

Computer Architecture  
ELE 475 / COS 475  
Slide Deck 1: Introduction and  
Instruction Set Architectures

David Wentzlaff

Department of Electrical Engineering  
Princeton University

# What is Computer Architecture?



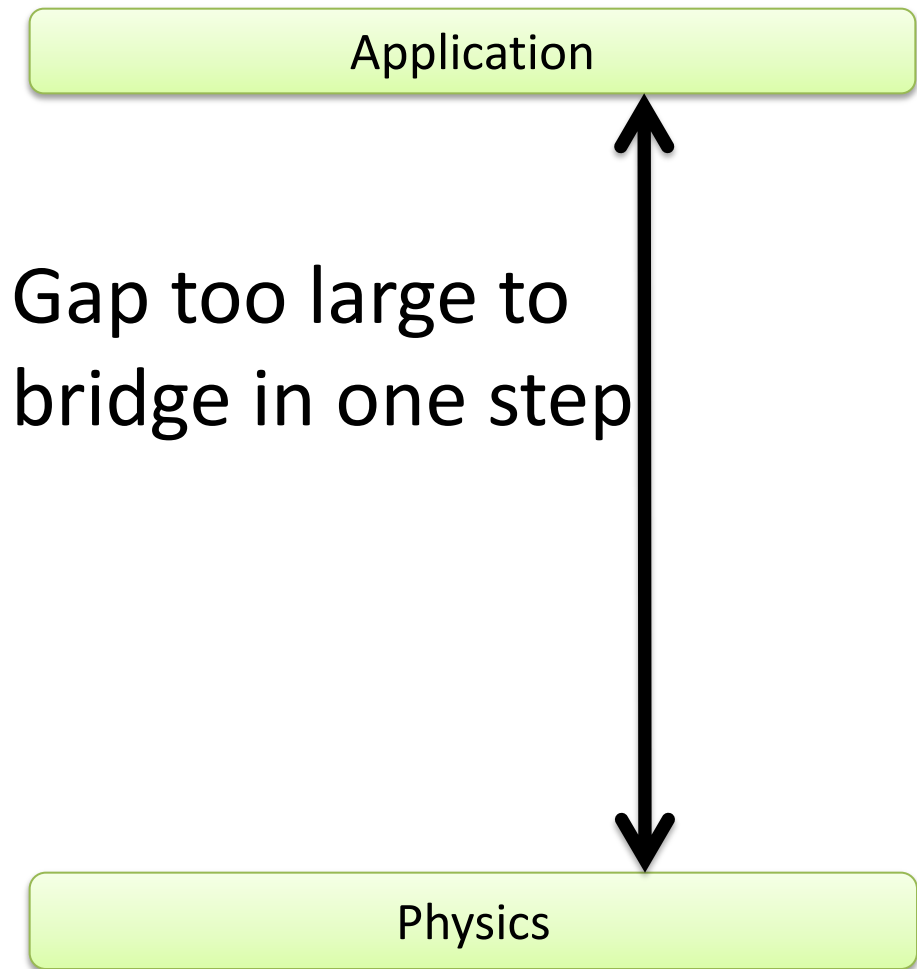
Application

# What is Computer Architecture?

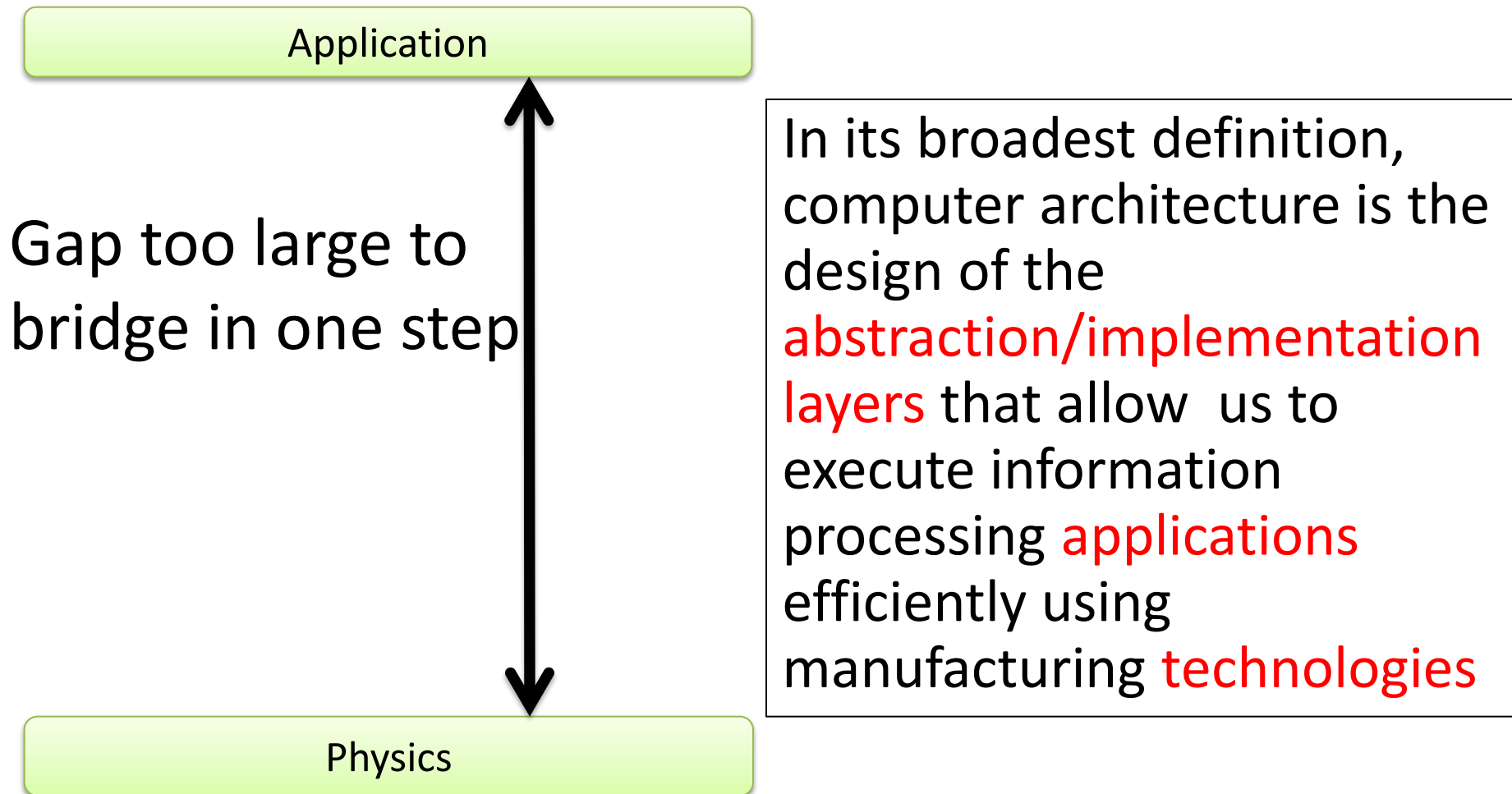
Application

Physics

# What is Computer Architecture?



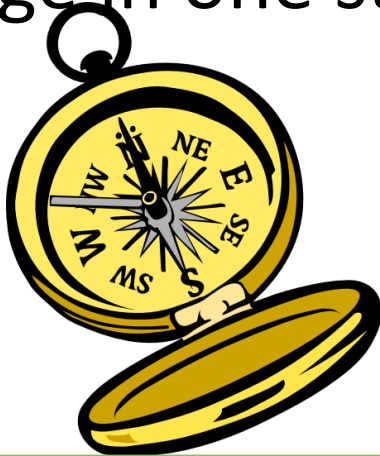
# What is Computer Architecture?



# What is Computer Architecture?

Application

Gap too large to  
bridge in one step



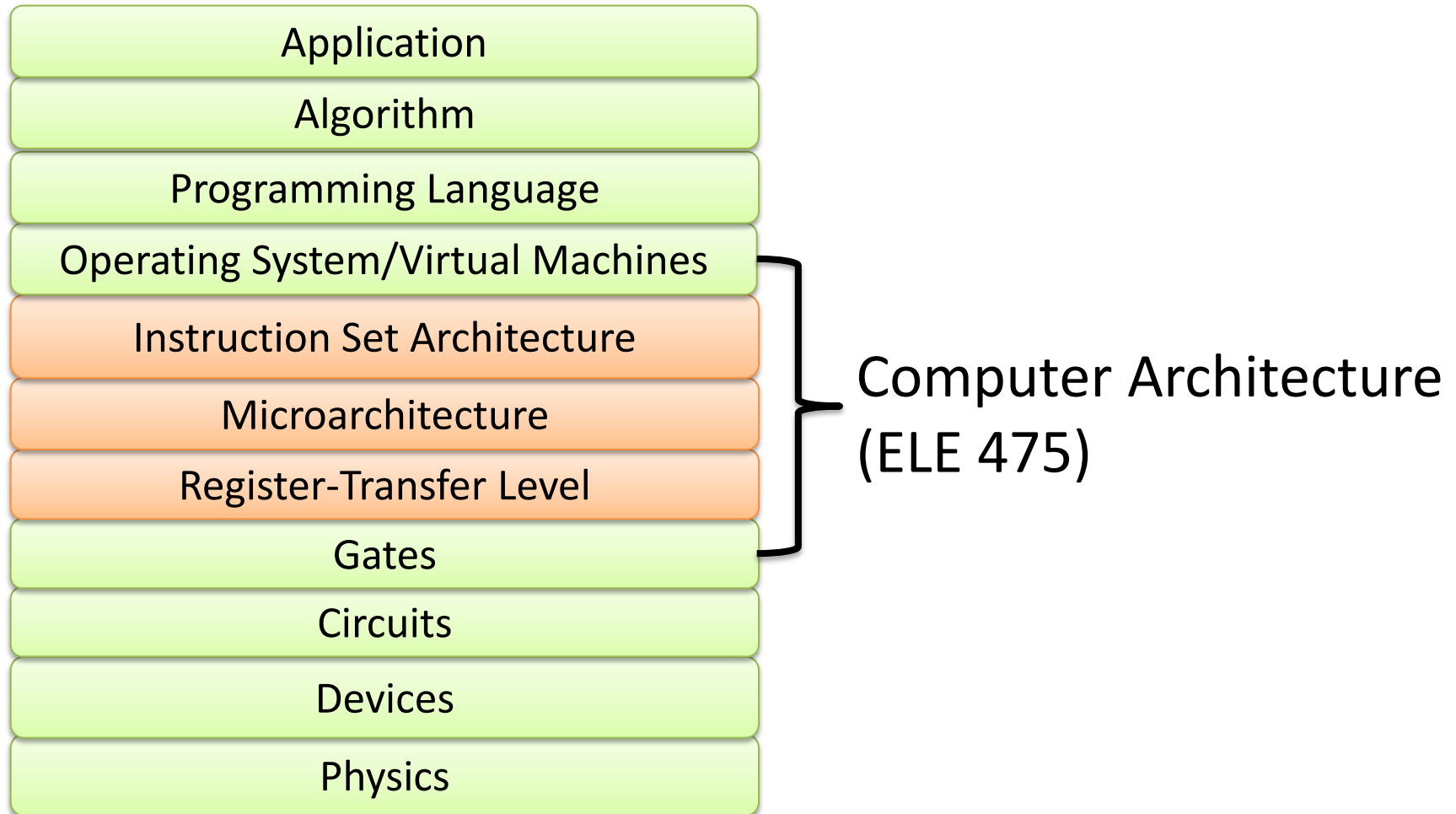
Physics

In its broadest definition, computer architecture is the design of the **abstraction/implementation layers** that allow us to execute information processing **applications** efficiently using manufacturing **technologies**

