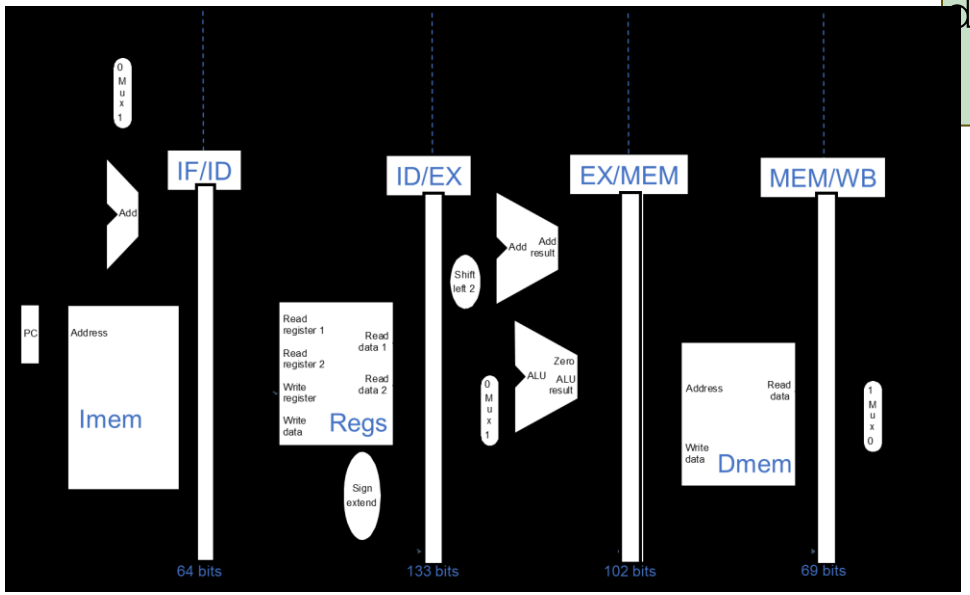
Branching – Forward and Backward Branches

High-level code

```
sum=0;
i =0;
n=10;
for (i=0; i<=n; i++) {
    Sum +=i;
}
printf("Sum of  %d… ");
```

MIPS assembly code

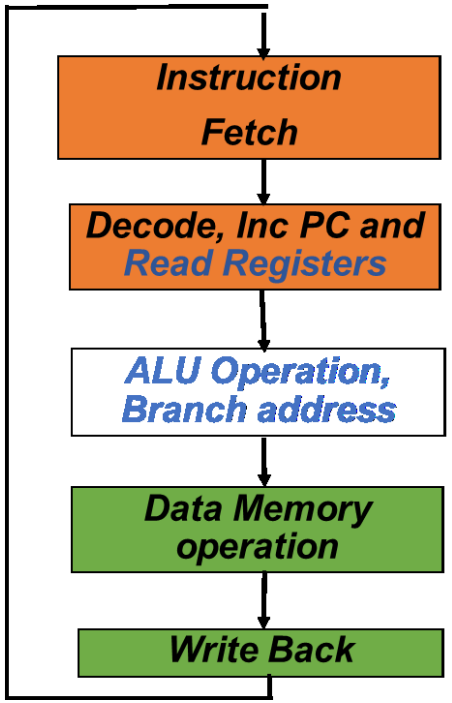
```
  add  $s1, $0, $0        # sum = 0
  add  $s0, $0, $0        # i   = 0
  addi $t0, $0, 10        # $t0 = 10
for:
  beq  $s0, $t0, done     # if i == 10, branch
  add  $s1, $s1, $s0      # sum = sum + i
  addi $s0, $s0, 1        # increment i
  j    for
done:
  syscall print
```



STALL ON BRANCH

- Wait until branch outcome determined before fetching next instruction

Instr	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
BEQ	IF (I1)	ID (I1)	EX (I1)	MEM (I1)	WB (I1)			
STALL		Bubble	Bubble	Bubble	Bubble	Bubble		
STALL			Bubble	Bubble	Bubble	Bubble	Bubble	
add				IF (I2)	ID (I2)	EX (I2)	MEM (I2)	
addi					IF (I3)	ID (I3)	EX (I3)	MEM (I3)



