Tradeoffs in processor design

CSC 236

Hardware performance

Three lectures

1. Architectural decisions

Hardware performance

Three lectures

1. Architectural decisions
2. Advances from von Neumann

Clock speed: does faster clock mean programs execute faster?

No … not necessarily.

Architecture matters

Hardware performance

Three lectures

1. Architectural decisions
2. Advances from von Neumann

```
mov ax, 10
add ax, 1
cmp ax, 11
je equals
```

```
equals:
    ; is ax == 11?
```

Hardware performance

Three lectures

1. Architectural decisions
2. Advances from von Neumann
3. Evolution: beyond 8086

ISA design

Number of explicit operands

Add X and Y ; store results in Z

$$Z = X + Y$$

Can do that operation with 3 operands.

Add X & Y ; store in Z

Explicit	Example	Implicit
3	add r0, r1, r2	0

ARM
r0 = r1 + r2

Add X & Y ; store in Z

Explicit	Example	Implicit
3	add r0, r1, r2	0
2	add ax, bx	1

8086

ax = ax + bx

Add X & Y ; store in Z

Explicit	Example	Implicit
3	add r0, r1, r2	0
2	add ax, bx	1
1	inc ax	2

8086

ax = ax + 1

Add X & Y ; store in Z

Explicit	Example	Implicit
3	add r0, r1, r2	0
2	add ax, bx	1
1	inc ax	2
0	add	3*

JVM

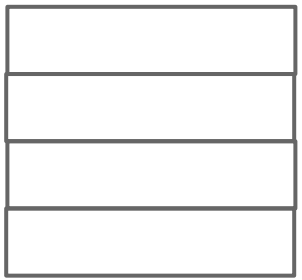
$stack_0 = stack_0 + stack_1$

*** sort of**

Zero operand instructions ⇒ stack machine

- Only data movement (ie, load) instructions have operands
- Binary operators pop 2 off stack; push one on
- Example: area of a circle

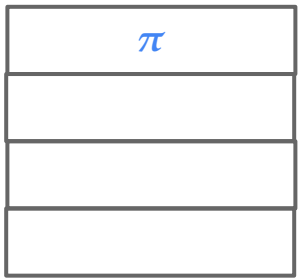
```
push pi
push r
dup
mul
mul
```



Zero operand instructions ⇒ stack machine

- Only data movements (ie, load) instructions have operands
- Binary operators pop 2 off stack; push one on
- Example: area of a circle

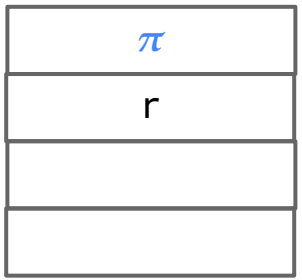
```
push pi
push r
dup
mul
mul
```



Zero operand instructions ⇒ stack machine

- Only data movements (ie, load) instructions have operands
- Binary operators pop 2 off stack; push one on
- Example: area of a circle

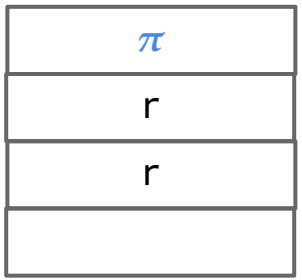
```
push pi
push r
dup
mul
mul
```



Zero operand instructions ⇒ stack machine

- Only data movements (ie, load) instructions have operands
- Binary operators pop 2 off stack; push one on
- Example: area of a circle

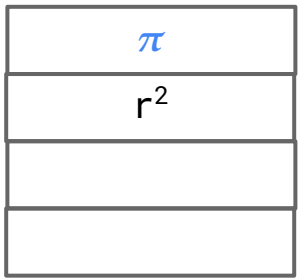
```
push pi
push r
dup
mul
mul
```



Zero operand instructions ⇒ stack machine

- Only data movements (ie, load) instructions have operands
- Binary operators pop 2 off stack; push one on
- Example: area of a circle

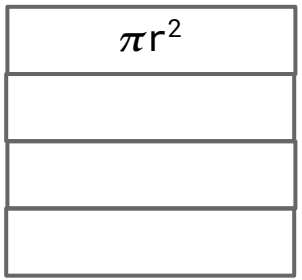
```
push pi
push r
dup
mul
mul
```



Zero operand instructions ⇒ stack machine

- Only data movements (ie, load) instructions have operands
- Binary operators pop 2 off stack; push one on
- Example: area of a circle

```
push pi
push r
dup
mul
mul
```