# Abstractions in Modern Computing Systems

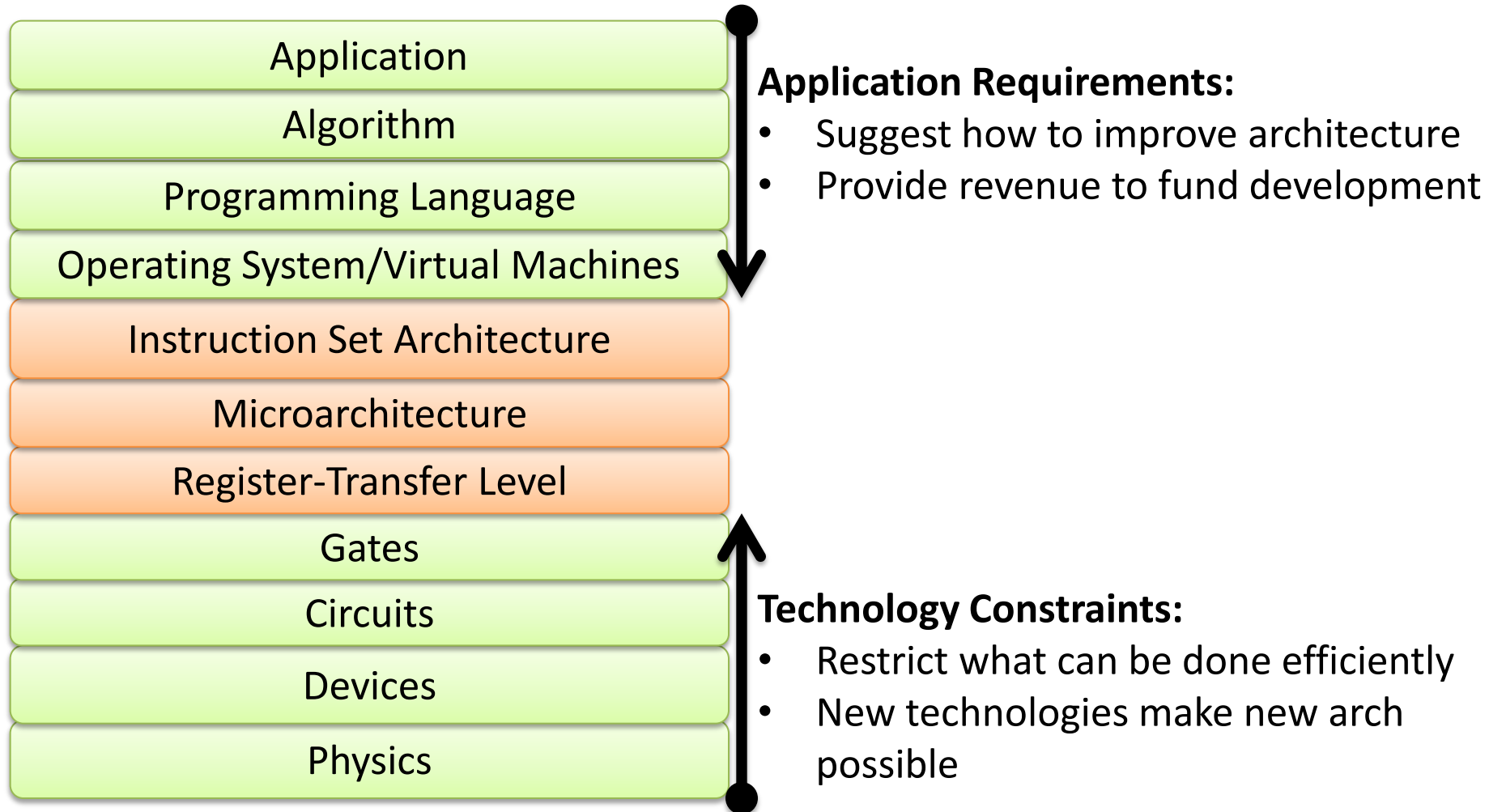


# Abstractions in Modern Computing Systems

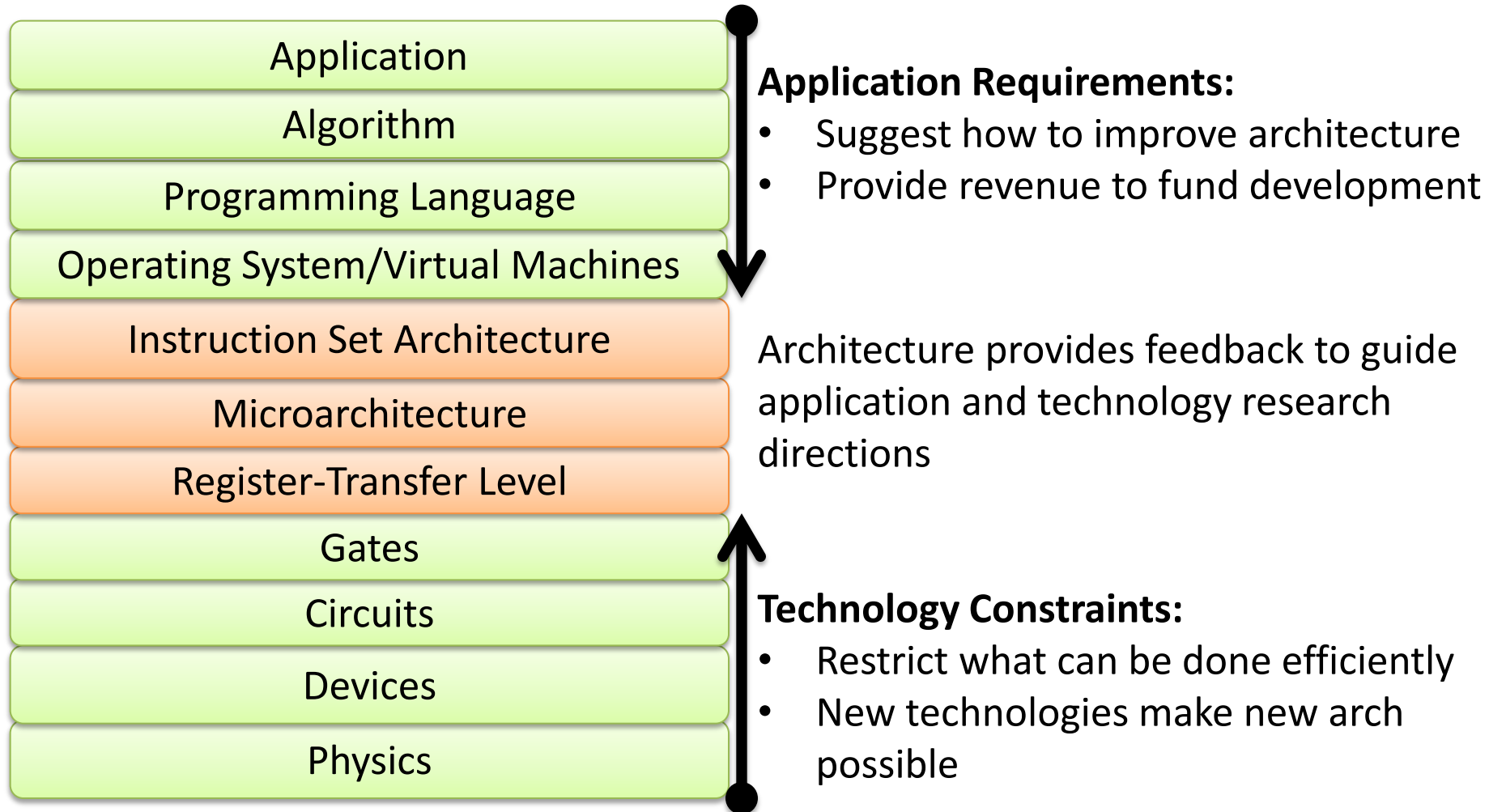




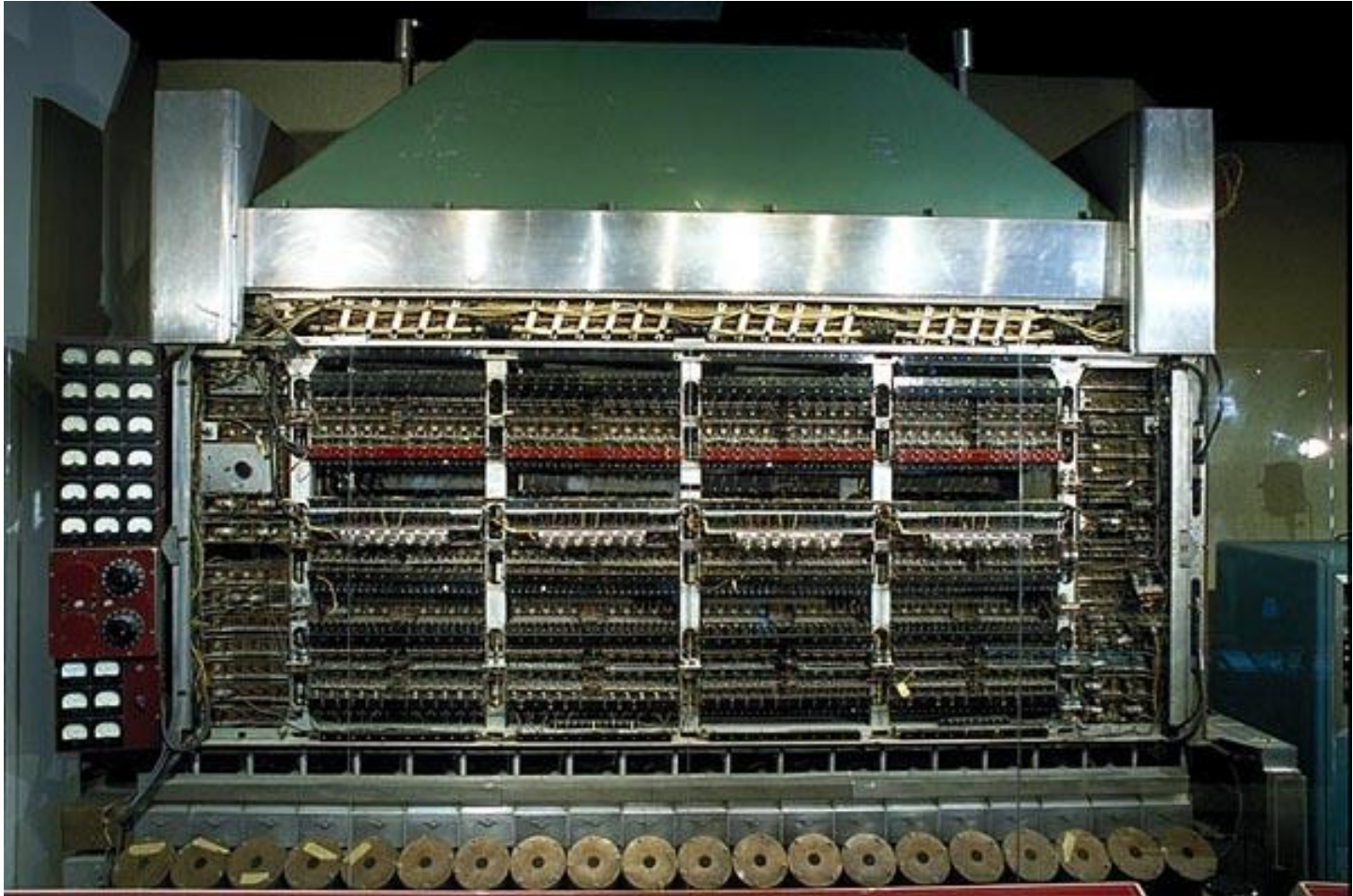
# Computer Architecture is Constantly Changing



# Computer Architecture is Constantly Changing

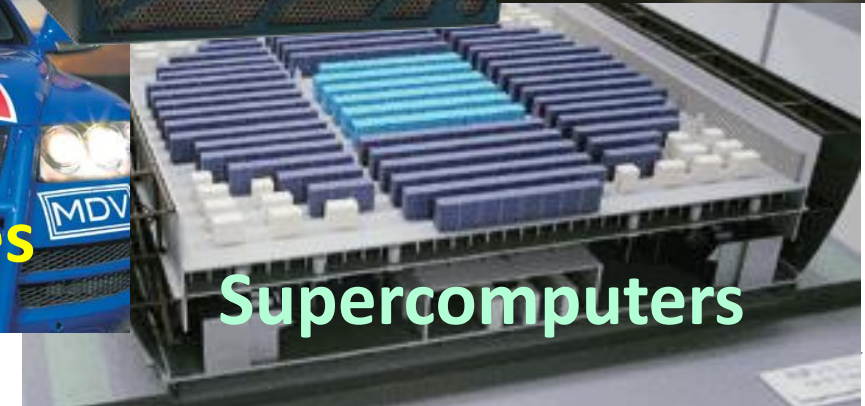
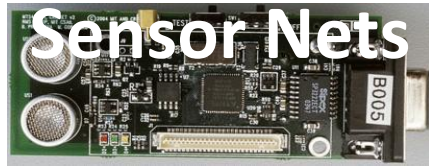


# Computers Then...



IAS Machine. Design directed by John von Neumann.  
First booted in Princeton NJ in 1952  
Smithsonian Institution Archives (Smithsonian Image 95-06151)