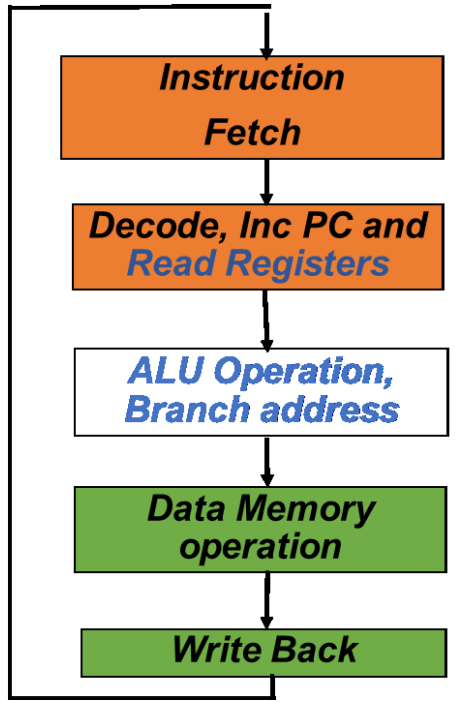
- In our earlier MIPS Pipeline design, We would need about 2 stalls!
  - i.e. IF of next instruction can begin only after completion of EX/ALU stage.
- By adding extra HW, we can at-best reduce to 1 stall by computing in ID stage.

STALL ON BRANCH

```
for:
  beq  $s0, $t0, done
  add  $s1, $s1, $s0
  addi $s0, $s0, 1
  j    for
done:
  syscall print
```

- Wait until branch outcome determined before fetching next instruction

Instr	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
BEQ	IF (I1)	ID (I1)	EX (I1)	MEM (I1)	WB (I1)			
STALL		Bubble	Bubble	Bubble	Bubble	Bubble		
STALL			Bubble	Bubble	Bubble	Bubble	Bubble	
add				IF (I2)	ID (I2)	EX (I2)	MEM (I2)	
addi					IF (I3)	ID (I3)	EX (I3)	MEM (I3)



- In our MIPS Pipeline design, We would need about 2 stalls!
  - i.e. IF of next instruction can begin only after completion of EX/ALU stage.
- By adding extra HW, we can at-best reduce to 1 stall by computing in ID stage.

PERFORMANCE ANALYSIS (NEXT PC = PC+4)

- correct PC $\Rightarrow$ no penalty 86% of the time
- incorrect PC $\Rightarrow$ 2 bubbles (baseline) 14% of the time
- Assume
  - no data hazards
  - 20% control flow instructions
  - 70% of control flow instructions are taken

CPI = [1 + (0.20*0.7) * 2] = [1 + 0.14 * 2] = 1.28

probability of a wrong Insn.

penalty for a wrong Insn.

```
  add  $s1, $0, $0        # sum = 0
  add  $s0, $0, $0        # i   = 0
  addi $t0, $0, 10        # $t0 = 10
for:
  beq  $s0, $t0, done     # if i == 10, branch
  add  $s1, $s1, $s0      # sum = sum + i
  addi $s0, $s0, 1        # increment i
  j    for
done:
  syscall print
```

What if we reduce the penalty to 1 cycle?

CPI = [1 + 0.14 * 1] = 1.14

Can we reduce either of the two penalty terms?

How to Handle Control Dependences

Potential solutions if the instruction is a control-flow instruction:

- Stall the pipeline until we know the next fetch address
- Employ delayed branching (branch delay slot)
- Guess the next fetch address (branch prediction)

DELAYED BRANCHING

- Change the semantics of a branch instruction
  - Branch after N instructions/N cycles

- Basic Idea: Delay the execution of a branch. N instructions (delay slots) that come after the branch are always executed regardless of branch direction.

