CS49K Programming Languages

Chapter 5: Target Machine Architecture

LECTURE 7:MACHINE

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Machine – Language – Grammar
- Types of Processing Units
- Instruction Level Parallelism
- Assembly Level View
- Memory Hierarchy (Fetch Cycles – Delay)
- Processor Arithmetic
- Instruction Set Architecture
- Compiling to modern processors

Overview of Machines

SECTION 1

Grammar

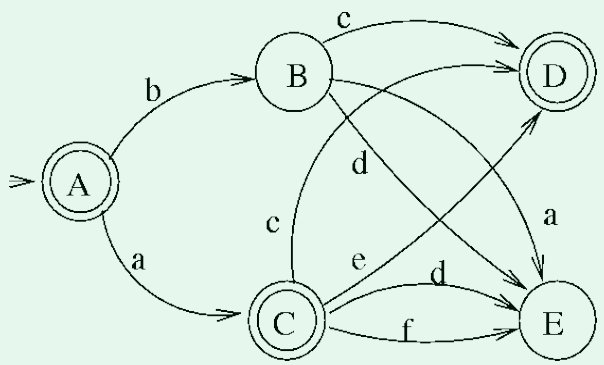
$$
\begin{aligned}
S &\rightarrow aS \mid bX \\
X &\rightarrow aX \mid bY \\
Y &\rightarrow aY \mid bZ \mid \Lambda \\
Z &\rightarrow aZ \mid \Lambda
\end{aligned}
$$