Three architectures

Look at how 3 different machines might calculate

$$Z = X + Y$$

M1: one explicit operand

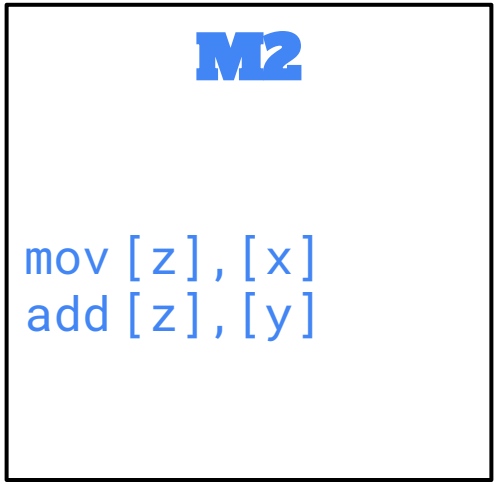
M2: two explicit operands

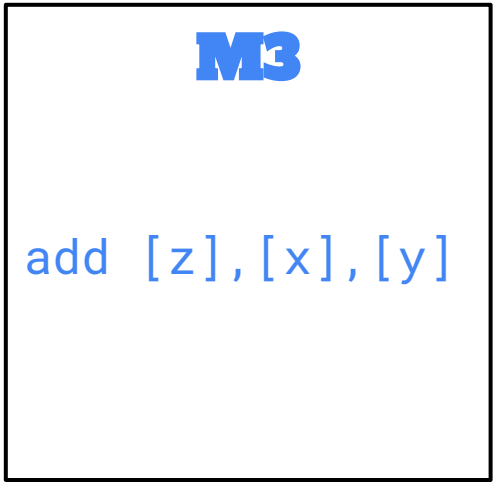
M3: three explicit operands

Comparison / Evaluation

- Highly simplified, idealized
  - Not to advocate an architecture
  - Not to teach architectural design/evaluation
- Just an example of a tradeoff
- Demonstrates that evaluation is
  - Complicated and
  - Somewhat unintuitive


M1
load [x]
add [y]
store [z]


M2
mov [z],[x]
add [z],[y]


M3
add [z],[x],[y]

M1

```
load   [x]
add    [y]
store  [z]
```

M2

```
mov [z],[x]
add [z],[y]
```

M3

```
add [z],[x],[y]
```

- **Accumulator machine**

  ```
  acc = X
  acc = acc + Y
  Z = acc
  ```

- 3 instructions

M1

```
load   [x]
add    [y]
store  [z]
```

M2

```
mov [z],[x]
add [z],[y]
```

M3

```
add [z],[x],[y]
```

```
Z = X
Z = Z + Y
```

- 2 instructions

M1

```
load   [x]
add    [y]
store  [z]
```

M2

```
mov [z],[x]
add [z],[y]
```

M3

```
add [z],[x],[y]
```

Z = X + Y

- 1 instruction

M1

```
load   [x]
add    [y]
store  [z]
```

M2

```
mov [z],[x]
add [z],[y]
```

M3

```
add [z],[x],[y]
```

Which of these represents 8086?

None

```
mov ax,[x]
add ax,[y]
mov [z],ax
```

M1

```
load   [x]
add    [y]
store  [z]
```

M2

```
mov [z],[x]
add [z],[y]
```

M3

```
add [z],[x],[y]
```

Which of these represents ARM?

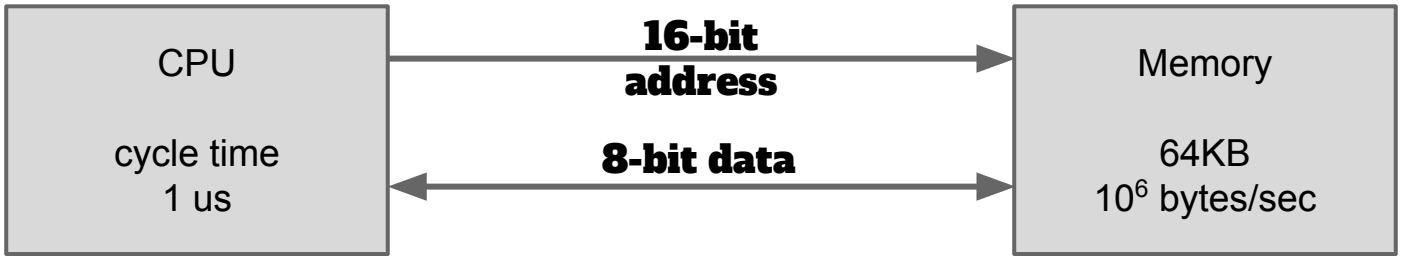
None

```
load r0,[x]
load r1,[y]
add r0,r0,r1
store [z],r1
```