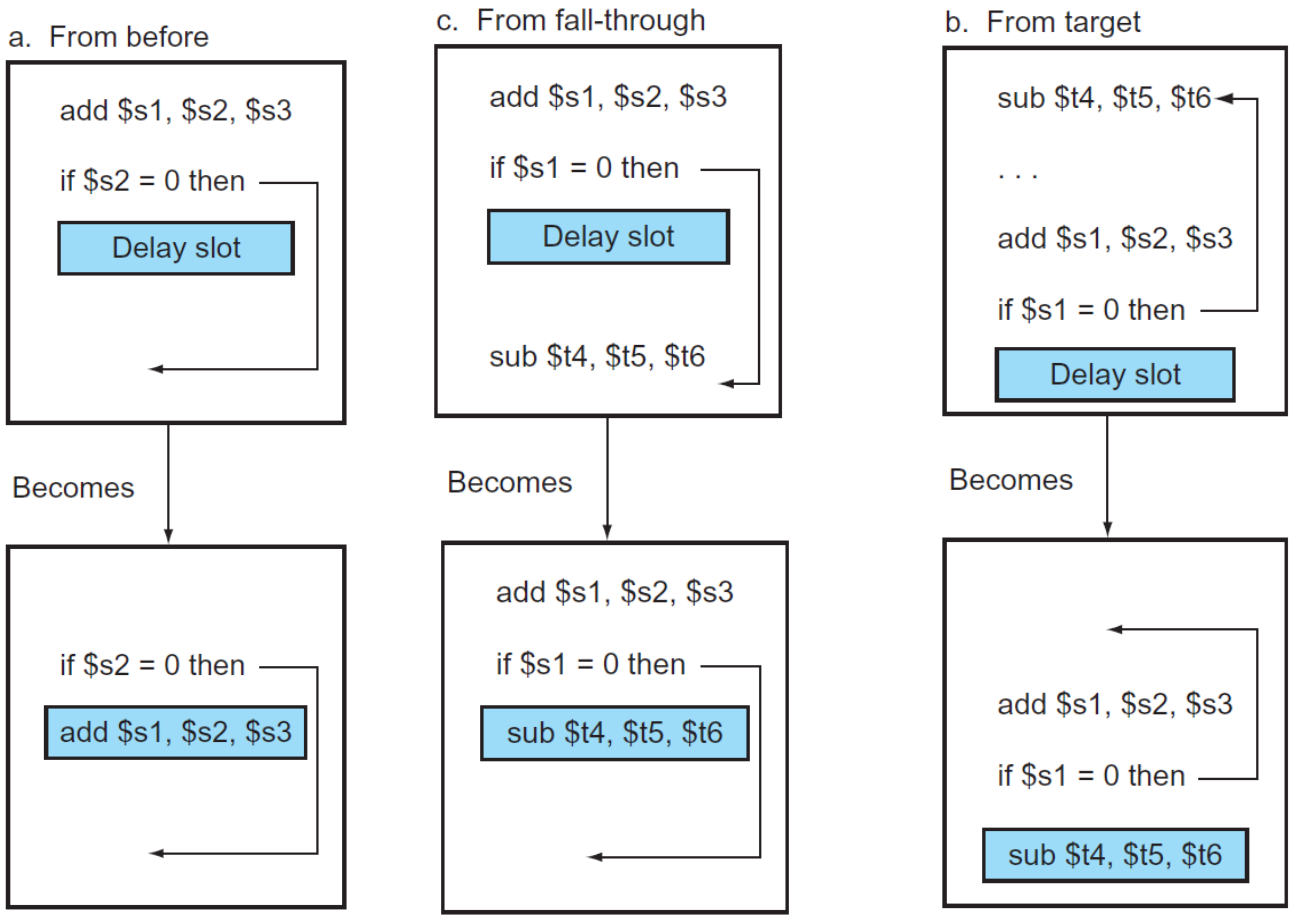
- Problem: How do you find instructions to fill the delay slots?
  - Branch must be independent of delay slot instructions

- ***Unconditional branch***: Easier to find instructions to fill the delay slot

- ***Conditional branch***: Condition computation should not depend on instructions in delay slots → ***difficult to fill the delay slot***

Filling the Delay Slot

Basic Idea: Reorder data Independent instructions which does not change the program semantics.



within same basic block

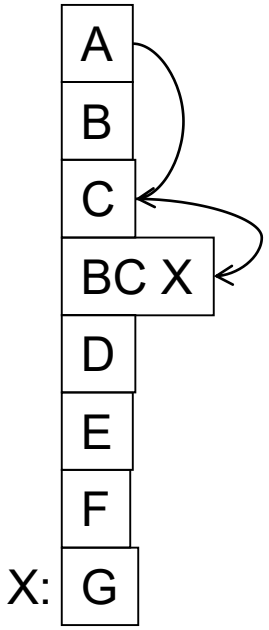
a new instruction added assuming Branch is not taken

a new instruction added assuming Branch is taken

Safe?