Formal

Language Input

```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```

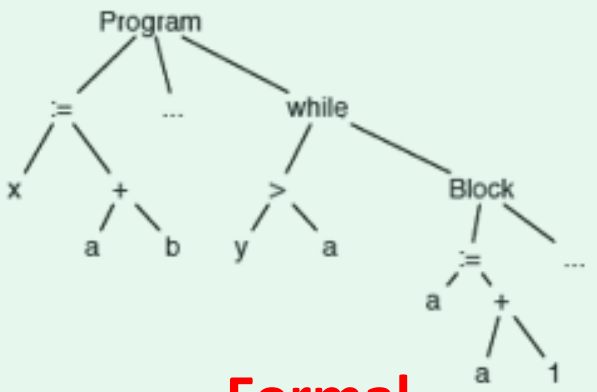
Informal



Formal

State Machine

Validated Tokens Syntax Tree



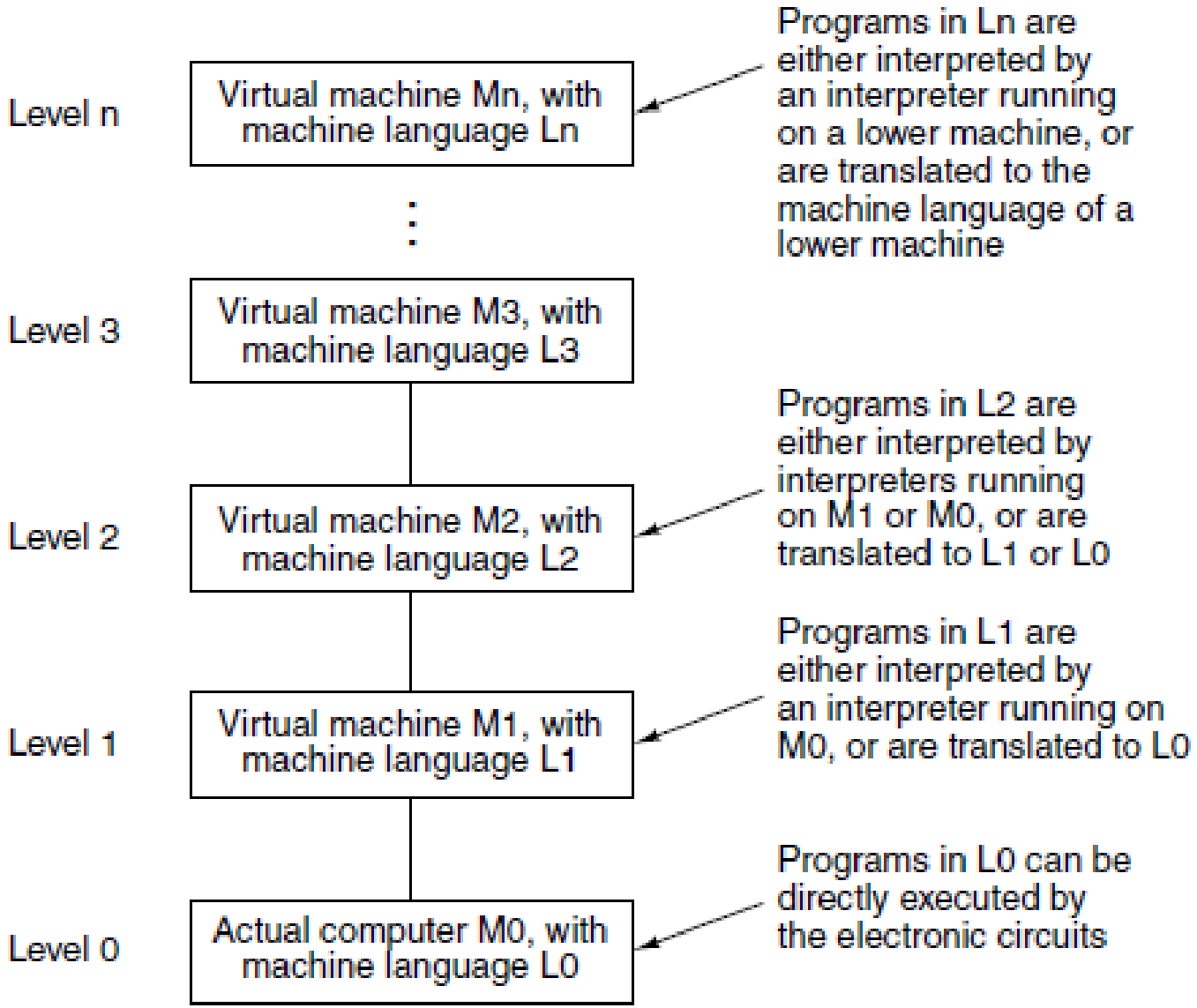
Formal

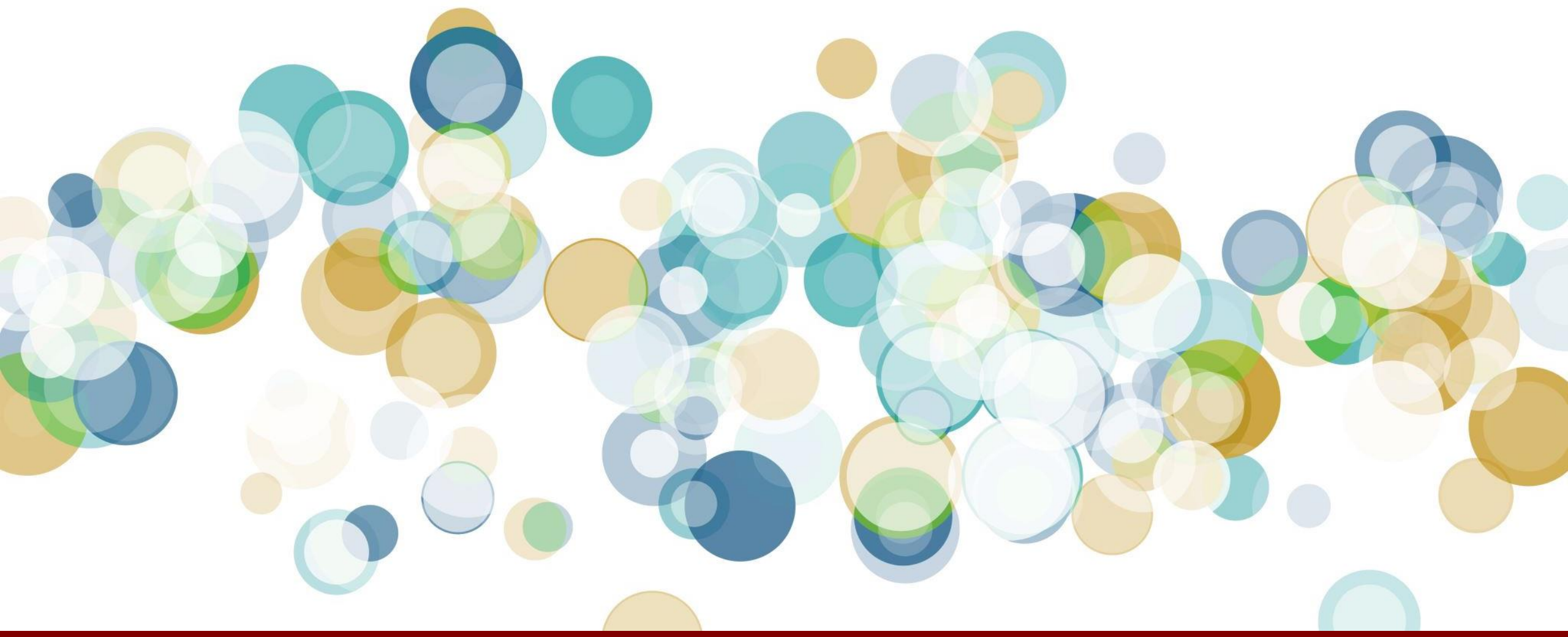
Target Code

Machine Code
Assembly
Byte Code
Object Code

Informal

Interpretation Levels

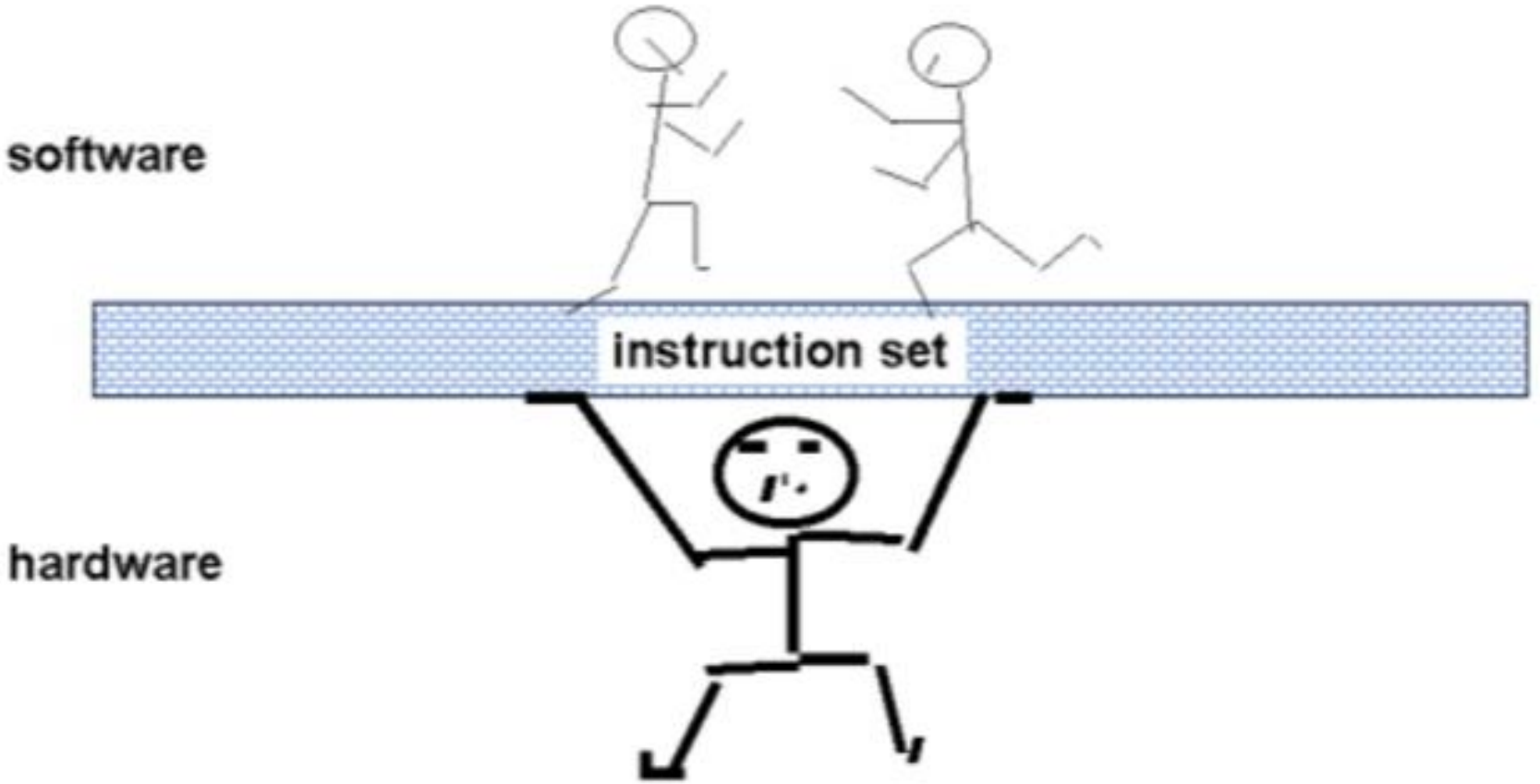


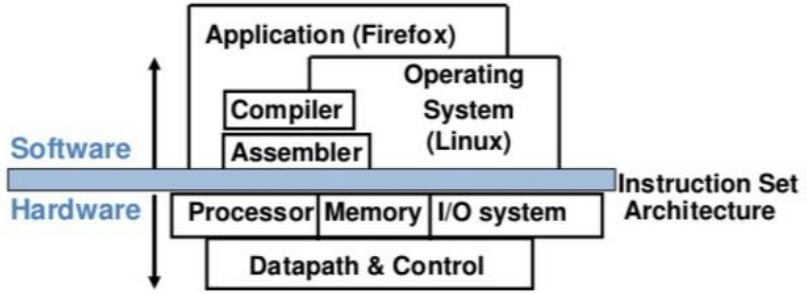


Real Machine

MACHINE 1

Instruction Set Architecture (ISA)





Applications

JVM & JDK

C-Compiler

X-ASM

Simulator

C-Compiler

Assembler

Bootstrap-BIOS

Operation System

Existing Machine (x86)

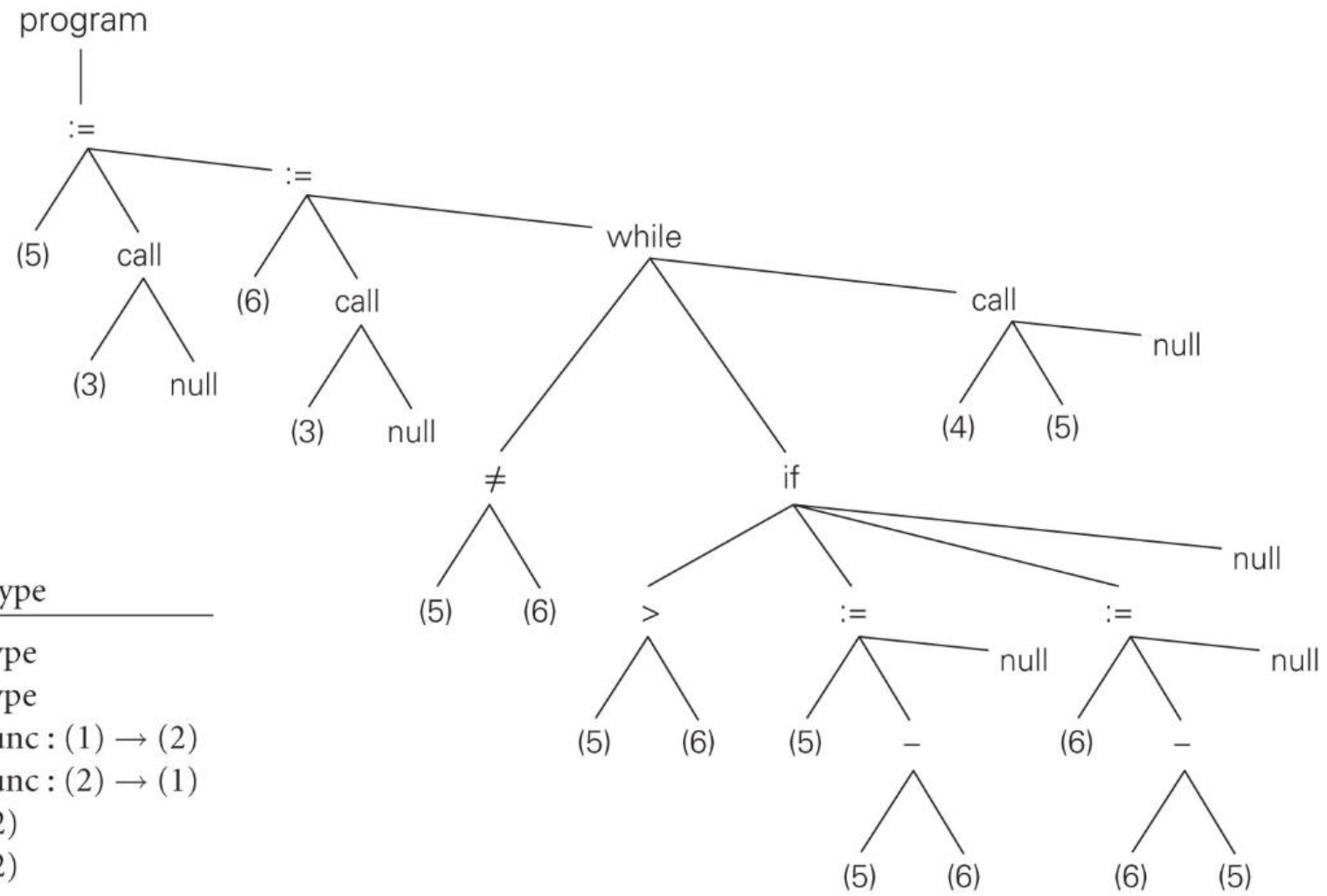
Instruction Set Architecture X

Processor

Memory

I/O

Datapath & Control



Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

Figure 15.2 Syntax tree and symbol table for the GCD program. The only difference from Figure 1.5 is the addition of explicit null nodes to indicate empty argument lists and to terminate statement lists.

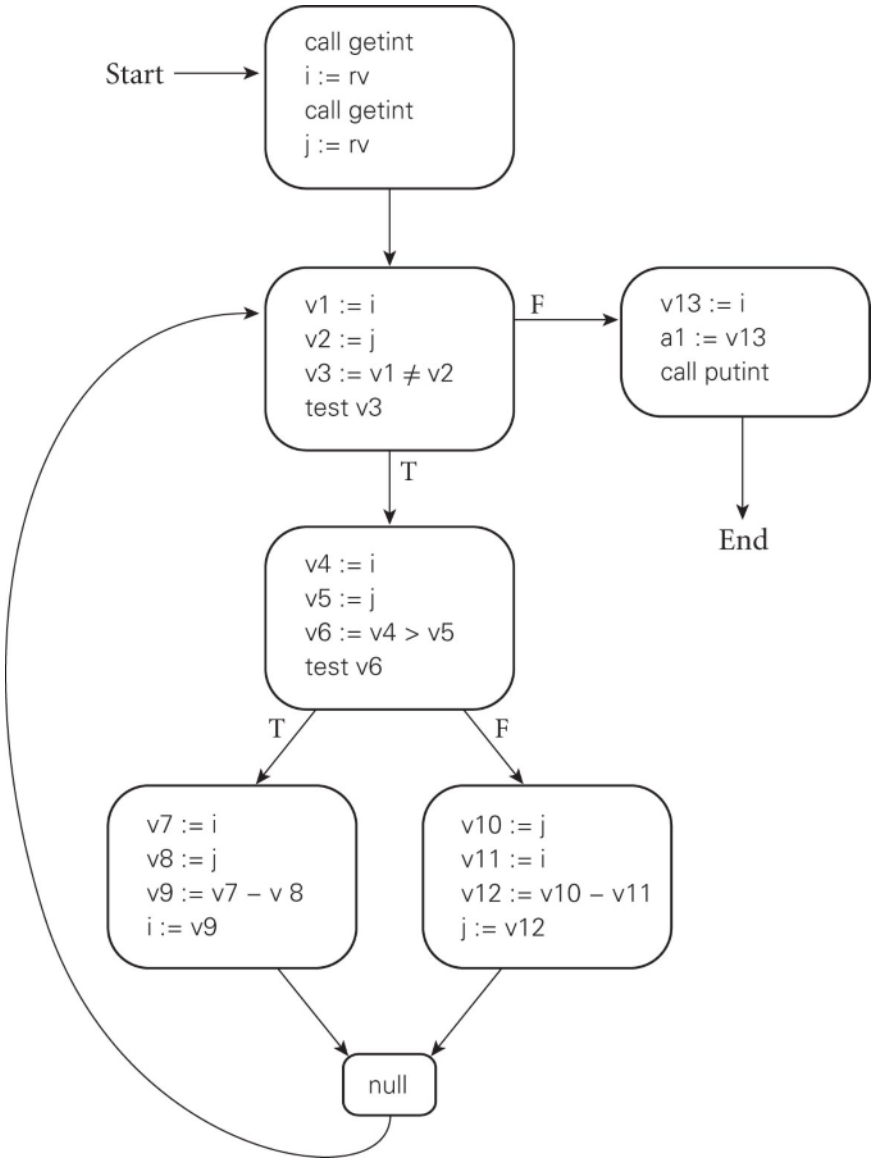
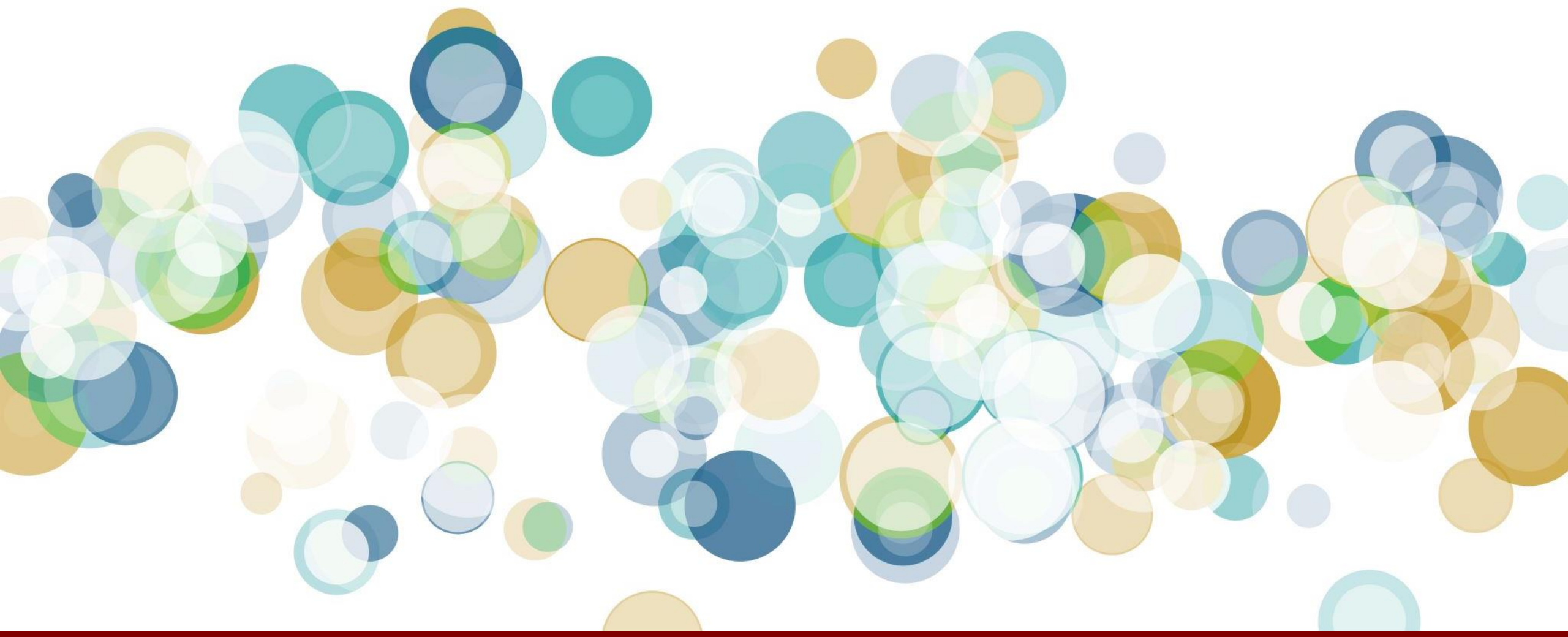


Figure 15.3 Control flow graph for the GCD program. Code within basic blocks is shown in the pseudo-assembly notation introduced on page 214, with a different virtual register (here named v1...v13) for every computed value. Registers a1 and rv are used to pass values to and from subroutines.

Chapter 5 and Chapter 15

Backend Compiler Issues are in Chapter 15. From GIMPLE, SSA, Optimization, Un-SSA, RTL, ASM, and Machine code.

Chapter 5 focused on ISA, Memory Coherence Model, Loop-level optimization (Some part of Chapter 6).

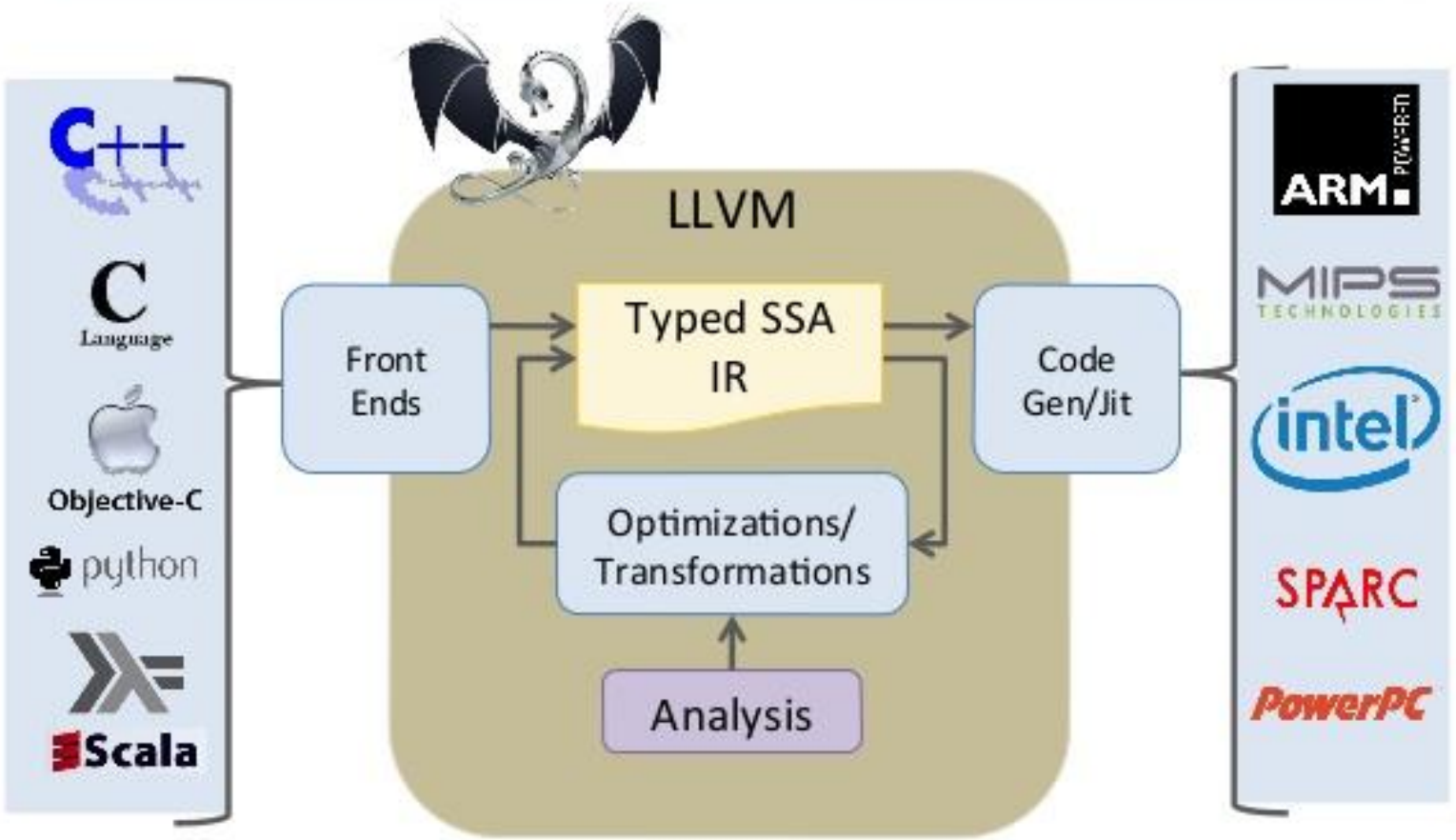


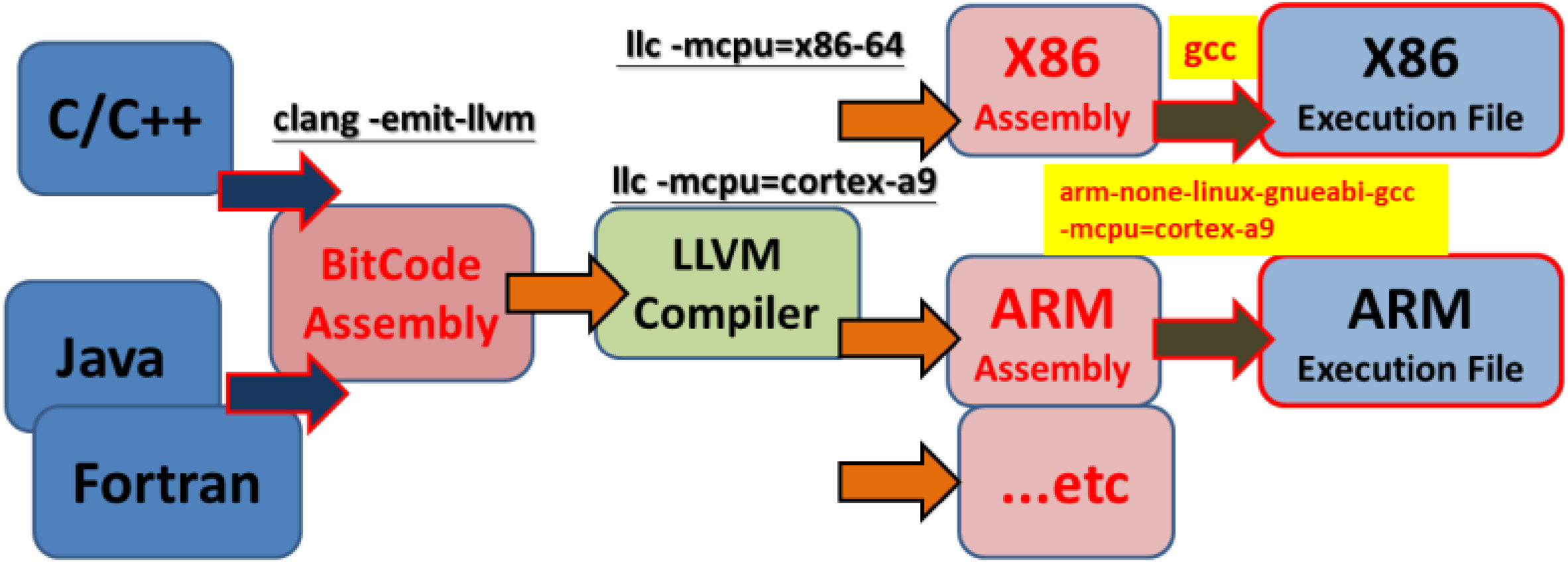
Low-Level Virtual Machine LLVM

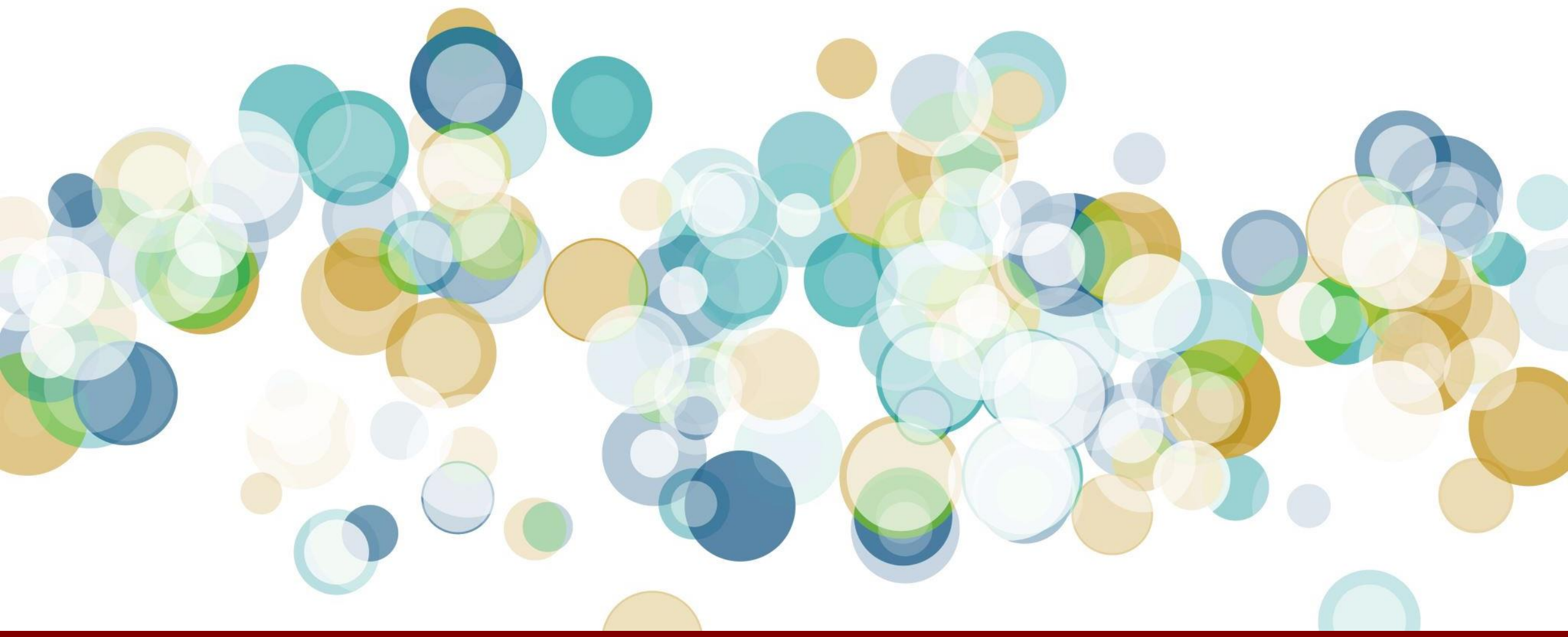
MACHINE 2

LLVM Compiler Infrastructure

[Lattner et al.]

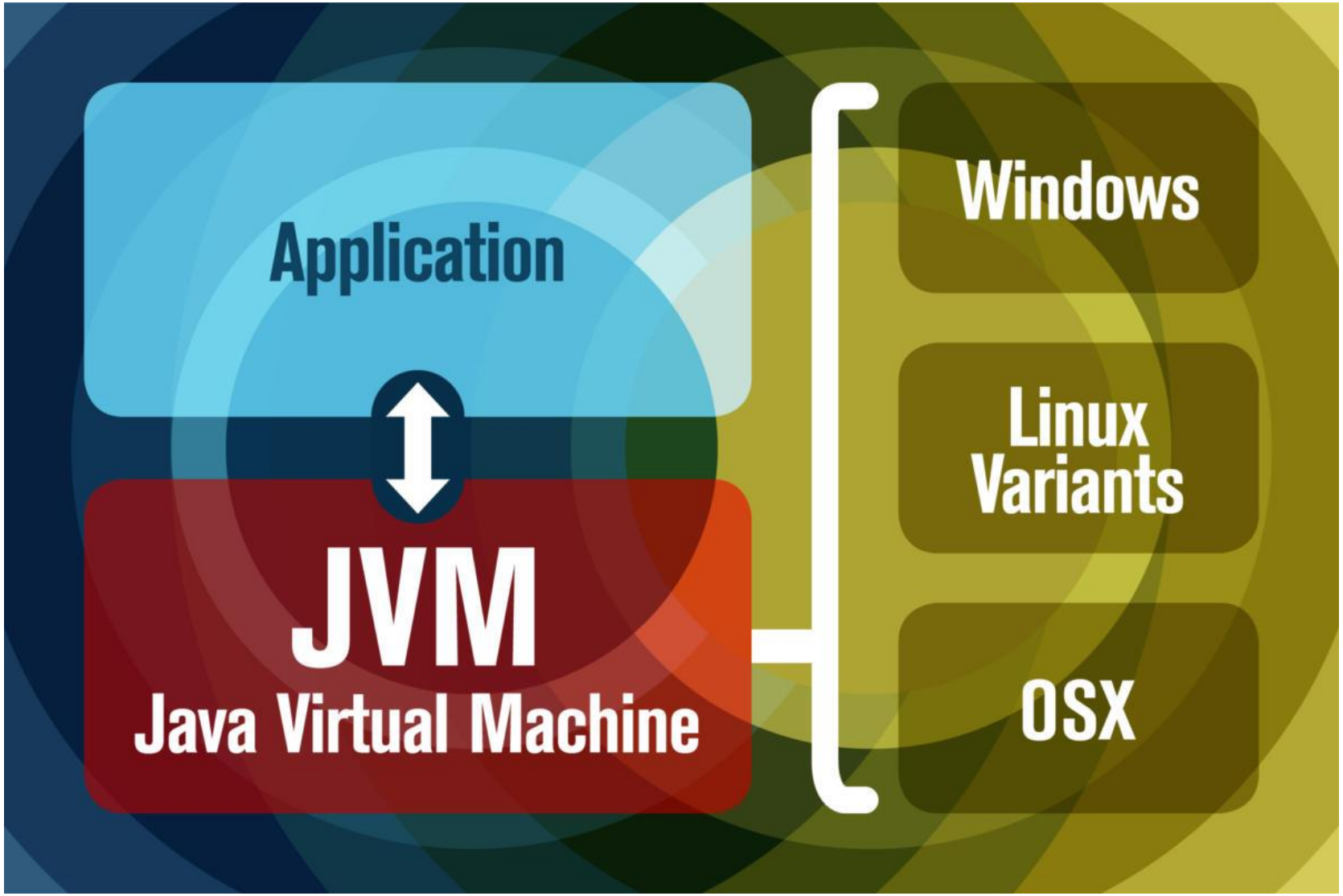


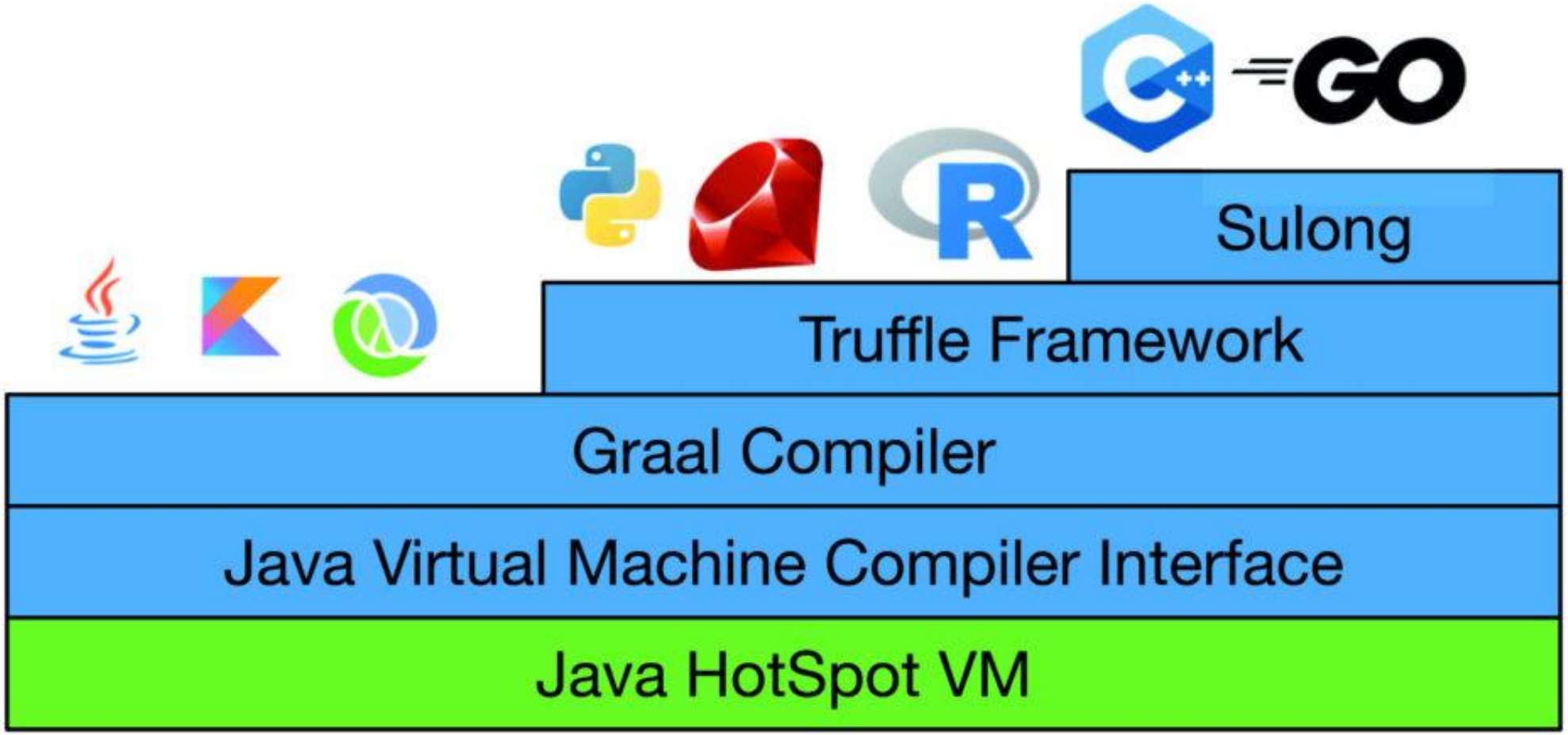

C/C++
Java
Fortran
clang -emit-llvm
BitCode Assembly
llc -mcpu=x86-64
llc -mcpu=cortex-a9
LLVM Compiler
X86 Assembly
ARM Assembly
...etc
gcc
X86 Execution File
arm-none-linux-gnueabi-gcc -mcpu=cortex-a9
ARM Execution File



Java Virtual Machine JVM

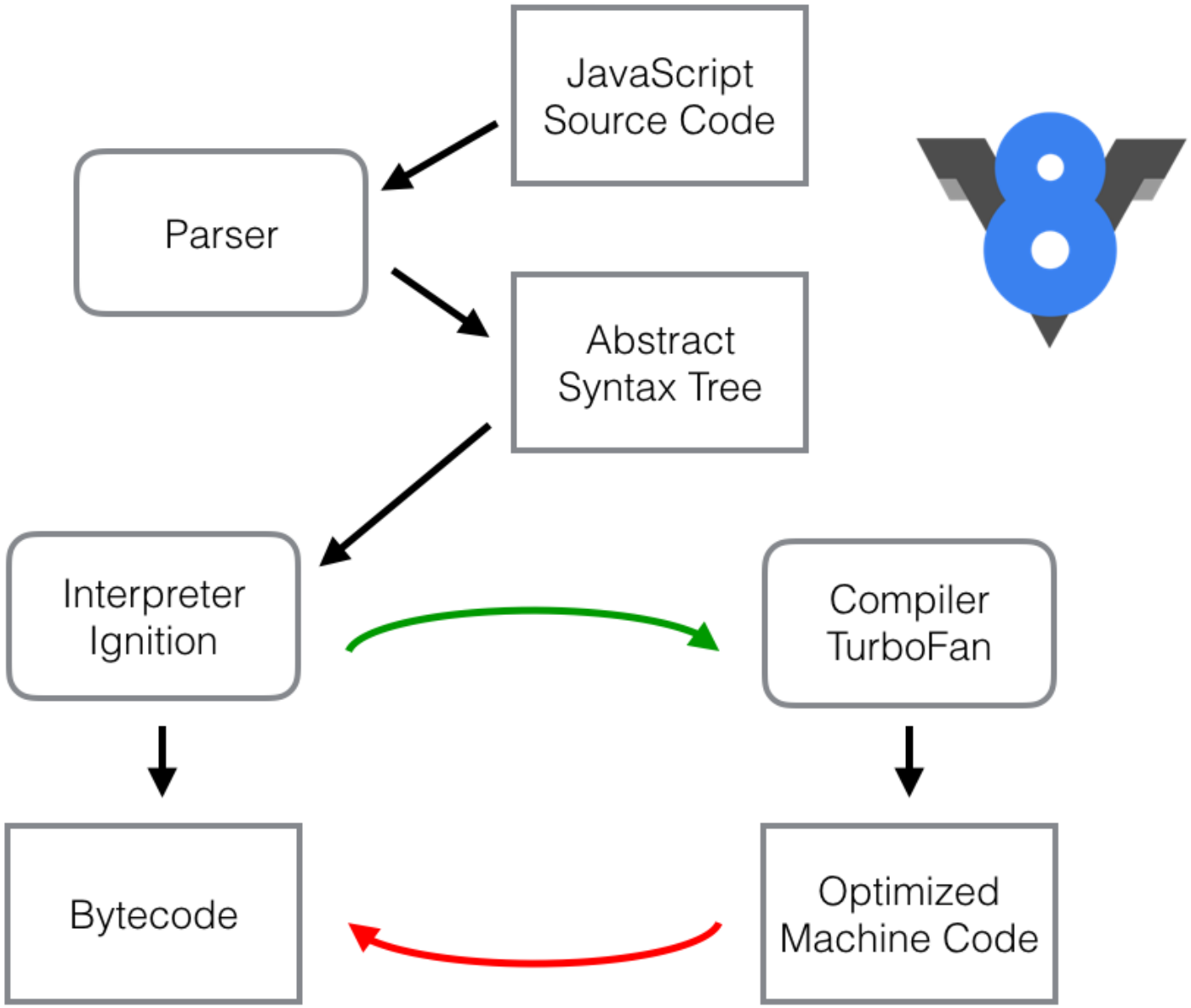
MACHINE 3


Application
JVM
Java Virtual Machine
Windows
Linux
Variants
OSX


Sulong
Truffle Framework
Graal Compiler
Java Virtual Machine Compiler Interface
Java HotSpot VM

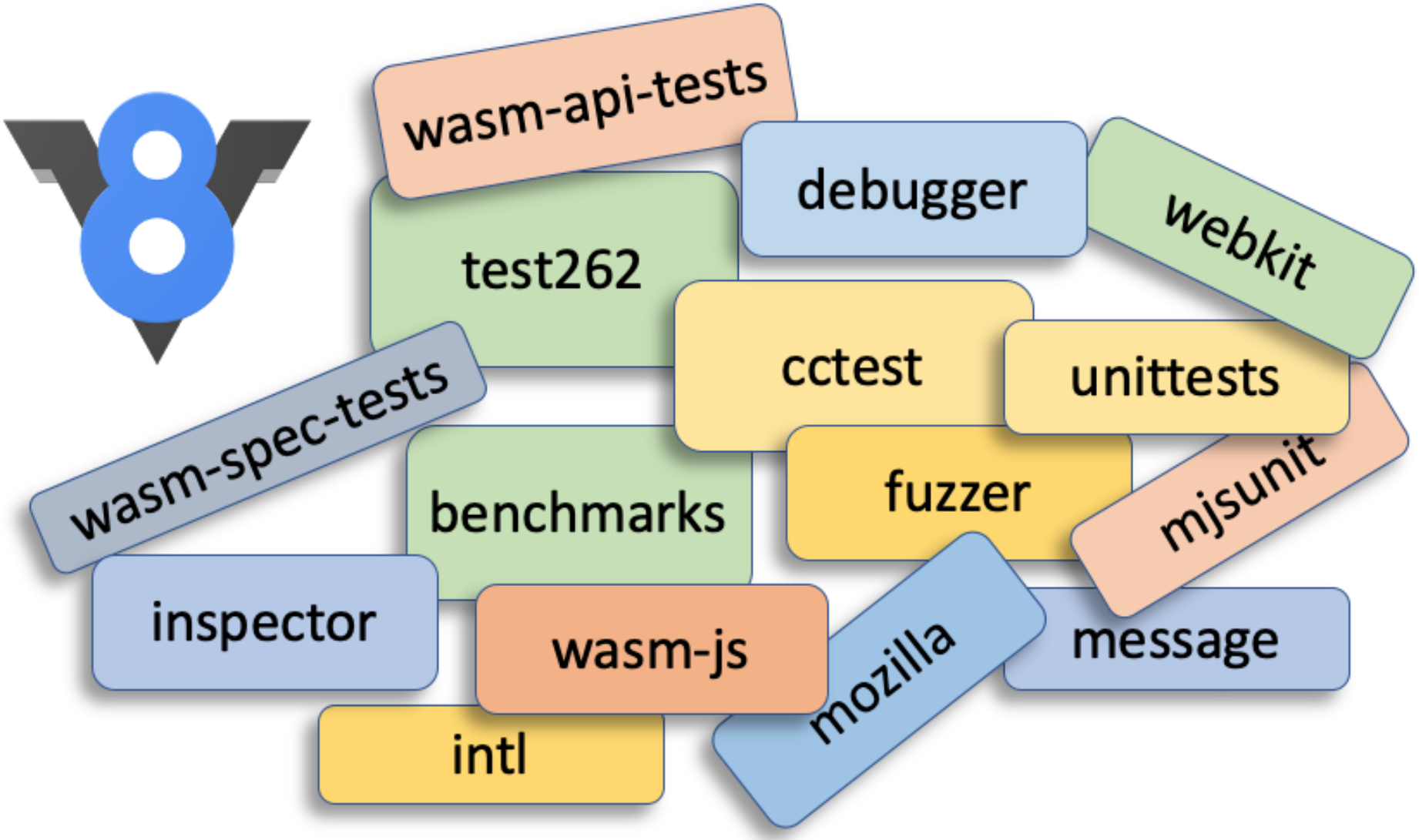
Google Chrome JavaScript V8 Engine

MACHINE 4



Google

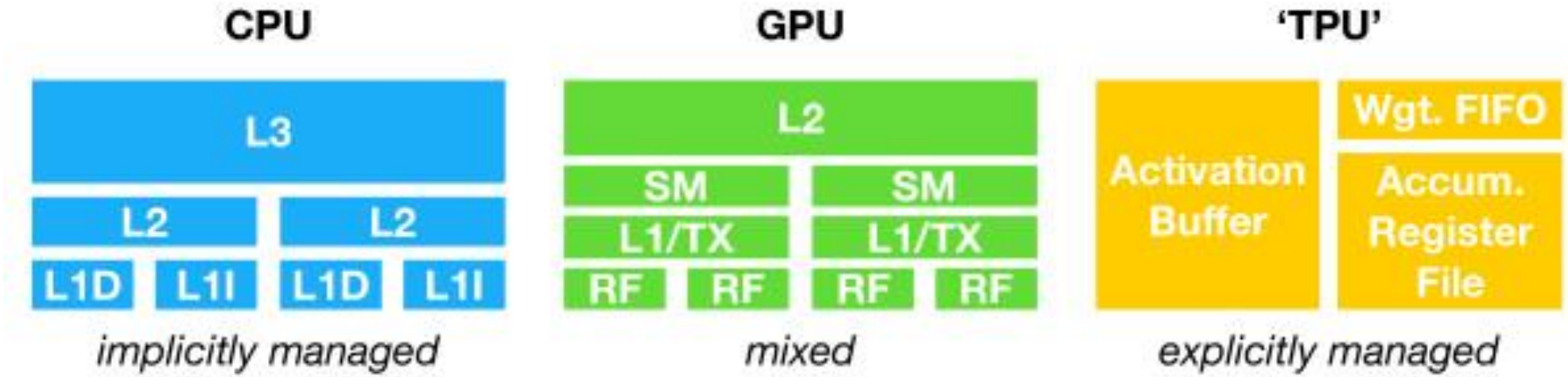
@fhinkel


wasm-api-tests
debugger
webkit
test262
cctest
unittests
wasm-spec-tests
benchmarks
fuzzer
mjsunit
inspector
wasm-js
mozilla
message
intl

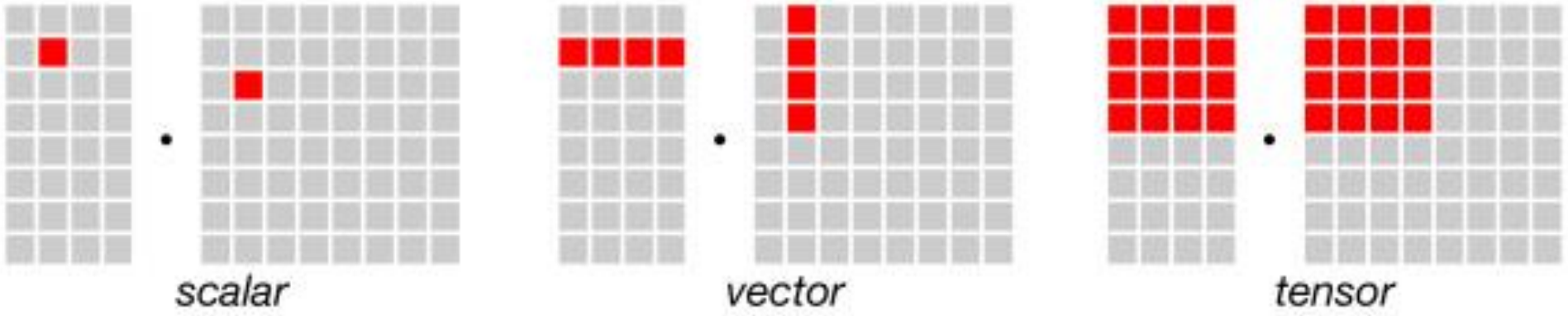
Processor Units

SECTION 2

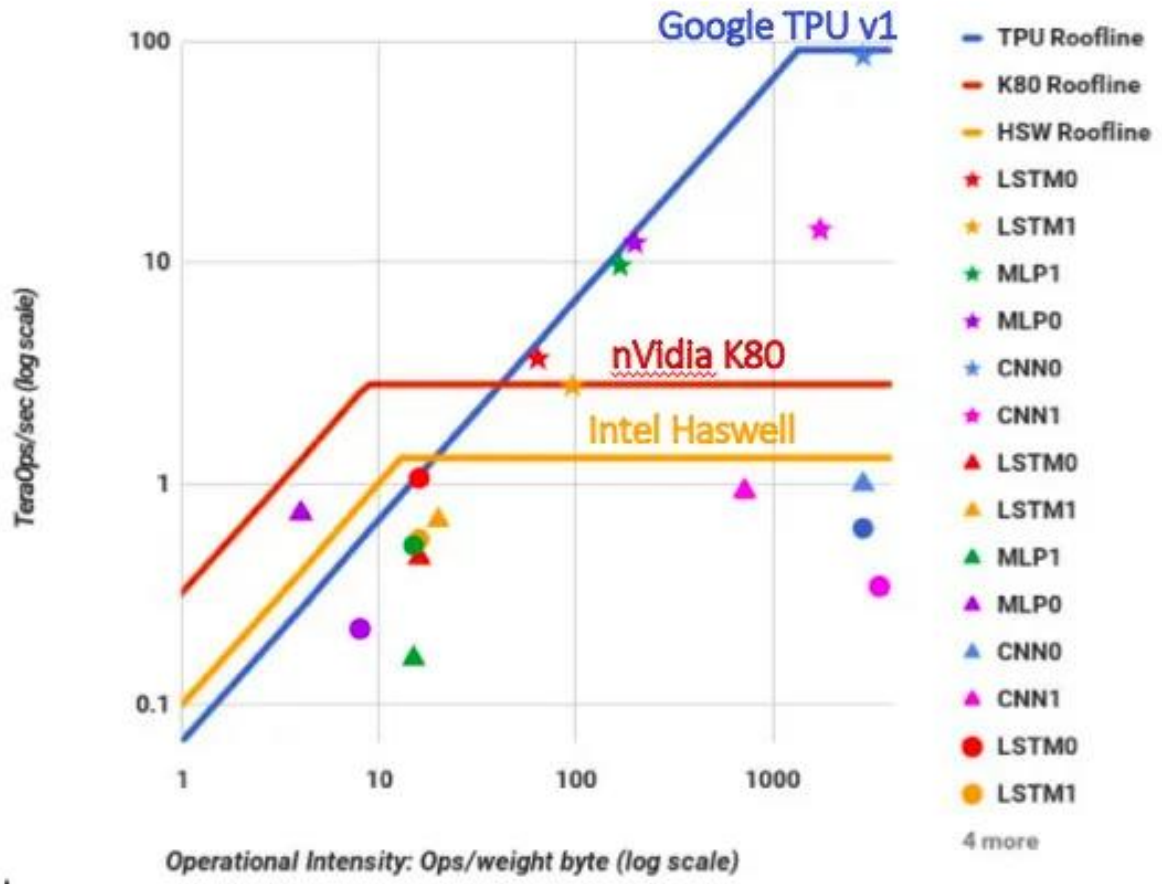
Memory Subsystem Architecture



Compute Primitive

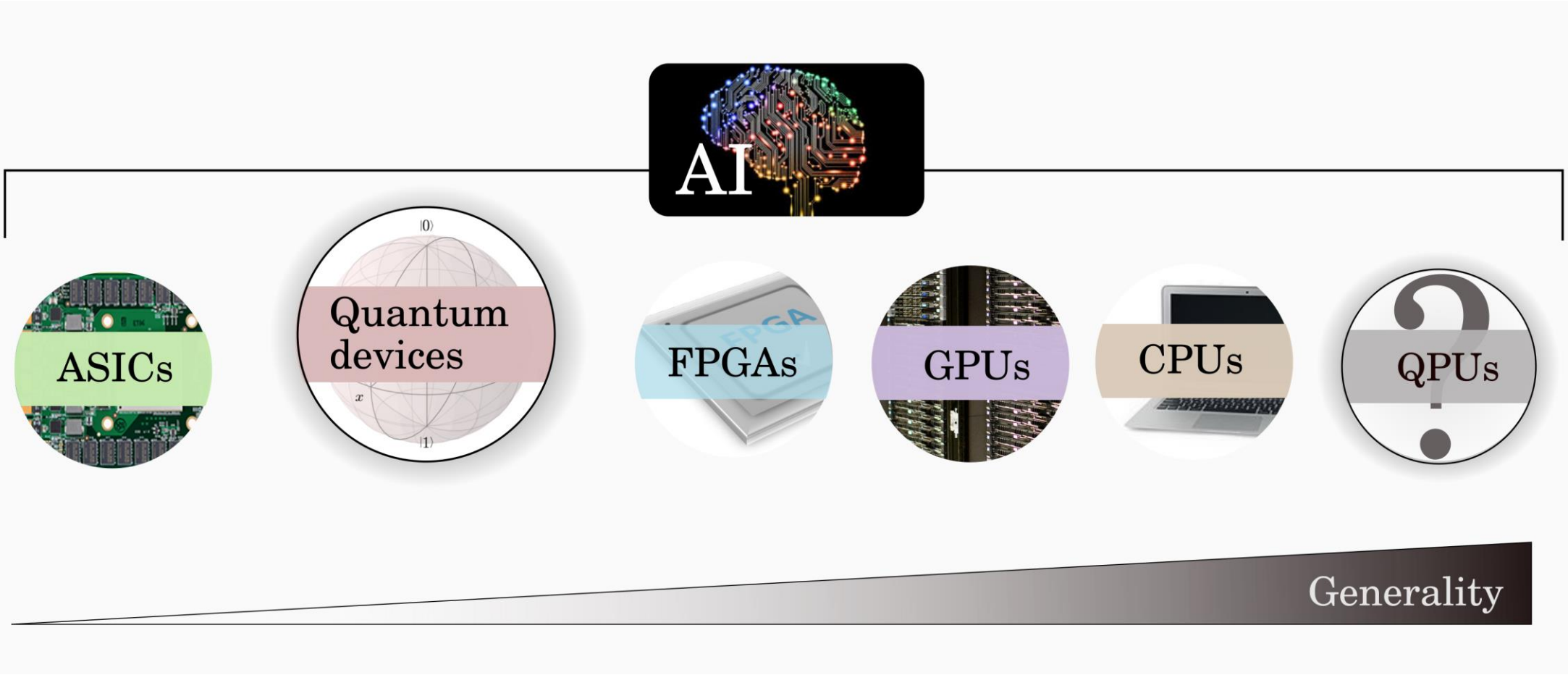


AI Application Performance



☆ = Google TPU v1 △ = nVidia K80 ◯ = Intel Haswell

N. Jouppi, et.al., "In-Datacenter Performance Analysis of a Tensor Processing Unit™,"
https://arxiv.org/ftp/arxiv/papers/1704/1704.04760.pdf


AI
ASICs
Quantum devices
|0⟩
|1⟩
FPGAs
GPUs
CPUs
QPUs
Generality

APU: Accelerated Processing Unit is the AMD's Fusion architecture that integrates both CPU and GPU on the same die.
BPU: Brain Processing Unit is the design of the AI chips by Horizon Robotics.
CPU: Central Processing Unit.
DPU:

- Deep Learning Processing Unit (DPU)
- Dataflow Processing Unit (DPU)