Thoughts to this point

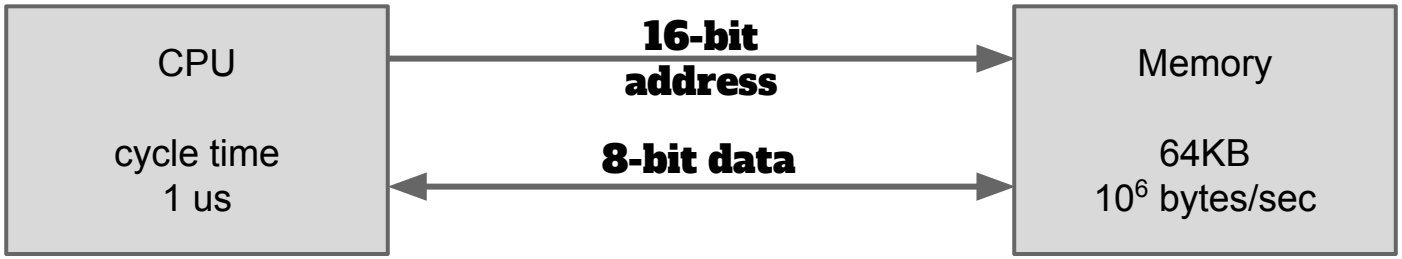
- As explicit operands increase
  - Instructions get more powerful
  - Instructions get more complex
  - Fewer instructions needed
- But that does not mean more is better

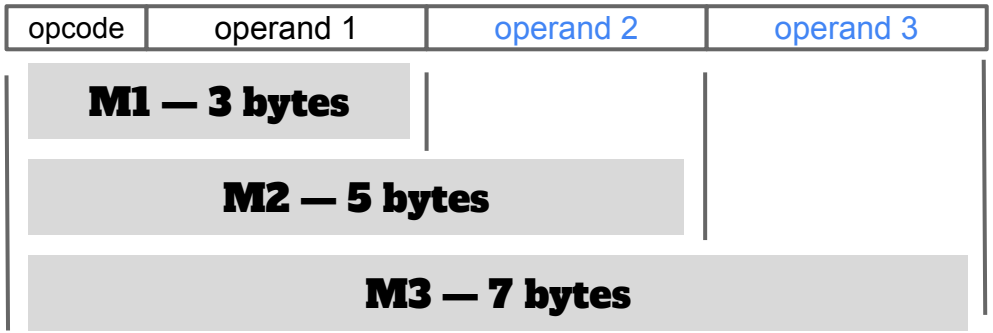
Idealized, fictitious machine



Compare M1, M2, M3 holding these parameters constant

Instruction size





Four steps

1. **Fetch instruction**
2. **Decode**
3. **Access data**
4. **Execute**

Four steps

1. **Fetch instruction** **1 µs / byte**
2. **Decode** **1 µs**
3. **Access data** **1 µs / operand**
4. **Execute** **2 µs**

	M1	M2	M3
Fetch	3	5	7
Decode	1	1	1
Data	1	2	3
Execute	2	2	2
1 instruction			
Instructions/sec			

	M1	M2	M3
Fetch	3	5	7
Decode	1	1	1
Data	1	2	3
Execute	2	2	2
1 instruction	7	10	13
Instructions/sec			

	M1	M2	M3
Fetch	3	5	7
Decode	1	1	1
Data	1	2	3
Execute	2	2	2
1 instruction	7	10	13
Instructions/sec	142,857	100,000	76,923

$$\text{instructions/second} = \frac{\text{cycles/second}}{\text{cycles/instruction}}$$

$10^6 / 7 = 142{,}857$

$10^6 / 10 = 100{,}000$

$10^6 / 13 = 76{,}923$

Which is faster?