Delayed Branching

Normal code:



Timeline:

if	ex
A	
B	A
C	B
BC	C
--	BC
G	--

6 cycles

Example:

```
A:  add r1, r2, r3
B:  add r4, r5, r6
C:  add r1, r1, r7
BC: beq r5, r6, X
G:  add r8, r9, r10
```

```
A:  add r1, r2, r3
C:  add r1, r1, r7
BC: beq r5, r6, X
B:  add r4, r5, r6
G:  add r8, r9, r10
```

```
A:  add r1, r2, r3
BC: beq r5, r6, X
B:  add r4, r5, r6
C:  add r1, r1, r7
G:  add r8, r9, r10
```

DELAYED BRANCHING

- Advantages:

  + Keeps the pipeline full with useful instructions in a simple way assuming

    1. All delay slots can be filled with useful instructions

    2. Number of delay slots == number of instructions to keep the pipeline full before the branch resolves.

- Disadvantages:

  -- Not easy to fill the delay slots (even with a 2-stage pipeline)

    1. Number of delay slots increases with pipeline depth, superscalar execution width

    2. Number of delay slots would vary for variable latency operations.

  -- Ties ISA semantics to hardware implementation

    -- SPARC, MIPS: 1 delay slot

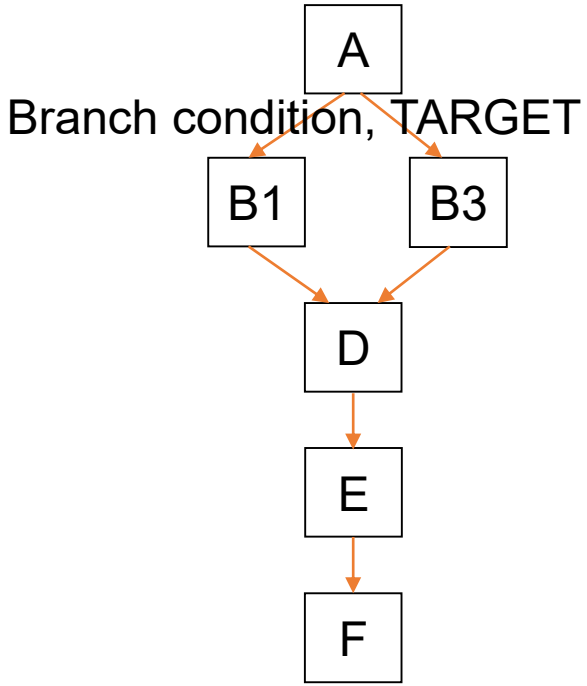
    -- What if pipeline implementation changes with the next design?

BRANCH PREDICTION

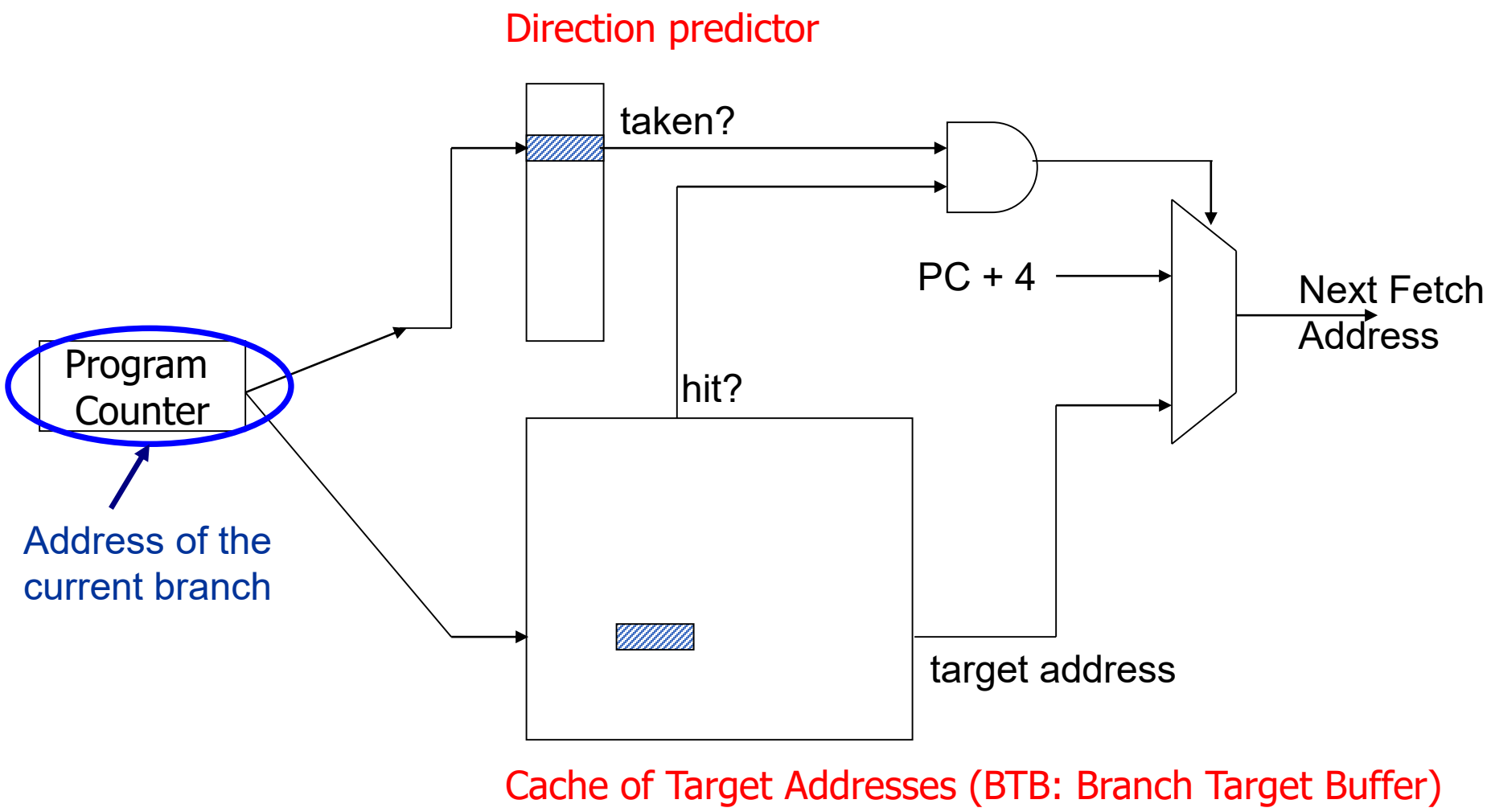
How do we keep the pipeline full in the presence of branches?