- DLA: Deep Learning Accelerator (DLA, or NVDIA) is an open and standardized architecture by Nvidia to address the computational demands of inference.

EPU: Emotion Processing Unit is designed by Emoshape
FPU: Floating Processing Unit (FPU).
GPU: Graphics Processing Unit (GPU)
HPU: Holographic Processing Unit (HPU) is the specific hardware of Microsoft's Hololens.
IPU:

- Integer Processing Unit.
- Intelligence Processing Unit (IPU) is specific for the graph related applications by GraphCore.
- Intelligence Processing Unit (IPU): Mythic, another start-up company also name their product as IPU.
- Image Processing Unit (IPU) is the Pixel Visual Core designed by Google and integrated in Google Pixel 2 released in 2017.

NPU: Neural Network Processing Unit (NPU) has become a general name of AI chip rather than a brand name of a company.
SPU: Stream Processing Unit (SPU) is related to the specialized hardware to process the data streams of video.
TPU: Tensor Processing Unit (TPU) is Google's specialized hardware for neural network.
VPU: Vision Processing Unit (VPU) is the specialized chip for computer vision workloads.
ZPU: is a small, portable CPU core.

Instruction Level Parallelism

SECTION 3

Computer Architectural Issues that Impacts Programming Language

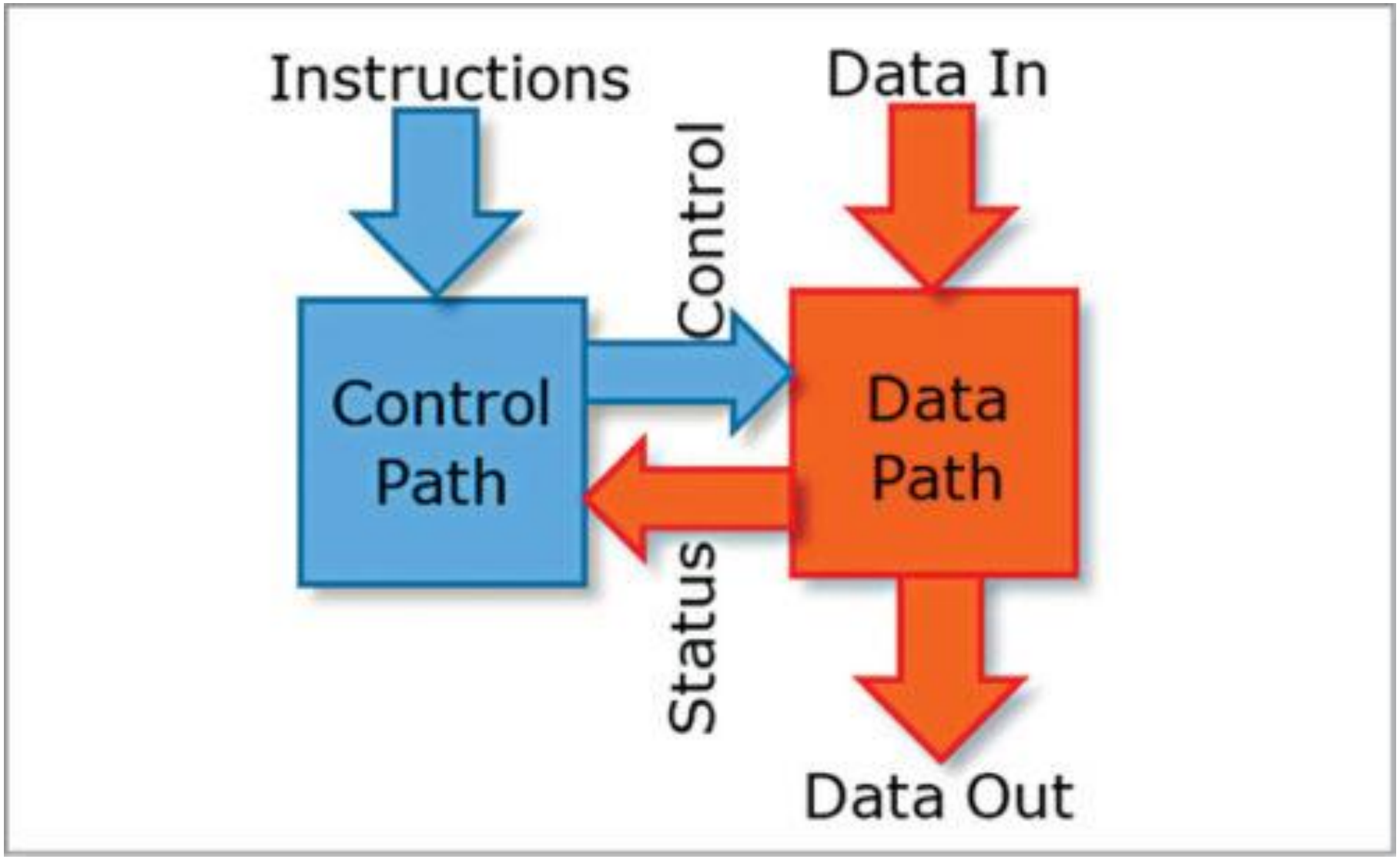
- Code Generation Optimization is largely depending on the instruction set architecture (ISA).
  - Instruction Level Parallelism (ILP)
  - CISC versus RISC (Ch. 5)
  - Scalar Instruction, Superscalar, Very Long Instruction Word Instruction (VLIW) (Ch. 6)
  - Loop Unrolling to exploit more ILP (Ch. 6)

Computer Architectural Issues that Impacts Programming Language

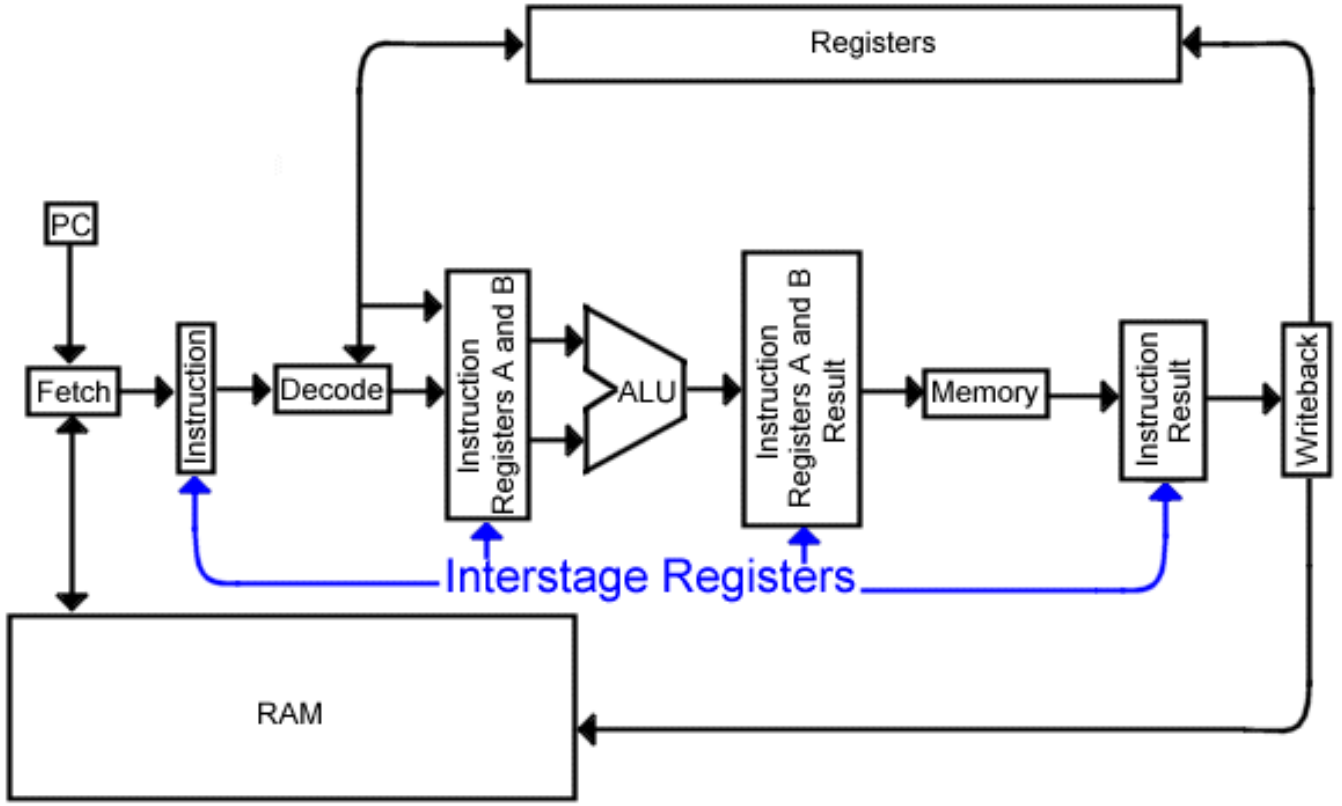
- Platform dependency (ISA-compatibility), x86, 32-bit z86, 64-bit x86, iOS, Linux
  - Virtual Machine
  - Interpreter
  - Native Code Compilation (NC, Just-in-Time Compilation)
- Multithreading (Task Level Parallelism) which requires language support (Concurrency) (Ch. 13)
- Multi-core (SIMD, MISD, MIMD) multi-processor. Take advantage of multi-processing.

Programming Language and Compiler Design

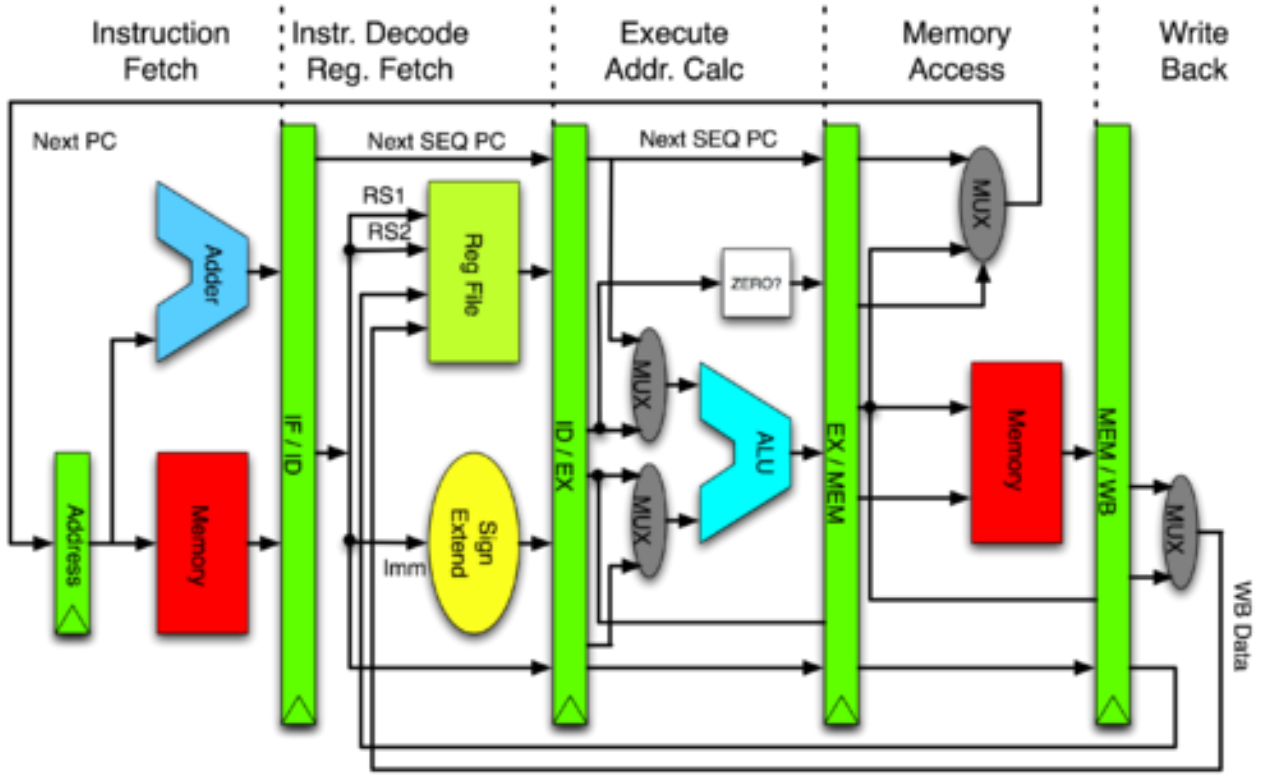
- To take advantage of the instruction level parallelism (ILP) by compiler code generation optimization
- Vector processing, Loop Unrolling as compiler technology to exploit parallelism at loop level.
- To take advantage of the task level parallelism (TLP) by concurrent programming and process scheduling (virtual machine or execution engine or OS loader)


Instructions
Data In
Control
Control Path
Data Path
Status
Data Out

Pipelining – Datapath without Pipelining



Pipelining – Data Path



Memory Hierarchy

Load Operations may cause the CPU Stalls.

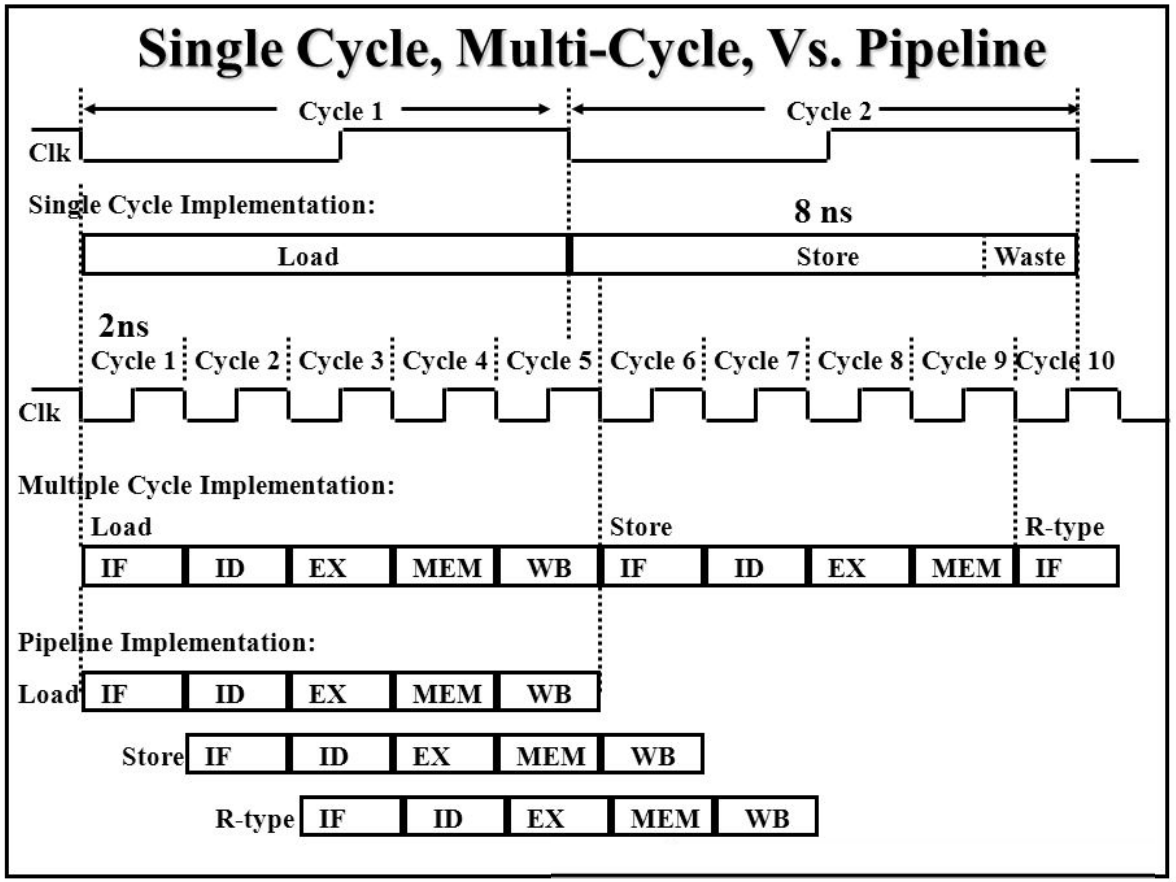
Memory Fetch Techniques:
Caching
Snooping Bus
Virtual Memory

	Typical access time	Typical capacity
Registers	0.2–0.5 ns	256–1024 bytes
Primary (L1) cache	0.4–1 ns	32 K–256 K bytes
L2 or L3 (on-chip) cache	4–30 ns	1–32 M bytes
off-chip cache	10–50 ns	up to 128 M bytes
Main memory	50–200 ns	256 M–16 G bytes
Flash	40–400 μs	4 G bytes to 1 T bytes
Disk	5–15 ms	500 G bytes and up
Tape	1–50 s	effectively unlimited

Figure 5.1 The memory hierarchy of a workstation-class computer. Access times and capacities are approximate, based on 2015 technology. Registers are accessed within a single clock cycle. Primary cache typically responds in 1 to 2 cycles; off-chip cache in more like 20 cycles. Main memory on a supercomputer can be as fast as off-chip cache; on a workstation it is typically much slower. Flash times vary with manufacturing technology, and are longer for writes than reads. Disk and tape times are constrained by the movement of physical parts.