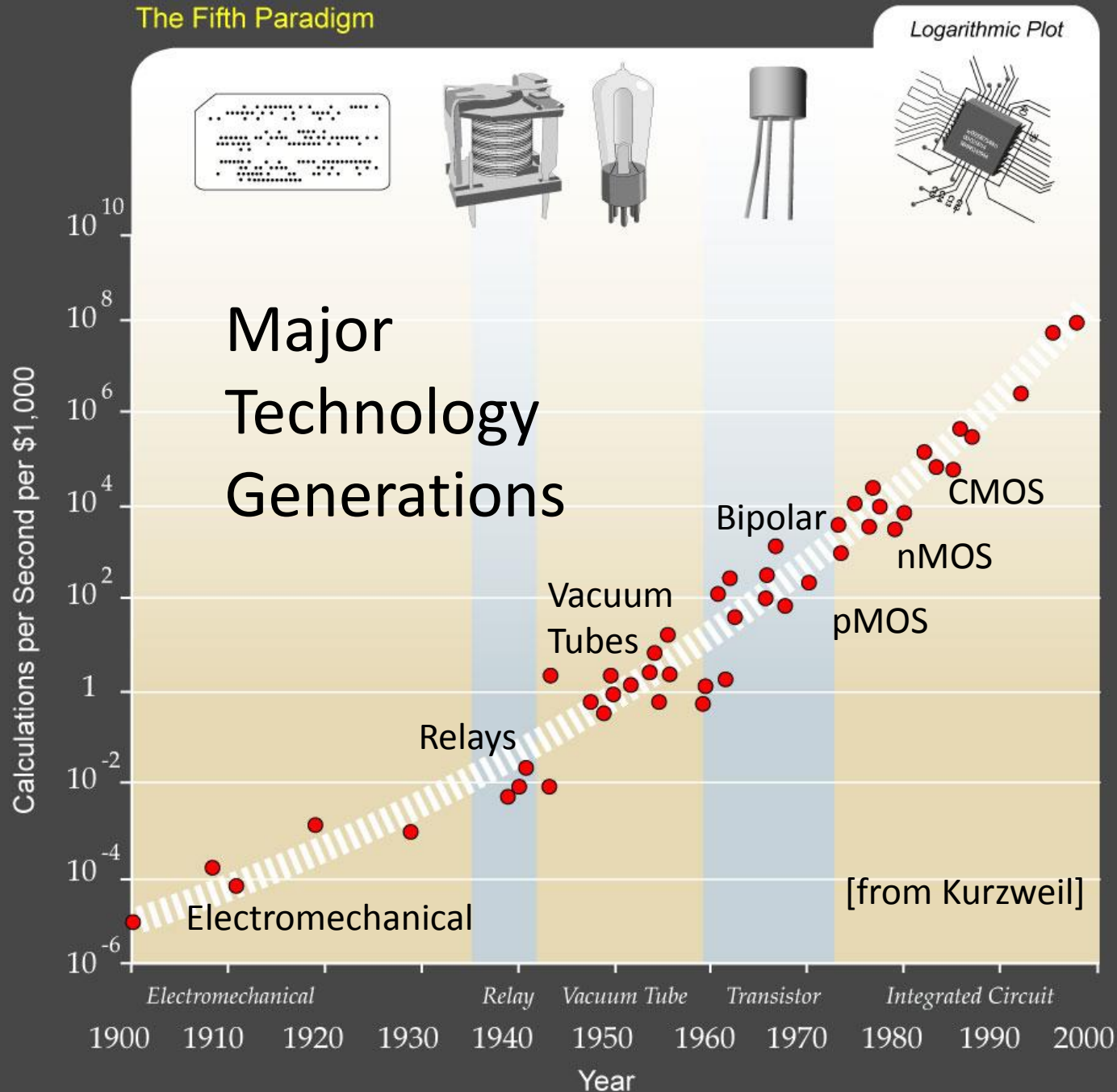
# Computers Now



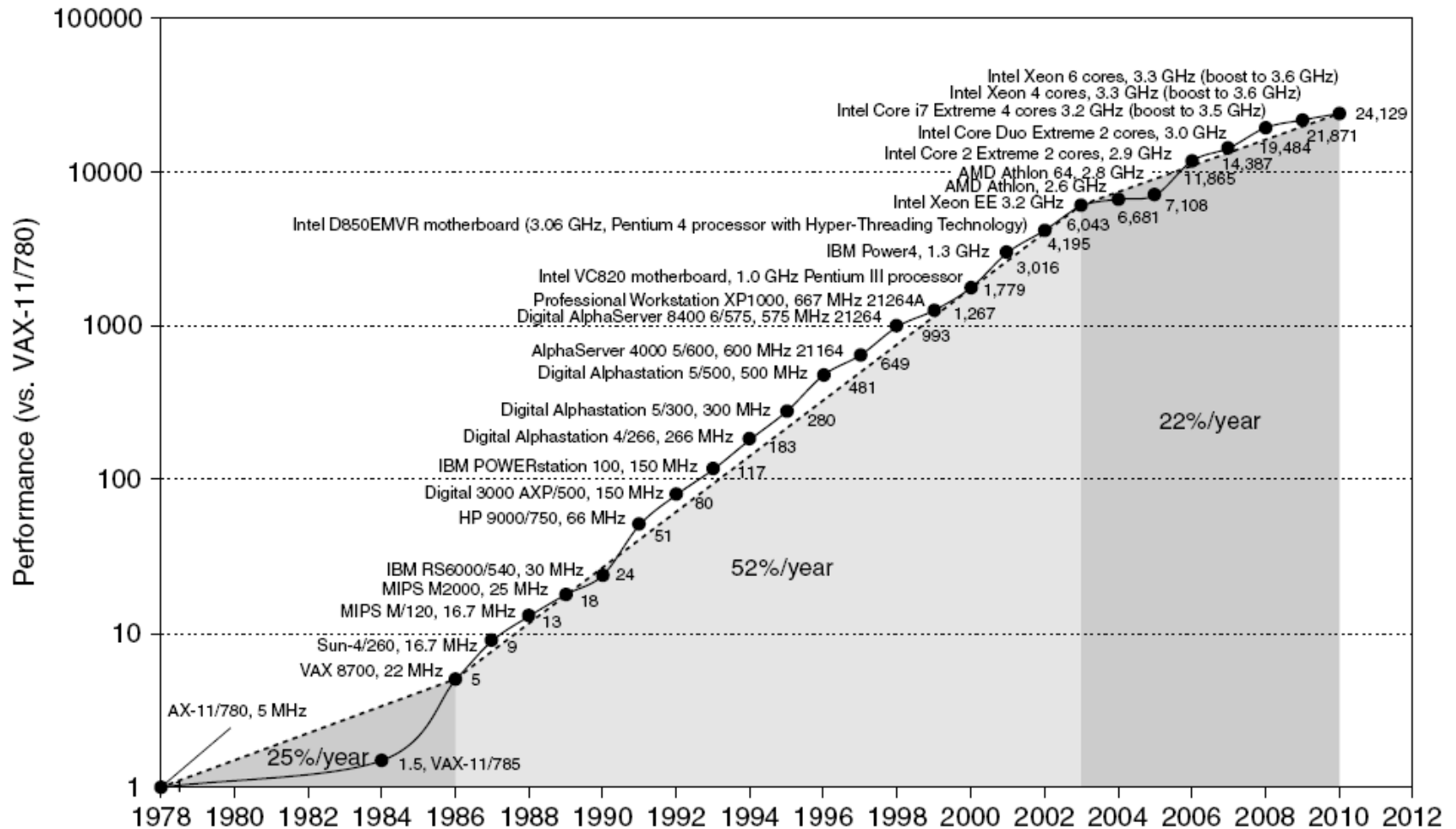


# Moore's Law

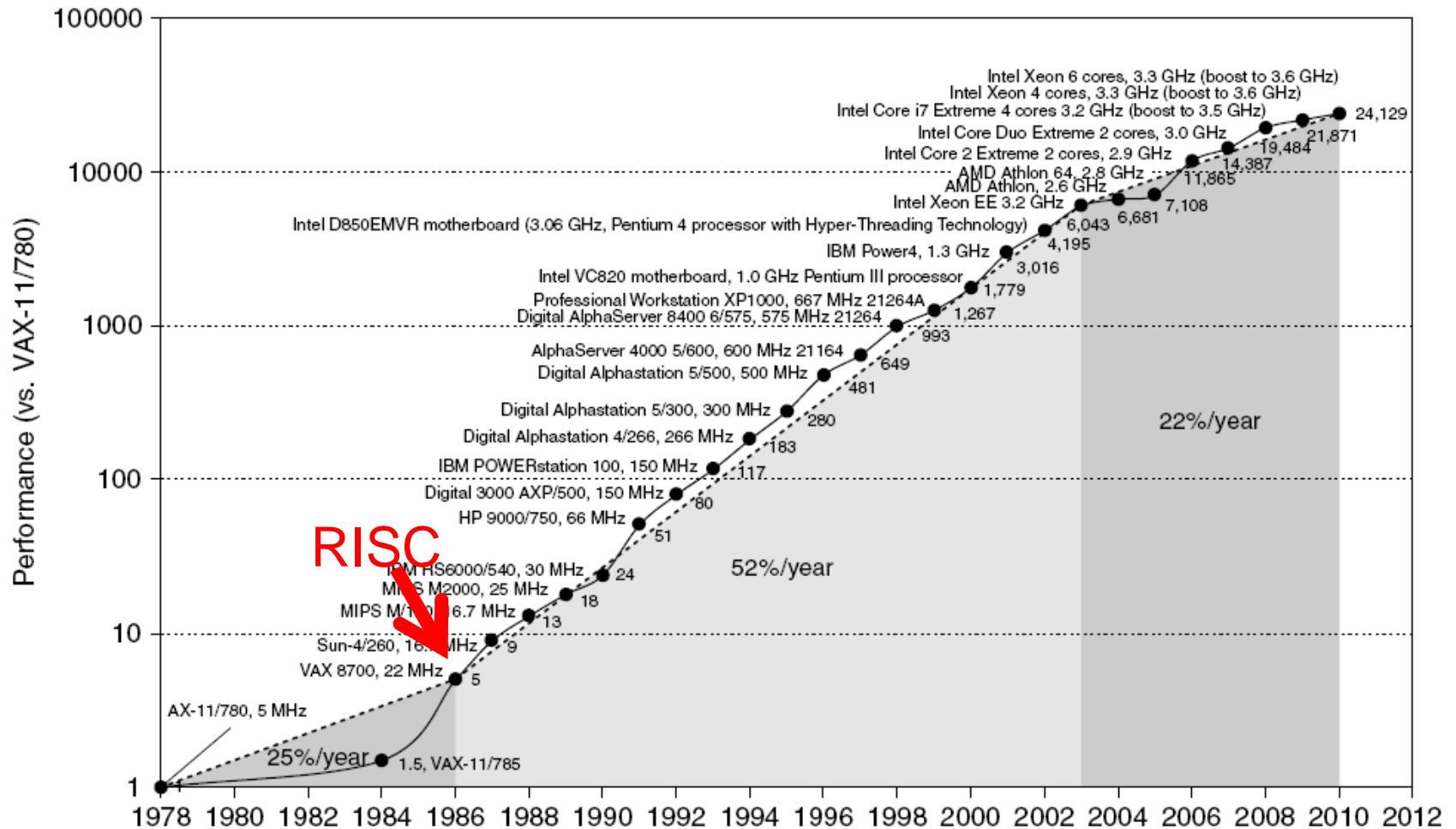
## The Fifth Paradigm



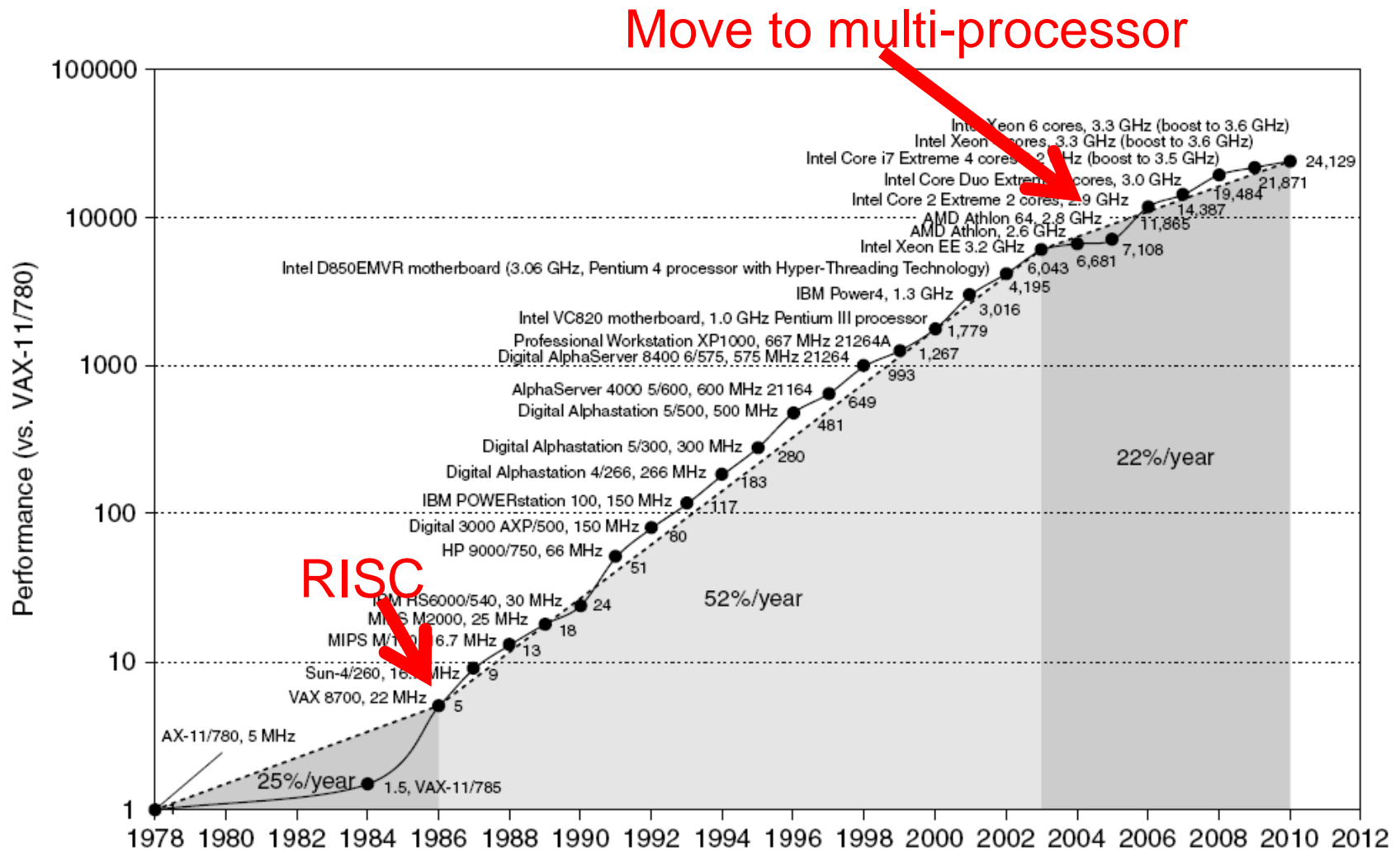
# Sequential Processor Performance



# Sequential Processor Performance



# Sequential Processor Performance





# Course Structure

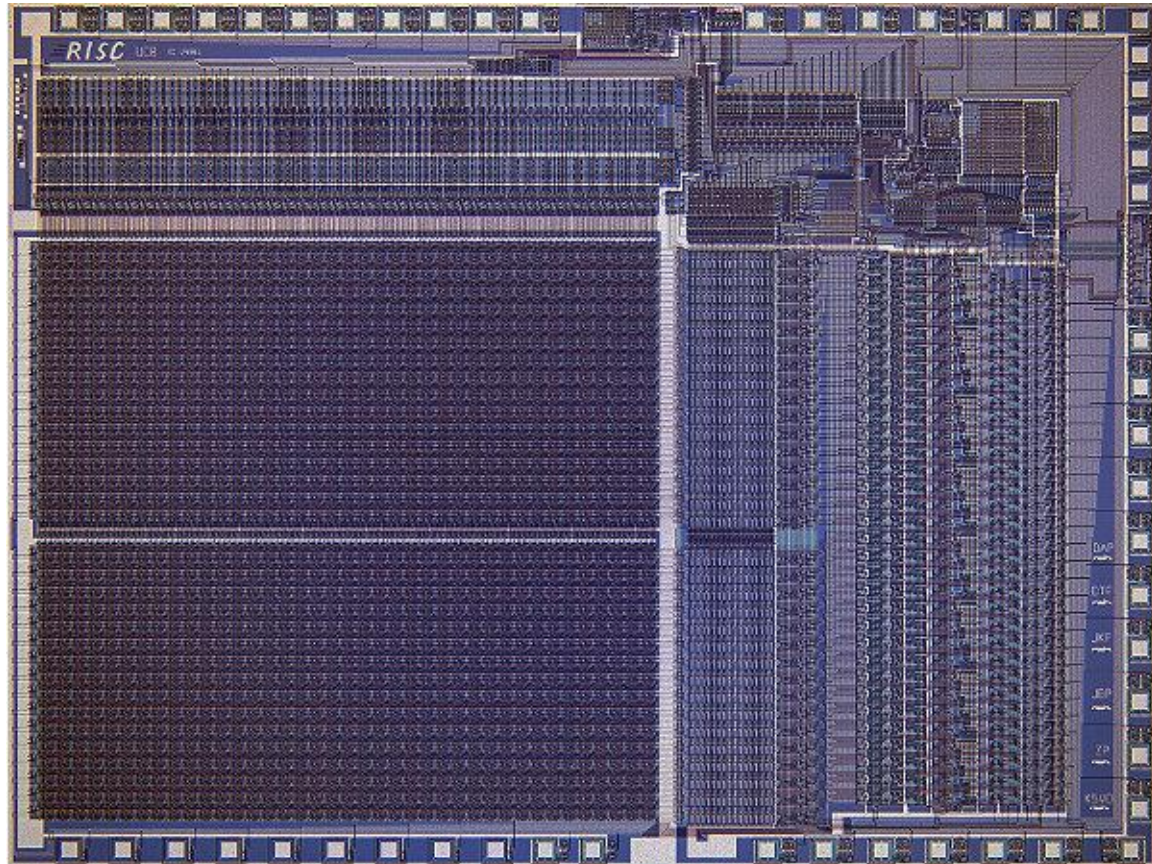
- Recommended Readings
- In-Lecture Questions
- Problem Sets
  - Very useful for exam preparation
  - Peer Evaluation
- Midterm
- Final Exam

# Course Content Computer Organization (ELE 375)

## Computer Organization

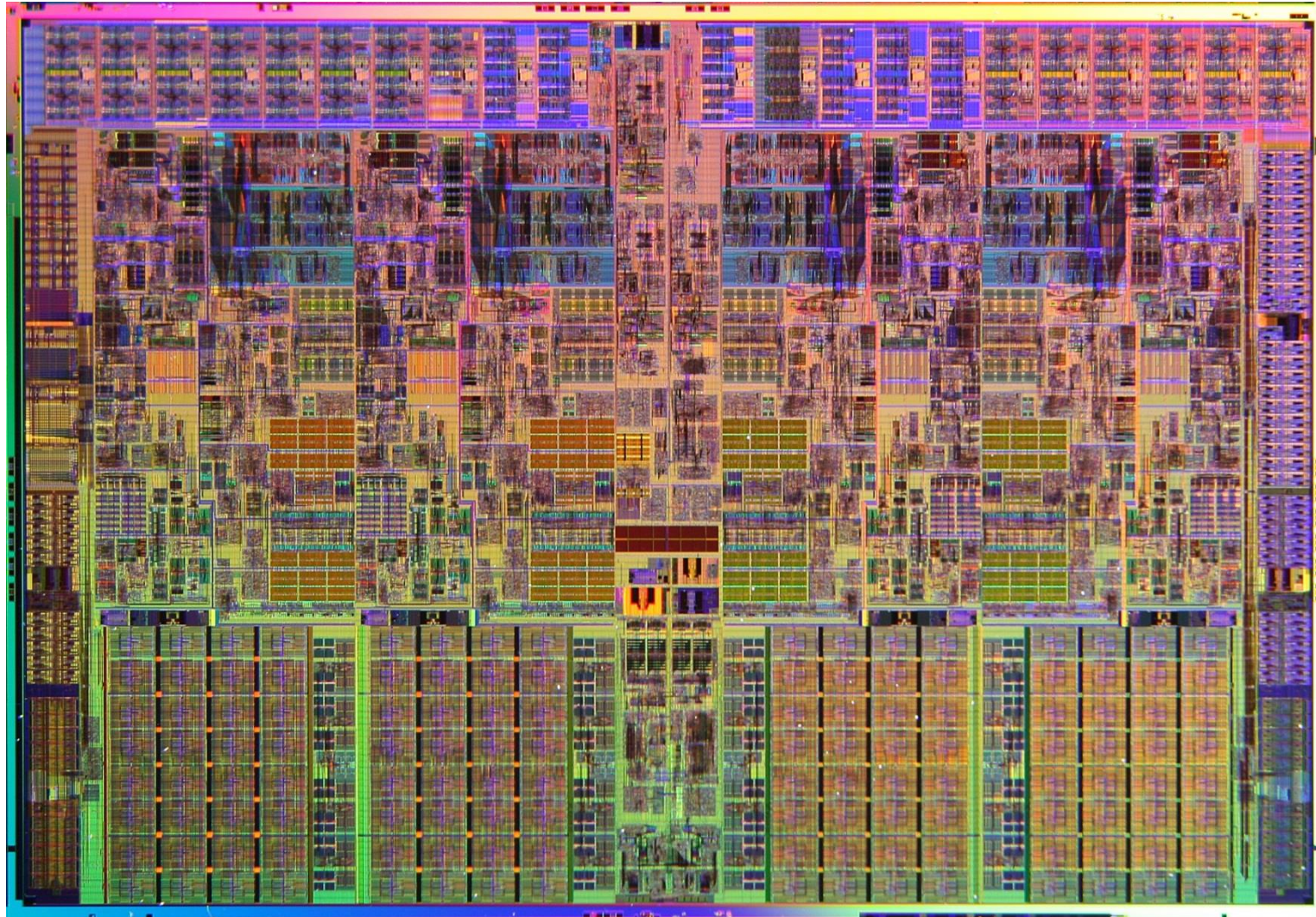
- Basic Pipelined Processor

~50,000 Transistors





# Course Content Computer Architecture (ELE 475)

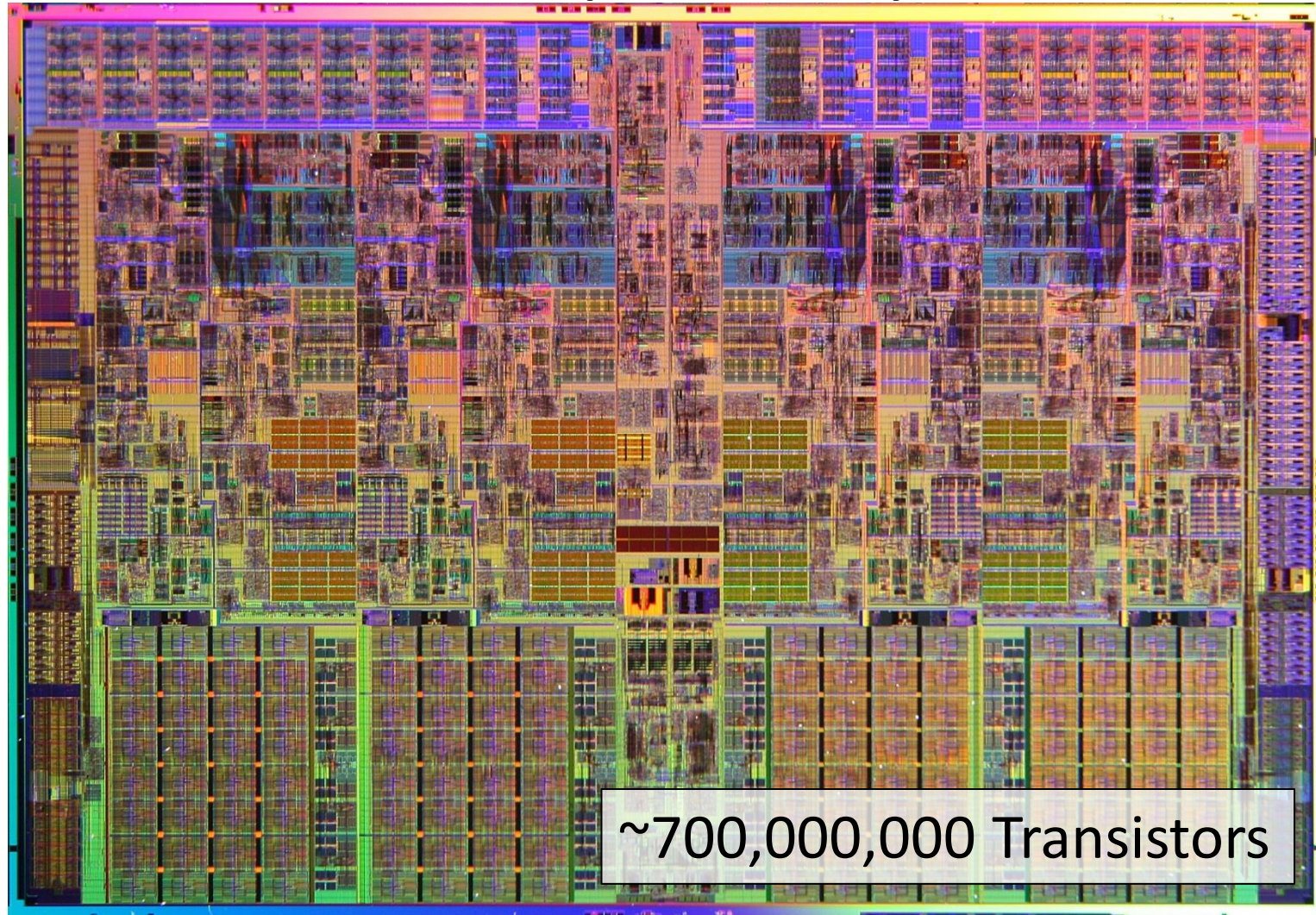


Intel Nehalem Processor, Original Core i7, Image Credit Intel:

[http://download.intel.com/pressroom/kits/corei7/images/Nehalem\\_Die\\_Shot\\_3.jpg](http://download.intel.com/pressroom/kits/corei7/images/Nehalem_Die_Shot_3.jpg)



# Course Content Computer Architecture (ELE 475)

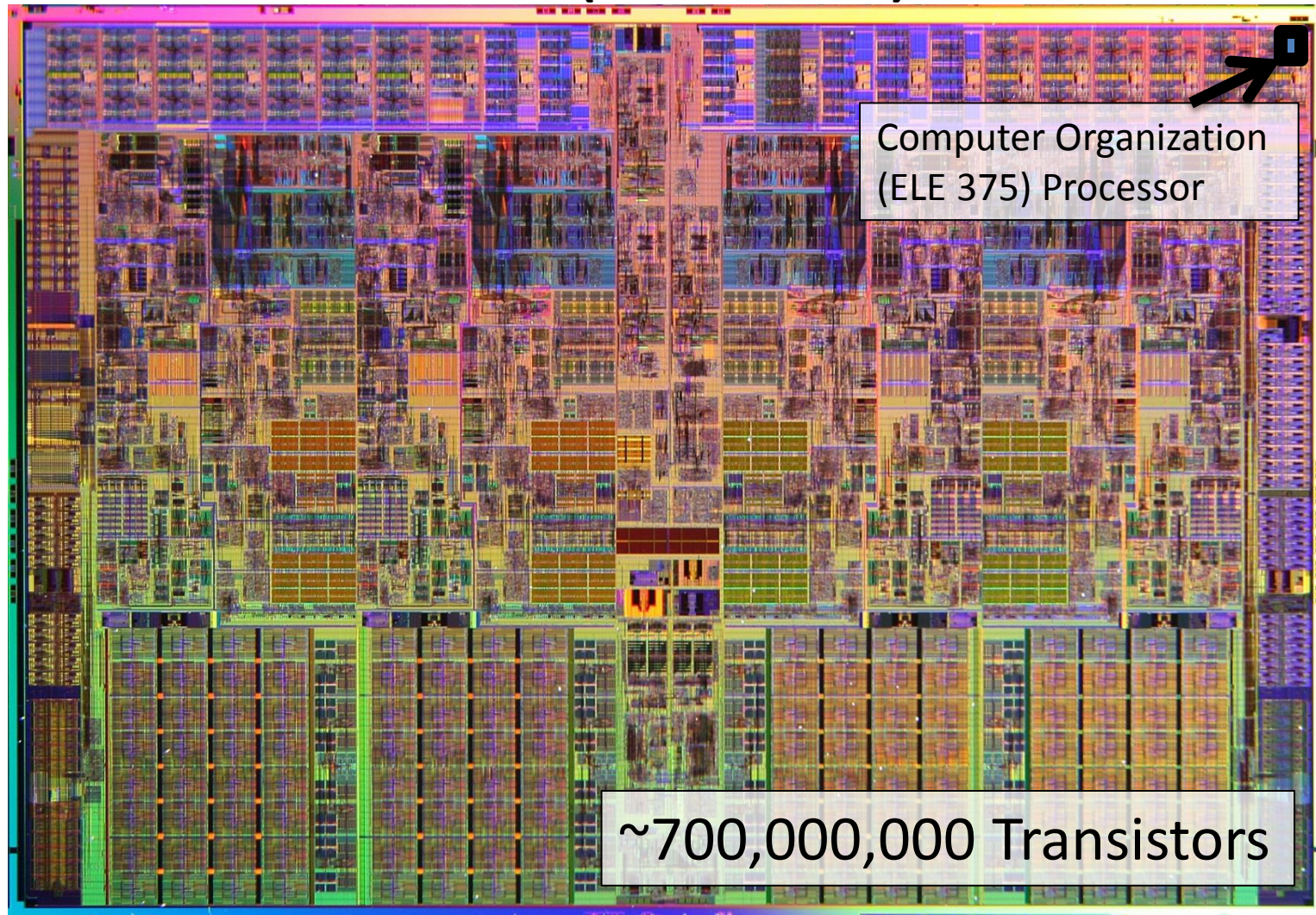


Intel Nehalem Processor, Original Core i7, Image Credit Intel:

[http://download.intel.com/pressroom/kits/corei7/images/Nehalem\\_Die\\_Shot\\_3.jpg](http://download.intel.com/pressroom/kits/corei7/images/Nehalem_Die_Shot_3.jpg)



# Course Content Computer Architecture (ELE 475)



Intel Nehalem Processor, Original Core i7, Image Credit Intel:

[http://download.intel.com/pressroom/kits/corei7/images/Nehalem\\_Die\\_Shot\\_3.jpg](http://download.intel.com/pressroom/kits/corei7/images/Nehalem_Die_Shot_3.jpg)



# Course Content Computer Architecture (ELE 475)

- Instruction Level Parallelism
  - Superscalar
  - Very Long Instruction Word (VLIW)
- Long Pipelines (Pipeline Parallelism)
- Advanced Memory and Caches
- Data Level Parallelism
  - Vector
  - GPU
- Thread Level Parallelism
  - Multithreading
  - Multiprocessor
  - Multicore
  - Manycore



Computer Organization  
(ELE 375) Processor

~700,000,000 Transistors

Intel Nehalem Processor, Original Core i7, Image Credit Intel:

[http://download.intel.com/pressroom/kits/corei7/images/Nehalem\\_Die\\_Shot\\_3.jpg](http://download.intel.com/pressroom/kits/corei7/images/Nehalem_Die_Shot_3.jpg)

# Architecture vs. Microarchitecture

“Architecture”/Instruction Set Architecture:

- Programmer visible state (Memory & Register)
- Operations (Instructions and how they work)
- Execution Semantics (interrupts)
- Input/Output
- Data Types/Sizes

Microarchitecture/Organization:

- Tradeoffs on how to implement ISA for some metric (Speed, Energy, Cost)
- Examples: Pipeline depth, number of pipelines, cache size, silicon area, peak power, execution ordering, bus widths, ALU widths

# Software Developments

- |            |   |
|------------|---|
| up to 1955 | Libraries of numerical routines <ul style="list-style-type: none"><li>- Floating point operations</li><li>- Transcendental functions</li><li>- Matrix manipulation, equation solvers, . . .</li></ul>                               |
| 1955-60    | <i>High level Languages</i> - Fortran 1956<br><i>Operating Systems</i> - <ul style="list-style-type: none"><li>- Assemblers, Loaders, Linkers, Compilers</li><li>- Accounting programs to keep track of usage and charges</li></ul> |



# Software Developments

up to 1955      Libraries of numerical routines

- Floating point operations
- Transcendental functions
- Matrix manipulation, equation solvers, . . .