- Predict the next instruction when a branch instruction is fetched

- Requires two kind of information:
  - i) Direction of the Branch ← Predict with some intelligence.
  - ii) Associated Target of a branch ← can be computed from the Instruction



Fetch Stage with BTB and Direction Prediction



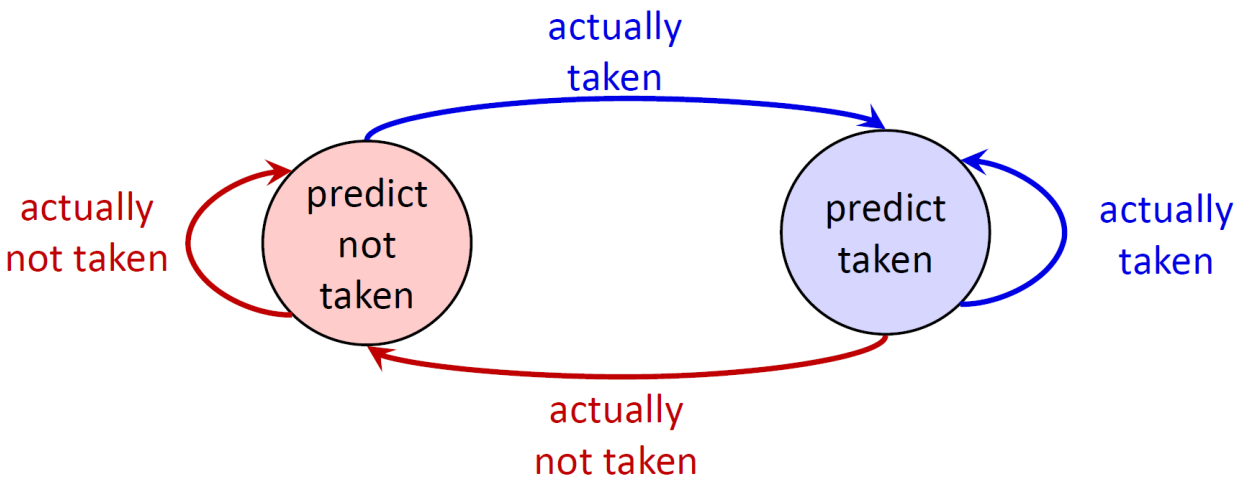
Branch Direction Prediction

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile/Program analysis/Programmer Hint based (likely direction)

- Run time (dynamic)
  - Last time prediction (single-bit)
  - N-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid (counter and level)