	M1	M2	M3
Instructions/sec	142,857	100,000	76,923

Which is faster?

	M1	M2	M3
Instructions/sec	142,857	100,000	76,923
Inst/calculation	3	2	1
Calculations/sec			

Which is faster?

	M1	M2	M3
Instructions/sec	142,857	100,000	76,923
Inst/calculation	3	2	1
Calculations/sec	47,619	50,000	76,923

Megahertz myth

From Wikipedia, the free encyclopedia

This article has multiple issues. Please help **improve it** or discuss these issues on the **talk page**. *(Learn how and when to remove these template messages)* [show]

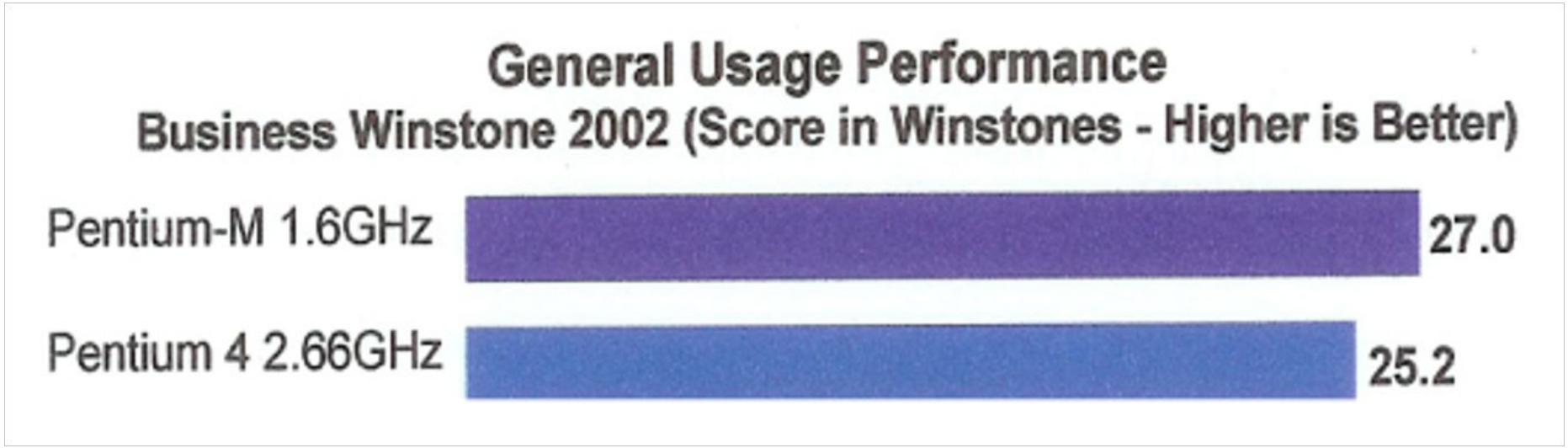
The **megahertz myth**, or in more recent cases the **gigahertz myth**, refers to the misconception of only using clock rate (for example measured in megahertz or gigahertz) to compare the performance of different microprocessors. While clock rates are a valid way of comparing the performance of different speeds of the same model and type of processor, other factors such as an amount of execution units, pipeline depth, cache hierarchy, branch prediction, and instruction sets can greatly affect the performance when considering different processors. For example, one processor may take two clock cycles to add two numbers and another clock cycle to multiply by a third number, whereas another processor may do the same calculation in two clock cycles. Comparisons between different types of processors are difficult because performance varies depending on the type of task. A benchmark is a more thorough way of measuring and comparing computer performance.

The myth started around 1984 when comparing the Apple II with the IBM PC. The argument was that the PC was five times faster than the Apple II, as its Intel 8088 processor had a clock speed roughly 4.7 times the clock speed of the MOS Technology 6502 used in the Apple. However, what really matters is not how finely divided a machine's instructions are, but how long it takes to complete a given task. Consider the LDA # (Load Accumulator Immediate) instruction. On a 6502 that instruction requires two clock cycles, or 2 µs at 1 MHz. Although the 4.77 MHz 8088's clock cycles are shorter, the LDA # needs at least[1] 4 of them, so it takes 4 / 4.77 MHz = 0.84 µs at least. So, at best, that instruction runs only a little more than 2 times as fast on the original IBM PC than on the Apple II.