Pipelining - CISC

Depending on the ILP



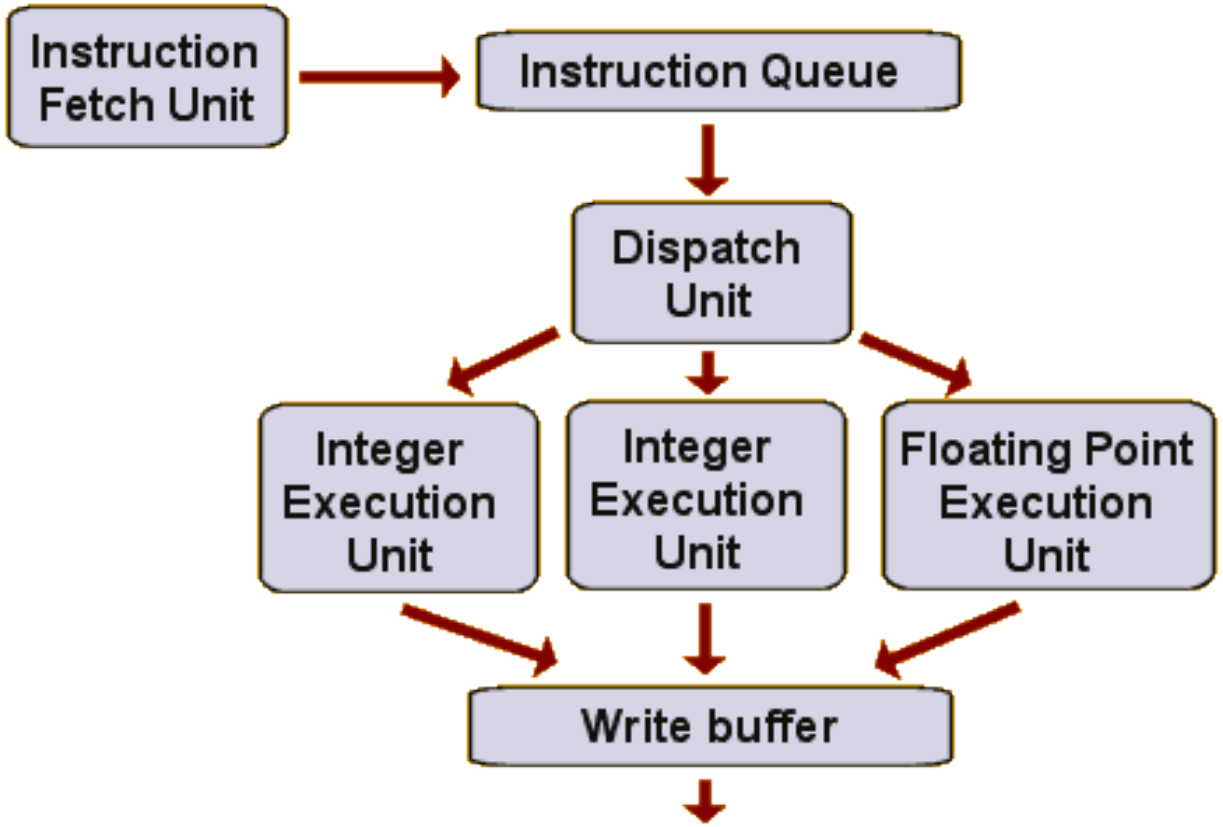
Pipelining - RISC

Depending on the ILP

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Superscalar

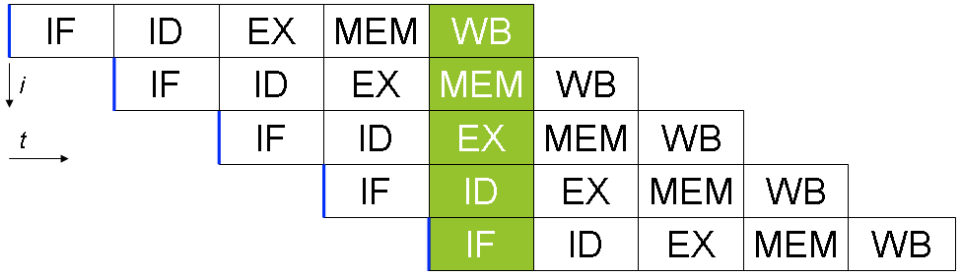
Degree of Parallelism (Int, FP)



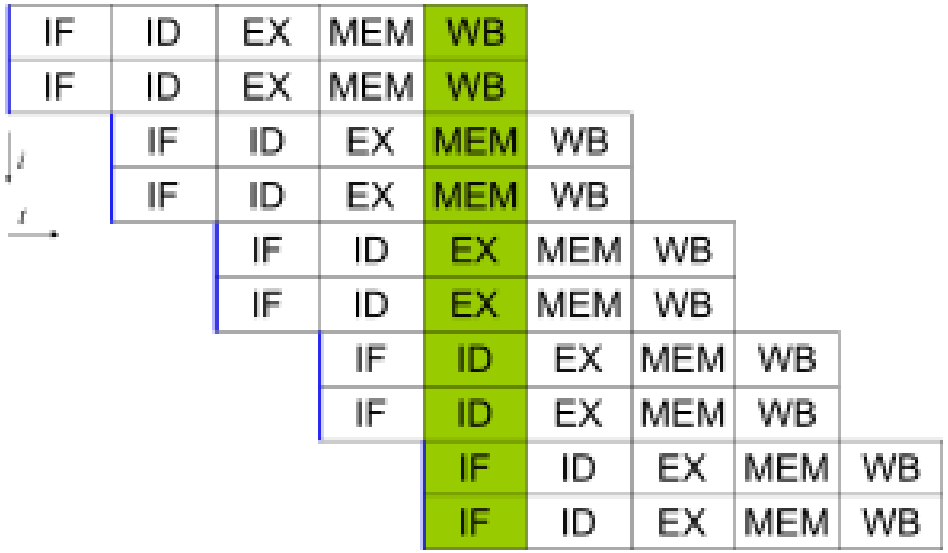
Superscalar

Degree of Parallelism (Int, FP)

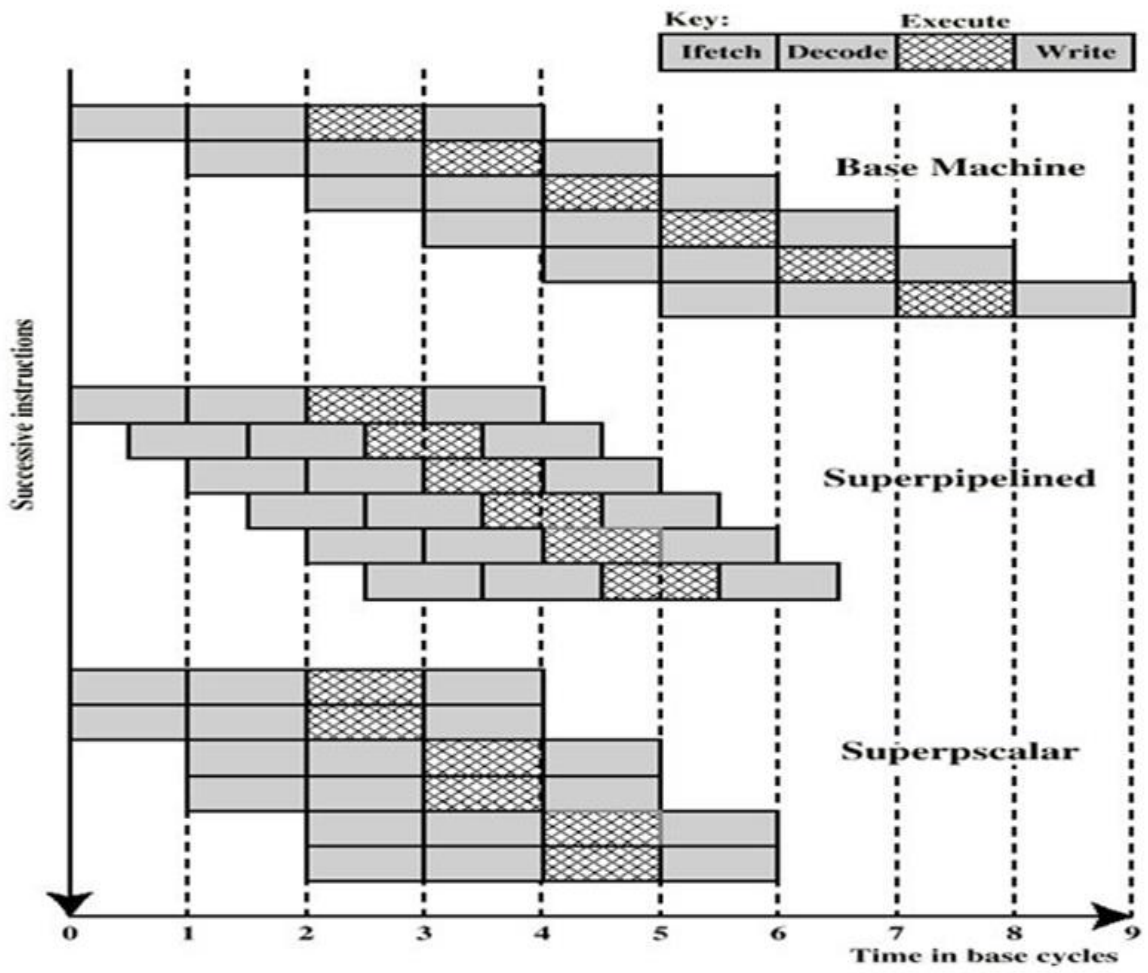
Pipelining for Scalar



Pipelining for Superscalar

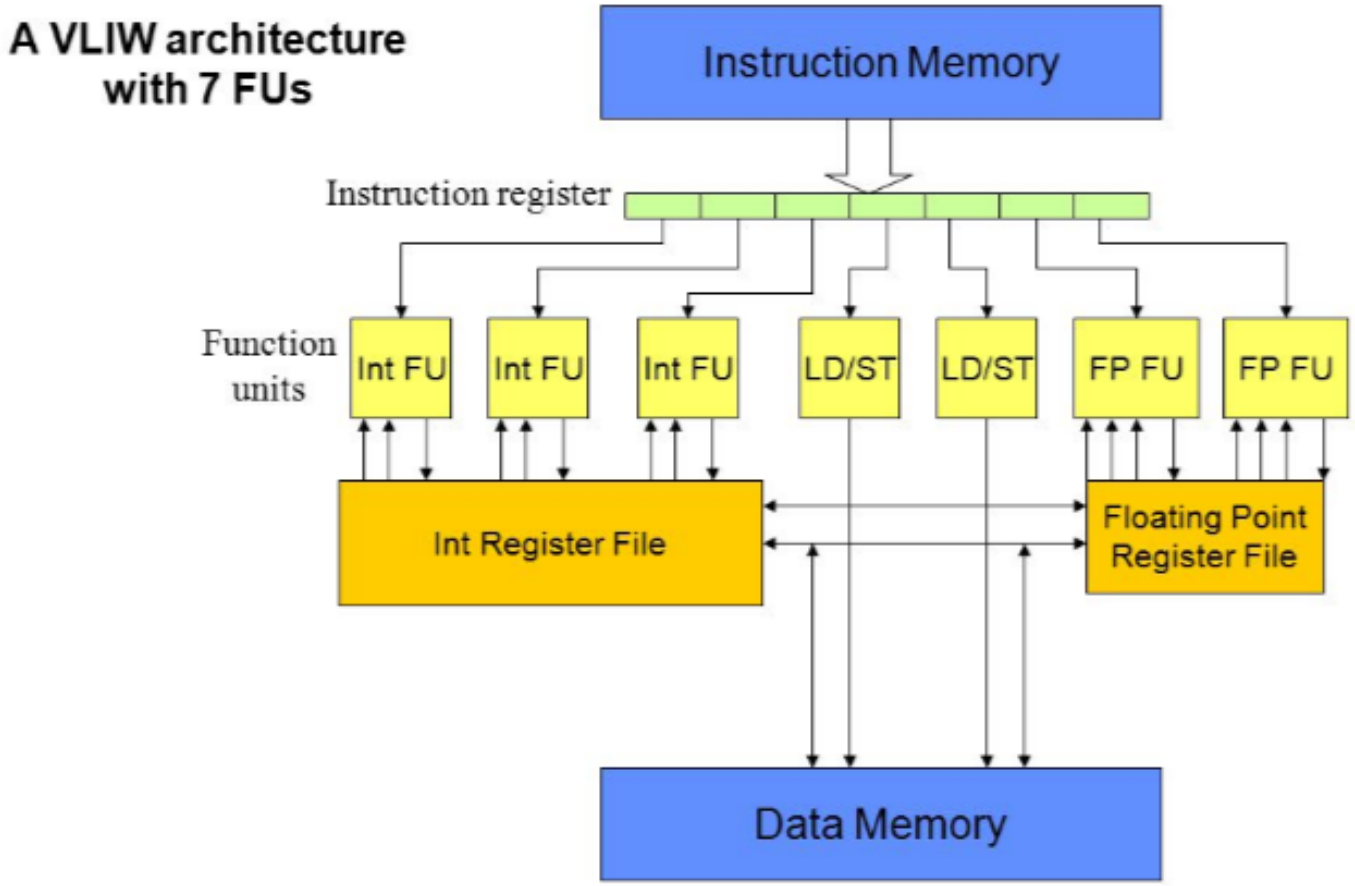


Super pipelined



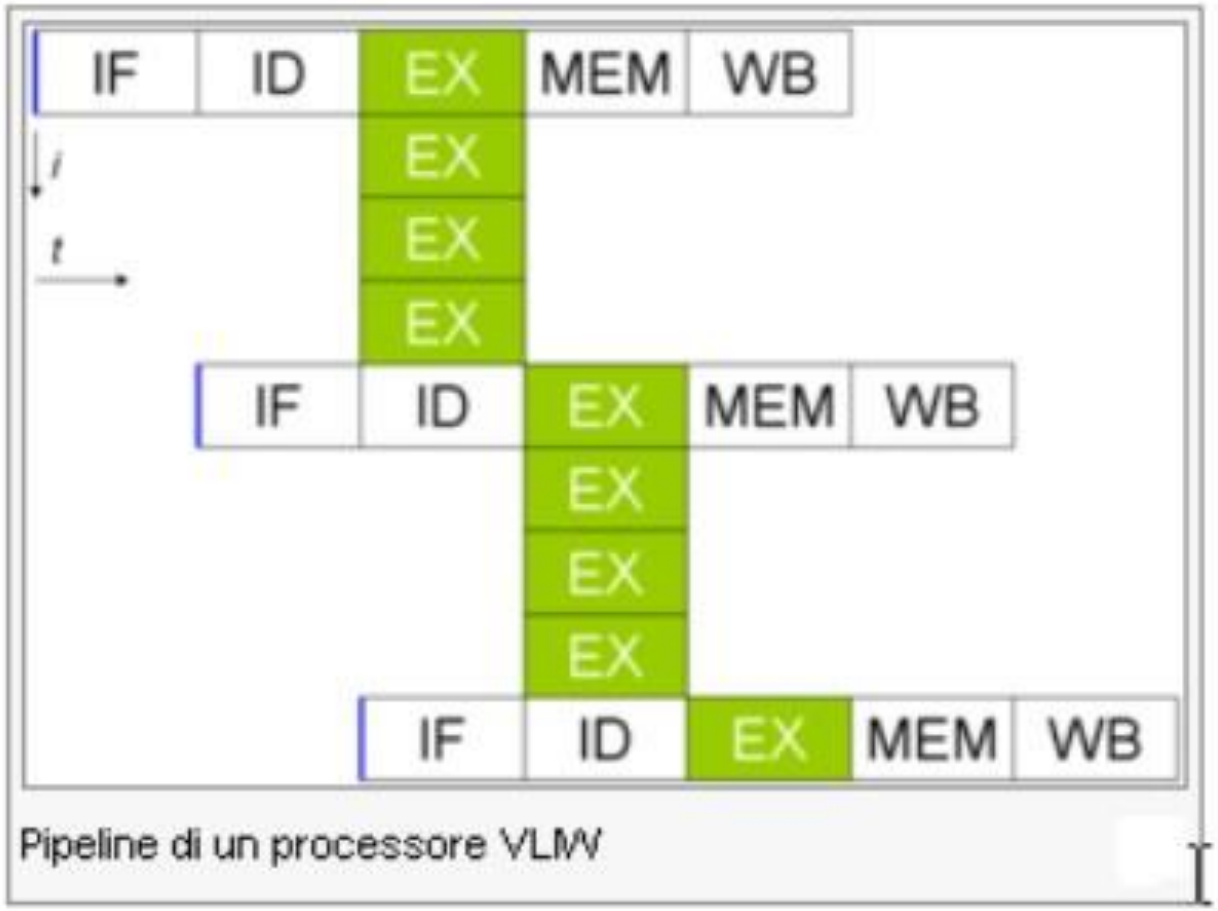
VLIW

Depending on Compiler a Lot



VLIW

Depending on Compiler a Lot



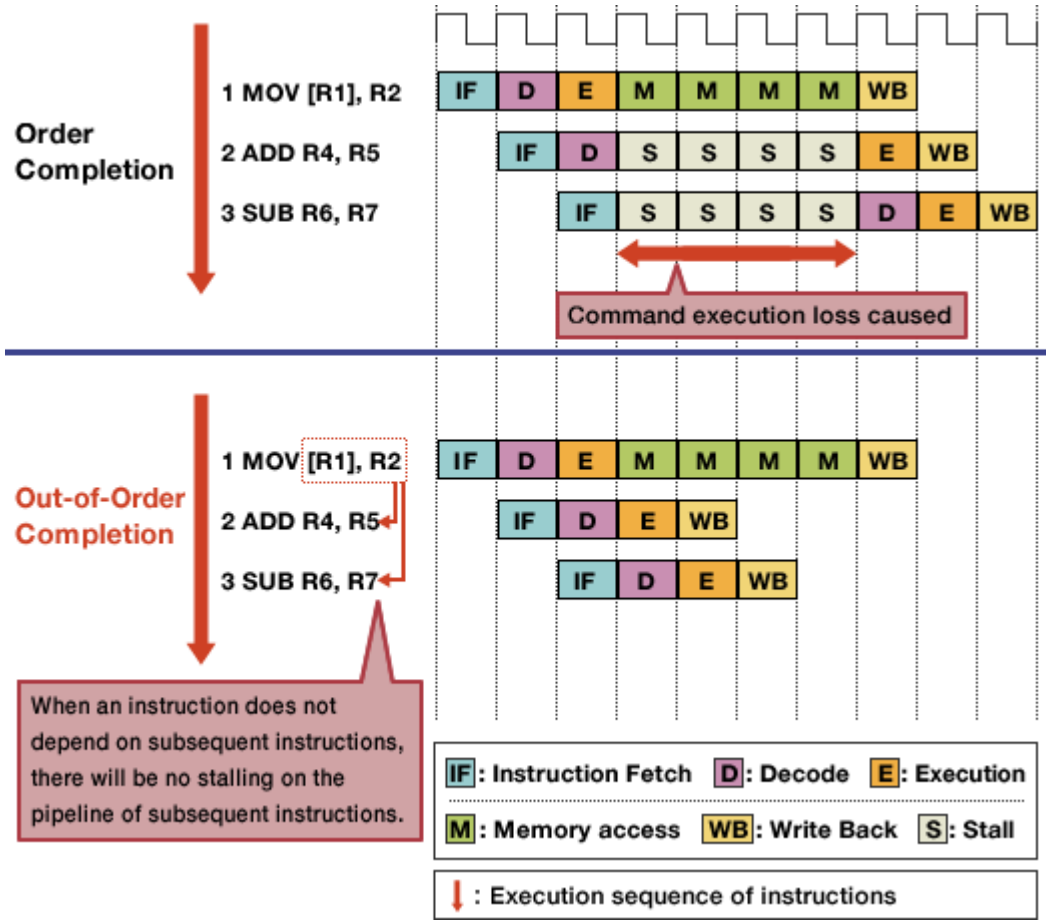
Pipeline di un processore VLIW

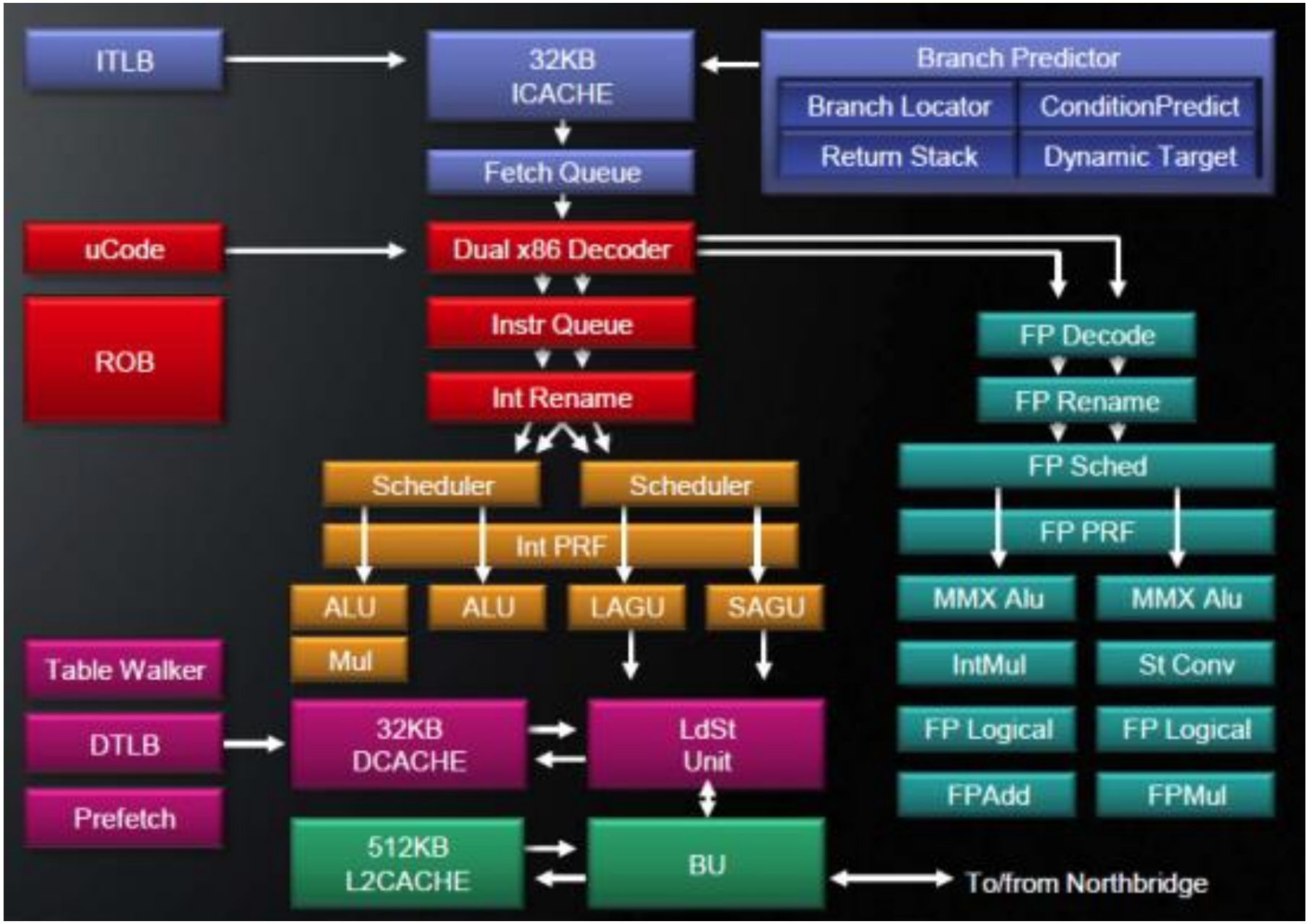
Out of Order Execution

IDLE State Avoidance

Out-of-Order Execution (Pentium II)

Cycle	1	2	3	4	5	6	7	8	9
$Instr_1$	Fetch	Decode	Execute			Write			
$Instr_2$		Fetch	Decode	Wait		Execute	Write		
$Instr_3$			Fetch	Decode	Execute	Write			
$Instr_4$				Fetch	Decode	Wait	Execute	Write	
$Instr_5$					Fetch	Decode	Execute	Write	
$Instr_6$						Fetch	Decode	Execute	Write

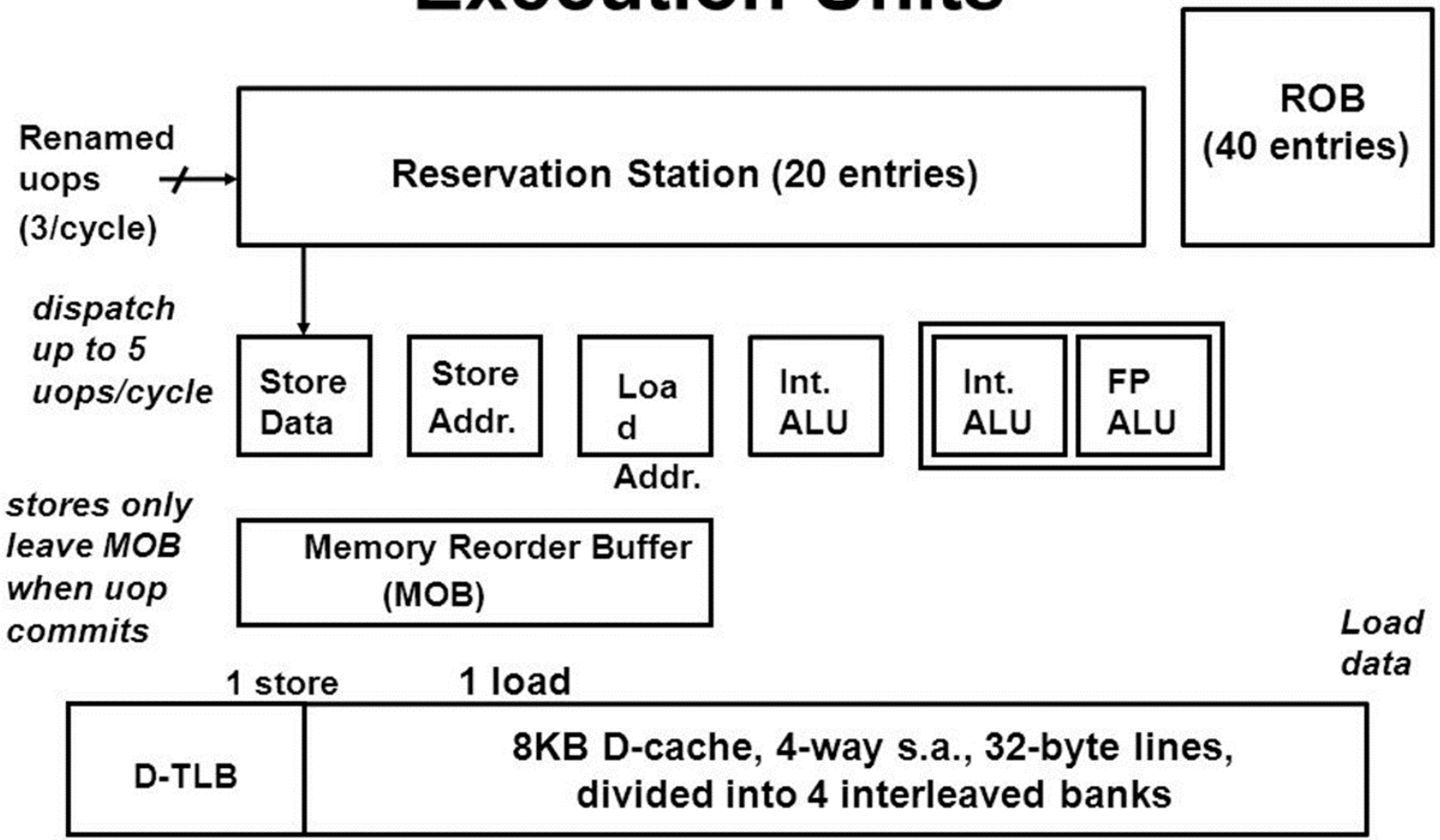




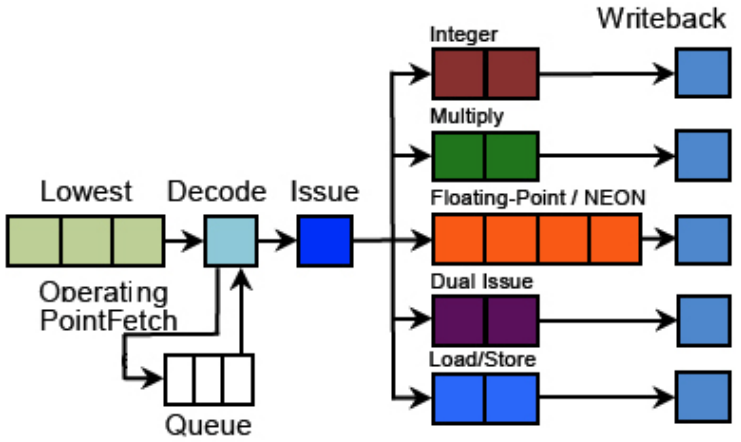
Out of Order Execution

Microcode

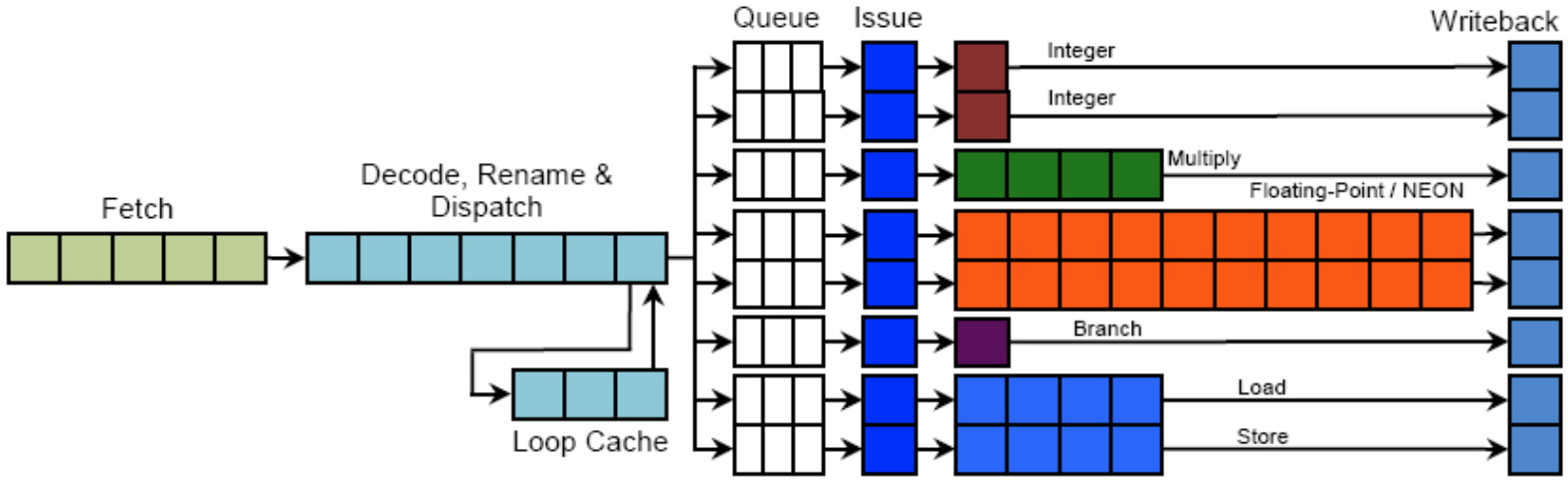
P6 Reservation Stations and Execution Units



D-TLB has 64 entries for 4KB pages fully assoc., plus 8 entries for 4MB pages, 4-way s.a.



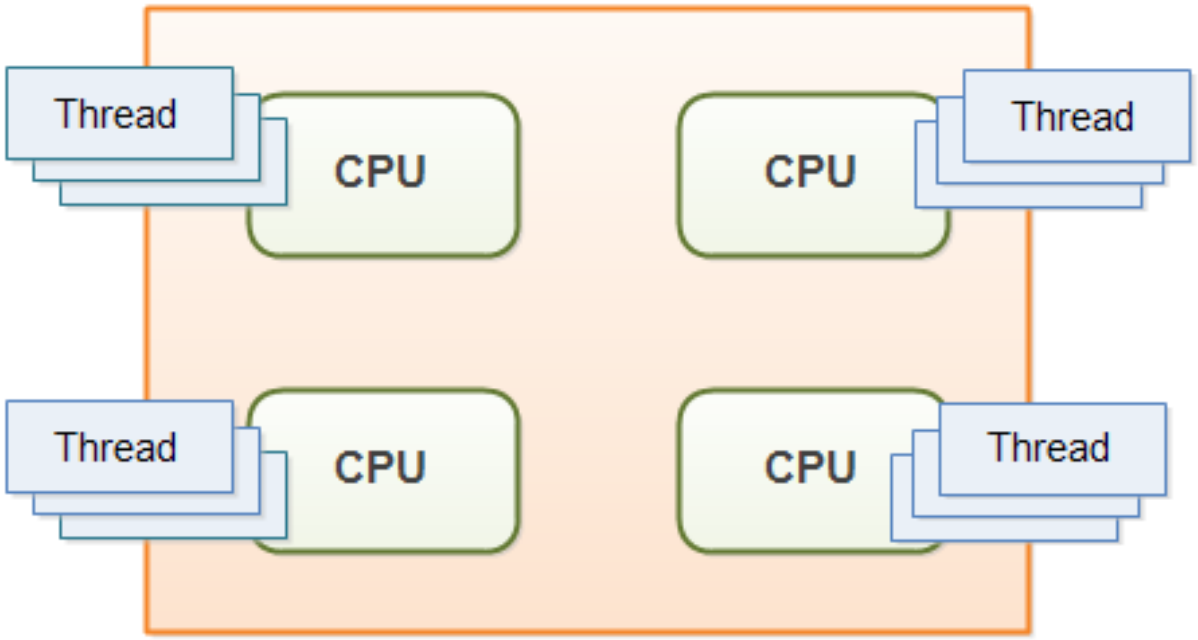
Cortex-A7 Pipeline

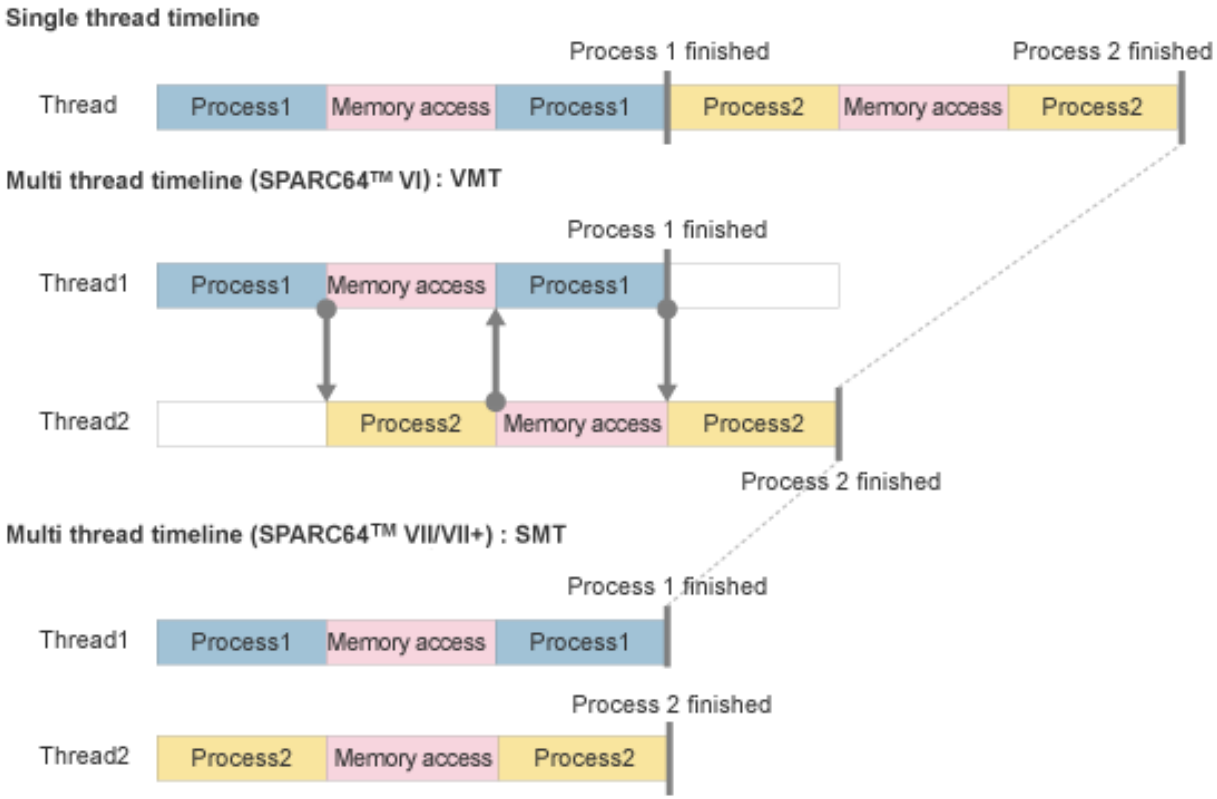


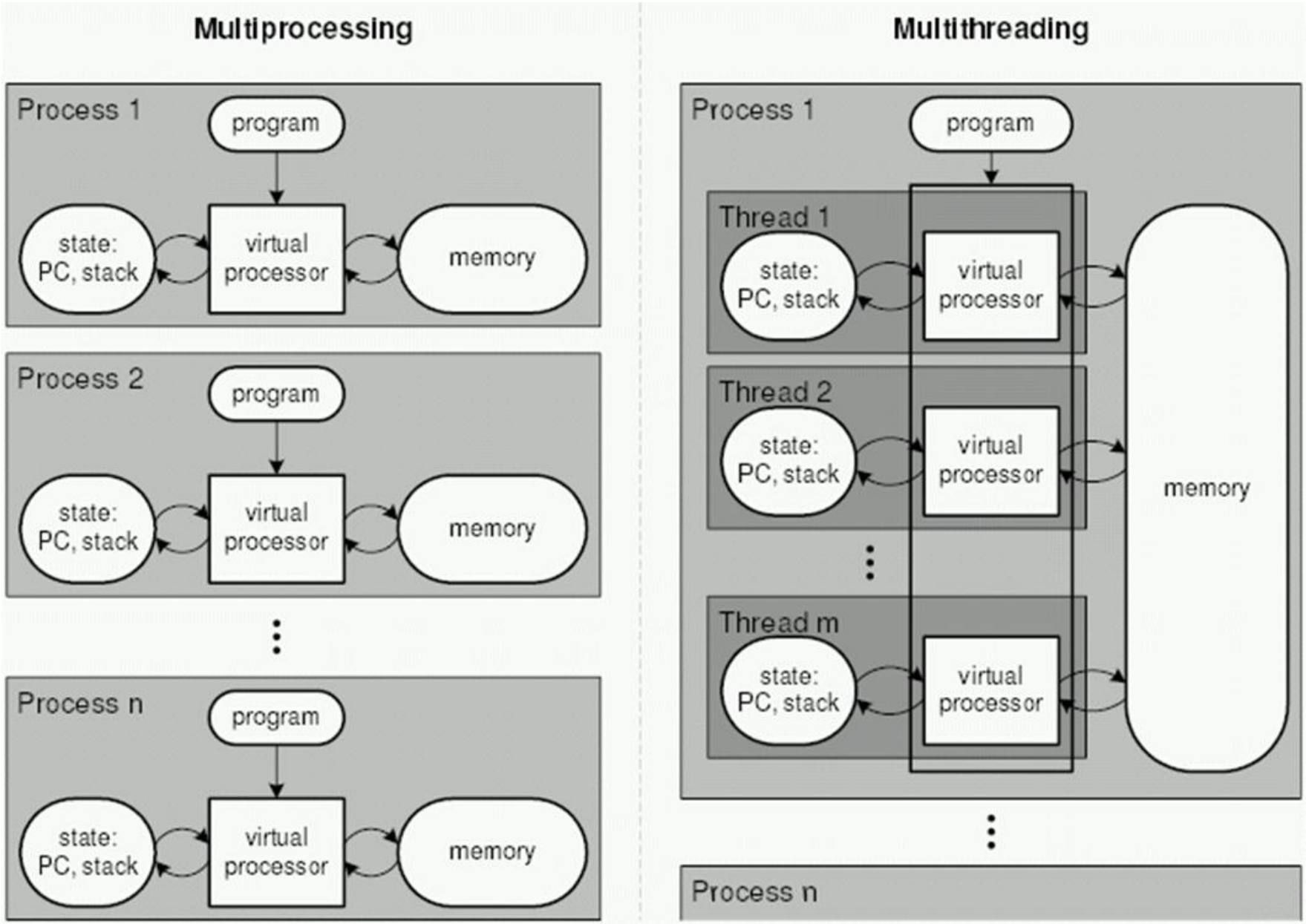
Cortex-A15 Pipeline

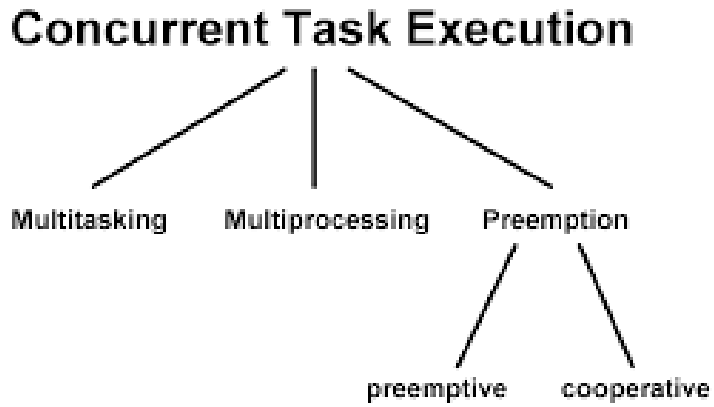
Multi-core (Quad-Core Processors)

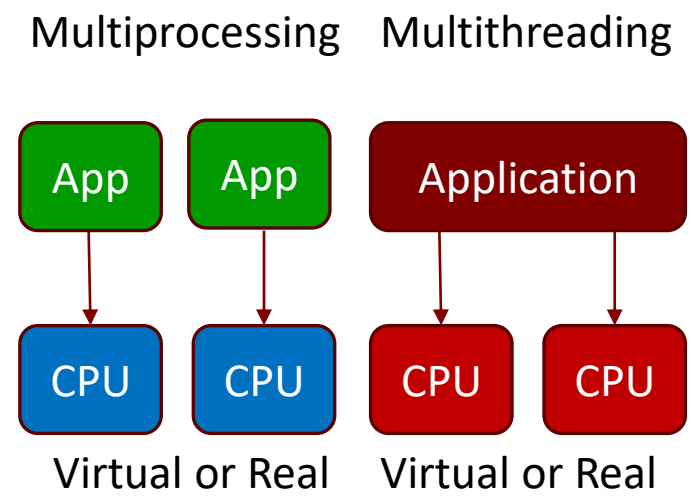
Task Level Parallelism (Concurrent Programming)





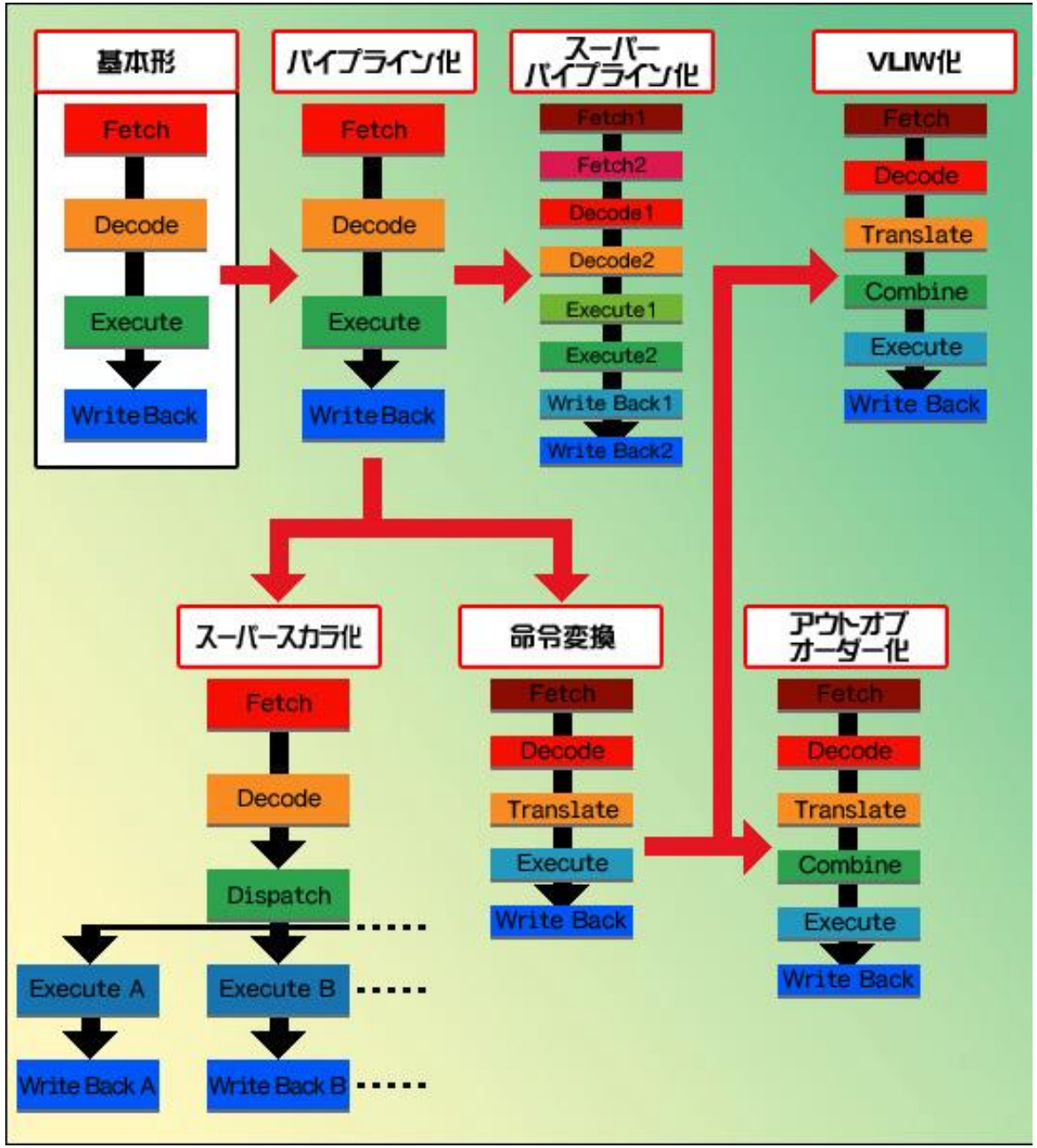

Multiprocessing
Process 1
program
state:
PC, stack
virtual
processor
memory
Process 2
program
state:
PC, stack
virtual
processor
memory
Process n
program
state:
PC, stack
virtual
processor
memory
Multithreading
Process 1
program
Thread 1
state:
PC, stack
virtual
processor
Thread 2
state:
PC, stack
virtual
processor
Thread m
state:
PC, stack
virtual
processor
memory
Process n


Concurrent Task Execution
Multitasking
Multiprocessing
Preemption
preemptive
cooperative


Multiprocessing
Multithreading
App
App
Application
CPU
CPU
CPU
CPU
Virtual or Real
Virtual or Real

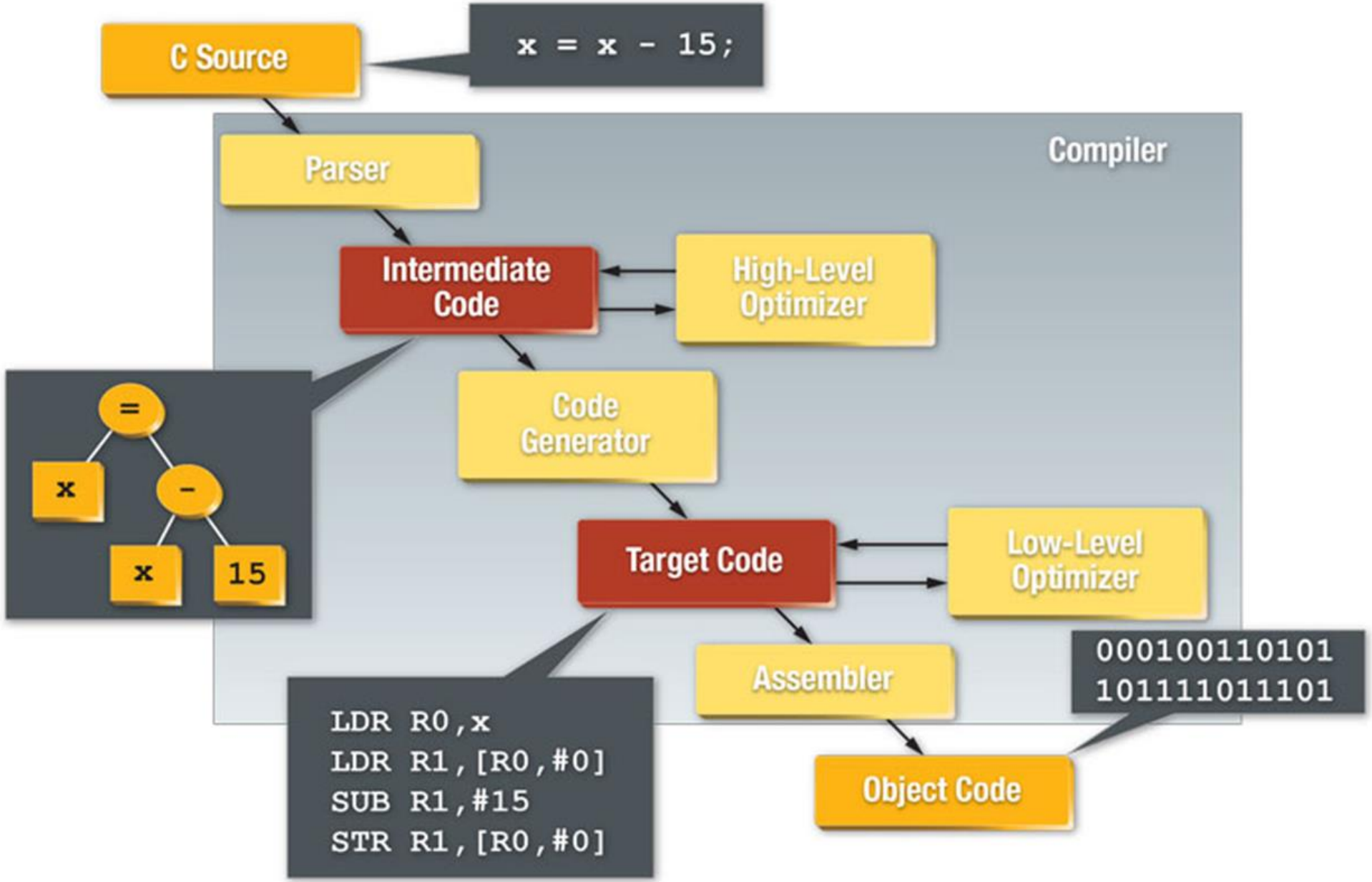
Datapath Design Styles

Japanese	English
基本形	Basic Format
パイプライン化	Pipelined
スーパーパイプライン化	Super pipelined
VLIW化	VLIW
スーパースカラ化	Superscalar
命令変換	Instruction Translation
アウトオブオーダー化	Out of Order Translation

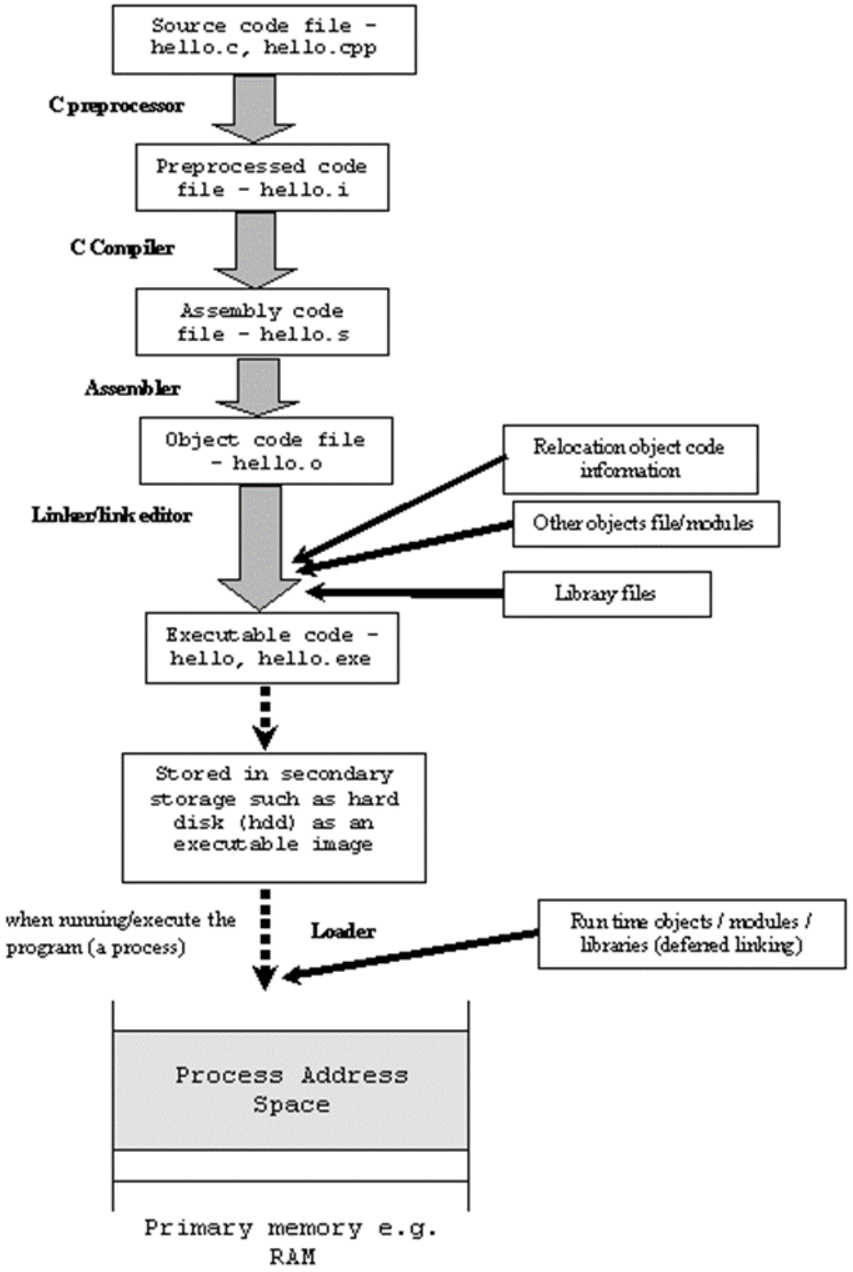


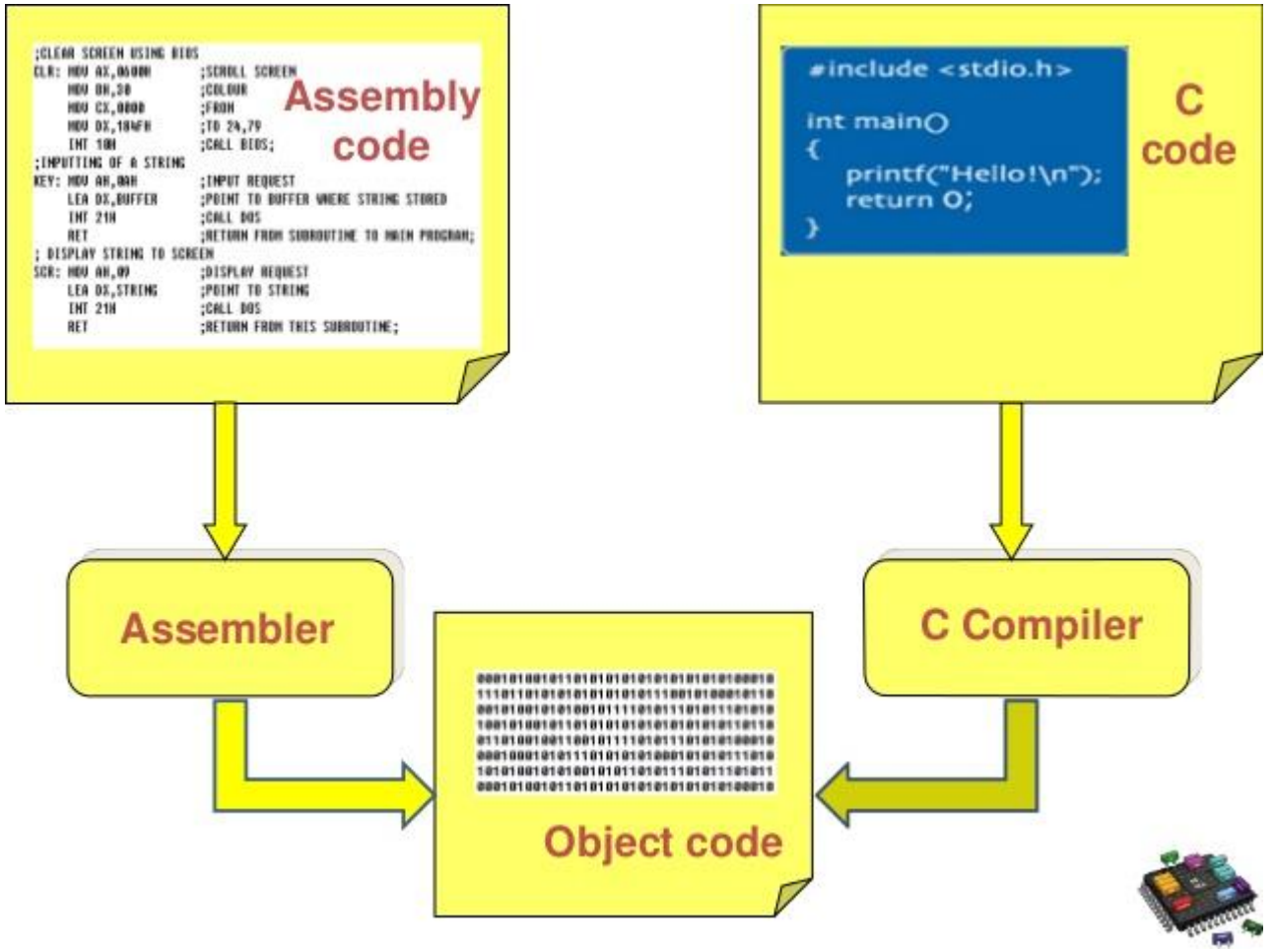
Assembly-Level View

SECTION 4


C Source
x = x - 15;
Compiler
Parser
Intermediate Code
High-Level Optimizer
=
x
-
x
15
Code Generator
Target Code
Low-Level Optimizer
LDR R0,x
LDR R1,[R0,#0]
SUB R1,#15
STR R1,[R0,#0]
Assembler
000100110101
101111011101
Object Code

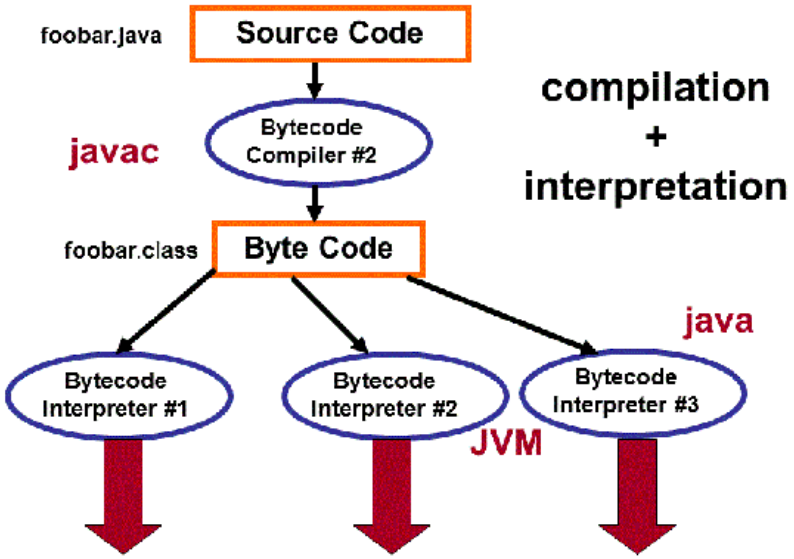
Assembly As Target Code for C

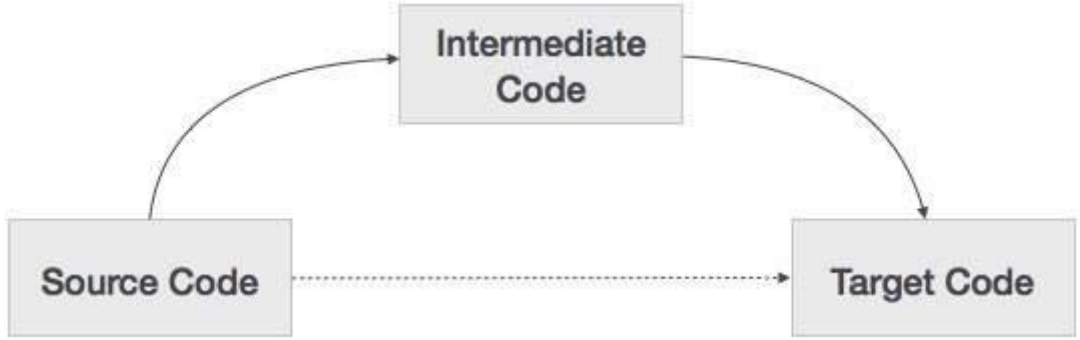


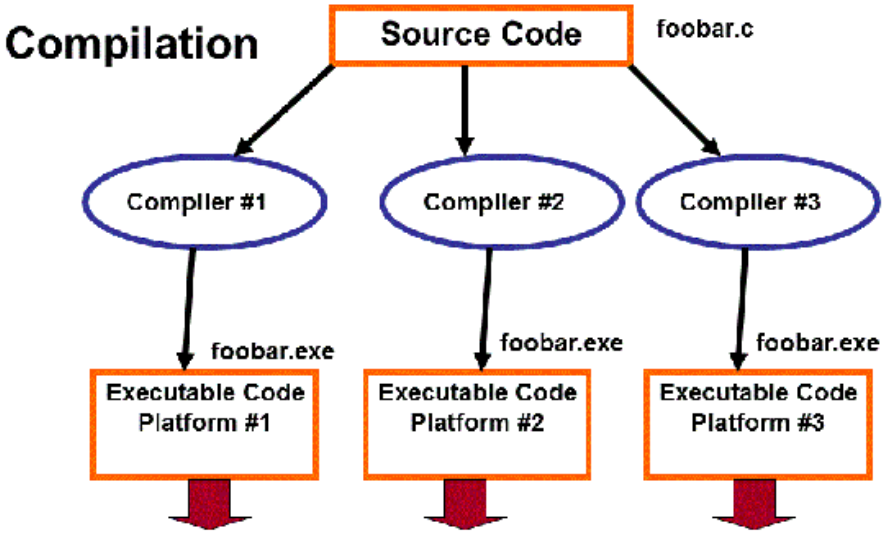


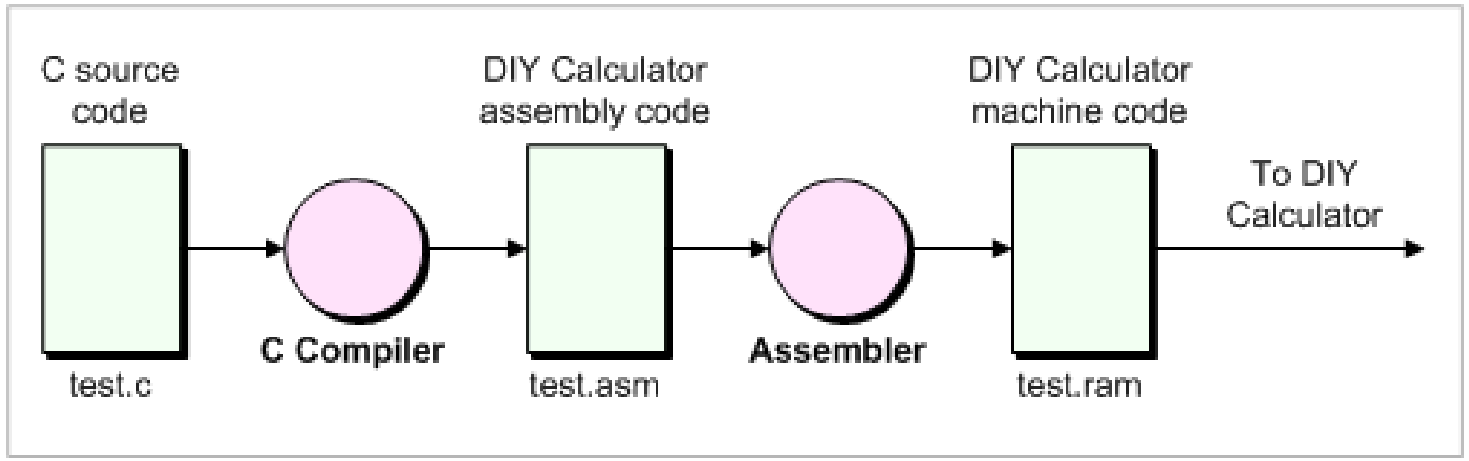
Assembly is One Form of Intermediate Code

Byte-Code is a Kind of Machine-Independent Assembly Code

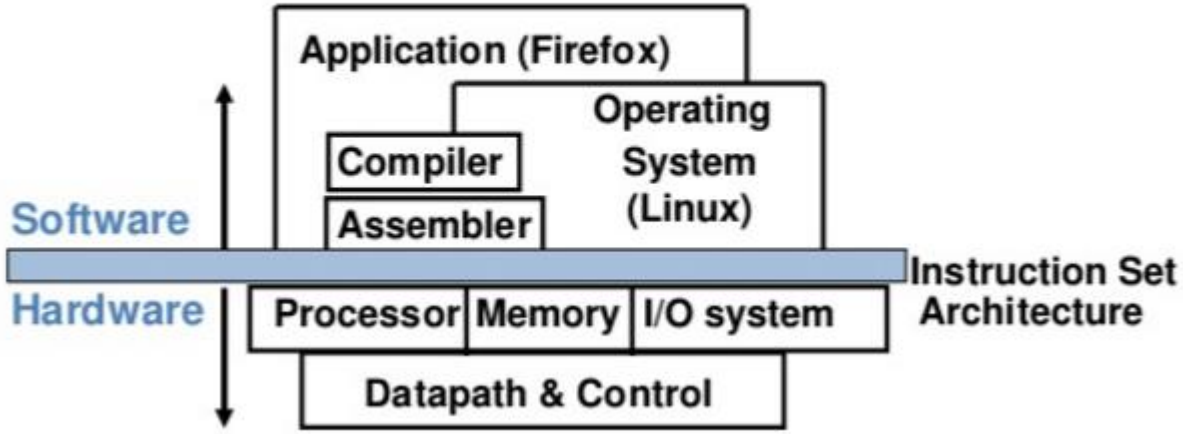








Instruction Set Architecture (ISA)



High Level Language to Assembly