1-Bit (Bimodal) Branch Predictor

- Very simple 1-bit direction predictor (0 = N, 1 = T)
  - Essentially: branch will go same way it went last time



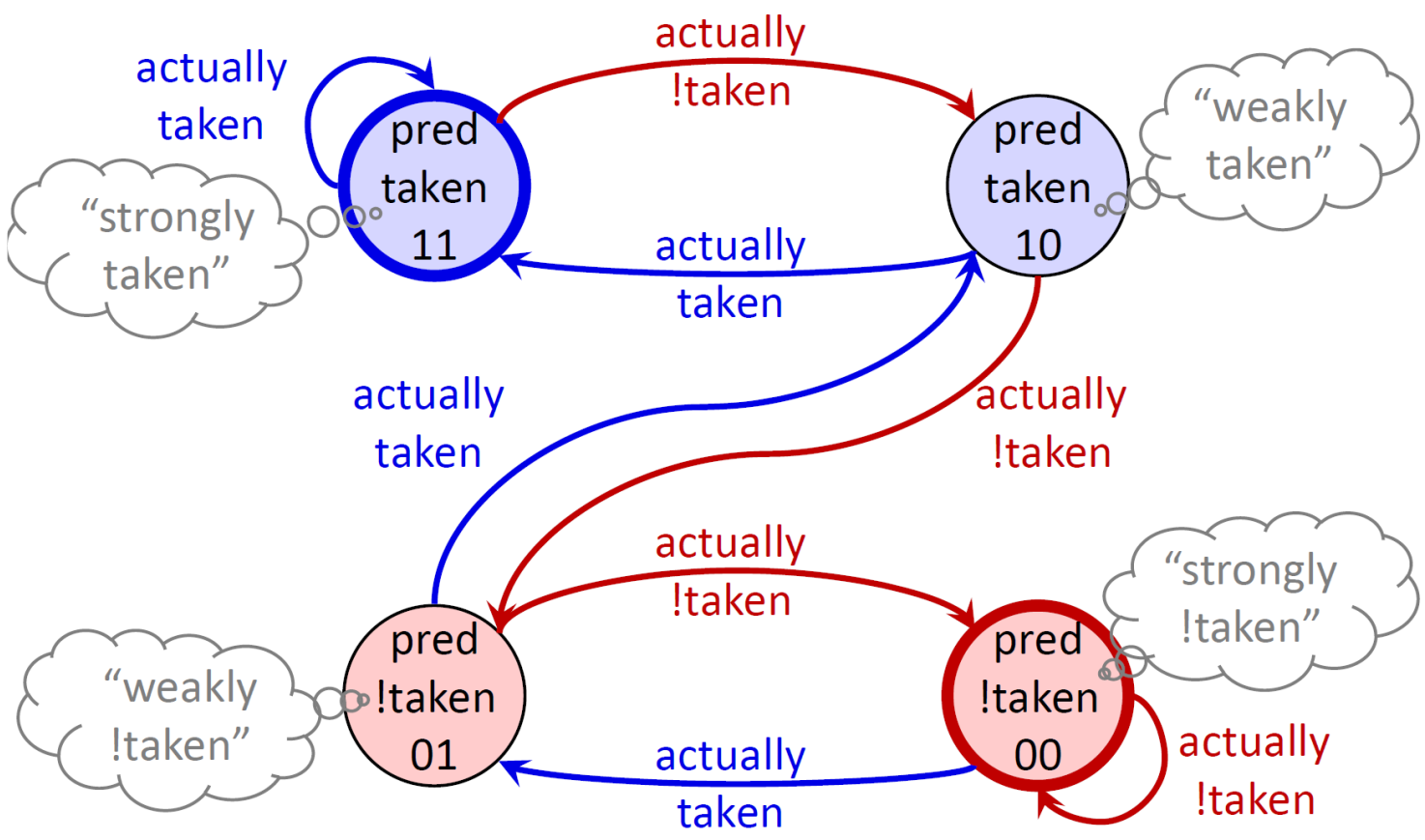
  - Problem: **inner loop branch** below

```
for (i=0;i<100;i++)
    for (j=0;j<3;j++)
        // some instructions
```

    - Two "built-in" mis-predictions per inner loop iteration
    - Branch predictor "changes its mind too quickly"