To compare systems you need

- Clock speed
- Instructions execution rate
- Capability of each instruction
- Specific task
  - ⇒ benchmark

Time = (instructions/task) x (cycles/instruction) x (seconds/cycle)


General Usage Performance
Business Winstone 2002 (Score in Winstones - Higher is Better)
Pentium-M 1.6GHz
27.0
Pentium 4 2.66GHz
25.2

Benchmarks

- Hard to write
- Workload-dependent
  - Business
  - Scientific
  - Gaming
  - ...

The compiler can obscure the task

You write:

```
int i;
for (i=0; i<1000000; i++)
   ;  // nothing

return i;
```

A clever compiler generates:

```
return 1000000;
```

Example

- Testing sorting on two massively parallel systems
- Generate 10M random integers, I
- Time: sort(I)
  - Essentially, zero seconds
  - No output … OK, let's sample the sorted list.
- Time: sort(I); print(I[:10])
  - One system 100x faster than other
  - Compiler changed sort to "top 10"
  - Sort is O(n log n); top 10 is O(n)

Military Benchmark circa 1970

Read tasks	Process	Print
2 min	50 min	8 min
	83%	

← 60 minutes →

circa 1990

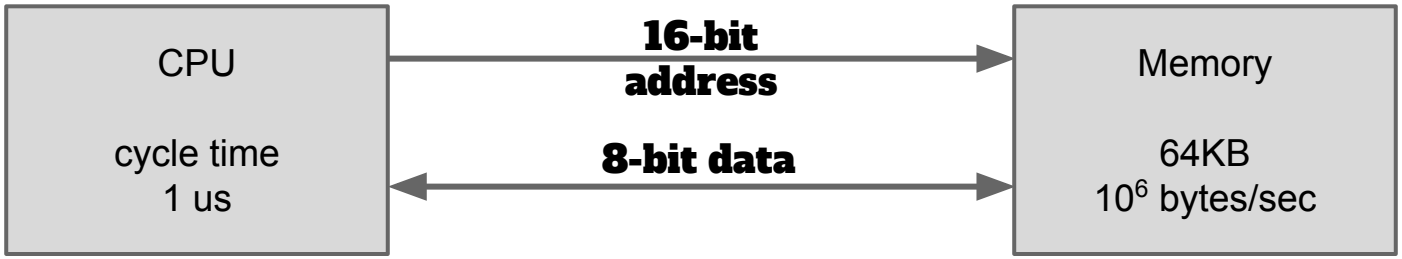
Read tasks	Process	Print
1 min	.05 min	4 min
	1%	

← 5.05 minutes →

Over the next 20 years:

- **CPU power increased 1000-fold**
- **I/O performance doubled.**

Example revisited — four ~~steps~~ *stages*



1.	**Fetch instruction**	**1 μs / byte**
2.	**Decode**	**1 μs**
3.	**Access data**	**1 μs / operand**
4.	**Execute**	**2 μs**

Consider a pipelined architecture

	P1	P2	P3
Fetch	3	5	7
Decode	1	1	1
Data	1	2	3
Execute	2	2	2
Pipeline period			
Prime pipeline			
IPS (run length of x)			

	P1	P2	P3
Fetch	3	5	7
Decode	1	1	1
Data	1	2	3
Execute	2	2	2
Pipeline period	**3**	**5**	**7**
Prime pipeline			
IPS (run length of x)			