```
do {
  g = g + A[i];
  i = i + j;
} while (i != h);
```

```
L1:  g = g + A[i];
     i = i + j;
     if (i = h) goto L1
```

```
g: $s1
h: $s2
i:  $s3
j:  $s4
Base of A: $s5
```

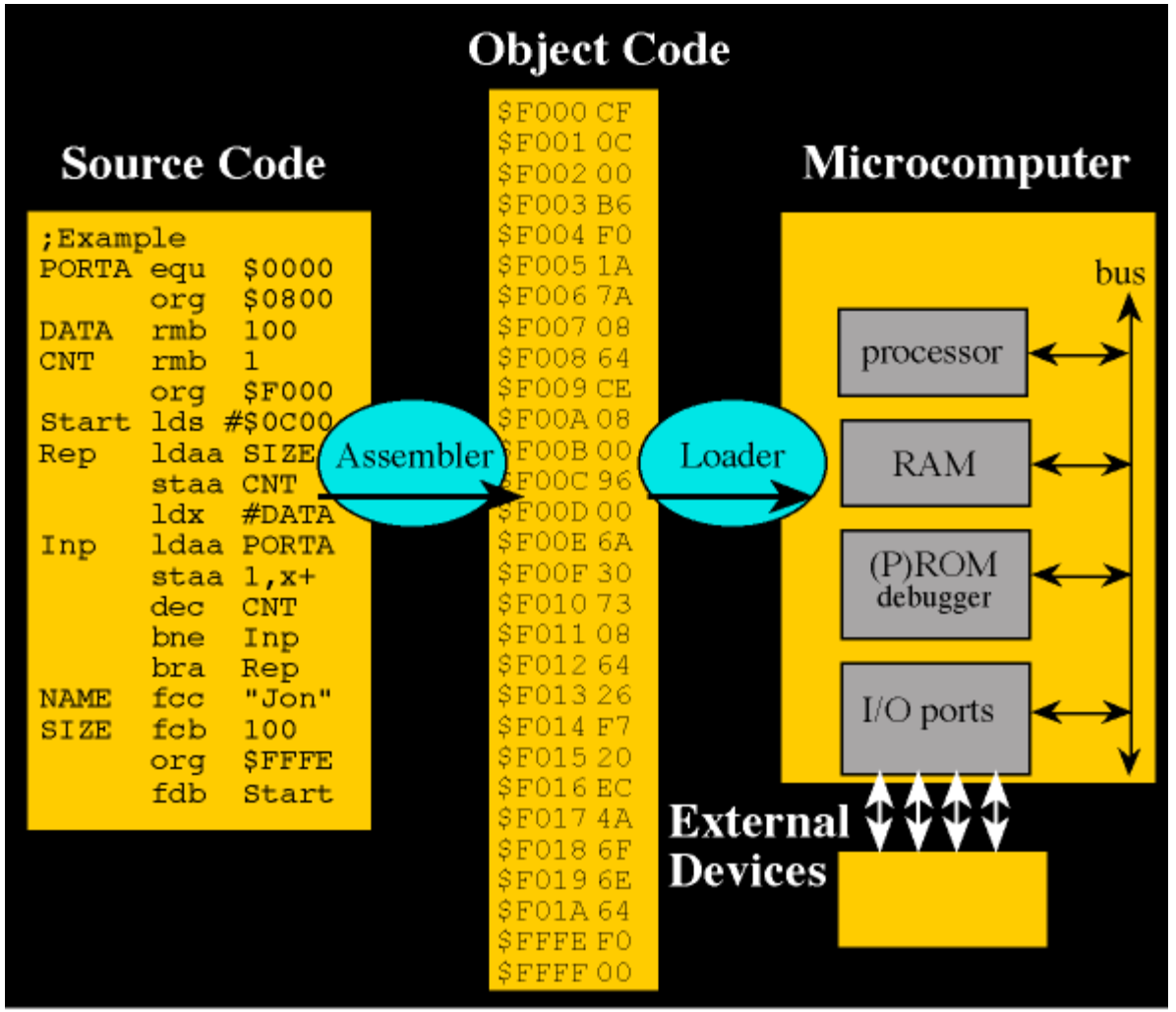
```
L1:  sll  $t1, $s3, 2     # $t1 = 4*i
     add $t1, $t1, $s5    #  $t1 = addr of A
     lw   $t1, 0($t1)     #  $t1 = A[i]
     add $s1, $s1, $t1    #  g = g + A[i]
     add $s3, $s3, $s4    #  i  = i + j
     bne $s3, $s2, L1     #  go to L1 if i != h
```

	15 14 13 12	11 10 9	8 7 6	5	4 3	2 1 0
ADD+	0001	DR	SR1	0	00	SR2
ADD+	0001	DR	SR1	1	imm5	
AND+	0101	DR	SR1	0	00	SR2
AND+	0101	DR	SR1	1	imm5	
BR	0000	n	z	p	PCoffset9	
JMP	1100	000	BaseR	000000		
JSR	0100	1	PCoffset11			
JSRR	0100	0	00	BaseR	000000	
LD+	0010	DR	PCoffset9			
LDI+	1010	DR	I	PCoffset9		
LDR+	0110	DR	BaseR	offset6		
LEA+	1110	DR	PCoffset9			
NOT+	1001	DR	SR	111111		
RET	1100	000	111	000000		
RTI	1000	000000000000				
ST	0011	SR	PCoffset9			
STI	1011	SR	PCoffset9			
STR	0111	SR	BaseR	offset6		
TRAP	1111	0000	trapvect8			
reserved	1101					

Assembly To Instruction