1955-60      *High level Languages* - Fortran 1956  
*Operating Systems* -

- Assemblers, Loaders, Linkers, Compilers
- Accounting programs to keep track of usage and charges

Machines required *experienced operators*

- Most users could not be expected to understand these programs, much less write them
- Machines had to be sold with a lot of resident software

# Compatibility Problem at IBM

By early 1960's, IBM had 4 incompatible lines of computers!

701	⇒	7094
650	⇒	7074
702	⇒	7080
1401	⇒	7010

# Compatibility Problem at IBM

By early 1960's, IBM had 4 incompatible lines of computers!

701	⇒	7094
650	⇒	7074
702	⇒	7080
1401	⇒	7010

Each system had its own

- Instruction set
- I/O system and Secondary Storage:  
magnetic tapes, drums and disks
- assemblers, compilers, libraries,...
- market niche business, scientific, real time, ...

# Compatibility Problem at IBM

By early 1960's, IBM had 4 incompatible lines of computers!

701	⇒	7094
650	⇒	7074
702	⇒	7080
1401	⇒	7010

Each system had its own

- Instruction set
- I/O system and Secondary Storage:  
magnetic tapes, drums and disks
- assemblers, compilers, libraries,...
- market niche business, scientific, real time, ...

⇒ IBM 360

# IBM 360 : Design Premises

*Amdahl, Blaauw and Brooks, 1964*

- The design must lend itself to *growth and successor machines*
- General method for connecting I/O devices
- Total performance - answers per month rather than bits per microsecond  $\Rightarrow$  *programming aids*
- Machine must be capable of *supervising itself* without manual intervention
- Built-in *hardware fault checking* and locating aids to reduce down time
- Simple to assemble systems with redundant I/O devices, memories etc. for *fault tolerance*
- Some problems required floating-point larger than 36 bits

# IBM 360: A *General-Purpose Register (GPR)* Machine

- Processor State
  - 16 General-Purpose 32-bit Registers
    - *may be used as index and base register*
    - *Register 0 has some special properties*
  - 4 Floating Point 64-bit Registers
  - A Program Status Word (PSW)
    - *PC, Condition codes, Control flags*
- A 32-bit machine with 24-bit addresses
  - But no instruction contains a 24-bit address!
- Data Formats
  - 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words

# IBM 360: A *General-Purpose Register (GPR)* Machine

- Processor State
  - 16 General-Purpose 32-bit Registers
    - *may be used as index and base register*
    - *Register 0 has some special properties*
  - 4 Floating Point 64-bit Registers
  - A Program Status Word (PSW)
    - *PC, Condition codes, Control flags*
- A 32-bit machine with 24-bit addresses
  - But no instruction contains a 24-bit address!
- Data Formats
  - 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words



*The IBM 360 is why bytes are 8-bits long today!*

# IBM 360: Initial Implementations

	<i>Model 30</i>	<i>. . .</i>	<i>Model 70</i>
<i>Storage</i>	8K - 64 KB		256K - 512 KB
<i>Datapath</i>	8-bit		64-bit
<i>Circuit Delay</i>	30 nsec/level		5 nsec/level
<i>Local Store</i>	Main Store		Transistor Registers
<i>Control Store</i>	Read only 1μsec		Conventional circuits

IBM 360 instruction set architecture (ISA) completely hid the underlying technological differences between various models.

Milestone: The first true ISA designed as portable hardware-software interface!



# IBM 360: Initial Implementations

	<i>Model 30</i>	<i>. . .</i>	<i>Model 70</i>
<i>Storage</i>	8K - 64 KB		256K - 512 KB
<i>Datapath</i>	8-bit		64-bit
<i>Circuit Delay</i>	30 nsec/level		5 nsec/level
<i>Local Store</i>	Main Store		Transistor Registers
<i>Control Store</i>	Read only 1μsec		Conventional circuits

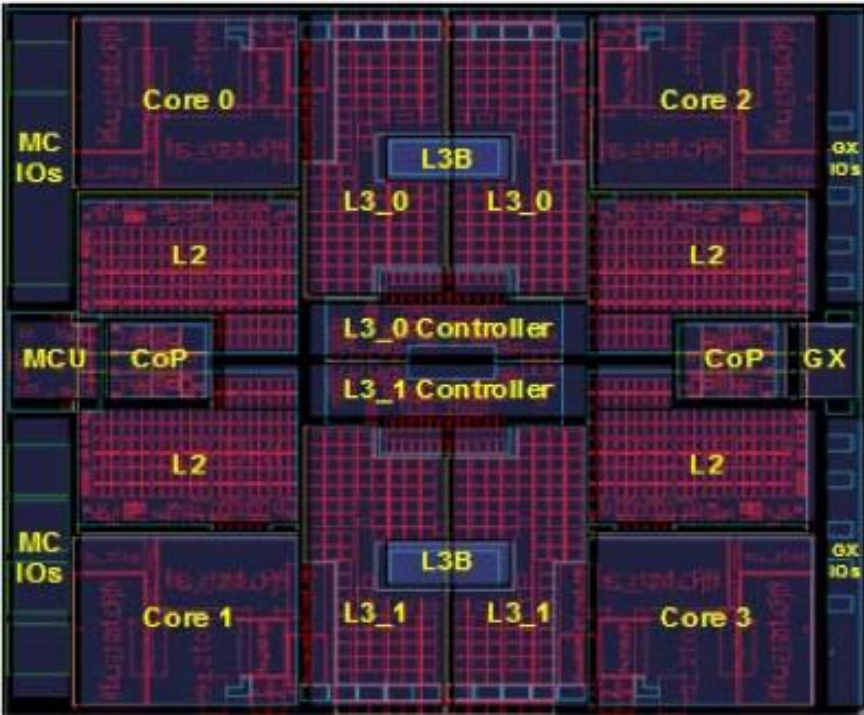
IBM 360 instruction set architecture (ISA) completely hid the underlying technological differences between various models.

Milestone: The first true ISA designed as portable hardware-software interface!

*With minor modifications it still survives today!*

# IBM 360: 47 years later...

## The zSeries z11 Microprocessor



[ IBM, Kevin Shum, HotChips, 2010]

Image Credit: IBM

Courtesy of International Business  
Machines Corporation, © International  
Business Machines Corporation.

- 5.2 GHz in IBM 45nm PD-SOI CMOS technology
- 1.4 billion transistors in 512 mm<sup>2</sup>
- 64-bit virtual addressing
  - original S/360 was 24-bit, and S/370 was 31-bit extension
- Quad-core design
- Three-issue out-of-order superscalar pipeline
- Out-of-order memory accesses
- Redundant datapaths
  - every instruction performed in two parallel datapaths and results compared
- 64KB L1 I-cache, 128KB L1 D-cache on-chip
- 1.5MB private L2 unified cache per core, on-chip
- On-Chip 24MB eDRAM L3 cache
- Scales to 96-core multiprocessor with 768MB of shared L4 eDRAM

# Same Architecture Different Microarchitecture

## AMD Phenom X4

- X86 Instruction Set
- Quad Core
- 125W
- Decode 3 Instructions/Cycle/Core
- 64KB L1 I Cache, 64KB L1 D Cache
- 512KB L2 Cache
- Out-of-order
- 2.6GHz

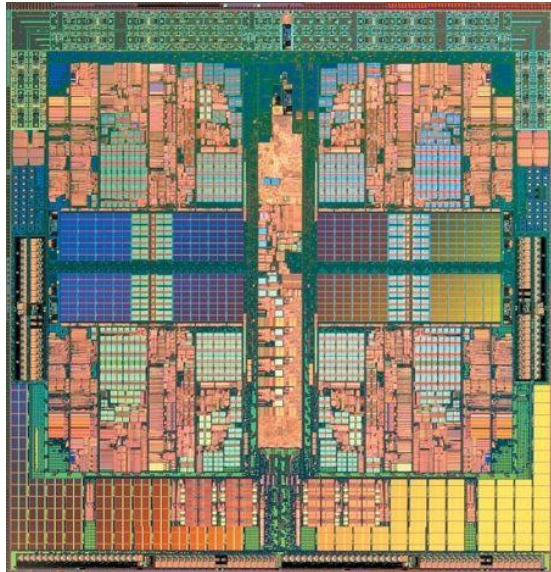


Image Credit: AMD

## Intel Atom

- X86 Instruction Set
- Single Core
- 2W
- Decode 2 Instructions/Cycle/Core
- 32KB L1 I Cache, 24KB L1 D Cache
- 512KB L2 Cache
- In-order
- 1.6GHz

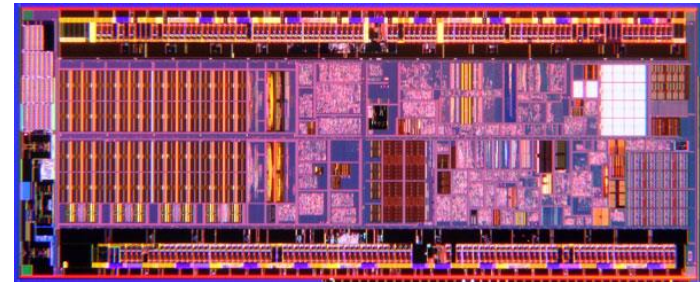


Image Credit: Intel

# Different Architecture

## Different Microarchitecture

### AMD Phenom X4

- X86 Instruction Set
- Quad Core
- 125W
- Decode 3 Instructions/Cycle/Core
- 64KB L1 I Cache, 64KB L1 D Cache
- 512KB L2 Cache
- Out-of-order
- 2.6GHz

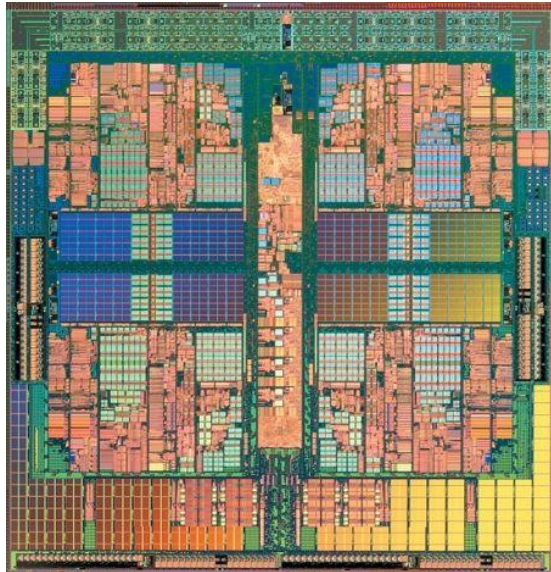


Image Credit: AMD

### IBM POWER7

- Power Instruction Set
- Eight Core
- 200W
- Decode 6 Instructions/Cycle/Core
- 32KB L1 I Cache, 32KB L1 D Cache
- 256KB L2 Cache
- Out-of-order
- 4.25GHz

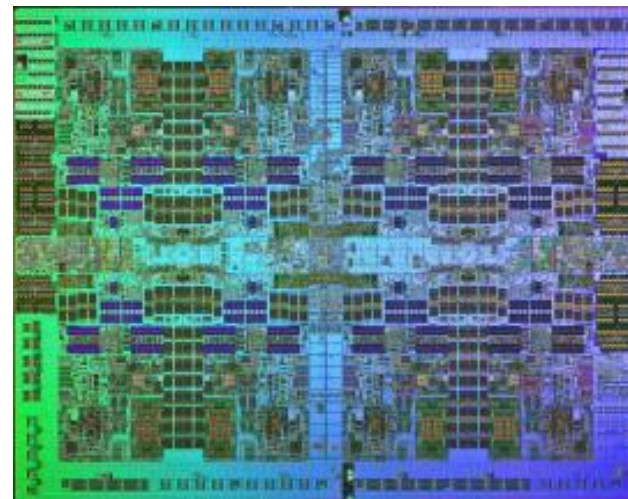


Image Credit: IBM

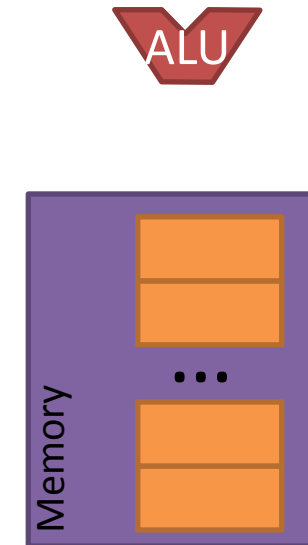
Courtesy of International Business Machines Corporation, © International Business Machines Corporation.

# Where Do Operands Come from And Where Do Results Go?

# Where Do Operands Come from And Where Do Results Go?



# Where Do Operands Come from And Where Do Results Go?





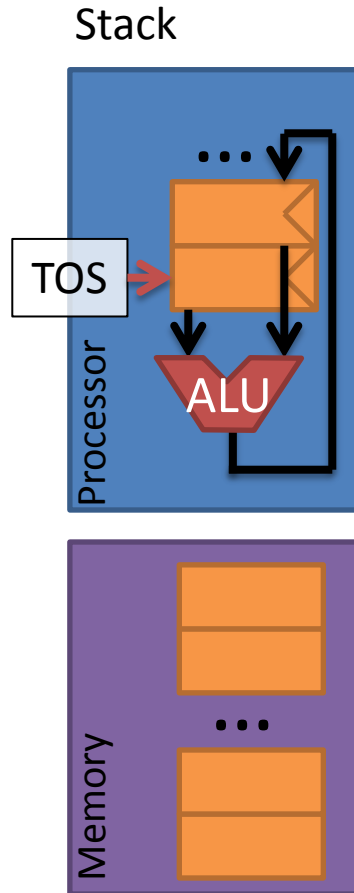
# Where Do Operands Come from And Where Do Results Go?



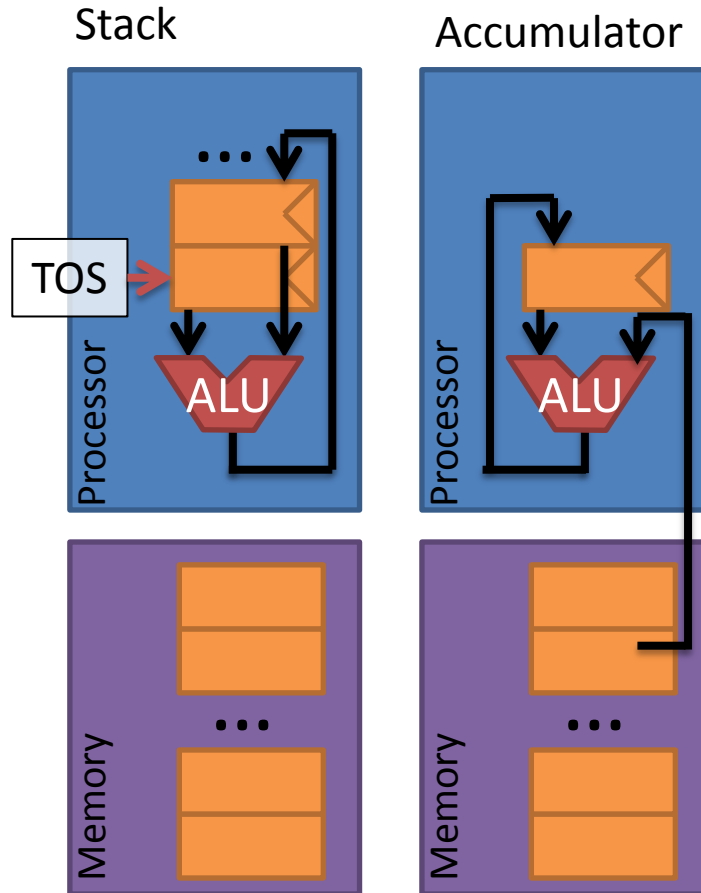


# Where Do Operands Come from And Where Do Results Go?

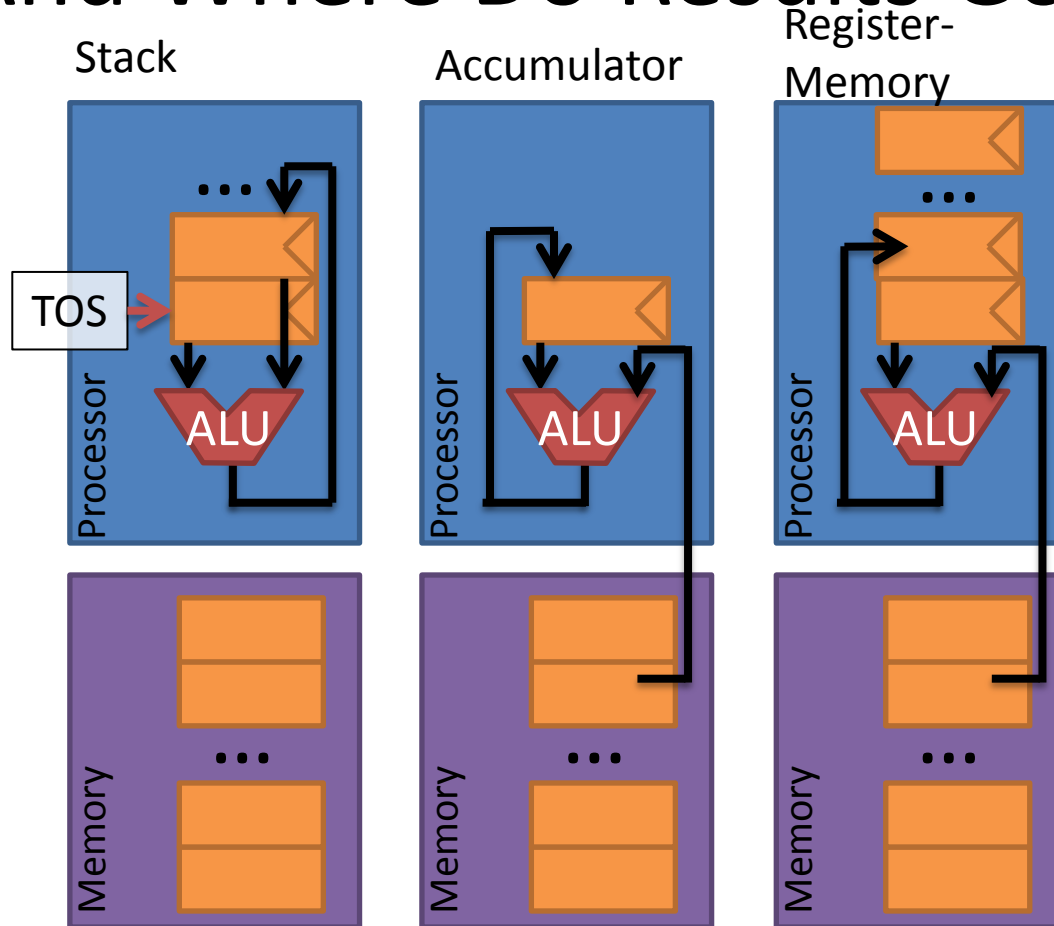
# Where Do Operands Come from And Where Do Results Go?