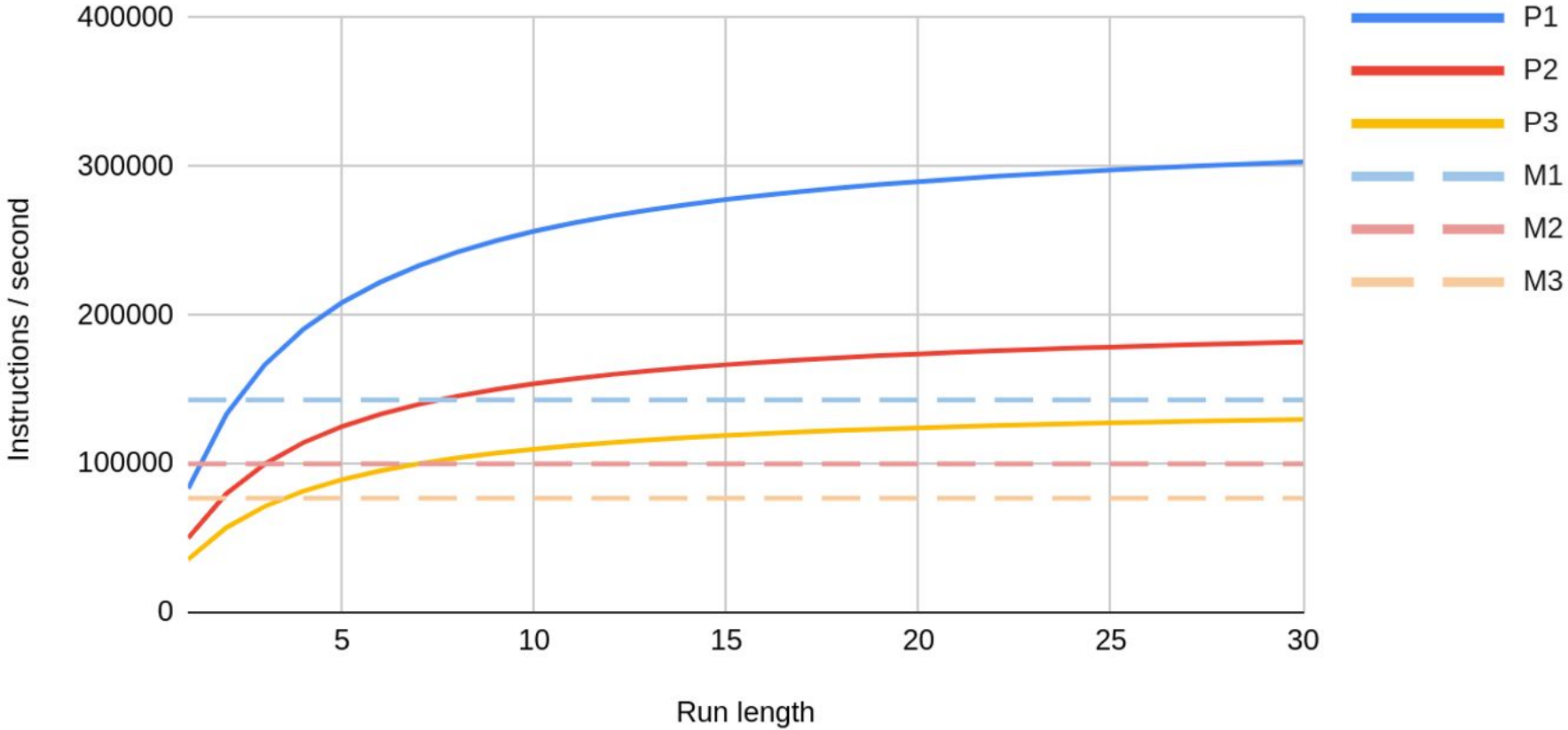
Period depends on the **slowest** pipeline stage.

	P1	P2	P3
Fetch	3	5	7
Decode	1	1	1
Data	1	2	3
Execute	2	2	2
Pipeline period	**3**	**5**	**7**
Prime pipeline	9	15	21
Instructions per Clock Tick	$\frac{x}{9 + 3x}$	$\frac{x}{15 + 5x}$	$\frac{x}{21 + 7x}$

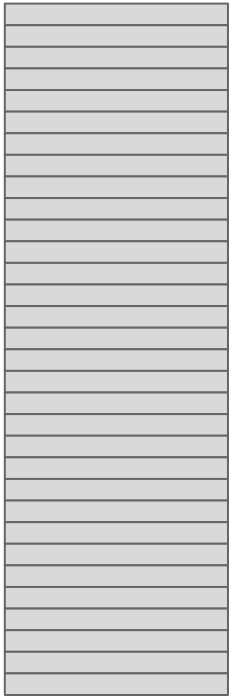
	P1	P2	P3
Fetch	3	5	7
Decode	1	1	1
Data	1	2	3
Execute	2	2	2
Pipeline period	**3**	**5**	**7**
Prime pipeline	9	15	21
Instructions second	$\frac{x}{9+3x} * \frac{10^6}{\text{sec}}$	$\frac{x}{15+5x} * \frac{10^6}{\text{sec}}$	$\frac{x}{21+7x} * \frac{10^6}{\text{sec}}$