Time	State	Prediction	Outcome	Result?
1	N	N	T	Wrong
2	T	T	T	Correct
3	T	T	T	Correct
4	T	T	N	Wrong
5	N	N	T	Wrong
6	T	T	T	Correct
7	T	T	T	Correct
8	T	T	N	Wrong
9	N	N	T	Wrong
10	T	T	T	Correct
11	T	T	T	Correct
12	T	T	N	Wrong

- **Two-bit saturating counters (2bc)** [Smith 1981]
  - Replace each single-bit prediction: (0,1,2,3) = (N,n,t,T)
  - One misprediction for each loop execution

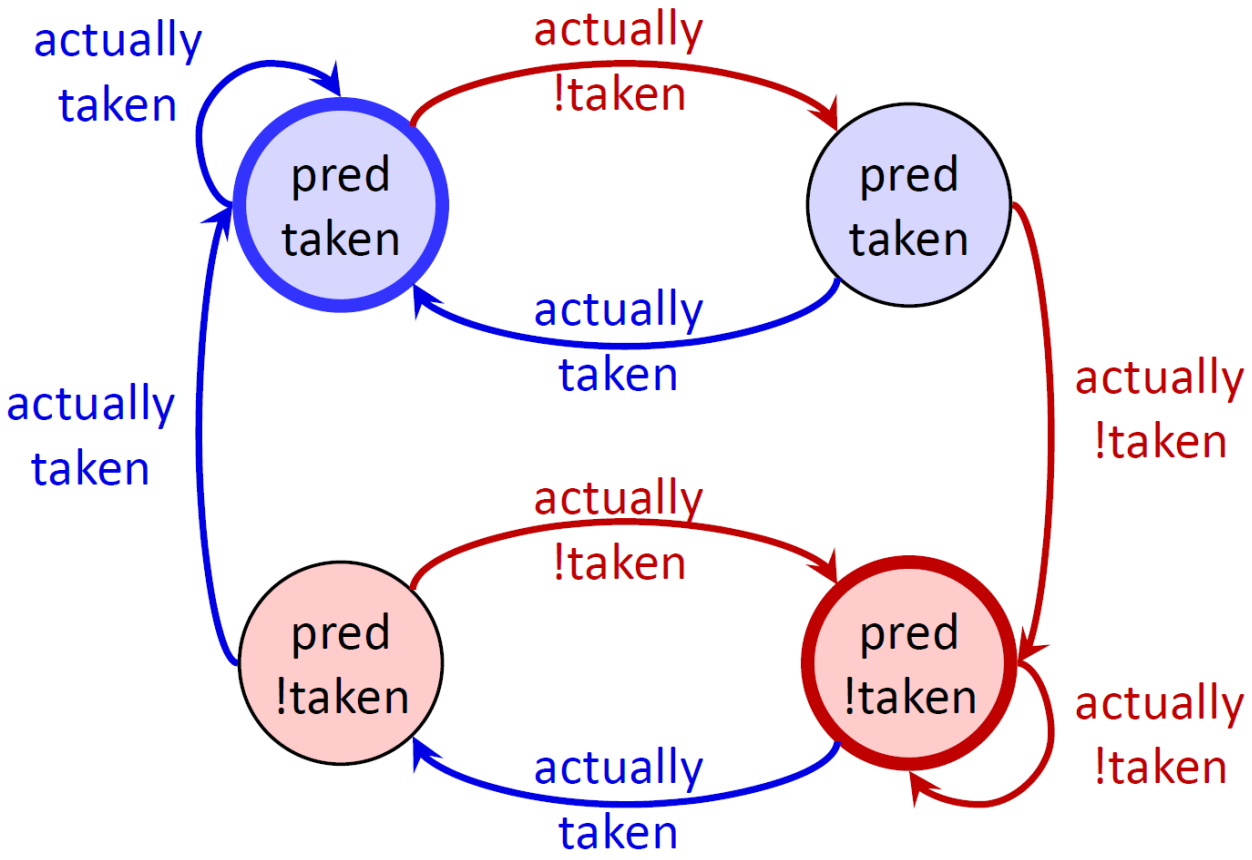



```
for (i=0;i<100;i++)
    for (j=0;j<3;j++)
        // some instructions
```

Time	State	Prediction	Outcome	Result?
1	N	N	T	Wrong
2	n	N	T	Wrong
3	t	T	T	Correct
4	T	T	N	Wrong
5	t	T	T	Correct
6	T	T	T	Correct
7	T	T	T	Correct
8	T	T	N	Wrong
9	t	T	T	Correct
10	T	T	T	Correct
11	T	T	T	Correct
12	T	T	N	Wrong

- Can we do even better?