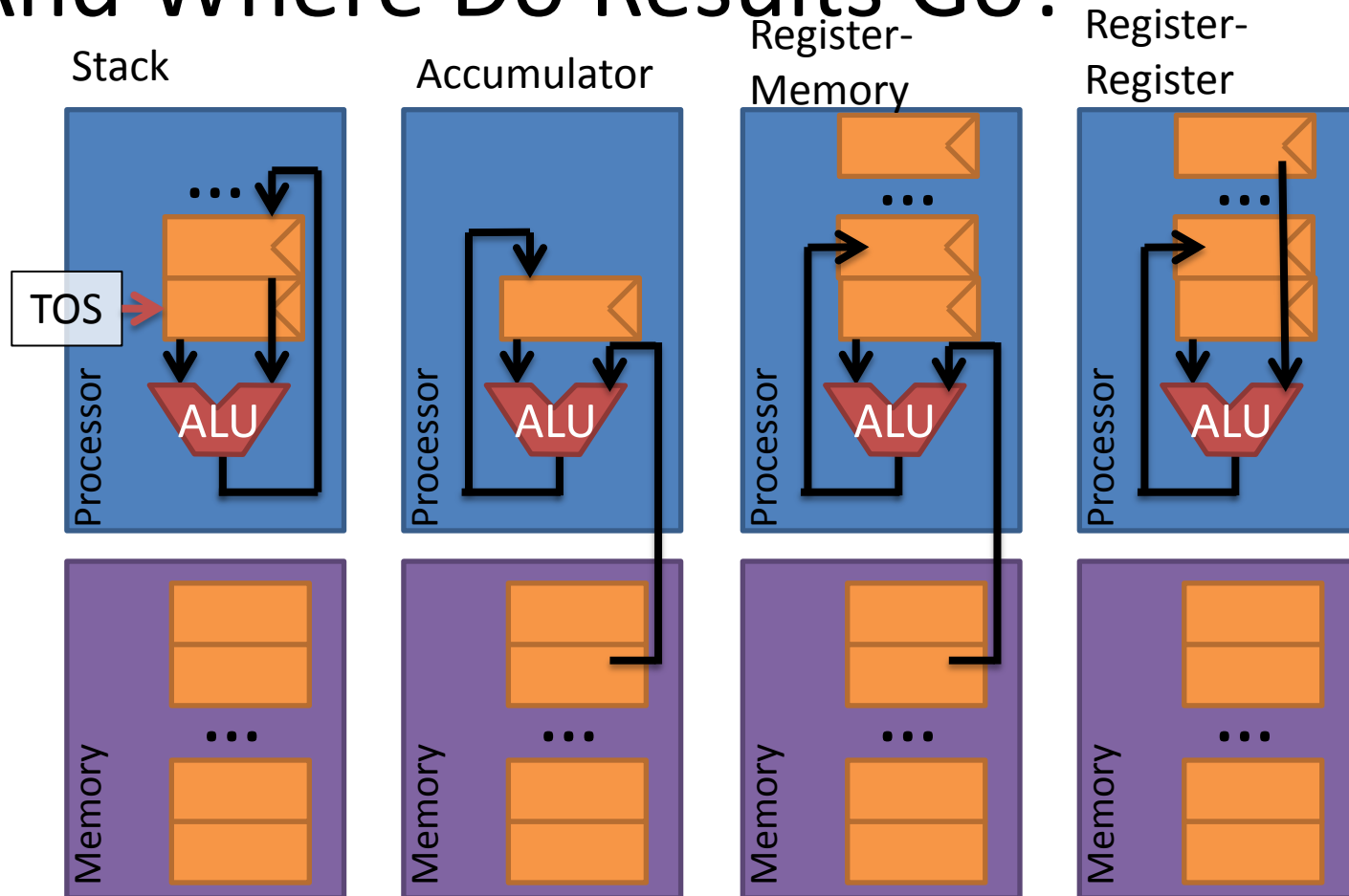
# Where Do Operands Come from And Where Do Results Go?



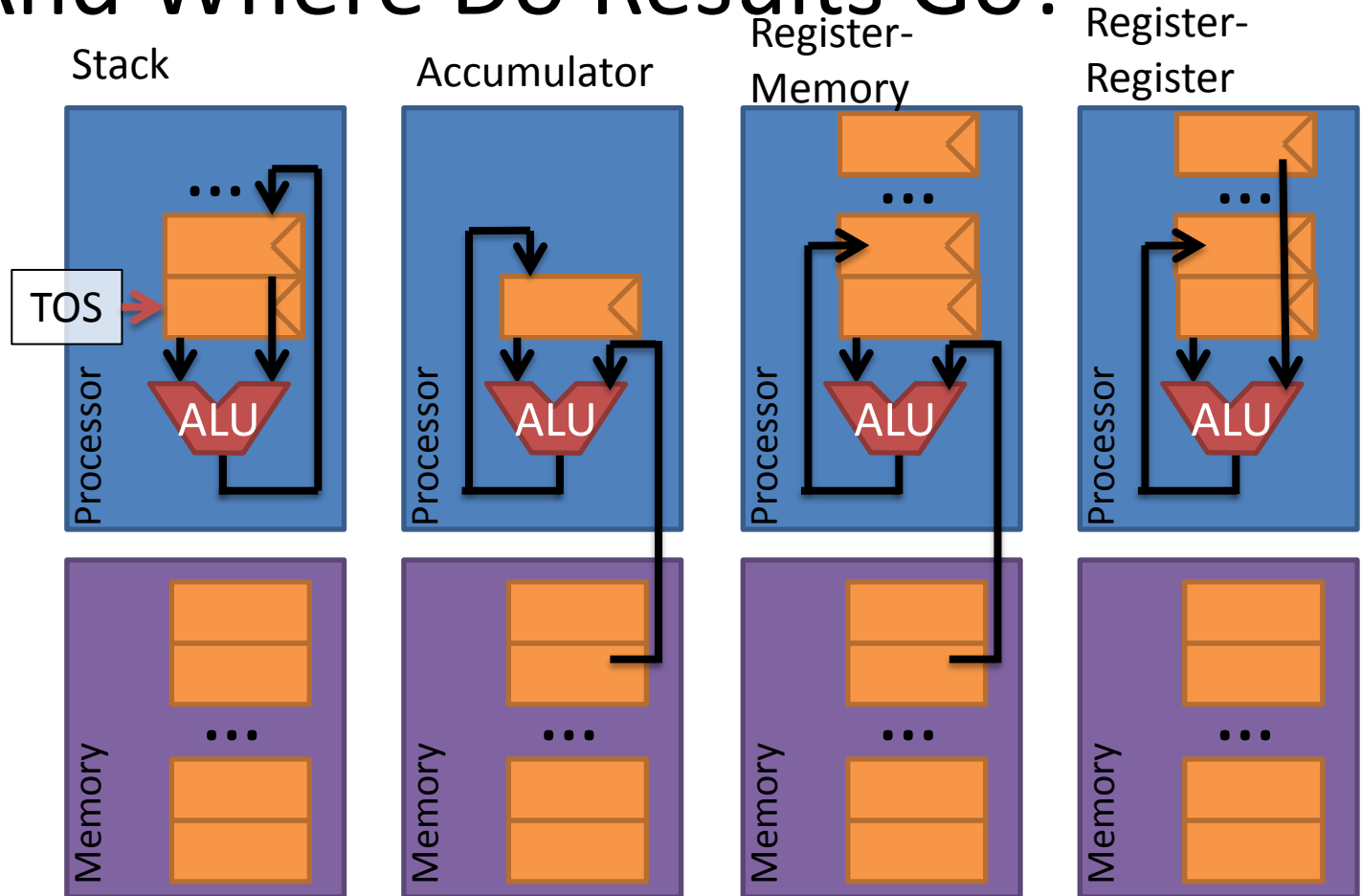
# Where Do Operands Come from And Where Do Results Go?



# Where Do Operands Come from And Where Do Results Go?



# Where Do Operands Come from And Where Do Results Go?



Number Explicitly  
Named Operands:

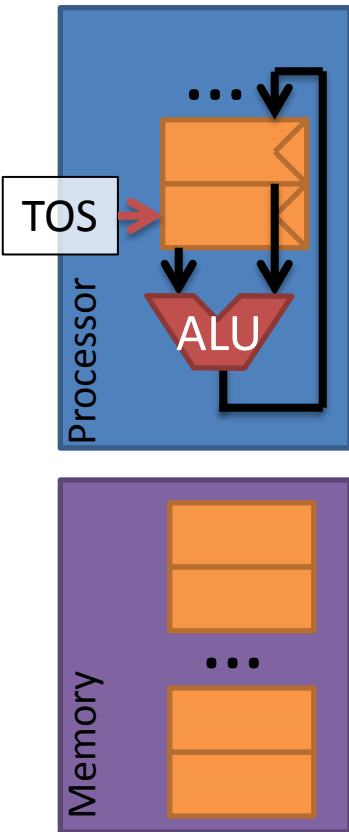
0

1

2 or 3

2 or 3

# Stack-Based Instruction Set Architecture (ISA)



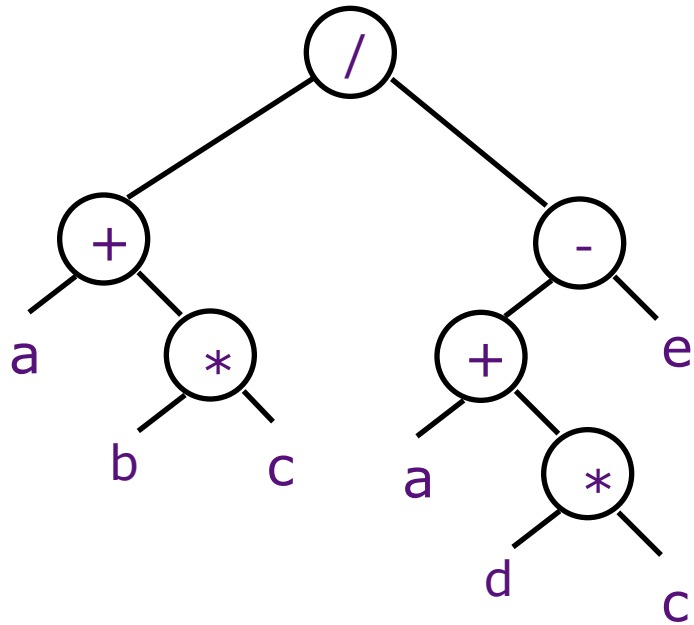
- Burrough's B5000 (1960)
- Burrough's B6700
- HP 3000
- ICL 2900
- Symbolics 3600

Modern

- Inmos Transputer
- Forth machines
- Java Virtual Machine
- Intel x87 Floating Point Unit

# Evaluation of Expressions

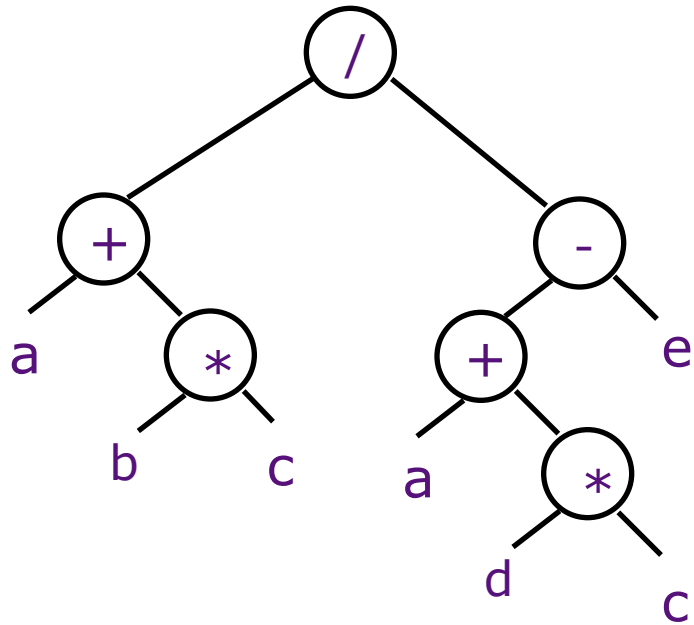
$(a + b * c) / (a + d * c - e)$





# Evaluation of Expressions

$(a + b * c) / (a + d * c - e)$

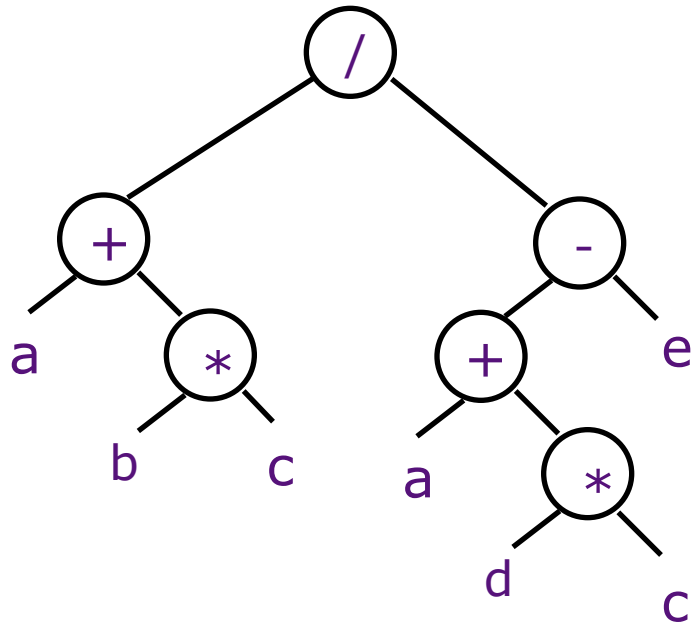


Reverse Polish

$a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /$

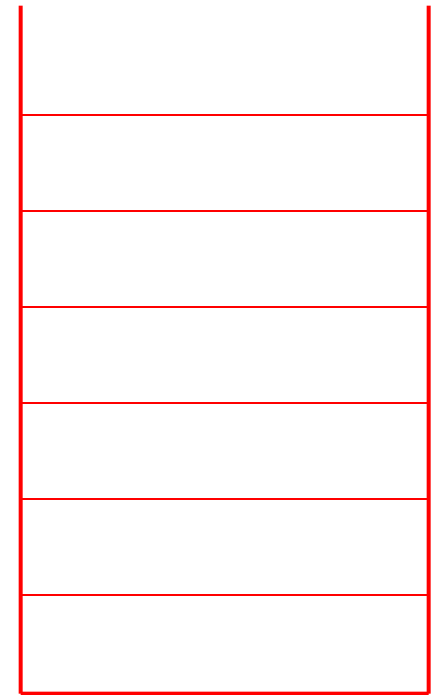
# Evaluation of Expressions

$(a + b * c) / (a + d * c - e)$



Reverse Polish

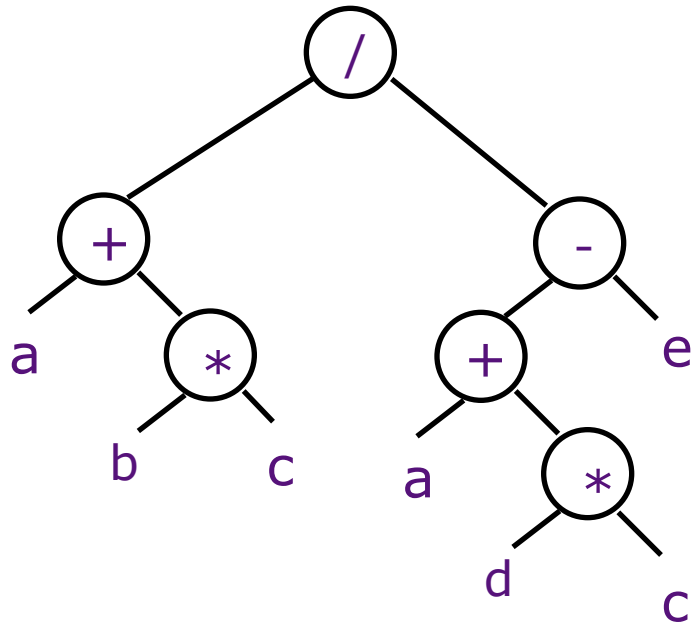
$a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /$