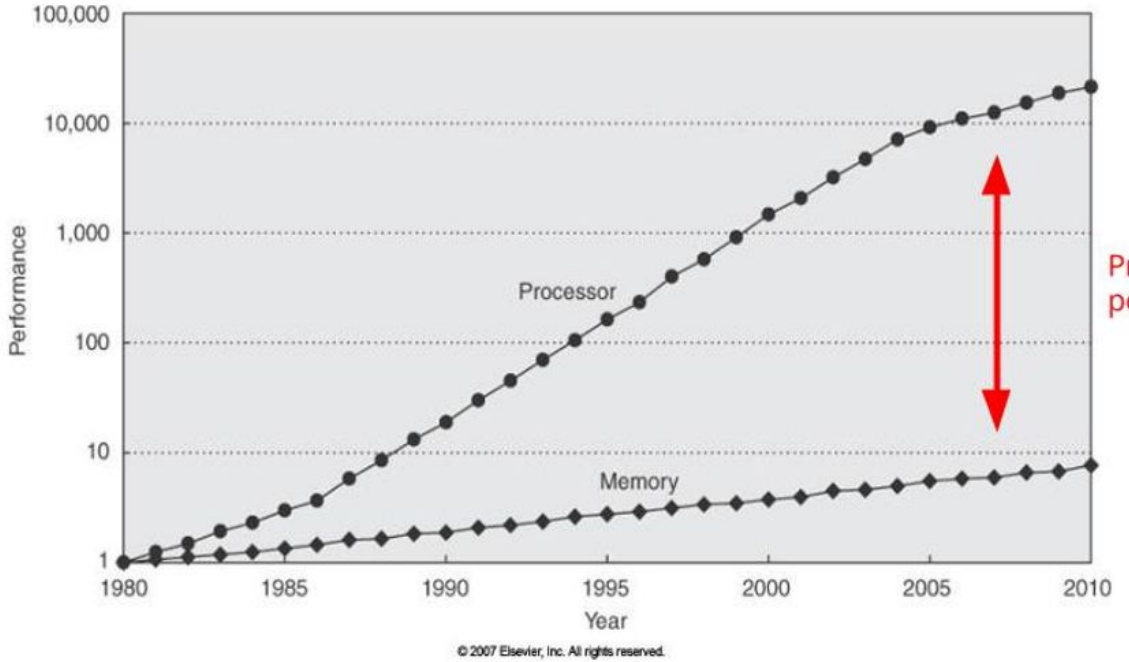
	P1	P2	P3
Fetch	3	5	7
Decode	1	1	1
Data	1	2	3
Execute	2	2	2
Pipeline period	**3**	**5**	**7**
Prime pipeline	9	15	21
IPS (run length of x)	$\frac{10^6 x}{3(3+x)}$	$\frac{10^6 x}{5(3+x)}$	$\frac{10^6 x}{7(3+x)}$

Pipelined Performance

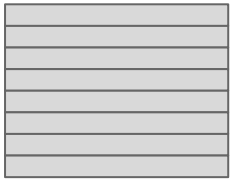


Memory hierarchy (not to scale)





Processor vs. Memory performance gap



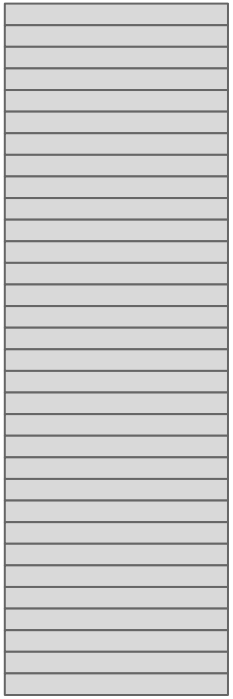
Memory speed lags behind CPU speed

Registers (dozens)

Memory (GBs)

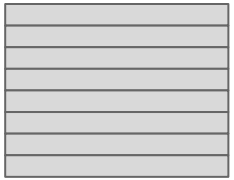
Memory hierarchy

(not to scale)