```
LOC  OBJECT CODE    LINE SOURCE TEXT
 VALUE

 00000006          00001 PORTB  EQU     06H          ;PortB data register
 00000010          00002 COUNT  EQU     10H          ;GPR register
 00000011          00003 MYREG  EQU     11H
                   00004
                   00005        ORG     0H
0000   3000        00006        movlw   B'00000000'
0001   0066        00007        tris    PORTB
                   00008
0002   0190        00009        CLRF    COUNT        ;COUNT = 0
0003   2???        00010 BACK   CALL    DISPLAY
0004   2???        00011        GOTO     BACK
                   00012
                   00013 ;increase value & send it to PORTB subroutine
0005   0A90        00014 DISPLAY INCF   COUNT,F      ;count = count + 1
0006   0810        00015        MOVF     COUNT,W
0007   0086        00016        MOVWF   PORTB
0008   2???        00017        CALL    DELAY
0009   0008        00018        RETURN               ;return to caller
                   00019
```



Assembly-Level View

- As mentioned early in this course, a compiler is simply a translator
  - It translates programs written in one language into programs written in another language
    - This other language can be almost anything
    - Most of the time, however, it's the machine language for some available computer

Assembly-Level View

- As a review, we will go over some of the material most relevant to language implementation, so that we can better understand
  - what the compiler has to do to your program
  - why certain things are fast and others slow
  - why certain things are easy to compile and others aren't

Assembly-Level View

- There are many different programming languages and there are many different machine languages
  - Machine languages show considerably less diversity than programming languages
  - Traditionally, each machine language corresponds to a different computer Architecture
  - The Implementation is how the architecture is realized in hardware

Assembly-Level View

- Formally, an architecture is the interface to the hardware
  - what it looks like to a user writing programs on the bare machine.
- In the last 30 years, the line between these has blurred to the point of disappearing
  - compilers have to know a LOT about the implementation to do a decent job

Assembly-Level View

- Changes in hardware technology (e.g., how many transistors can you fit on one chip?) have made new implementation techniques possible
  - the architecture was also modified
  - *Example*: RISC (reduced instruction set computer) revolution ~30 years ago
- In the discussion below, we will focus on modern RISC architectures, with a limited amount of coverage of their predecessors, the CISC architectures

Memory Hierarchy and Data Representation

SECTION 5

Memory Hierarchy

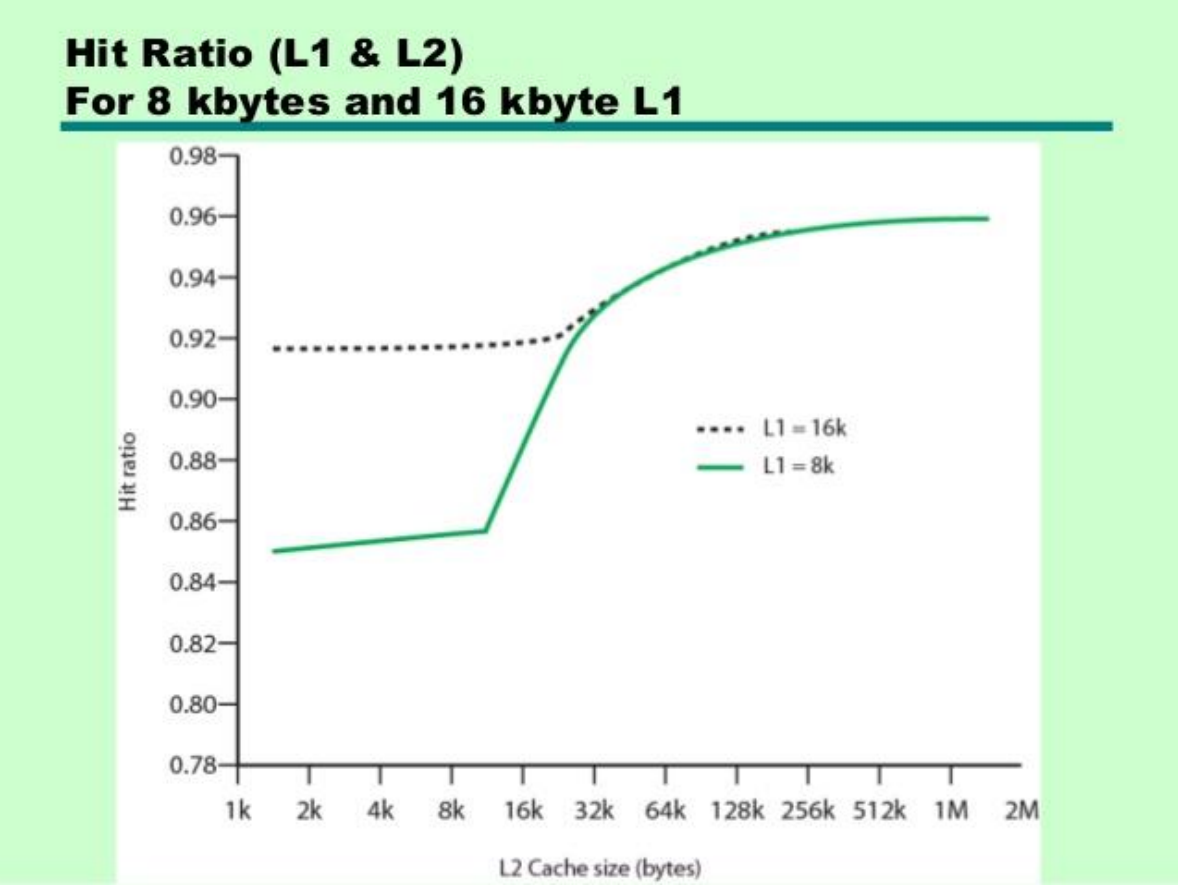
- Memory is too big to fit on one chip with a processor
  - Because memory is off-chip (in fact, on the other side of the bus), getting at it is much slower than getting at things on-chip **(Note: No longer a problem for single chip computer.)**
  - Most computers therefore employ a MEMORY HIERARCHY, in which things that are used more often are kept close at hand

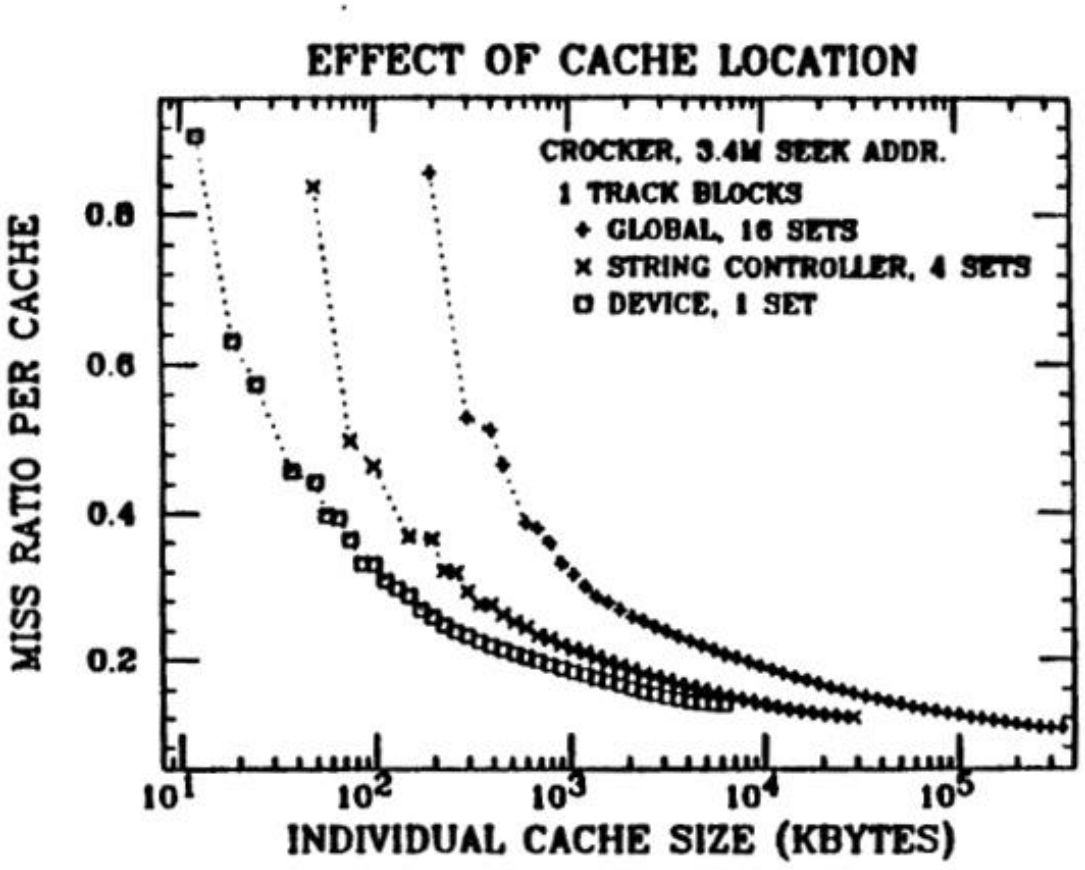
	Typical access time	Typical capacity
Registers	0.2–0.5 ns	256–1024 bytes
Primary (L1) cache	0.4–1 ns	32 K–256 K bytes
L2 or L3 (on-chip) cache	4–30 ns	1–32 M bytes
off-chip cache	10–50 ns	up to 128 M bytes
Main memory	50–200 ns	256 M–16 G bytes
Flash	40–400 μs	4 G bytes to 1 T bytes
Disk	5–15 ms	500 G bytes and up
Tape	1–50 s	effectively unlimited

Figure 5.1 The memory hierarchy of a workstation-class computer. Access times and capacities are approximate, based on 2015 technology. Registers are accessed within a single clock cycle. Primary cache typically responds in 1 to 2 cycles; off-chip cache in more like 20 cycles. Main memory on a supercomputer can be as fast as off-chip cache; on a workstation it is typically much slower. Flash times vary with manufacturing technology, and are longer for writes than reads. Disk and tape times are constrained by the movement of physical parts.

Hit Ratio and Miss Ratio of Cache

It is directly related to the performance of the processor





Memory Hierarchy

- Some of these levels are visible to the programmer; others are not
- For our purposes here, the levels that matter are *registers* and *main memory*
- Registers are special locations that can hold (a very small amount of) data that can be accessed very quickly
  - A typical RISC machine has a few (often two) sets of registers that are used to hold integer and floating point operands

Memory Hierarchy

- It also has several special-purpose registers, including the
  - program counter (PC)
    - holds the address of the next instruction to be executed
    - usually incremented during fetch-execute cycle
  - processor status register
    - holds a variety of bits of little interest in this course (privilege level, interrupt priority level, trap enable bits)

Data Representation

- Memory is usually (but not always) byte-addressable, meaning that each 8-bit piece has a unique address
  - Data longer than 8 bits occupy multiple bytes
  - Typically
    - an integer occupies 16, 32, or (recently) 64 bits
    - a floating point number occupies 32, 64, or (recently) 128 bits

Computer Numbers

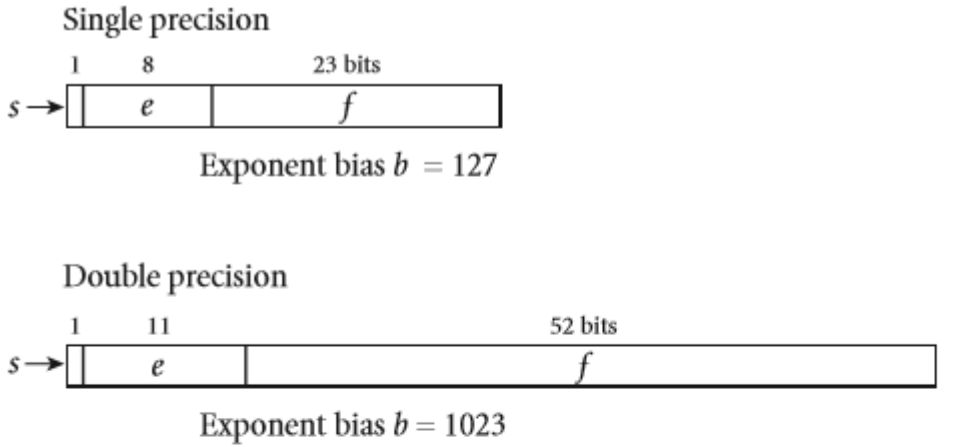
Integers and Floating Points

0000	0	1000	8
0001	1	1001	9
0010	2	1010	a
0011	3	1011	b
0100	4	1100	c
0101	5	1101	d
0110	6	1110	e
0111	7	1111	f

Figure 5.3 The hexadecimal digits.

0111	7	1111	−1
0110	6	1110	−2
0101	5	1101	−3
0100	4	1100	−4
0011	3	1011	−5
0010	2	1010	−6
0001	1	1001	−7
0000	0	1000	−8

Figure 5.4 **Four-bit two's complement numbers.** Note that there is a negative number (−8) that doesn't have a positive equivalent. There is only one zero, however.



	e	f	Value
Zero	0	0	± 0
Infinity	2b + 1	0	± ∞
Normalized	1 ≤ e ≤ 2b	<any>	± 1.f × 2^{e-b}
Denormalized	0	≠0	± 0.f × 2^{1-b}
NaN	2b + 1	≠0	NaN

Figure 5.5 **The IEEE 754 floating-point standard.** For normalized numbers, the exponent is $e - 127$ or $e - 1023$, depending on precision. The significand is $(1 + f) \times 2^{-23}$ or $(1 + f) \times 2^{-52}$, again depending on precision. Field f is called the *fractional part*, or *fraction*. Bit patterns in which e is all ones (255 for single-precision, 2047 for double-precision) are reserved for infinities and NaNs. Bit patterns in which e is zero but f is not are used for subnormal (gradual underflow) numbers.

Data Representation

- It is important to note that, unlike data in high-level programming languages, memory is untyped (bits are just bits)
- *Operations* are typed, in the sense that different operations *interpret* the bits in memory in different ways
- Typical DATA FORMATS include
  - instruction
  - integer (various lengths)
  - floating point (various lengths)

Data Representation

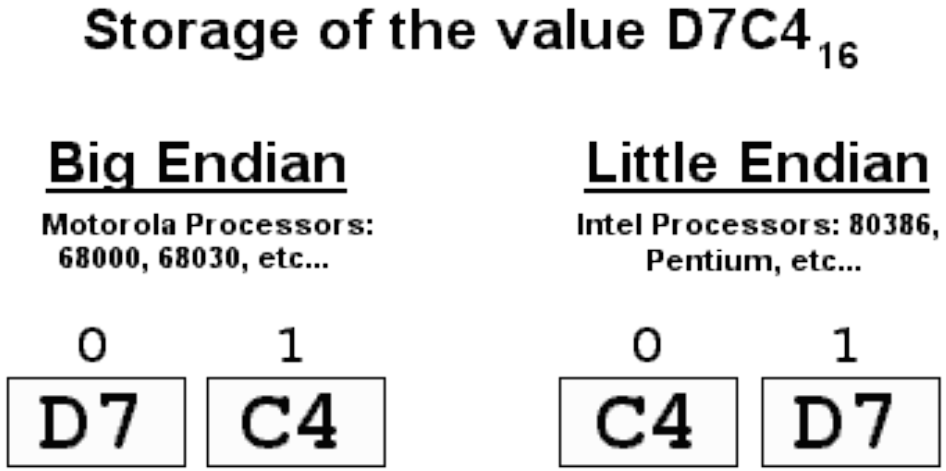
- Other concepts we will not detail but covered in other courses
  - Big-endian vs. little-endian (for details see Figure 5.2)
  - Integer arithmetic
    - 2's complement arithmetic
  - Floating-point arithmetic
    - IEEE standard, 1985

Data Representation

•Big endian machine: Stores data **big-end first**. When looking at multiple bytes, the first byte (lowest address) is the biggest.
•Little endian machine: Stores data **little-end first**. When looking at multiple bytes, the first byte is **smallest**.

•It varies machine to machine and language to language. Language with virtual machine like Java, it is interpreted by Language. (Machine effect is hidden.)

•As **BigInteger** is big-endian, while **BitSet** is little-endian, the bytes will be reversed when trying to convert directly from one to the other via toByteArray(). [Java Language]



Data Representation

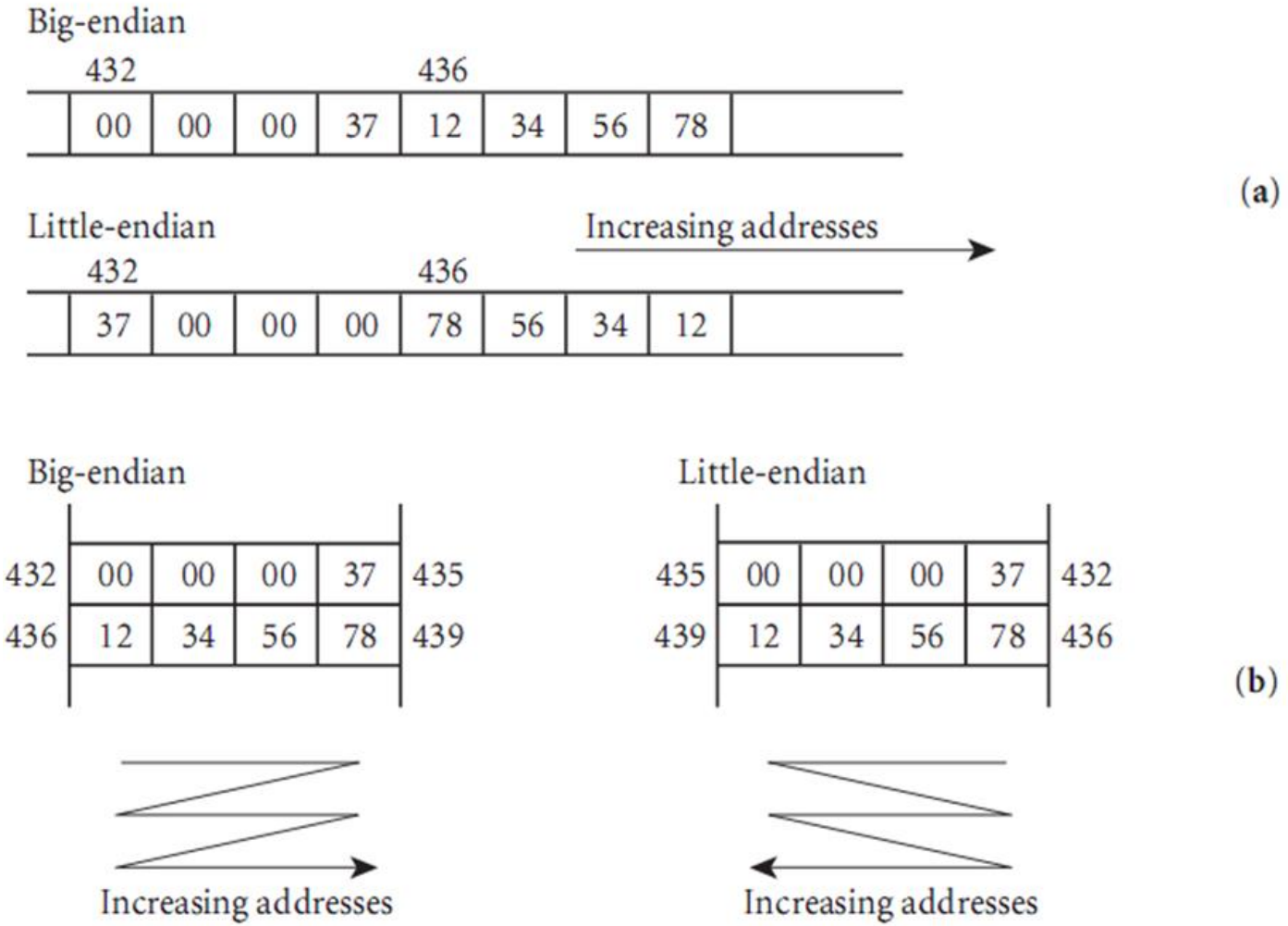
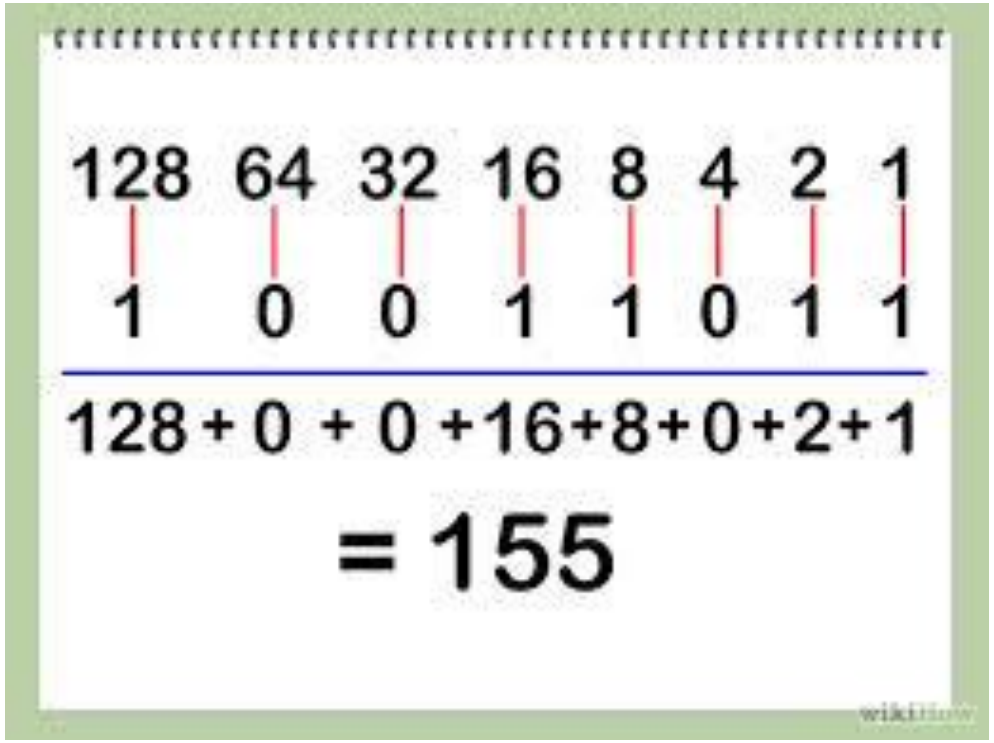


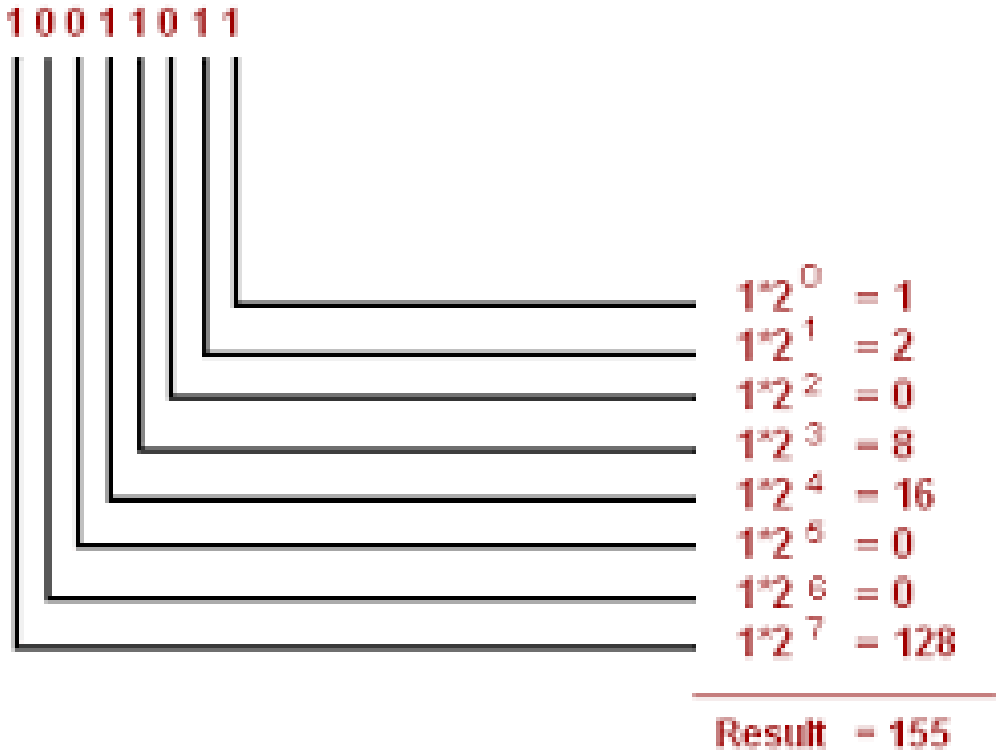
Figure 5.2 **Big-endian and little-endian byte orderings.** (a) Two four-byte quantities, the numbers 37_{16} and $12\ 34\ 56\ 78_{16}$, stored at addresses 432 and 436, respectively. (b) The same situation, with memory visualized as a byte-addressable array of words.

Processor Arithmetic

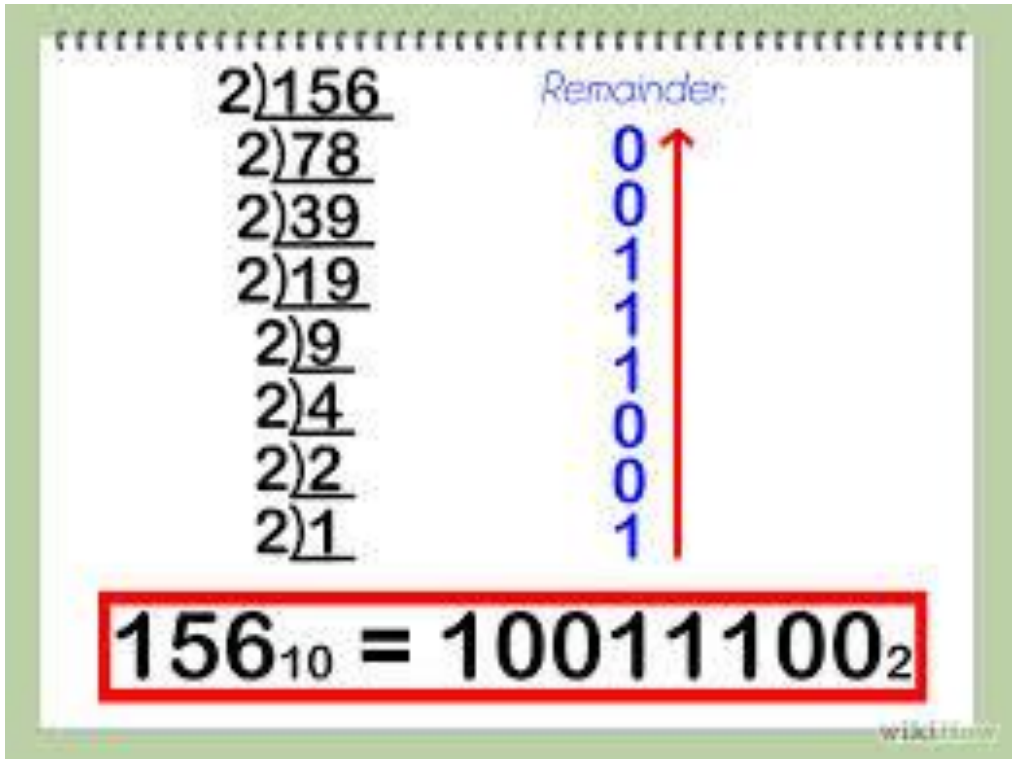
SECTION 6

Binary/Decimal Conversion



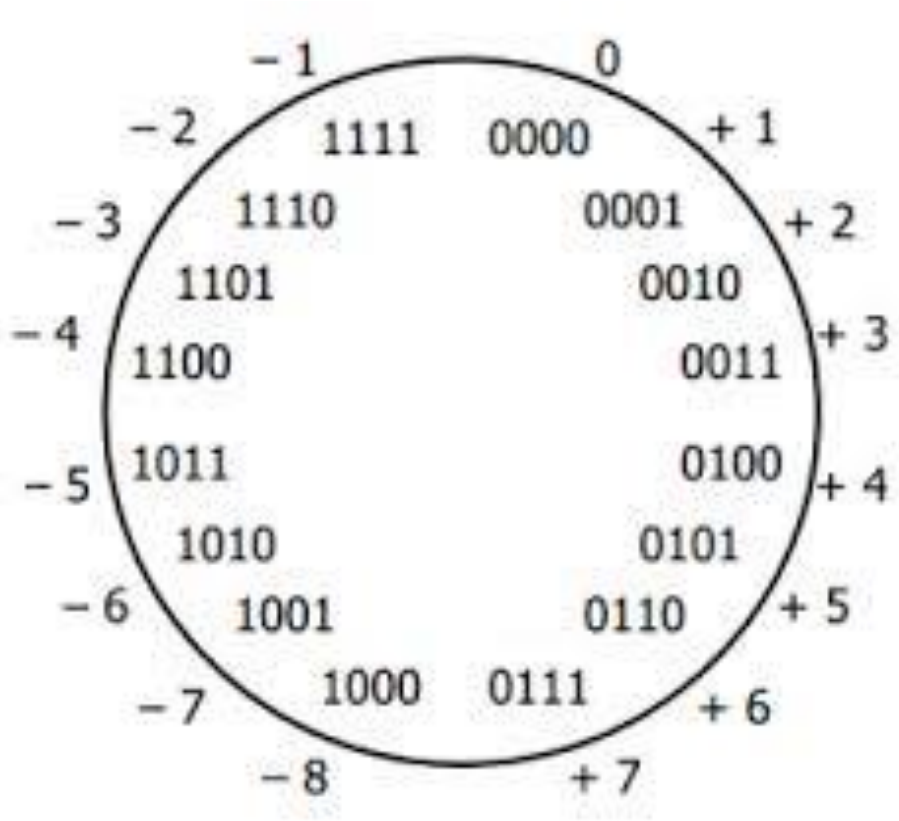


Decimal to Binary



Divider	Dividend	Remainder
2	202	0
2	101	1
2	50	0
2	25	1
2	12	0
2	6	0
2	3	1
		1

Two's Complement



Negative number is represented as two's complement.

For byte number's (8 bits):

$-X = (2^8 - 1) - X + 1;$

$X + (-X) = X + (2^8 - X) = 2^8 = 0\ ;$

eg.

A = 0100 -> A's One's Complement = 1011 ->

A's Two's Complement -> 1100

The number 2^8 is a overflow for the byte format, because unsigned byte number range

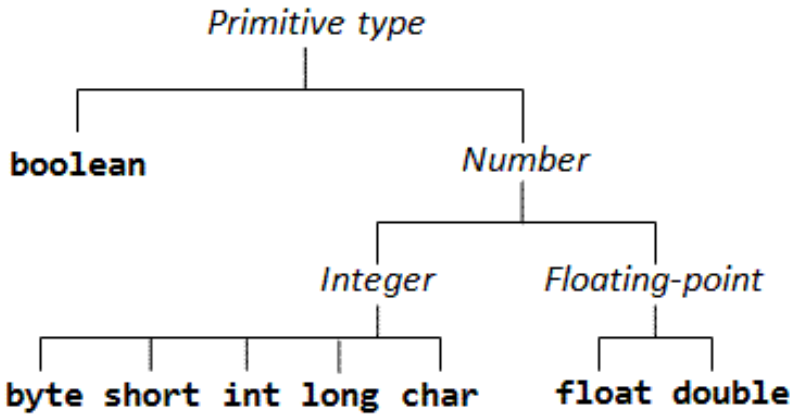
from 0 to $2^8 - 1$ = 11111111.

Therefore, this method can work for computer.

Finding 2's Complement

	-128	64	32	16	8	4	2	1	
X	0	1	0	0	1	1	1	1	**Number : 79 decimal**
$(2^8-1)-X$	1	0	1	1	0	0	0	0	**Flip the bits**
	0	0	0	0	0	0	0	1	**Add 1**
$NegX = (2^8-1)-X+1$	1	0	1	1	0	0	0	1	**Number: -79 in 2's Complement format**

Java's special number rules
(different from other languages)



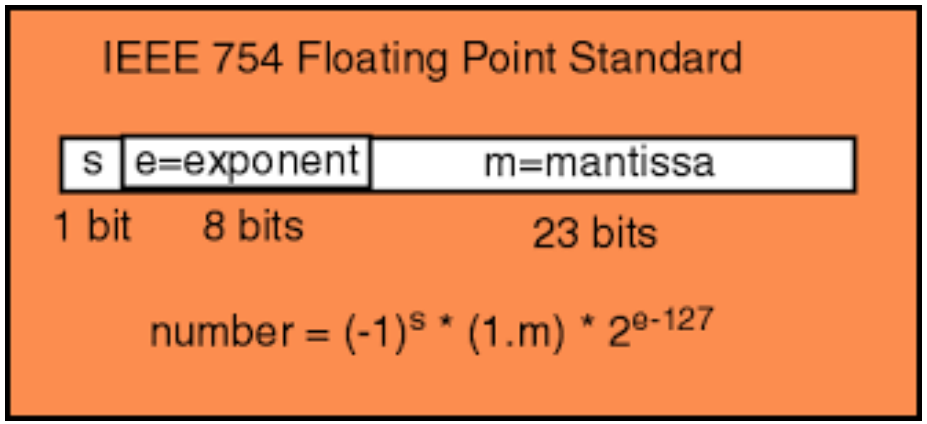
Java doesn't have unsigned number primitives.

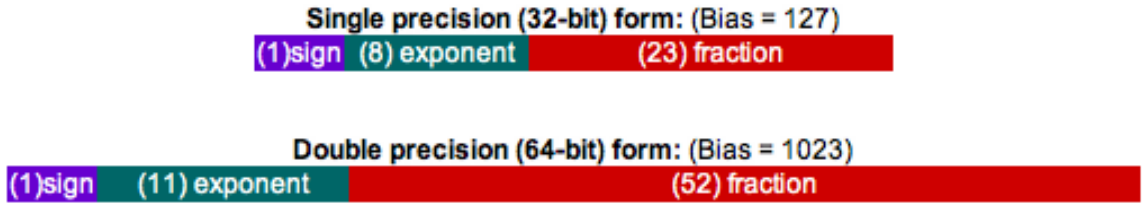
unsigned number is seldom used.

If you need to use unsigned number, use **char** data type instead. Because char does not follow the number operation rules while **char** can still operate the **bit-wise** operations.

Java's char is 16 bit. (supporting Unicode: UTF-16)

IEEE 754 binary floating point representation. (Java's Float Standard)





Binary Addition

Binary Arithmetic Rules

0	1	0	1
+0	+0	+1	$+_1 1$
0	1	1	0
0	1	0	1
$+\ 0_1$	$+_1 0_1$	$+_1 1_1$	$+_1 1_1$
1	0	0	1

Key:
Carry Out
Carry in

C		101111000
X	190	10111110
Y	+ 141	+ 10001101
$X + Y$	331	101001011

C		001011000
X	173	10101101
Y	+ 44	+ 00101100
$X + Y$	217	11011001

C		011111110
X	127	01111111
Y	+ 63	+ 00111111
$X + Y$	190	10111110

C		000000000
X	170	10101010
Y	+ 85	+ 01010101
$X + Y$	255	11111111

Subtraction (A-B) = (A + -B)

using Two's compliment addition for subtraction

- 12_{ten} - 5_{ten} (using Java int data type example)

$$
\begin{array}{ll}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1100 \quad (12_{ten}) \\
- & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0101 \quad (\ 5_{ten}) \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111 \quad (\ 7_{ten})
\end{array}
$$

- 12_{ten} - 5_{ten} = 12_{ten} + (- 5_{ten})

$$
\begin{array}{ll}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1100 \quad (12_{ten}) \\
+ & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011 \quad (-5_{ten}) \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111 \quad (\ 7_{ten})
\end{array}
$$

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

ISA Related Programming Language and Compiler Design Issues

- Instruction Choice (Code Optimization and Pipelining)
- Find the proper processing unit
- Memory Address Mode to reduce memory access latency
- Big endian and little endian conversion and the right number format (2's complement and etc).
- Data Storage format (stack, heap, data pool)
- Integer and Floating Point arithmetic (check the FPU, IPU and ALU available on the machine)
- Loop unrolling, superscalar, VLIW, pipelining, super-pipelining and code reordering optimization to exploit ILP.
- Languages which require more programmer's work on defining data types tend to be faster. Languages which has interpreter or compiler to decide the data types tend to be slower.

Instruction Set Architecture I

ISA

SECTION 7

Instruction-Set Architecture

- The set of instructions executed by a modern processor may include:
  - data movement (load, store, push, pop, movem, swap - registers)
  - arithmetic and logical (negate, extend, add, subtract, multiply, divide, and, or, shift)
  - control transfer (jump, call, trap - jump into the operating system, return - from call or trap, conditional branch)

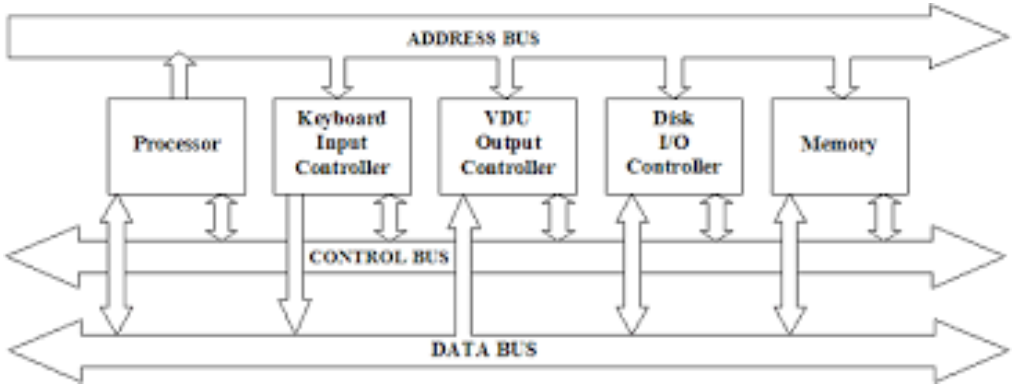
Three Instruction Types

Logic, Switch and Control (Gate Types)

Computation — arithmetic and logical operations, tests, and comparisons on values held in registers (and possibly in memory)

Data movement — loads from memory to registers, stores from registers to memory, copies from one register (or memory location) to another

Control flow — conditional and unconditional branches (gotos), subroutine calls and returns, traps into the operating system



Instruction-Set Architecture

Addressing Modes

- Instructions can specify many different ways to obtain their data (Addressing Modes)
  - data in instruction
  - data in register
  - address of data in instruction
  - address of data in register
  - address of data computed from two or more values contained in the instruction and/or registers

Instruction-Set Architecture

Addressing Modes

- On a RISC machine, arithmetic/logic instructions use only the first two of these ADDRESSING MODES
  - load and store instructions use the others
- On a CISC machine, all addressing modes are generally available to all instructions
  - CISC machines typically have a richer set of addressing modes, including some that perform
    - multiple indirections, and/or
    - arithmetic computations on values in memory in order to calculate an effective address

As	Ad	Addressing Mode	Syntax	Description
00	0	Register Mode	Rn	Register contents are operand
01	1	Indexed Mode	X(Rn)	(Rn + X) points to the operand. X is stored in the next word
01	1	Symbolic Mode	ADDR	(PC + X) points to the operand. X is stored in the next word. Indexed Mode X(PC) is used
01	1	Absolute Mode	&ADDR	The word following the instruction contains the absolute address.
10	-	Indirect Register Mode	@Rn	Rn is used as a pointer to the operand
11	-	Indirect Autoincrement	@Rn+	Rn is used as a pointer to the operand. Rn is incremented afterwards
11	-	Immediate Mode	#N	The word following the instruction contains the immediate constant N. Indirect Autoincrement Mode @PC+ is used

Arrays

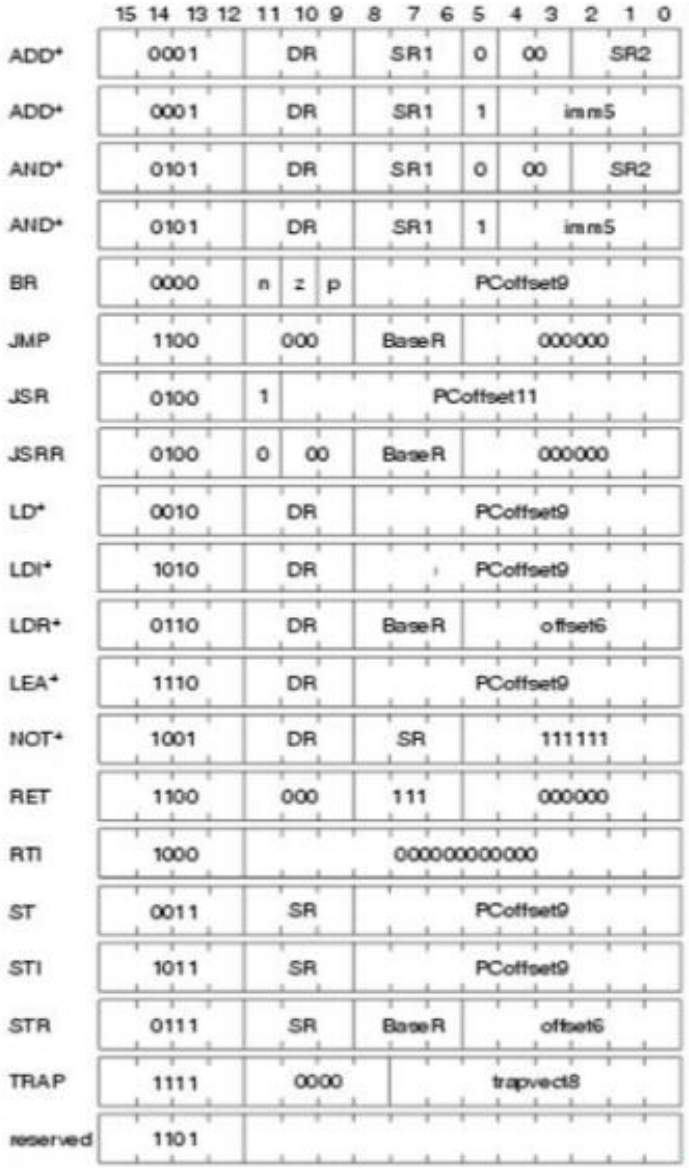
Relative to The Program Counter (Functional Calls)

Pointer

Pointer Objects/Data Fields

Constants & Literals

Addressing Modes



	15 14 13 12	11 10 9	8 7 6	5	4 3	2 1 0
ADD*	0001	DR	SR1	0	00	SR2
ADD*	0001	DR	SR1	1	imm5	
AND*	0101	DR	SR1	0	00	SR2
AND*	0101	DR	SR1	1	imm5	
BR	0000	n z p	PCoffset9			
JMP	1100	000	BaseR	000000		
JSR	0100	1	PCoffset11			
JSRR	0100	0 00	BaseR	000000		
LD*	0010	DR	PCoffset9			
LDI*	1010	DR	PCoffset9			
LDR*	0110	DR	BaseR	offset6		
LEA*	1110	DR	PCoffset9			
NOT*	1001	DR	SR	111111		
RET	1100	000	111	000000		
RTI	1000	000000000000				
ST	0011	SR	PCoffset9			
STI	1011	SR	PCoffset9			
STR	0111	SR	BaseR	offset6		
TRAP	1111	0000	trapvect8			
reserved	1101					

- **Register**

  (Operand is in one of the 8 registers)

- **PC-relative**

  (Operand is "offset" from where the PC points

  - offsets are sign extended to 16 bits)

- **Base + Offset (Base relative)**

  (Operand is "offset" from the contents of a register)

- **Immediate**

  (Operand is *in* the instruction)

- **Indirect**

  (The "Operand" ***points*** to the real address of Operand

  - rather than being the operand)

Functional Group	Example Mnemonics
Move Instructions	MOV
Math Instructions	MUL DIV ADD SUB
Logic Instructions	AND IOR XOR NEG
Rotate/Shift Instructions	ASR LSR SL
Bit Instructions	BSET BCLR BTG BTST
Compare/Skip/Branch	BTSC BTSS CPBEQ CPBGT
Flow Control Instructions	BRA CALL RCALL REPEAT
Shadow/Stack Instructions	LNK POP PUSH ULNK
Control Instructions	NOP CLRWDT PWRSAV RESET
DSP Instructions	MAC LAC SAC SFTAC

Conditions and Branches

All instruction sets provide a branching mechanism to update the program counter under program control. Branches allow compilers to implement conditional statements, subroutines, and loops. Conditional branches may be controlled in several ways.

A := B + C
if A = 0 then body

x86