Evaluation Stack

# Evaluation of Expressions

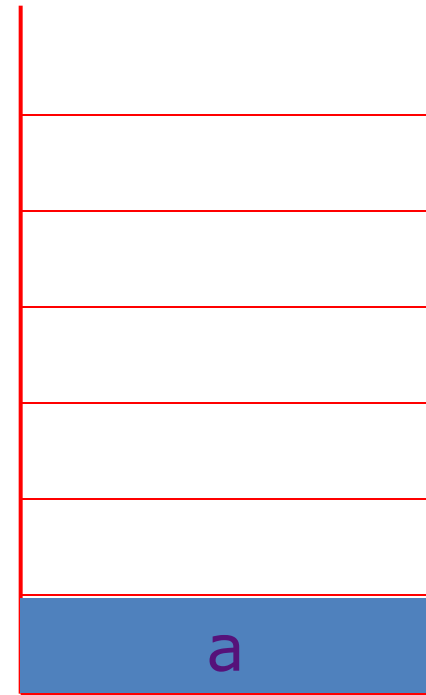
$(a + b * c) / (a + d * c - e)$



Reverse Polish

$a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /$

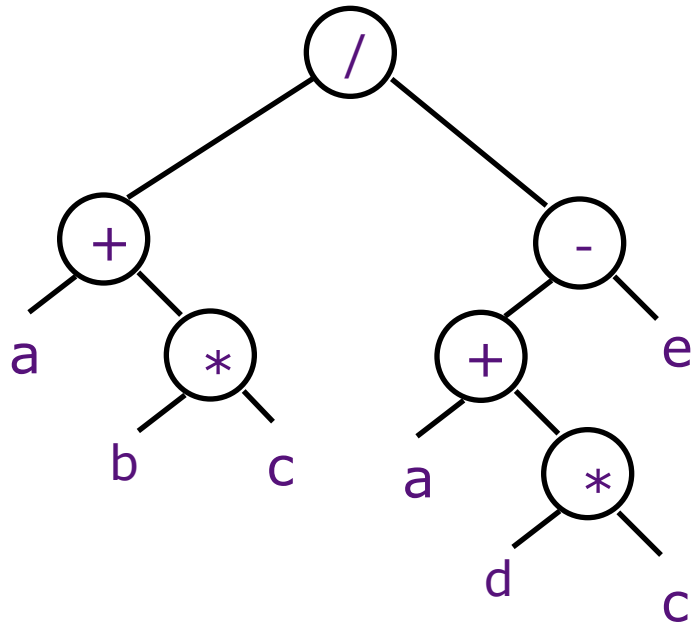
↑  
push a



Evaluation Stack

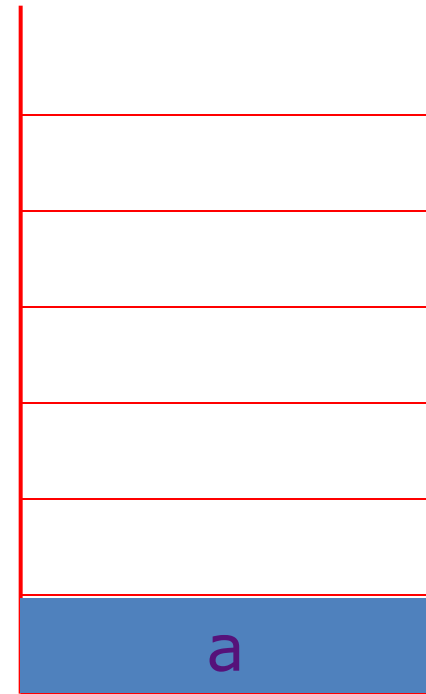
# Evaluation of Expressions

$(a + b * c) / (a + d * c - e)$



Reverse Polish

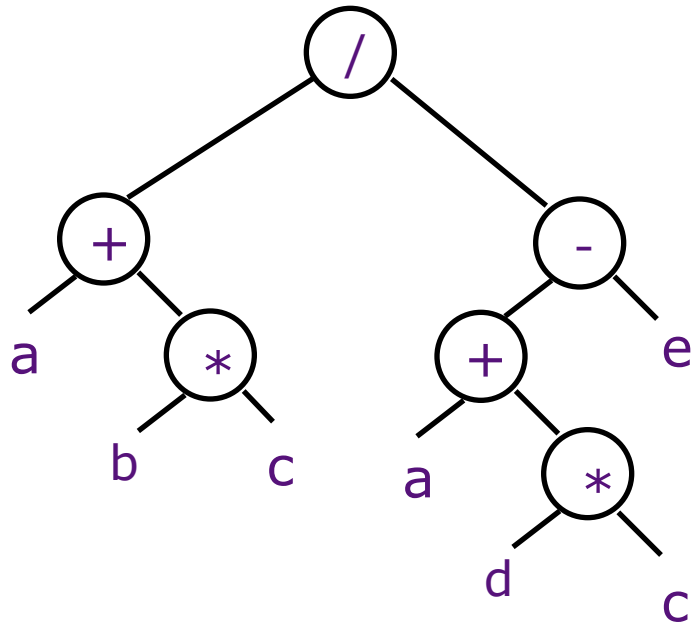
$a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /$



Evaluation Stack

# Evaluation of Expressions

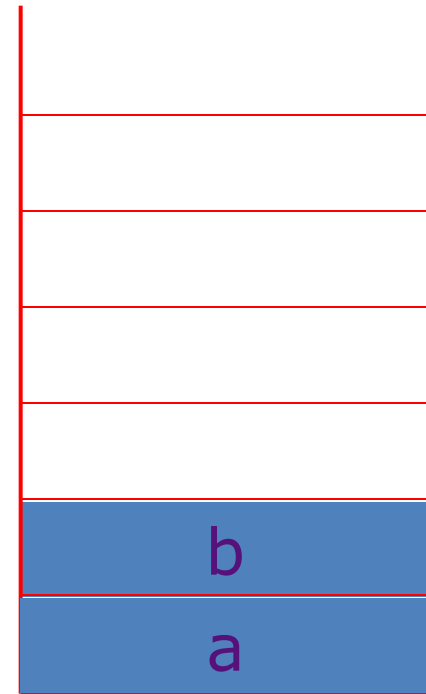
$(a + b * c) / (a + d * c - e)$



Reverse Polish

$a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /$

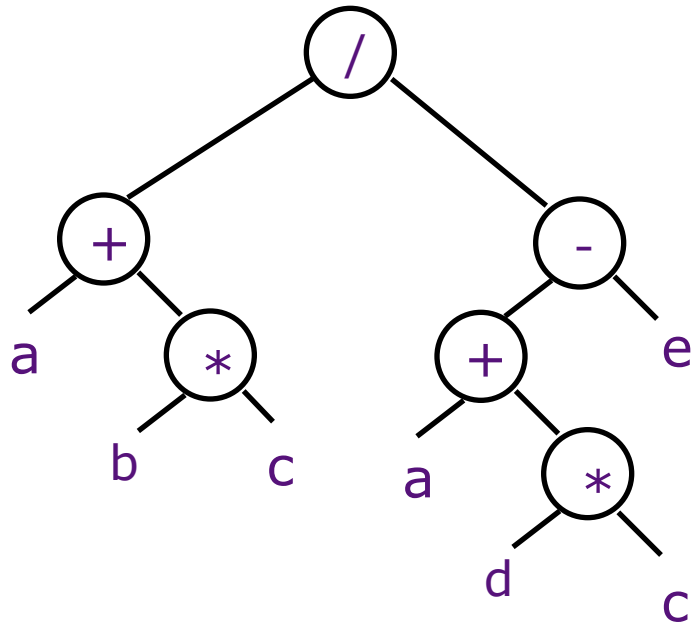
↑  
push b



Evaluation Stack

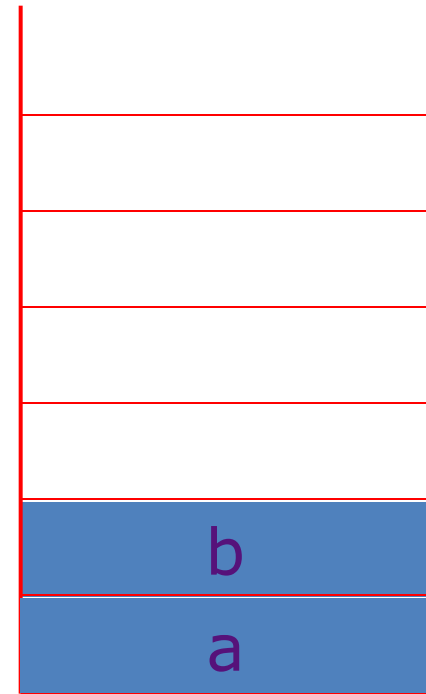
# Evaluation of Expressions

$(a + b * c) / (a + d * c - e)$



Reverse Polish

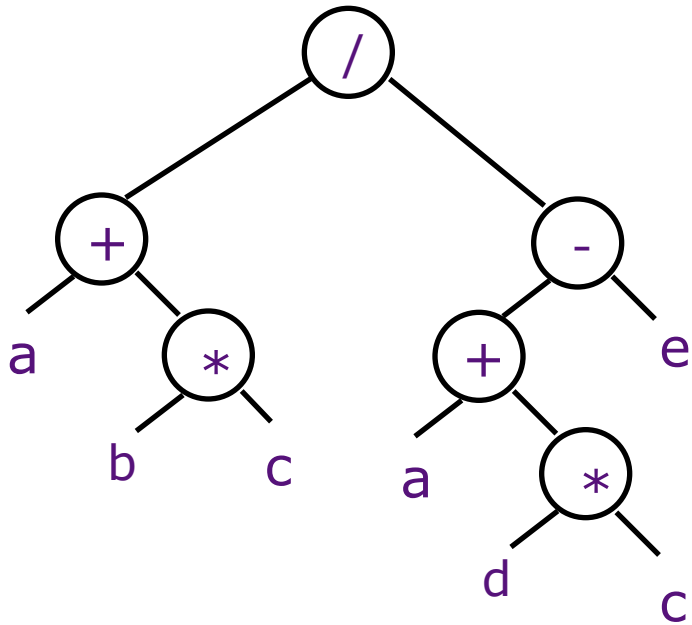
$a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /$



Evaluation Stack

# Evaluation of Expressions

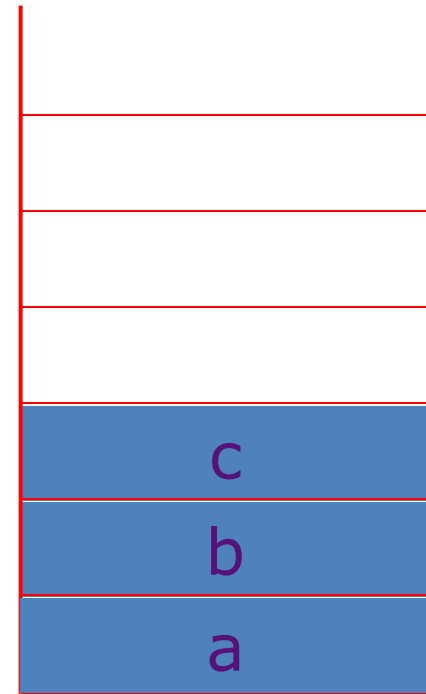
$(a + b * c) / (a + d * c - e)$



Reverse Polish

$a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /$

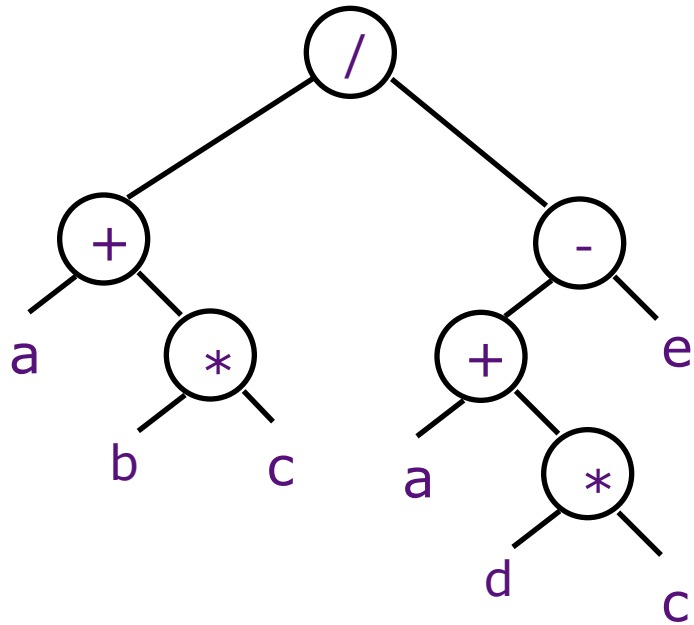
↑  
push c



Evaluation Stack

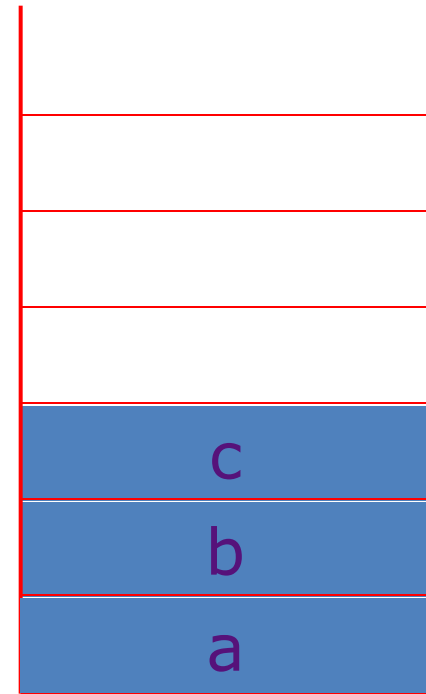
# Evaluation of Expressions

$(a + b * c) / (a + d * c - e)$



Reverse Polish

$a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /$

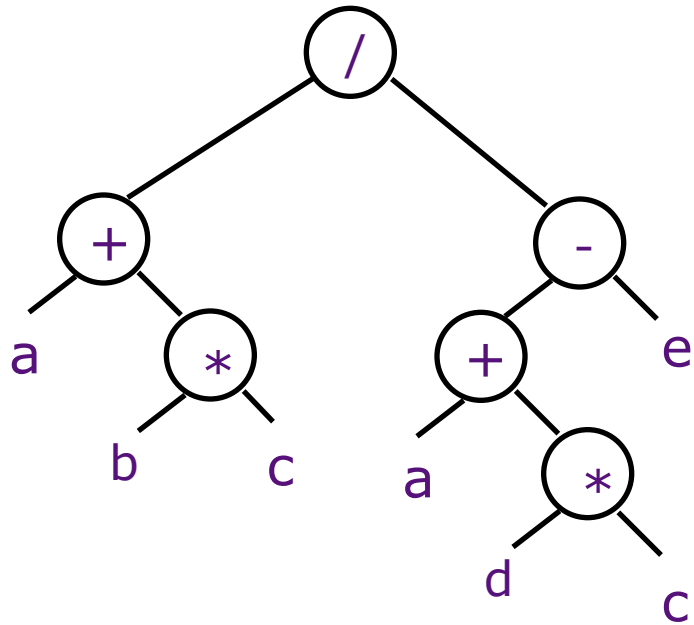


Evaluation Stack



# Evaluation of Expressions

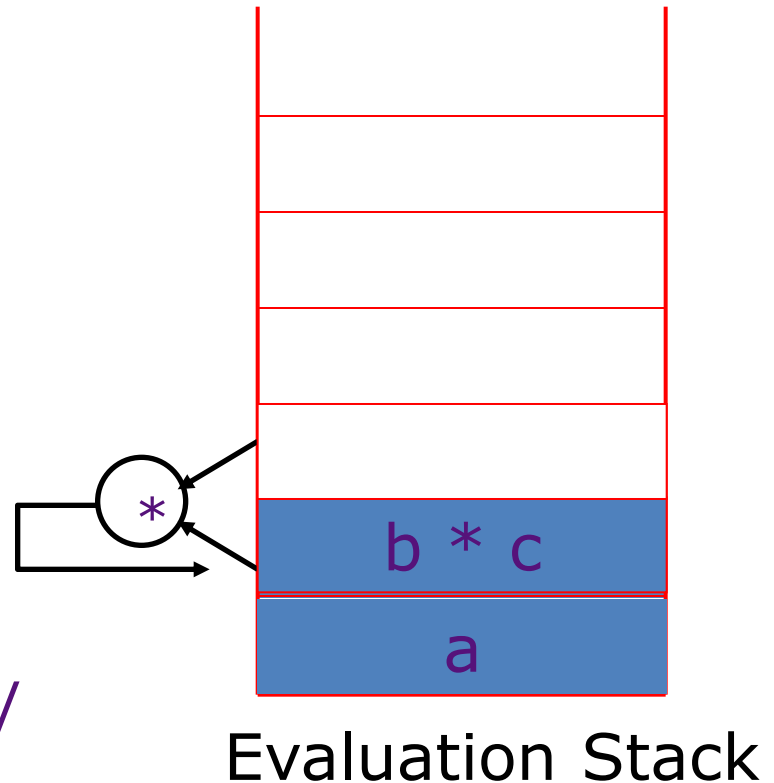
$(a + b * c) / (a + d * c - e)$



Reverse Polish

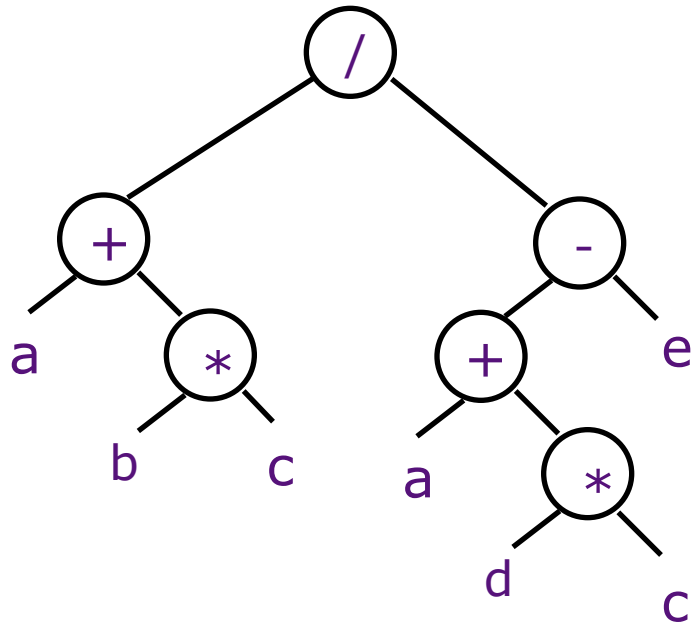
$a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /$

↑  
multiply



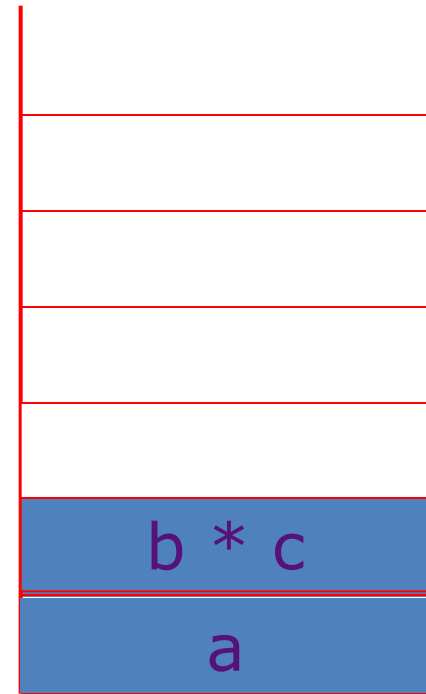
# Evaluation of Expressions

$(a + b * c) / (a + d * c - e)$



Reverse Polish

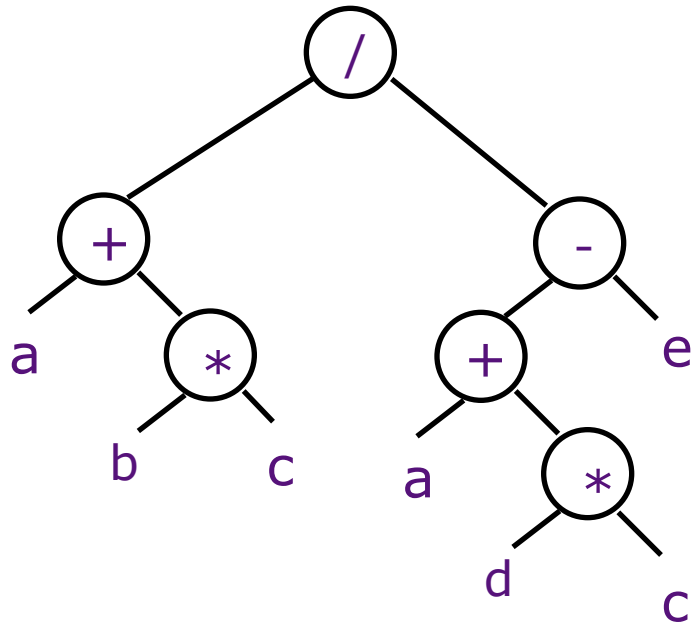
$a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /$