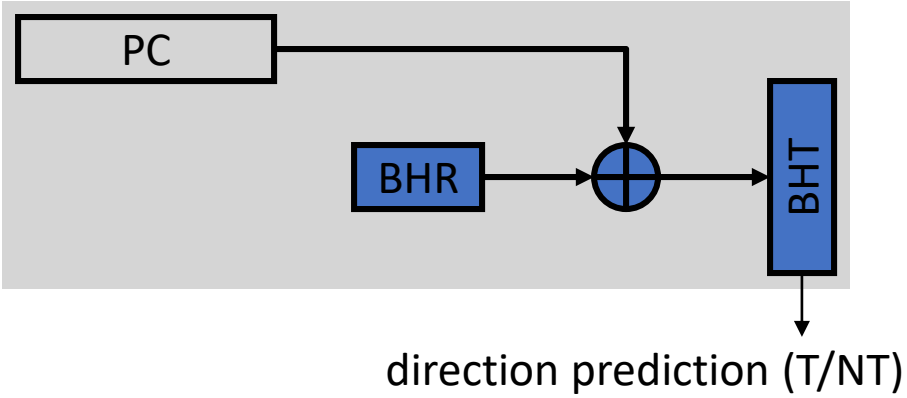
- **Two-bit Hysteris counters**
  - Same as before: (0,1,2,3) = (N,n,t,T)
- Instead of saturation, use history of previously taken decision
- Force predictor to mis-predict twice before "changing its mind"



BRANCHES MAY BE CORRELATED

- Consider:

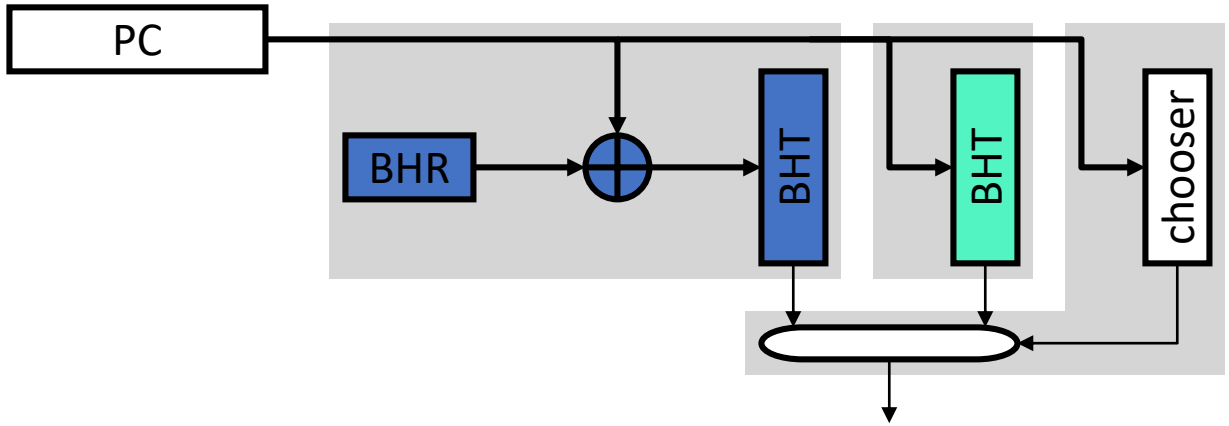
```
for (i=0; i<1000000; i++) {    // Highly biased
    if (i % 2 == 0) {          // Locally correlated
        … y = random()%3 …
    }
    if (y == 0) {   // Unpredictable
        …
    }
    if (i % 2 == 0) {
        …              // Globally correlated
    }
}
```



- Exploits observation that branch outcomes are correlated
- How do we incorporate history into our predictions?
  - Use PC xor BHR to index into BHT. Why?
- Maintains recent branch outcomes in **Branch History Register (BHR)**
  - In addition to BHT of counters (typically 2-bit sat. counters)

- **Tournament (Hybrid) predictor** [McFarling 1993]
  - Idea: combine two predictors
    - **Simple bimodal predictor** for history-independent branches
    - **Correlated predictor** for branches that need history
    - **Chooser** assigns branches to one predictor or the other
    - Branches start in simple BHT, move mis-prediction threshold

+ 90–95% accuracy



More Branch Predictor works and competitions:
https://www.jilp.org/vol13/index.html