```
    movl    C, %eax     ; move long-word C into register eax
    addl    B, %eax     ; add
    movl    %eax, A     ; and store
    jne     L1          ; branch (jump) if result not equal to zero
     body
L1:
```

The first three instructions all set the condition codes. The fourth (jne) tests the codes in the wake of the movl that stores to A. It branches if the codes indicate that the value was not zero.

Conditions and Branches

For cases in which the outcome of a branch depends on a value that has just been computed or moved, most machines provide compare and test instructions. Again on the x86:

if A≤B then body

x86

```
    movl    A, %eax     ; move long-word A into register eax
    cmpl    B, %eax     ; compare to B
    jg      L1          ; branch (jump) if greater
     body
L1:
```

if A > 0 then body

x86

```
    testl   %eax, %eax  ; compare %eax (A) to 0
    jle     L2          ; branch if less than or equal
     body
L2:
```

ISA will make a difference in performance

On the x86, the code fragment C := max(A, B) might naively be translated

```
    movl    A, %ecx
    movl    B, %edx
    cmpl    %edx, %ecx  ; compare %edx (A) to %ecx (B)
    jle     L1          ; branch if less than or equal
    movl    %ecx, C     ; store A to C
    jmp     L2
L1:
    movl    %edx, C     ; store B to C
L2:
```

With a conditional move instruction it can become the following instead:

```
    movl    B, %ecx
    movl    A, %edx
    cmpl    %ecx, %edx  ; compare %edx (A) to %ecx (B)
    cmovgl  %edx, %ecx  ; move A into %ecx if greater
    movl    %ecx, C     ; store to C
```

A few ISAs, including 32-bit ARM and IA-64 (Itanium), allow almost any instruction to be marked as conditional. This more general mechanism is known as **predication**.

Instruction Set Architecture II

Architecture Implementation

SECTION 8

Architecture Implementation

- As technology advances, there are occasionally times when some threshold is crossed that suddenly makes it possible to design machines in a very different way
  - One example of such a *paradigm shift* occurred in the mid 1980s with the development of RISC (reduced instruction set computer) architectures

Architecture Implementation

- During the 1950s and the early 1960s, the instruction set of a typical computer was implemented by soldering together large numbers of discrete components that performed the required operations
  - To build a faster computer, one generally designed extra, more powerful instructions, which required extra hardware
  - This has the unfortunate effect of requiring assembly language programmers to learn a new language

Architecture Implementation

- IBM hit upon an implementation technique called MICROPROGRAMMING:
  - *same* instruction set across a whole line of computers, from cheap to fast machines
  - basic idea of microprogramming
    - build a *micro engine* in hardware that executed a interpreter program *in firmware*
      - interpreter implemented IBM 360 instruction set
    - more expensive machines had fancier micro engines
      - more of the 360 functionality in hardware
      - top-of-the-line machines had everything in hardware

Architecture Implementation

- Microprogramming makes it easy to extend the instruction set
  - people ran studies to identify instructions that often occurred in sequence (e.g., the sequence that jumps to a subroutine and updates bookkeeping information in the stack)
  - then provided new instructions that performed the function of the sequence
  - By clever programming in the firmware, it was generally possible to make the new instruction faster than the old sequence, and programs got faster.

Architecture Implementation

- The microcomputer revolution of the late 1970s (another *paradigm shift*) occurred when it became possible to fit a micro engine onto a single chip (microprocessor):
  - personal computers were born
  - by the mid 1980s, VLSI technology reached the point where it was possible to eliminate the micro engine and still implement a processor on a single chip

Architecture Implementation

- With a hardware-only processor on one chip, it then became possible to apply certain performance-enhancing tricks to the implementation, but only if the instruction set was very simple and predictable
  - This was the RISC revolution
  - Its philosophy was to give up "nice", fancy features in order to make common operations fast

Architecture Implementation

- RISC machines:
  - a common misconception is that small instruction sets are the distinguishing characteristic of a RISC machine
  - better characterization: RISC machines are machines in which at least one new instruction can (in the absence of conflicts) be started every cycle (hardware clock tick)
    - all possible mechanisms have been exploited to minimize the duration of a cycle, and to maximize the number of functional units that operate in parallel during a given cycle

Architecture Implementation

- Reduced cycle time comes from making all instructions simple and regular
  - Simple instructions never have to run slowly because of extra logic necessary to implement the complicated instructions
  - Maximal parallelism comes from giving the instructions a very regular, predictable format
    - the interactions between instructions are clear and the processor can begin working on the next instruction before the previous one has finished

Compiling to modern processors I

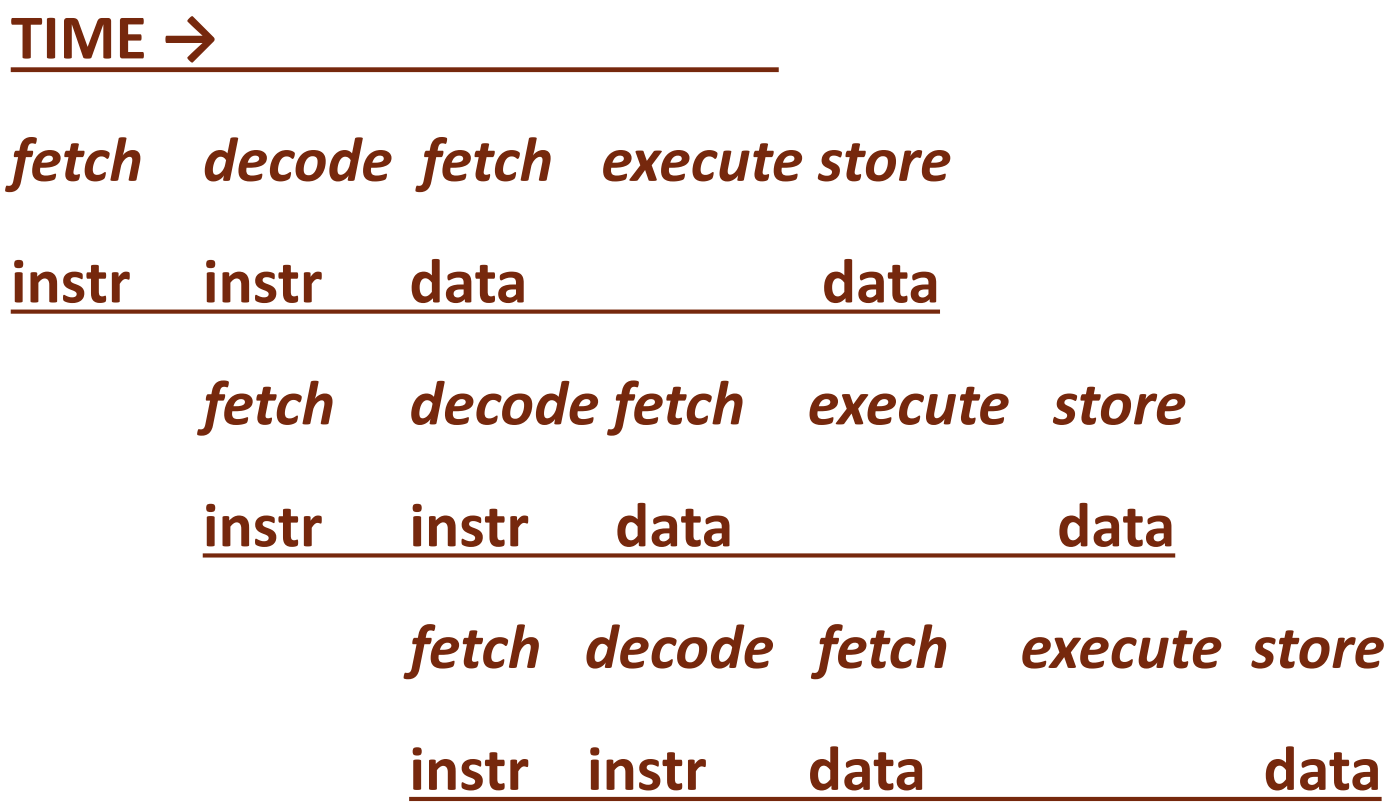
Pipelining

SECTION 9

Compiling for Modern Processors

PIPELINING is probably the most important performance enhancing trick

- It works kind of like this:



Compiling for Modern Processors

- The processor has to be careful not to execute an instruction that depends on a *previous* instruction that hasn't finished yet
  - The compiler can improve the performance of the processor by generating code in which the number of dependencies that would *stall* the pipeline is minimized
  - This is called INSTRUCTION SCHEDULING; it's one of the most important machine-specific optimizations for modern compilers

Compiling for Modern Processors

- Loads and load delays are influenced by
  - Dependences
    - Flow dependence (Control Dependency)
    - Anti-dependence
    - Output dependence (Data Dependency)
- Branches
  - since control can go both ways, branches create delays

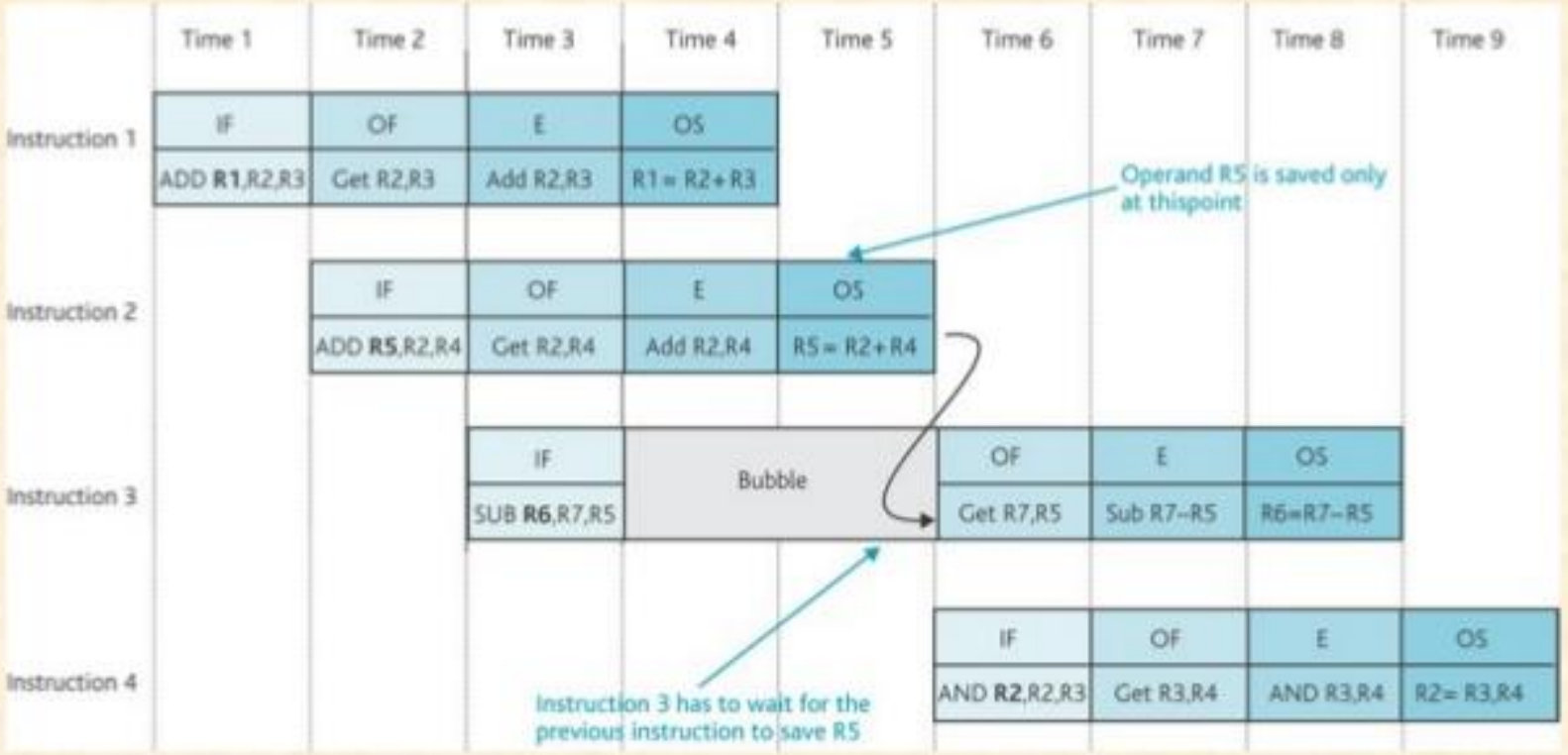
Dependency

- **Flow dependence (also called true dependence)**
  - **S1:** a = c*10
  - **S2:** d = 2*a + c

- **Anti-dependence**
  - **S1:** e = f*4 + g
  - **S2:** g = 2*h

S2 need to wait until S1 release the g variable.
This can be removed with out-of-order execution as long as S1 get the right input data.

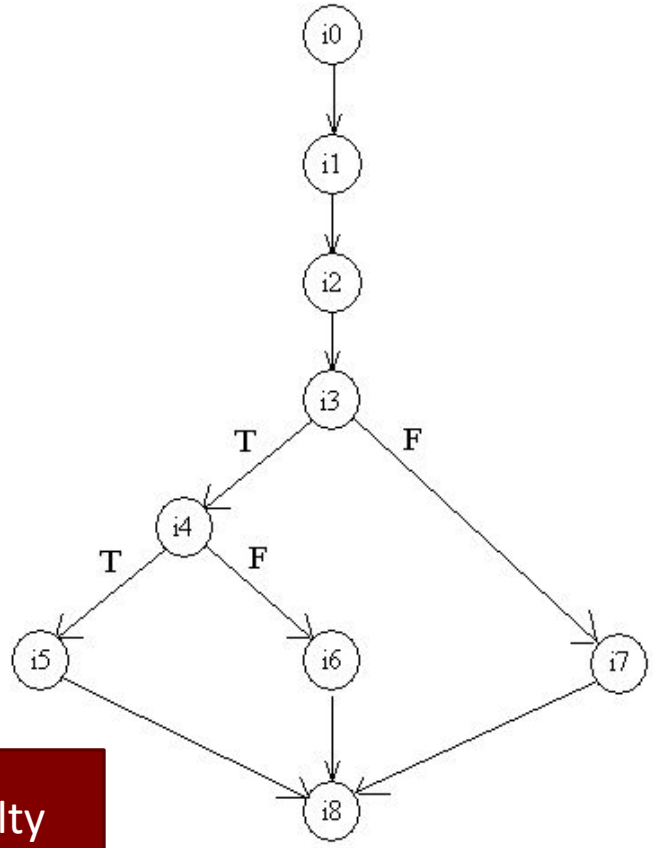
Data Dependency



- The pipeline is stalled after the fetch phase of instruction 3 for two clocked cycles.

Control Dependency Graph

```
i0:    r1 = op1;
i1:    r2 = op2;
i2:    r3 = op3;
i3:    if (r2 > r1)
i4:       { if (r3 > r1)
i5:            r4 = r3;
i6:         else r4 = r1}
i7:      else r4 = r2;
i8:           f5 = r4 * r4
```



Use Branch Prediction to Minimize the Penalty

Compiling for Modern Processors

Usual goal: *minimize pipeline stalls*

Delay slots

- loads and branches take longer than *ordinary* instructions
- loads have to go to memory, which is slow
- branches disrupt the pipeline
- processor have interlock hardware
- early RISC machines often provided *delay slots* for the second (maybe third) cycle of a load or store instruction, during which something else can occur
- the instruction in a branch delay slot gets executed whether the branch occurs or not

Compiling for Modern Processors

Delay slots (continued)

- the instruction in a load delay slot can't use the loaded value
- as pipelines have grown deeper, people have generally realized that delay slots are more trouble than they're worth
- most current processor implementations interlock all loads, so you don't have to worry about the correctness issues of load delay
- some machines still have branch delay slots (so they can run code written in the late '80s)
  - later implementations usually provide a *nullifying* alternative that skips the instruction in the slot if static branch prediction is wrong

Compiling to modern processors II

RISC VS CISC

SECTION 10

Compilation to CPU

COMPILATION 1

Compiling for Modern Processors

- Unfortunately, even this *start a new instruction every cycle* characterization of RISC machines is inadequate
  - In all honesty, there is no good clear definition of what RISC means
- Most recent RISC machines (and also the most recent x86 machines) are so-called SUPERSCALAR implementations that can start *more* than one instruction each cycle

Compiling for Modern Processors

- If it's a CISC machine, the number of instructions per second depends crucially on the mix of instructions produced by the compiler
  - the MHz number gives an upper bound (again assuming a single set of functional units)
  - if it's a "multi-issue“ (superscalar) processor like the PowerPC G3 or Intel machines since the Pentium Pro, the upper bound is higher than the MHz number

Compiling for Modern Processors

- As technology improves, complexity is beginning to creep back into RISC designs
- Right now we see "RISC" machines with on-chip
  - vector units
  - memory management units
  - large caches

Compiling for Modern Processors

- We also see "CISC" machines (the Pentium family) with RISC-like subsets (single-cycle hard-coded instructions)
- In the future, we might see
  - large amounts of main memory
  - multiple processors
  - network interfaces (now in prototypes)
  - additional functions
    - digital signal processing

Compiling for Modern Processors

- In addition, the 80x86 instruction set will be with us for a long time, due to the huge installed base of IBM-compatible PCs
  - After a failed attempt to introduce its own RISC architecture (the i860), Intel worked with HP on the RISC-like IA64, architecture
  - Currently x86-64 (AMD's design) is the 64-bit extension of the x86 architecture that dominates the market

Compiling for Modern Processors

- In a sense, code for RISC machines resembles microcode
  - Complexity that used to be hidden in firmware must now be embedded in the compiler
  - Some of the worst of the complexity (e.g. branch delay slots) can be hidden by the assembler (as it is on MIPS machines)
    - it is definitely true that it is harder to produce good (fast) code for RISC machines than it is for CISC machines

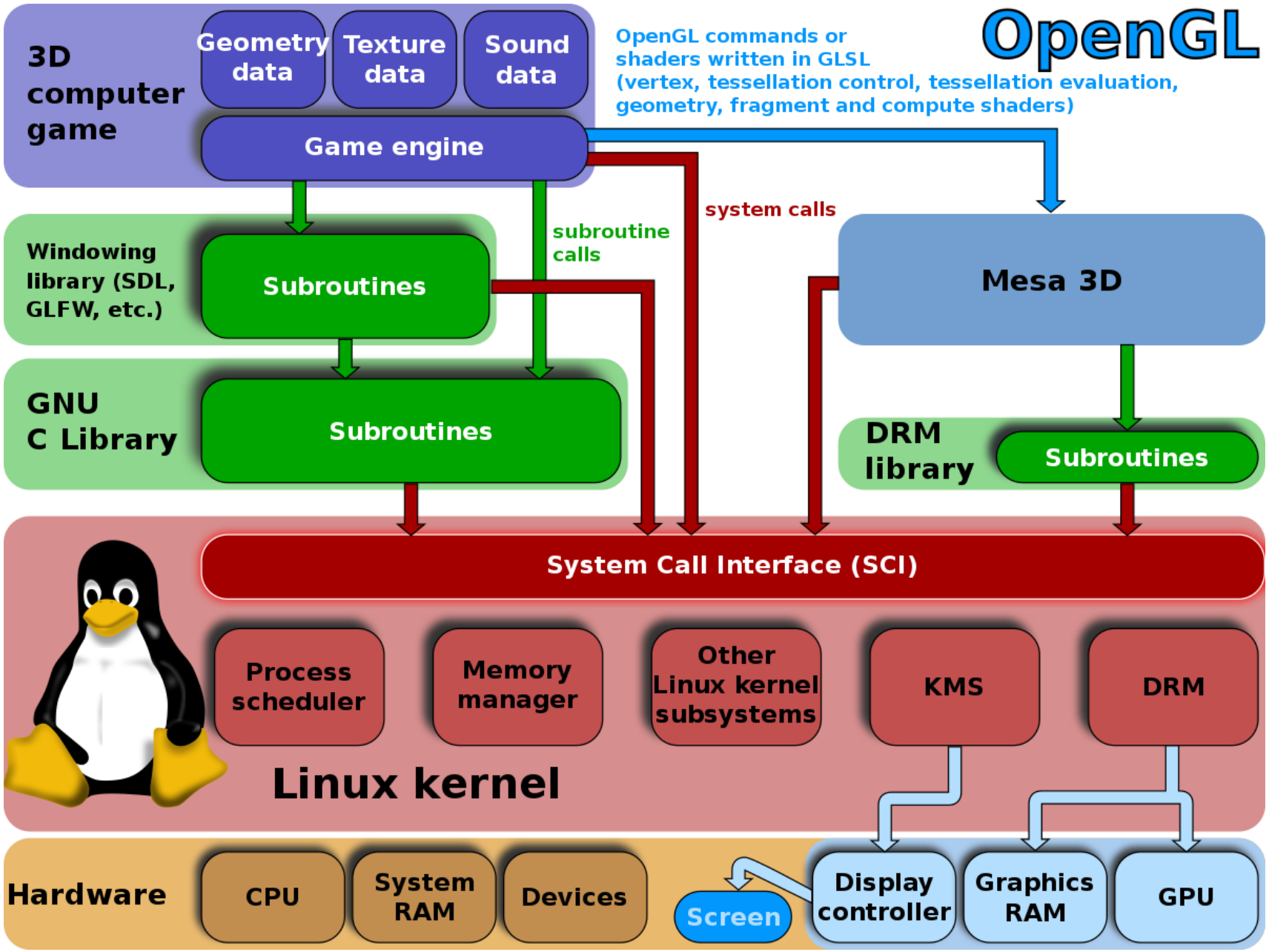
Compiling for Modern Processors

Example: the Pentium chip runs a little bit faster than a 486 if you use the same old binaries

- If you recompile with a compiler that knows to use a RISC-like subset of the instruction set, with appropriate instruction scheduling, the Pentium can run *much* faster than a 486

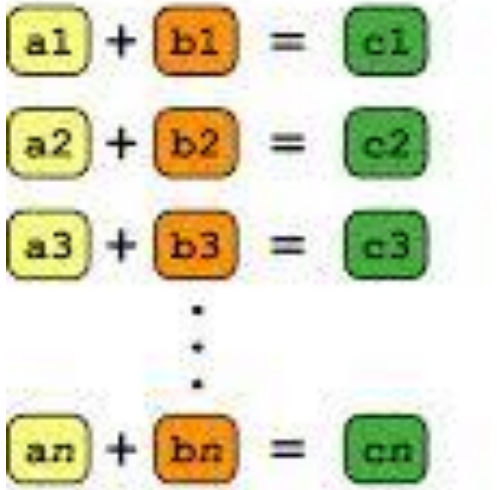
Compilation to GPU

COMPILATION 2


3D computer game
Geometry data
Texture data
Sound data
Game engine
OpenGL commands or shaders written in GLSL (vertex, tessellation control, tessellation evaluation, geometry, fragment and compute shaders)
OpenGL
system calls
subroutine calls
Windowing library (SDL, GLFW, etc.)
Subroutines
Mesa 3D
GNU C Library
Subroutines
DRM library
Subroutines
System Call Interface (SCI)
Process scheduler
Memory manager
Other Linux kernel subsystems
KMS
DRM
Linux kernel
Hardware
CPU
System RAM
Devices
Screen
Display controller
Graphics RAM
GPU

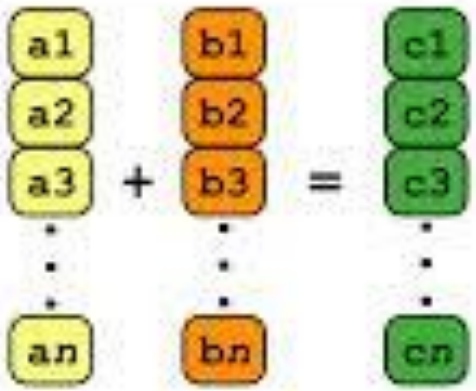
Vector Processing Unit

Scalar Processing

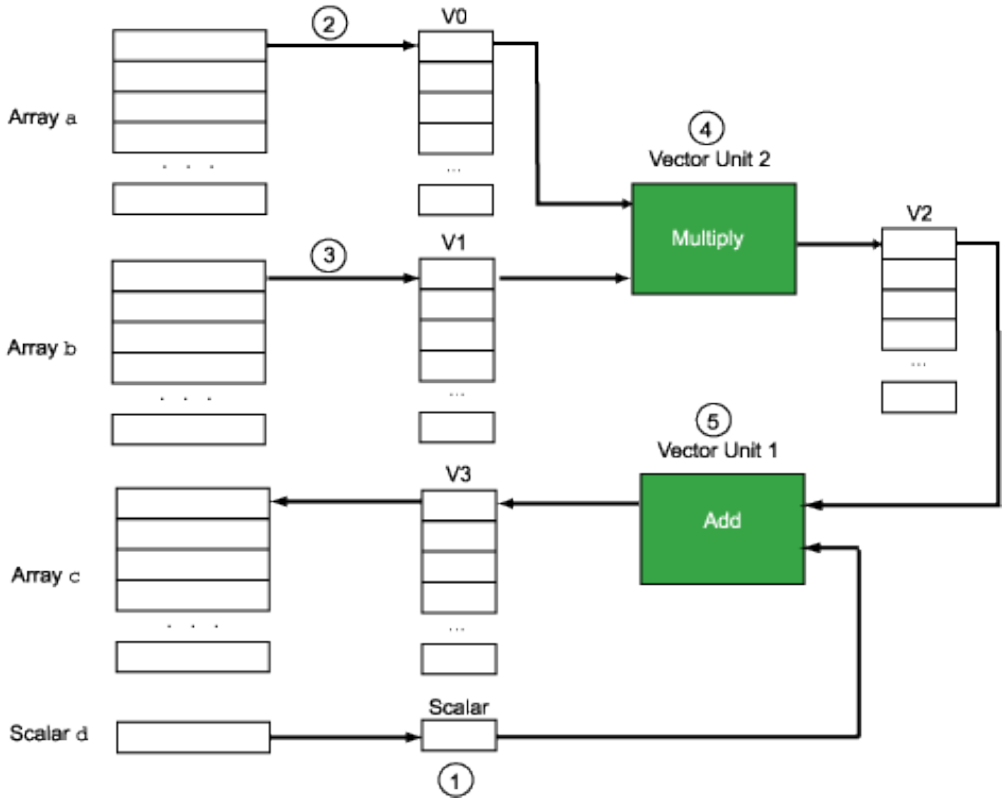



```
for i = 1 to n
    c[i] = a[i] + b[i]
end
```

Vector Processing

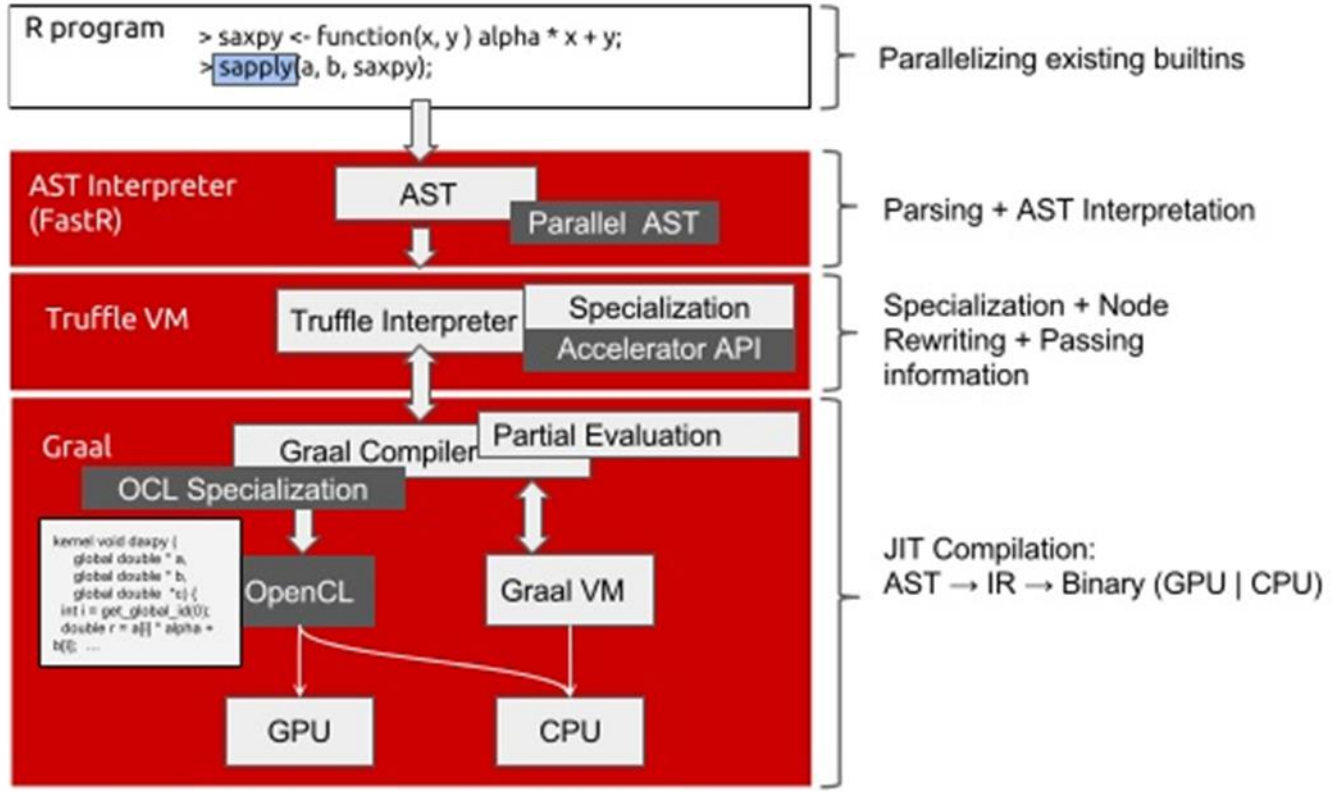