Evaluation Stack

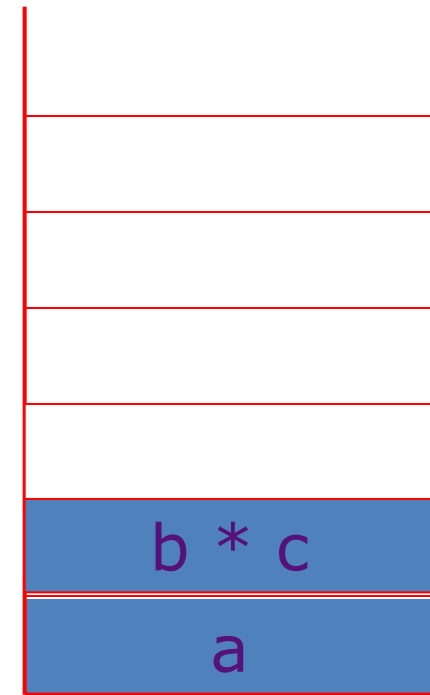
# Evaluation of Expressions

$(a + b * c) / (a + d * c - e)$



Reverse Polish

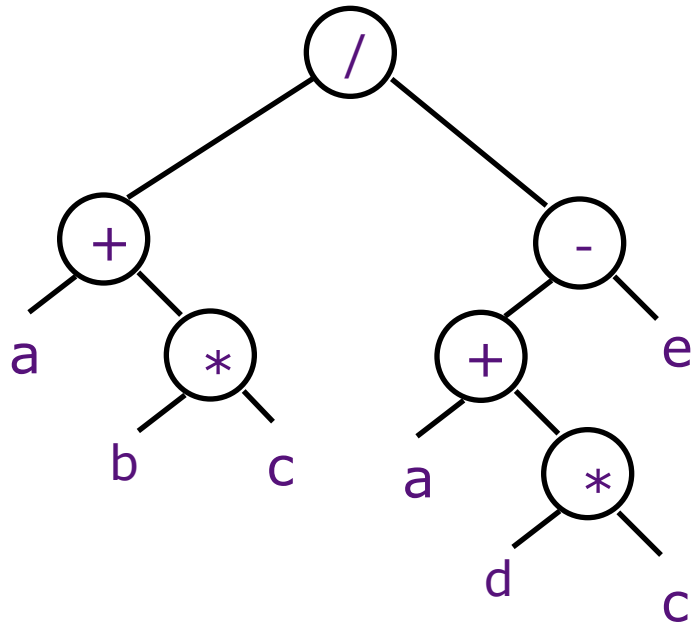
$a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /$   
↑  
**add**



Evaluation Stack

# Evaluation of Expressions

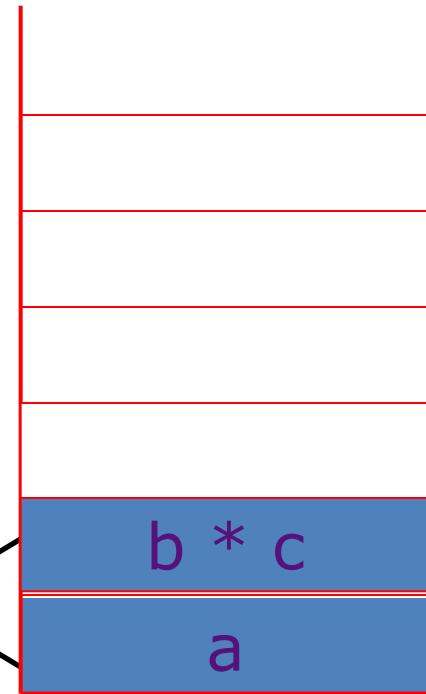
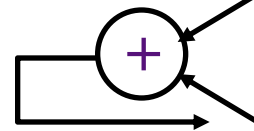
$(a + b * c) / (a + d * c - e)$



Reverse Polish

$a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /$

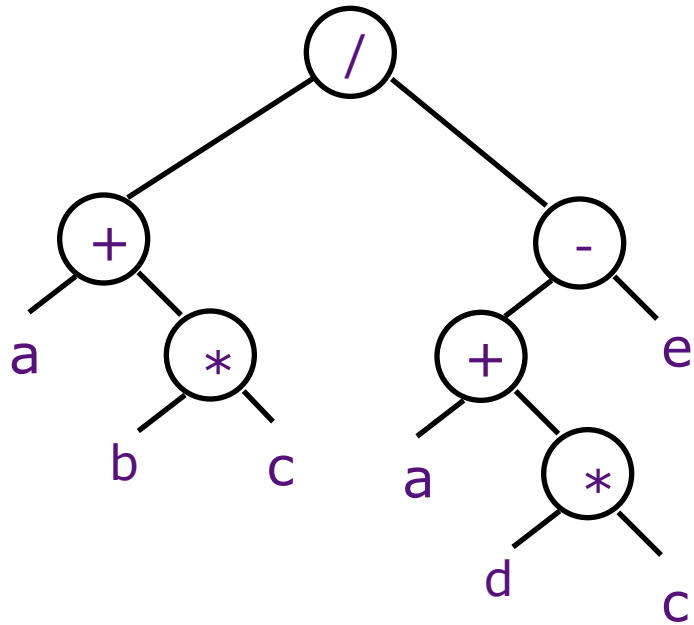
↑  
add



Evaluation Stack

# Evaluation of Expressions

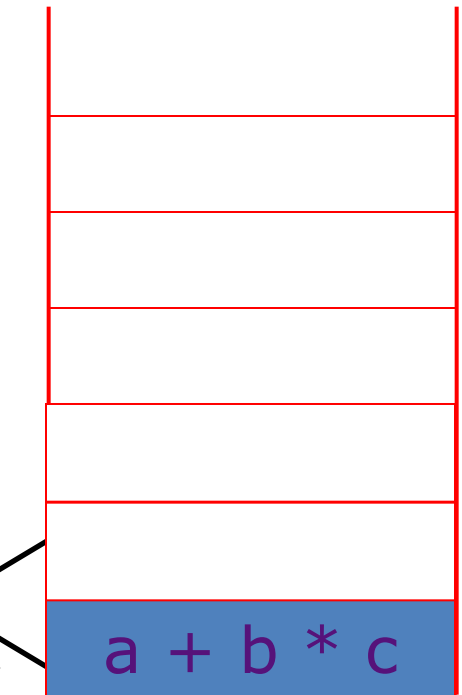
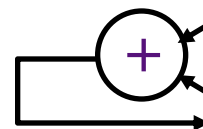
$(a + b * c) / (a + d * c - e)$



Reverse Polish

$a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /$

↑  
**add**



Evaluation Stack

# Hardware organization of the stack

- Stack is part of the processor state
  - ⇒ *stack must be bounded and small*  
≈ number of Registers,  
*not* the size of main memory
- Conceptually stack is unbounded
  - ⇒ *a part of the stack is included in the processor state; the rest is kept in the main memory*

# Stack Operations and Implicit Memory References

- Suppose the top 2 elements of the stack are kept in registers and the rest is kept in the memory.

Each <i>push</i> operation	⇒	1 memory reference
<i>pop</i> operation	⇒	1 memory reference

# Stack Operations and Implicit Memory References

- Suppose the top 2 elements of the stack are kept in registers and the rest is kept in the memory.

Each <i>push</i> operation	⇒	1 memory reference
<i>pop</i> operation	⇒	1 memory reference

*No Good!*



# Stack Operations and Implicit Memory References

- Suppose the top 2 elements of the stack are kept in registers and the rest is kept in the memory.

Each *push* operation       $\Rightarrow$       1 memory reference  
      *pop* operation       $\Rightarrow$       1 memory reference

*No Good!*

- Better performance by keeping the top N elements in registers, and memory references are made only when register stack overflows or underflows.

# Stack Size and Memory References

a b c \* + a d c \* + e - /

<i>program</i>	<i>stack (size = 2)</i>	<i>memory refs</i>
push a	R0	a
push b	R0 R1	b
push c	R0 R1 R2	c, ss(a)
*	R0 R1	sf(a)
+	R0	
push a	R0 R1	a
push d	R0 R1 R2	d, ss(a+b*c)
push c	R0 R1 R2 R3	c, ss(a)
*	R0 R1 R2	sf(a)
+	R0 R1	sf(a+b*c)
push e	R0 R1 R2	e,ss(a+b*c)
-	R0 R1	sf(a+b*c)
/	R0	

# Stack Size and Memory References

a b c \* + a d c \* + e - /

<i>program</i>	<i>stack (size = 2)</i>	<i>memory refs</i>
push a	R0	a
push b	R0 R1	b
push c	R0 R1 R2	c, ss(a)
*	R0 R1	sf(a)
+	R0	
push a	R0 R1	a
push d	R0 R1 R2	d, ss(a+b*c)
push c	R0 R1 R2 R3	c, ss(a)
*	R0 R1 R2	sf(a)
+	R0 R1	sf(a+b*c)
push e	R0 R1 R2	e,ss(a+b*c)
-	R0 R1	sf(a+b*c)
/	R0	

4 stores, 4 fetches (implicit) 67

# Stack Size and Expression Evaluation

a b c \* + a d c \* + e - /

<i>program</i>	<i>stack (size = 4)</i>
push a	R0
push b	R0 R1
push c	R0 R1 R2
*	R0 R1
+	R0
push a	R0 R1
push d	R0 R1 R2
push c	R0 R1 R2 R3
*	R0 R1 R2
+	R0 R1
push e	R0 R1 R2
-	R0 R1
/	R0