Memory (GBs)

cache

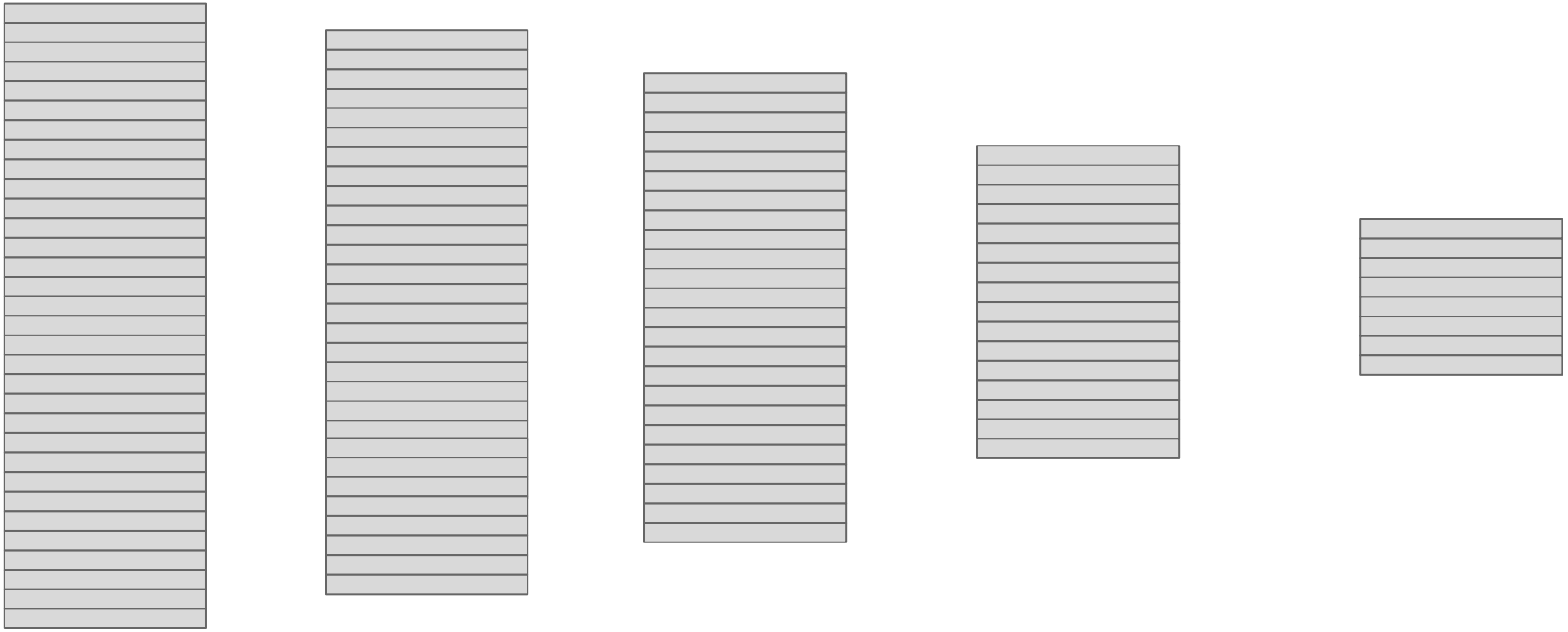


Registers (dozens)

A cache is a *latency tolerating device*.

Memory hierarchy

(not to scale)



Memory (GBs)

L2 & L3 (MBs)

L1 (KBs)

Registers (dozens)

Latencies

Core i7 Xeon 5500 Series Data Source Latency (approximate)

local	L1 CACHE hit,	~4 cycles (2.1 - 1.2 ns)	**1.5ns**
local	L2 CACHE hit,	~10 cycles (5.3 - 3.0 ns)	**x3**
local	L3 CACHE hit, line unshared	~40 cycles (21.4 - 12.0 ns)	**x10**
local	L3 CACHE hit, shared line in another core	~65 cycles (34.8 - 19.5 ns)	**x12**
local	L3 CACHE hit, modified in another core	~75 cycles (40.2 - 22.5 ns)	**x20**
remote	L3 CACHE	~100-300 cycles (160.7 - 30.0 ns)	**x60**
local	DRAM	~60 ns	**x40**
remote	DRAM	~100 ns	**x67**

Registers!

Registers — AMD 29000

- 192 registers
- 64 global
- 16 local
  - In 8 “contexts”
  - Can support subroutine call depth of 8
- What happens when exceed 8?

Registers — AMD 29000

- 192 registers
- 64 global
- 16 local
  - In 8 “contexts”
  - Can support subroutine call depth of 8
- What happens when exceed 8?
  - Push & pop
  - Loss of performance