# Stack Size and Expression Evaluation

a b c \* + a d c \* + e - /

*a and c are  
"loaded" twice*

⇒

*not the best  
use of registers!*

*program*

push a

push b

push c

\*

+

push a

push d

push c

\*

+

push e

-

/

*stack (size = 4)*

R0

R0 R1

R0 R1 R2

R0 R1

R0

R0 R1

R0 R1 R2

R0 R1 R2 R3

R0 R1 R2

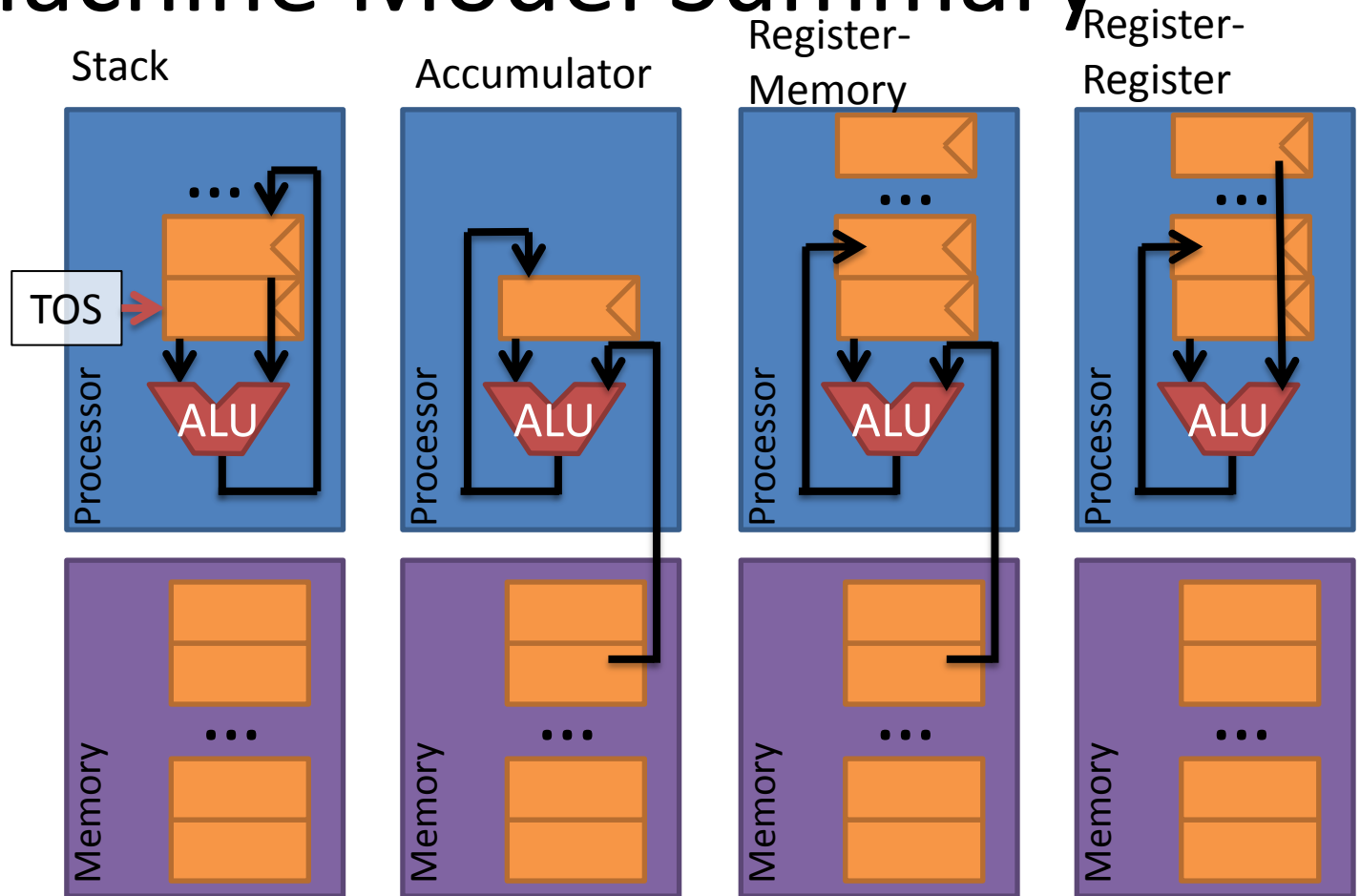
R0 R1

R0 R1 R2

R0 R1

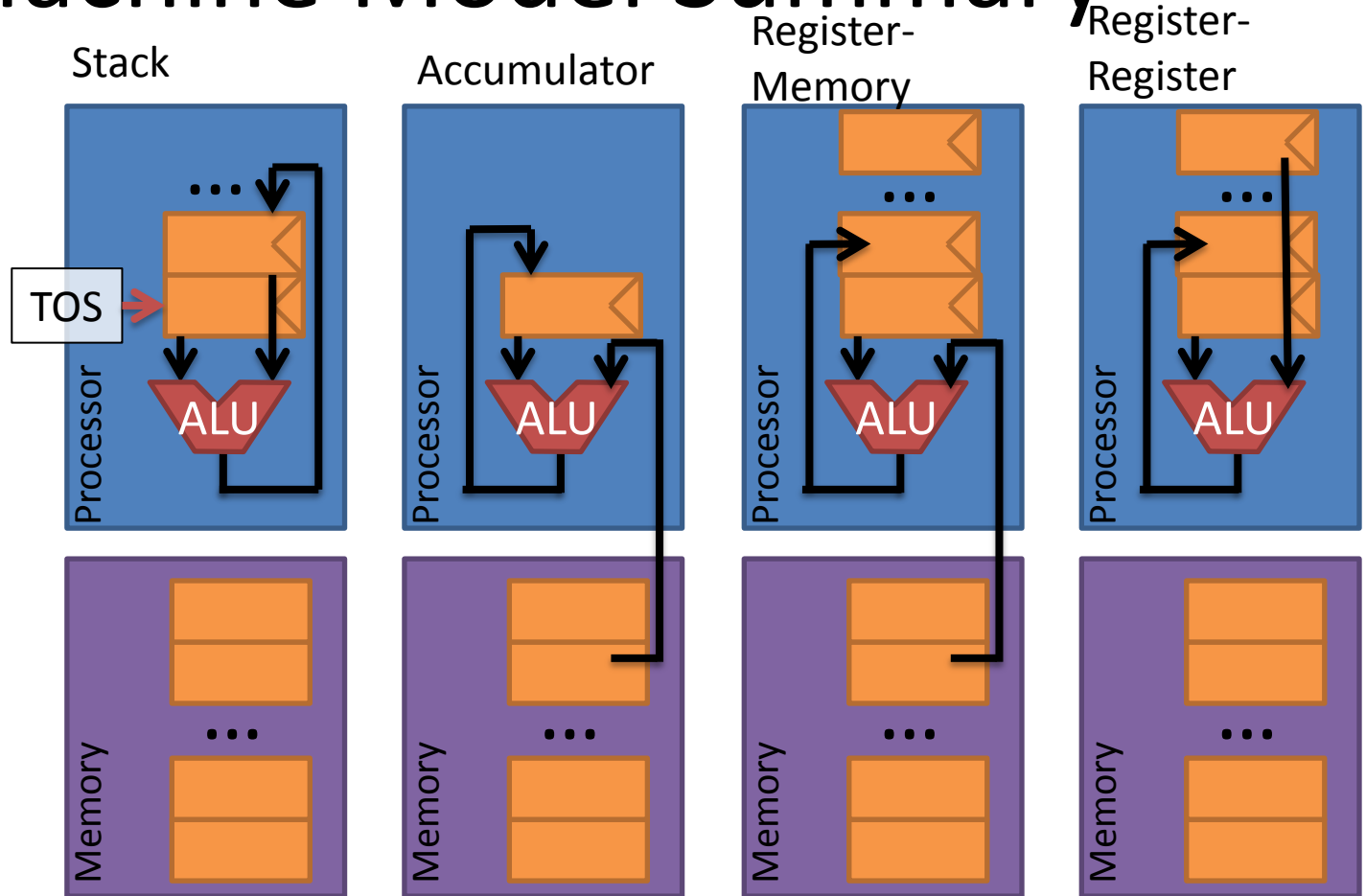
R0

# Machine Model Summary



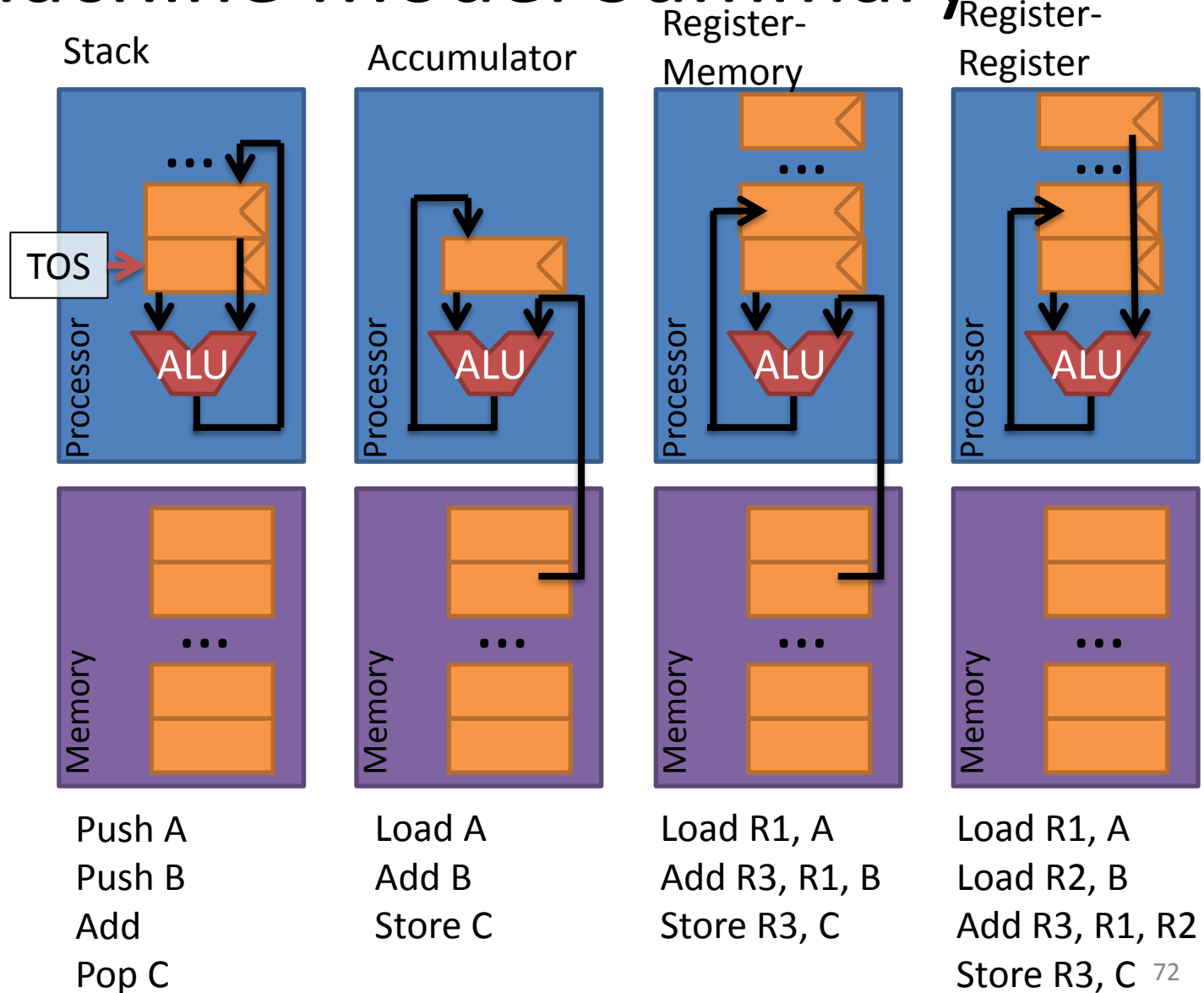


# Machine Model Summary



$$C = A + B$$

# Machine Model Summary



$$C = A + B$$

# Classes of Instructions

- Data Transfer
  - LD, ST, MFC1, MTC1, MFC0, MTC0
- ALU
  - ADD, SUB, AND, OR, XOR, MUL, DIV, SLT, LUI
- Control Flow
  - BEQZ, JR, JAL, TRAP, ERET
- Floating Point
  - ADD.D, SUB.S, MUL.D, C.LT.D, CVT.S.W,
- Multimedia (SIMD)
  - ADD.PS, SUB.PS, MUL.PS, C.LT.PS
- String
  - REP MOVSB (x86)

# Addressing Modes:

## How to Get Operands from Memory

Addressing Mode	Instruction	Function
Register	Add R4, R3, R2	Regs[R4] <- Regs[R3] + Regs[R2] **
Immediate	Add R4, R3, #5	Regs[R4] <- Regs[R3] + 5 **
Displacement	Add R4, R3, 100(R1)	Regs[R4] <- Regs[R3] + Mem[100 + Regs[R1]]
Register Indirect	Add R4, R3, (R1)	Regs[R4] <- Regs[R3] + Mem[Regs[R1]]
Absolute	Add R4, R3, (0x475)	Regs[R4] <- Regs[R3] + Mem[0x475]
Memory Indirect	Add R4, R3, @(R1)	Regs[R4] <- Regs[R3] + Mem[Mem[R1]]
PC relative	Add R4, R3, 100(PC)	Regs[R4] <- Regs[R3] + Mem[100 + PC]
Scaled	Add R4, R3, 100(R1)[R5]	Regs[R4] <- Regs[R3] + Mem[100 + Regs[R1] + Regs[R5] * 4]

# Data Types and Sizes

- Types
  - Binary Integer
  - Binary Coded Decimal (BCD)
  - Floating Point
    - IEEE 754
    - Cray Floating Point
    - Intel Extended Precision (80-bit)
  - Packed Vector Data
  - Addresses
- Width
  - Binary Integer (8-bit, 16-bit, 32-bit, 64-bit)
  - Floating Point (32-bit, 40-bit, 64-bit, 80-bit)
  - Addresses (16-bit, 24-bit, 32-bit, 48-bit, 64-bit)

# ISA Encoding

**Fixed Width:** Every Instruction has same width

- Easy to decode

(RISC Architectures: MIPS, PowerPC, SPARC, ARM...)

Ex: MIPS, every instruction 4-bytes

**Variable Length:** Instructions can vary in width

- Takes less space in memory and caches

(CISC Architectures: IBM 360, x86, Motorola 68k, VAX...)

Ex: x86, instructions 1-byte up to 17-bytes



# ISA Encoding

**Fixed Width:** Every Instruction has same width

- Easy to decode

(RISC Architectures: MIPS, PowerPC, SPARC, ARM...)

Ex: MIPS, every instruction 4-bytes

**Variable Length:** Instructions can vary in width

- Takes less space in memory and caches

(CISC Architectures: IBM 360, x86, Motorola 68k, VAX...)

Ex: x86, instructions 1-byte up to 17-bytes

**Mostly Fixed or Compressed:**

- Ex: MIPS16, THUMB (only two formats 2 and 4 bytes)
- PowerPC and some VLIWs (Store instructions compressed, decompress into Instruction Cache)

**(Very) Long Instruction Word:**

- Multiple instructions in a fixed width bundle
- Ex: Multiflow, HP/ST Lx, TI C6000

# x86 (IA-32) Instruction Encoding

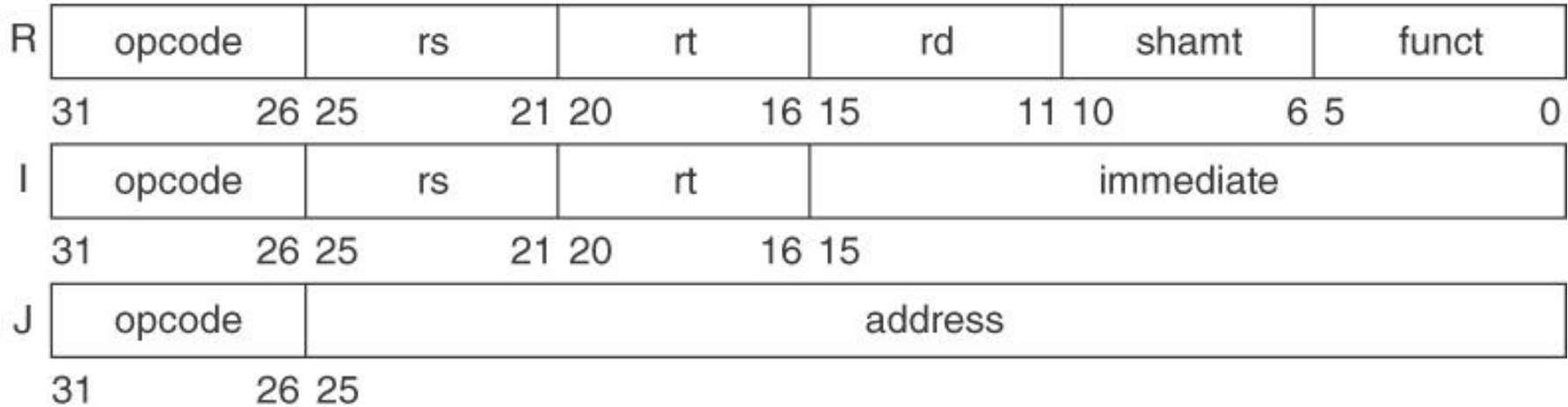
Instruction Prefixes	Opcode	ModR/M	Scale, Index, Base	Displacement	Immediate
Up to four Prefixes (1 byte each)	1,2, or 3 bytes	1 byte (if needed)	1 byte (if needed)	0,1,2, or 4 bytes	0,1,2, or 4 bytes

x86 and x86-64 Instruction Formats

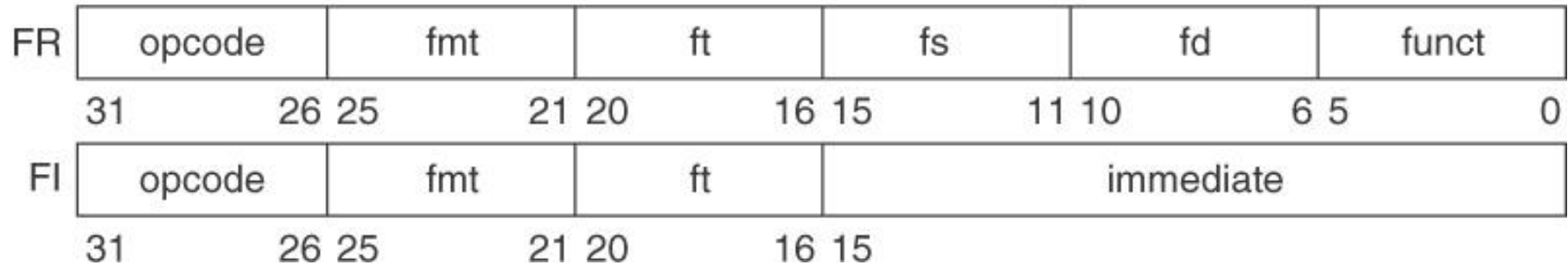
Possible instructions 1 to 18 bytes long

# MIPS64 Instruction Encoding

## Basic instruction formats



## Floating-point instruction formats



# Real World Instruction Sets

Arch	Type	# Oper	# Mem	Data Size	# Regs	Addr Size	Use
Alpha	Reg-Reg	3	0	64-bit	32	64-bit	Workstation
ARM	Reg-Reg	3	0	32/64-bit	16	32/64-bit	Cell Phones, Embedded
MIPS	Reg-Reg	3	0	32/64-bit	32	32/64-bit	Workstation, Embedded
SPARC	Reg-Reg	3	0	32/64-bit	24-32	32/64-bit	Workstation
TI C6000	Reg-Reg	3	0	32-bit	32	32-bit	DSP
IBM 360	Reg-Mem	2	1	32-bit	16	24/31/64	Mainframe
x86	Reg-Mem	2	1	8/16/32/ 64-bit	4/8/24	16/32/64	Personal Computers
VAX	Mem-Mem	3	3	32-bit	16	32-bit	Minicomputer
Mot. 6800	Accum.	1	1/2	8-bit	0	16-bit	Microcontroler

# Why the Diversity in ISAs?

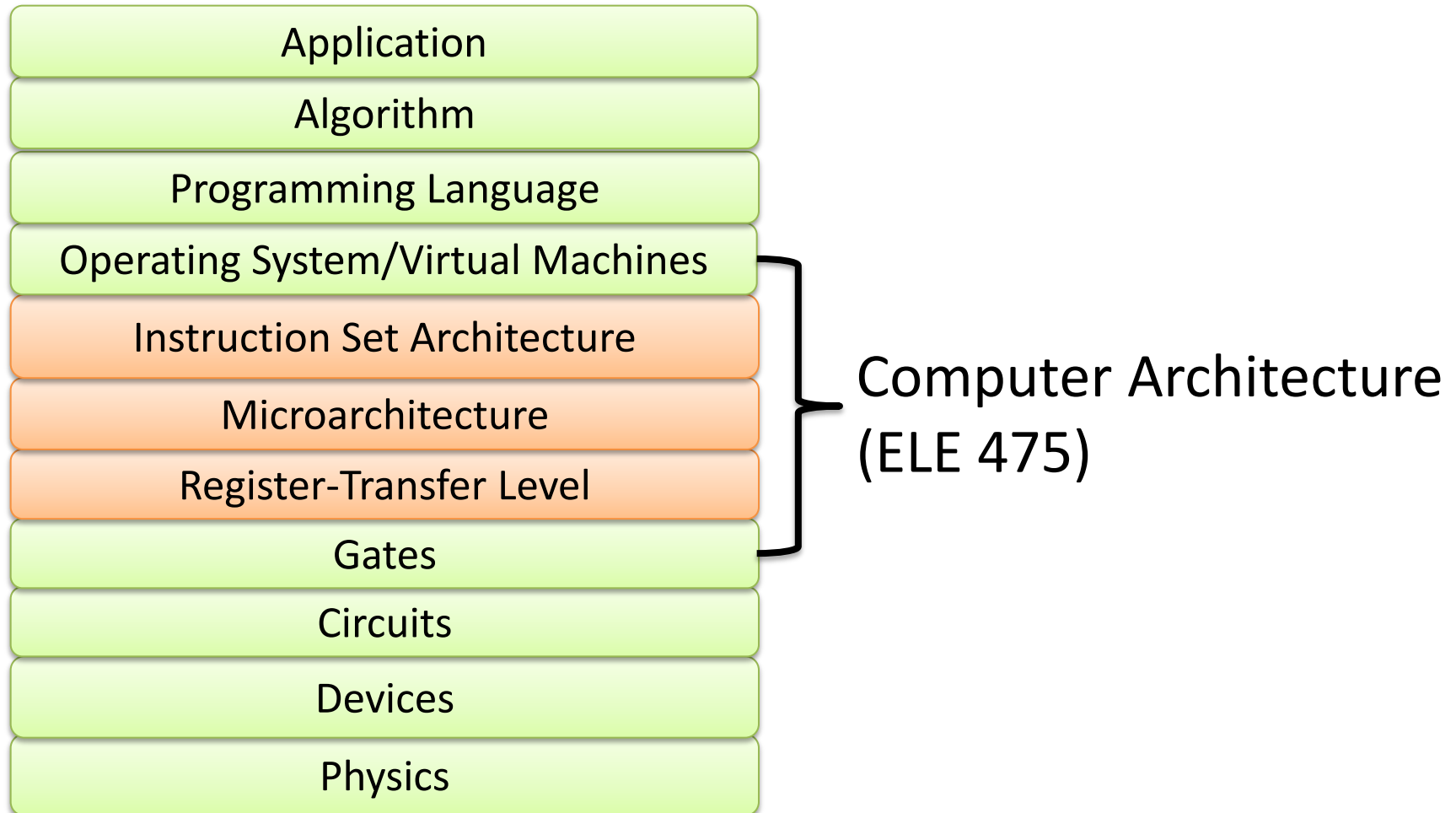
## Technology Influenced ISA

- Storage is expensive, tight encoding important
- Reduced Instruction Set Computer
  - Remove instructions until whole computer fits on die
- Multicore/Manycore
  - Transistors not turning into sequential performance

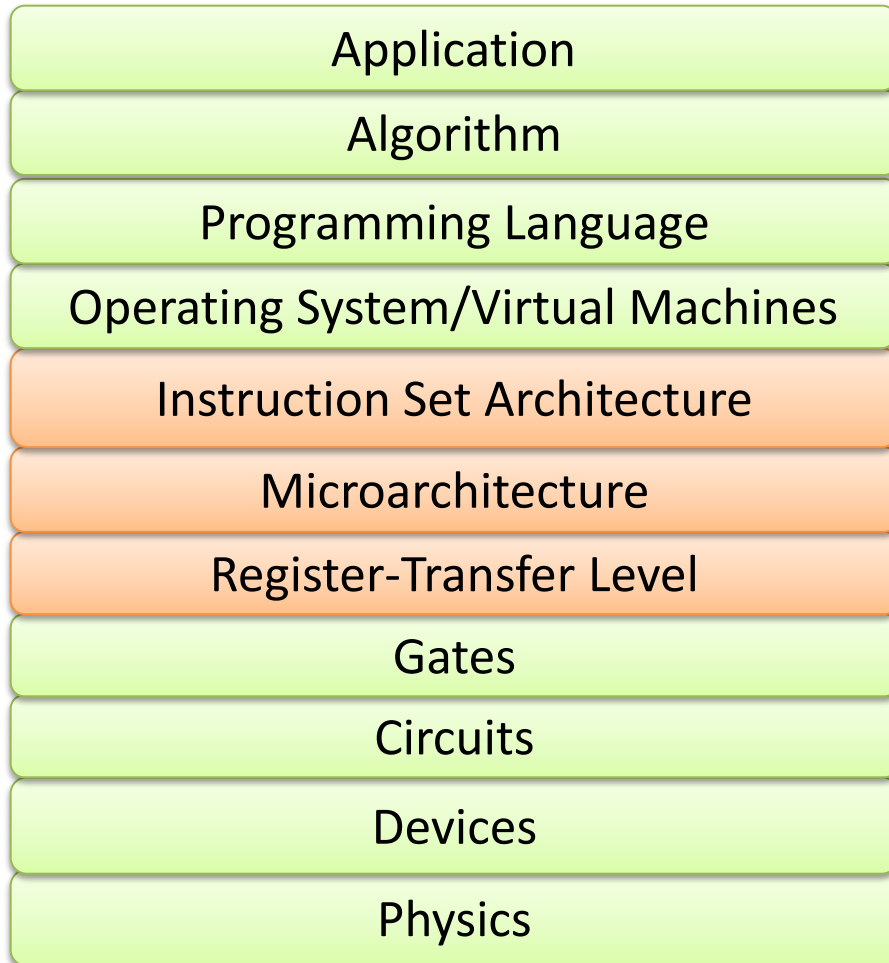
## Application Influenced ISA

- Instructions for Applications
  - DSP instructions
- Compiler Technology has improved
  - SPARC Register Windows no longer needed
  - Compiler can register allocate effectively

# Recap



# Recap



- ISA vs Microarchitecture
- ISA Characteristics
  - Machine Models
  - Encoding
  - Data Types
  - Instructions
  - Addressing Modes

# Computer Architecture Lecture 1

Next Class: Microcode and Review of Pipelining



# Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Christopher Batten (Cornell)
- MIT material derived from course 6.823
- UCB material derived from course CS252 & CS152
- Cornell material derived from course ECE 4750

Copyright © 2013 David Wentzlaff