```
c[1:n] = a[1:n] + b[1:n]
```



Enabling OpenCL JIT Compilation from the AST Interpreter

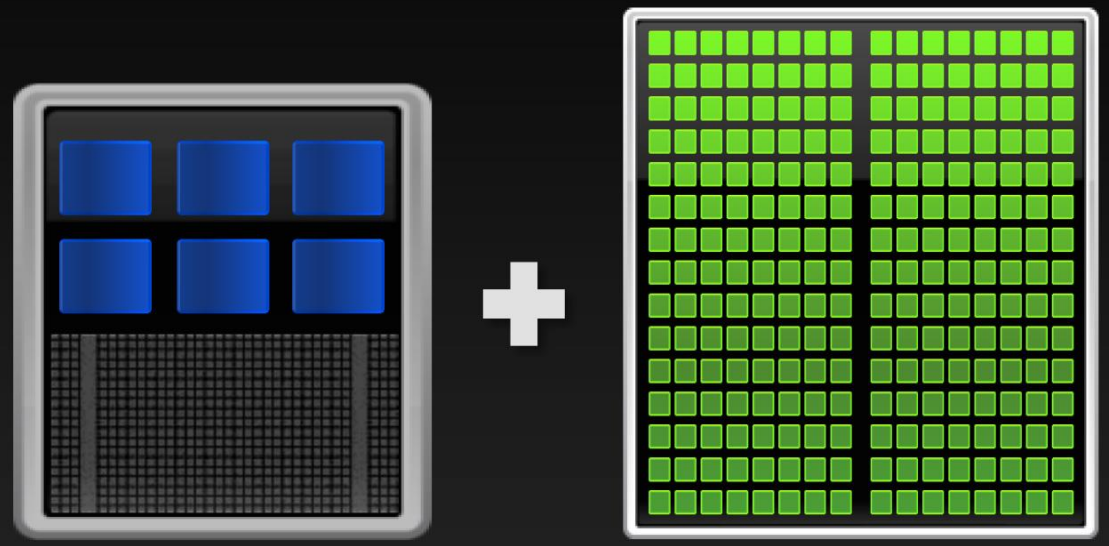
Compiler approach, middle-ware stack



Announcing CUDA 5.5: Full Support for ARM

ARM or x86 CPU
Optimized for Few Serial Tasks

GPU Accelerator
Optimized for Many Parallel Tasks



CUDA 5.5 Highlights

Support for ARM Platforms

- Native compilation on ARM

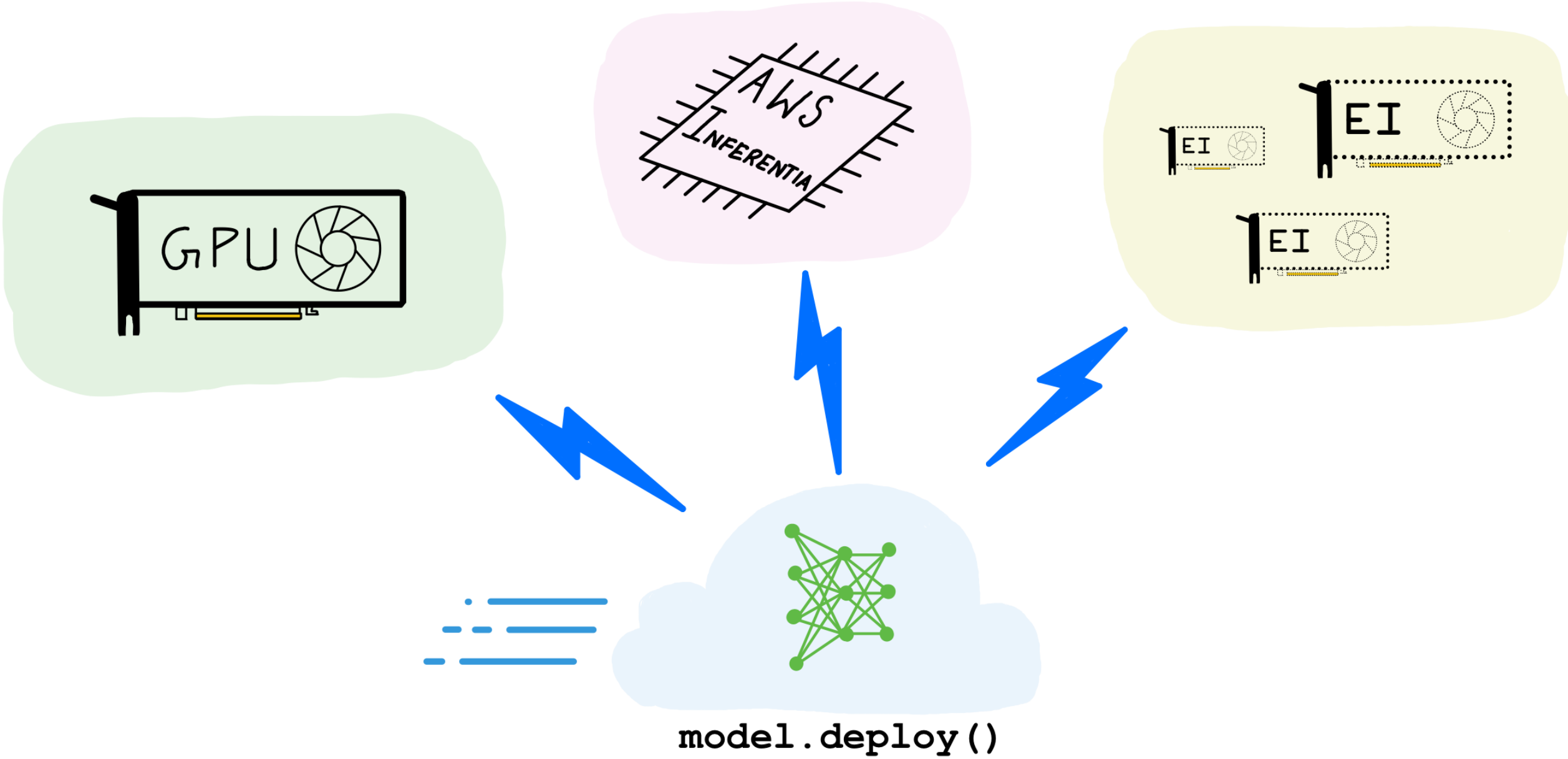
Optimized for MPI

- Faster Hyper-Q for all Linux distros
- MPI workload prioritization

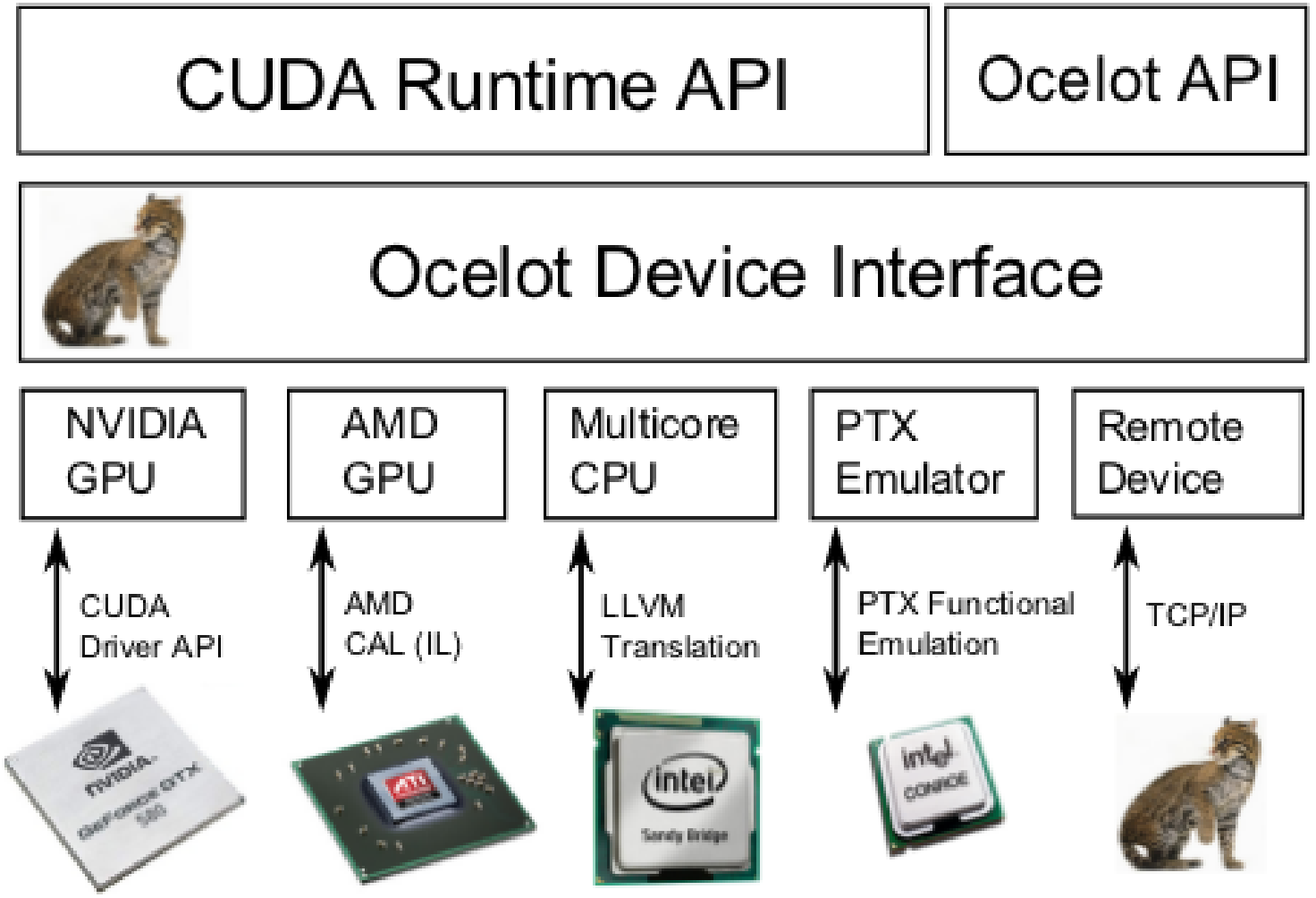
Guided Performance Analysis

- Step-by-step optimization

Available Now
www.nvidia.com/getcuda

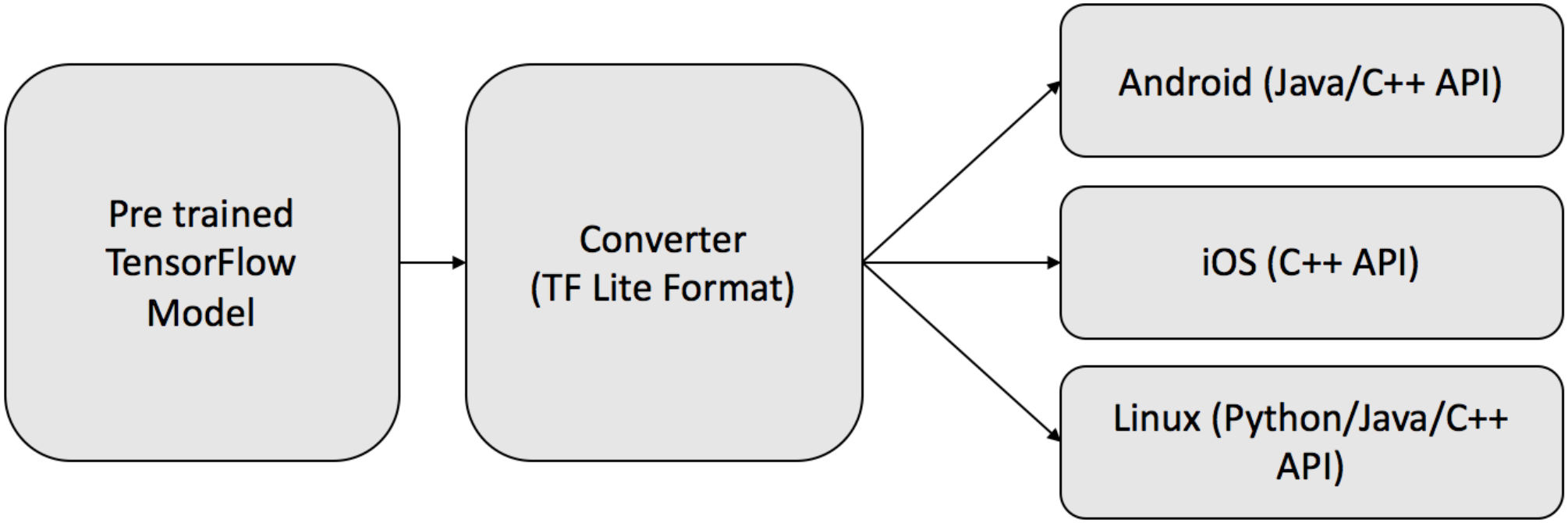

GPU
AWS
INFERENTIA
EI
EI
EI
model.deploy()

CUDA Application



Compilation to GPU

COMPILATION 1


Pre trained TensorFlow Model
Converter (TF Lite Format)
Android (Java/C++ API)
iOS (C++ API)
Linux (Python/Java/C++ API)

TensorFlow graph

```
def model():
  y = tf.add(x, 1)
  tf.reduce_sum(y)
```

XLA representation

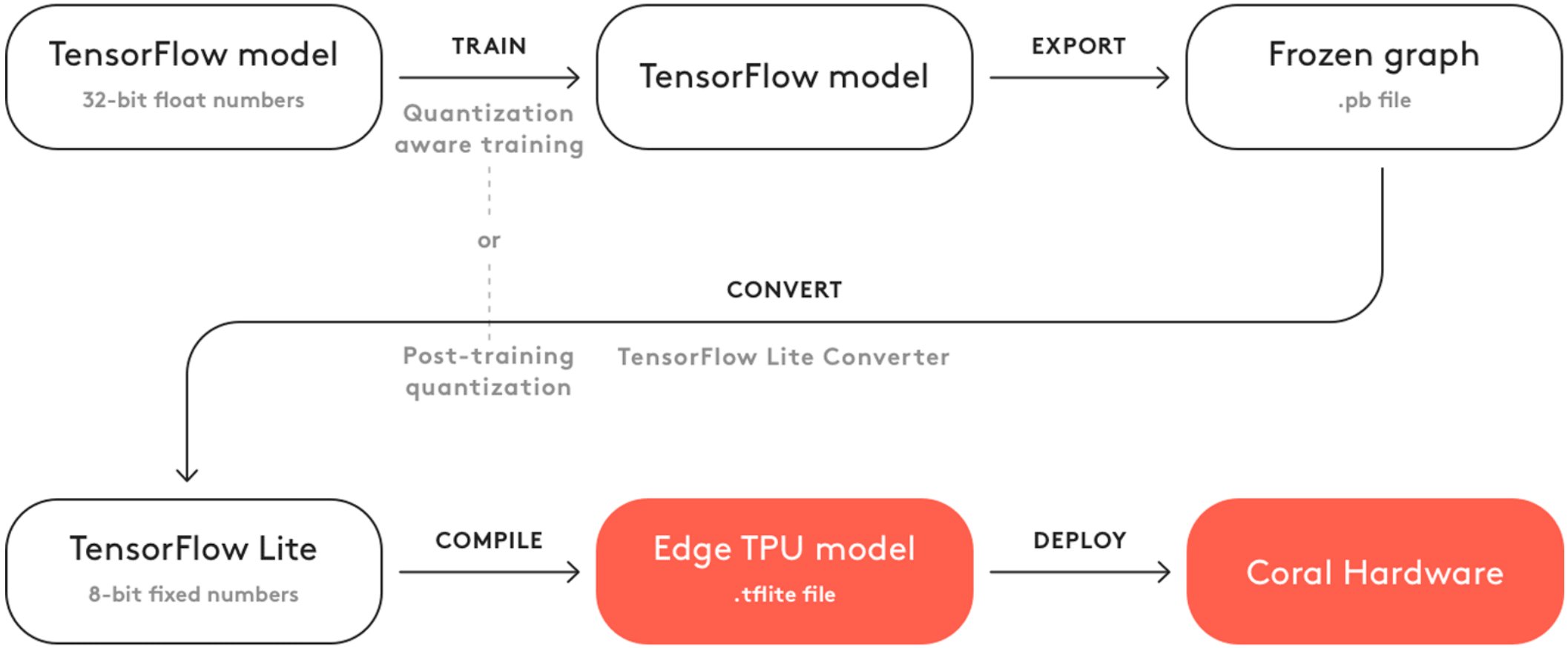
```
broadcast.4.133 = f32[100,25]{1,0} broadcas
transpose.4.129 = f32[100,32]{1,0} transpos
dot.4.130 = f32[100,25]{1,0} dot(transpose.
cross-replica-sum.4.131 = f32[100,25]{1,0}
multiply.4.134 = f32[100,25]{1,0} multiply(
subtract.4.135 = f32[100,25]{1,0} subtract(
reshape.4.136 = f32[2500]{0} reshape(subtra
```

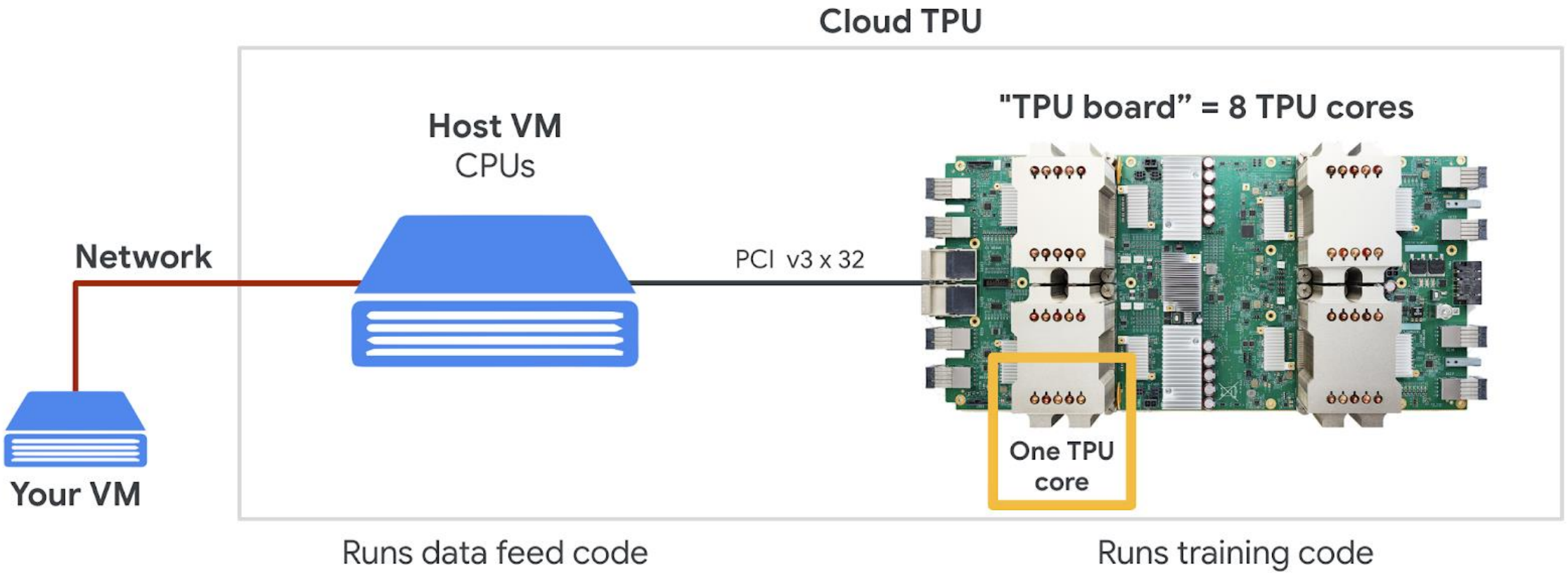
TPU machine code

```
89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52 00 00
03 C0 00 00 02 1C 08 02 00 00 00 B6 3F 50 2C 00 00 80
00 49 44 41 54 78 DA EC 9D 67 58 1C 57 96 F7 E7 D9 F7
C3 EE B3 FB 65 BF EC F3 EC 6C 98 D9 19 D9 33 4E B3 63
8F 67 76 3C 1E DB 33 63 7B 6C 2B 21 14 10 12 28 E7 2C
A1 2C 2B 23 10 20 24 91 33 88 20 40 12 39 67 10 49 E4
8C 40 84 6E 68 72 EE 6E A0 C9 B2 F4 FE AB 8F 54 6E 01
42 28 80 00 9D E3 A3 F2 A5 EA D6 AD EA AA 1B 7E F7 D6
B9 E7 FE A4 FA C2 2F 59 59 59 59 59 59 59 59 59 59 59
```

TF to XLA bridge

XLA compiler


TensorFlow model
32-bit float numbers
TRAIN
Quantization aware training
TensorFlow model
EXPORT
Frozen graph
.pb file
or
CONVERT
Post-training quantization
TensorFlow Lite Converter
TensorFlow Lite
8-bit fixed numbers
COMPILE
Edge TPU model
.tflite file
DEPLOY
Coral Hardware


Cloud TPU
"TPU board" = 8 TPU cores
Host VM
CPUs
Network
PCI v3 x 32
One TPU core
Your VM
Runs data feed code
Runs training code

TensorFlow Hello World

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

model.compile(optimizer='adam',loss=loss_fn,metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)

model.evaluate(x_test,  y_test, verbose=2)
```

Compiling to modern processors III

Processor Units

SECTION 11

Compiling for Modern Processors

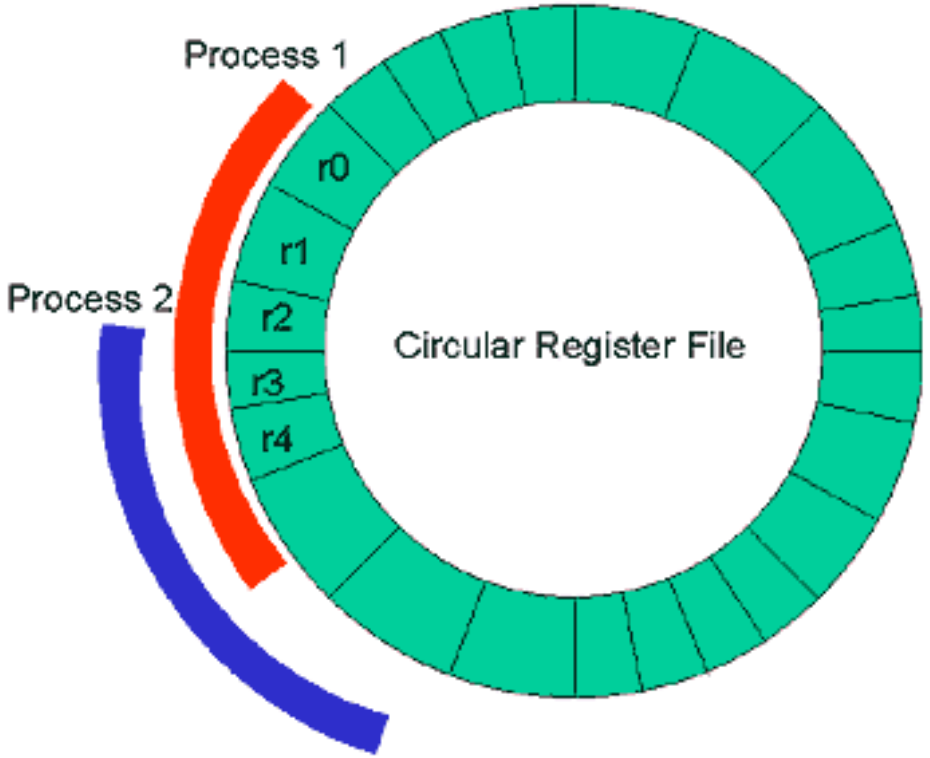
- Multiple functional units
  - superscalar machines can issue (start) more than one instruction per cycle, if those instructions don't need the same functional units
  - for example, there might be two instruction fetch units, two instruction decode units, an integer unit, a floating point adder, and a floating point multiplier

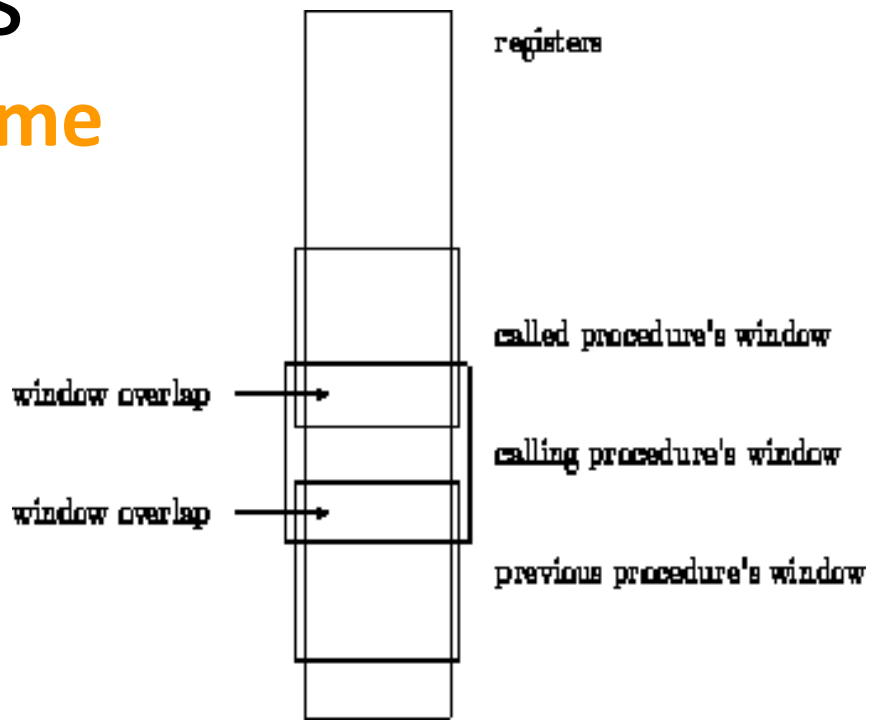
Compiling for Modern Processors

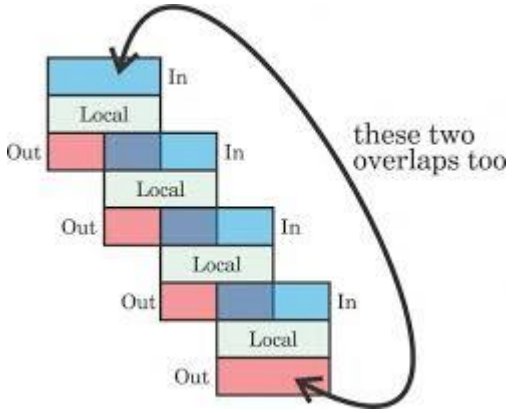
- Because memory is so much slower than registers, (several hundred times slower at present) keeping the *right* things in registers is extremely important
  - RISC machines often have at least two different classes of registers (so they don't have to support all operations on all registers) which the compiler has to keep track of

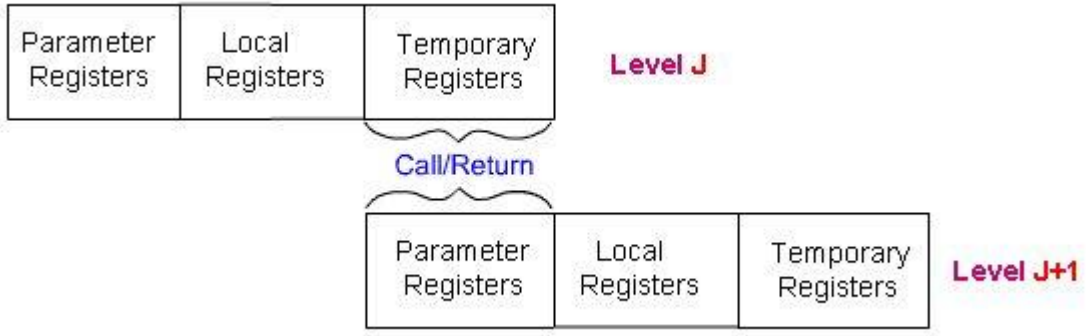
Register Windows

Save Memory Access Time









Compiling for Modern Processors

- Some (SPARC) have a complicated collection of overlapping REGISTER WINDOWS
- Finally, good register allocation sometimes conflicts with good instruction scheduling
  - code that makes ideal use of functional units may require more registers than code that makes poorer use of functional units
  - good compilers spend a *great* deal of effort
    - make sure that the data they need most is in register

Compiling for Modern Processors

- Note that instruction scheduling and register allocation often conflict
- Limited instruction formats/more primitive instructions
  - Many operations that are provided by a single instruction on a CISC machine take multiple instructions on a RISC machine
  - For example, some RISC machines don't provide a 32-bit multiply; you have to build it out of 4-bit (or whatever) multiplies

Compiling For Modern Machines

- To make all instructions the same length
  - data values and parts of addresses are often scaled and packed into odd pieces of the instruction
  - loading from a 32-bit address contained in the instruction stream takes two instructions, because one instruction isn't big enough to hold the whole address *and* the code for *load*
    - first instruction loads part of the address into a register
    - second instruction adds the rest of the address into the register and performs the load

Summary

- Some major RISC architectures:
  - ARM (Android, iPhone, Windows Phone, etc.)
  - MIPS (PlayStation, Nintendo, etc.)
  - SPARC (Oracle, Fujitsu)
  - Power/Power PC (IBM, Motorola. Apple)
  - RISC-V (open source 32-bit)
- Currently there is growing demand for 64-bit addressing (Intel, AMD)

Compiling to modern processors IV

Multithread and Multicore

SECTION 12

Advances in Processor Technology

- **Faster clocks.** Since smaller transistors can charge and discharge more quickly, higher-density chips can run at a higher clock rate. The Intel 8080 ran at 2MHz in 1974. Rates in excess of 2GHz (1000× faster) are commonplace today.
- **Instruction-level parallelism (ILP).** As noted in the previous subsection, modern processors employ pipelined, superscalar, and out-of-order execution to keep a very large number of instructions "in flight," and to execute those instructions as soon as their operands become available.

Advances in Processor Technology

- **Speculation.** To keep the pipeline full, a modern processor guesses which way control will go at every branch, and speculatively executes instructions along the predicted control path. Some processors employ additional forms of speculation as well: they may, for example, guess the value that will be returned by a read that misses in the cache. So long as guesses are right, the processor avoids "unnecessary" waiting. It must always check after the fact, however, and be prepared to undo any erroneous operations in the event that a guess was wrong.
- **Larger caches.** As noted in Sidebar C 5.2, caches play a critical role in coping with the processor-memory gap induced by higher clock rates. Higher VLSI density makes room for larger caches.

Technologies

- Pipelining
- Super-pipelining
- Superscalar
- Vector processing
- VLIW
- Instruction Translation to μcode
- Re-ordering of μcode (Reservation Station and Reordering of Registers)
- Renaming of Registers

Compiling to modern processors V

x86 Instruction Set (Optional)

SECTION 13

x86 Architecture

Processor Modes

A processor implementing the IA-32 architecture can execute in:

- **Protected Mode**: The normal mode of operation.
- **Real Mode**: A highly restricted operating mode in which only the first 1 MB of physical memory is directly accessible. Not discussed here.
- **System Management Mode (SMM)**: Not discussed here.

A processor implementing the Intel 64 architecture has the three above modes plus IA-32e mode, which has two sub modes:

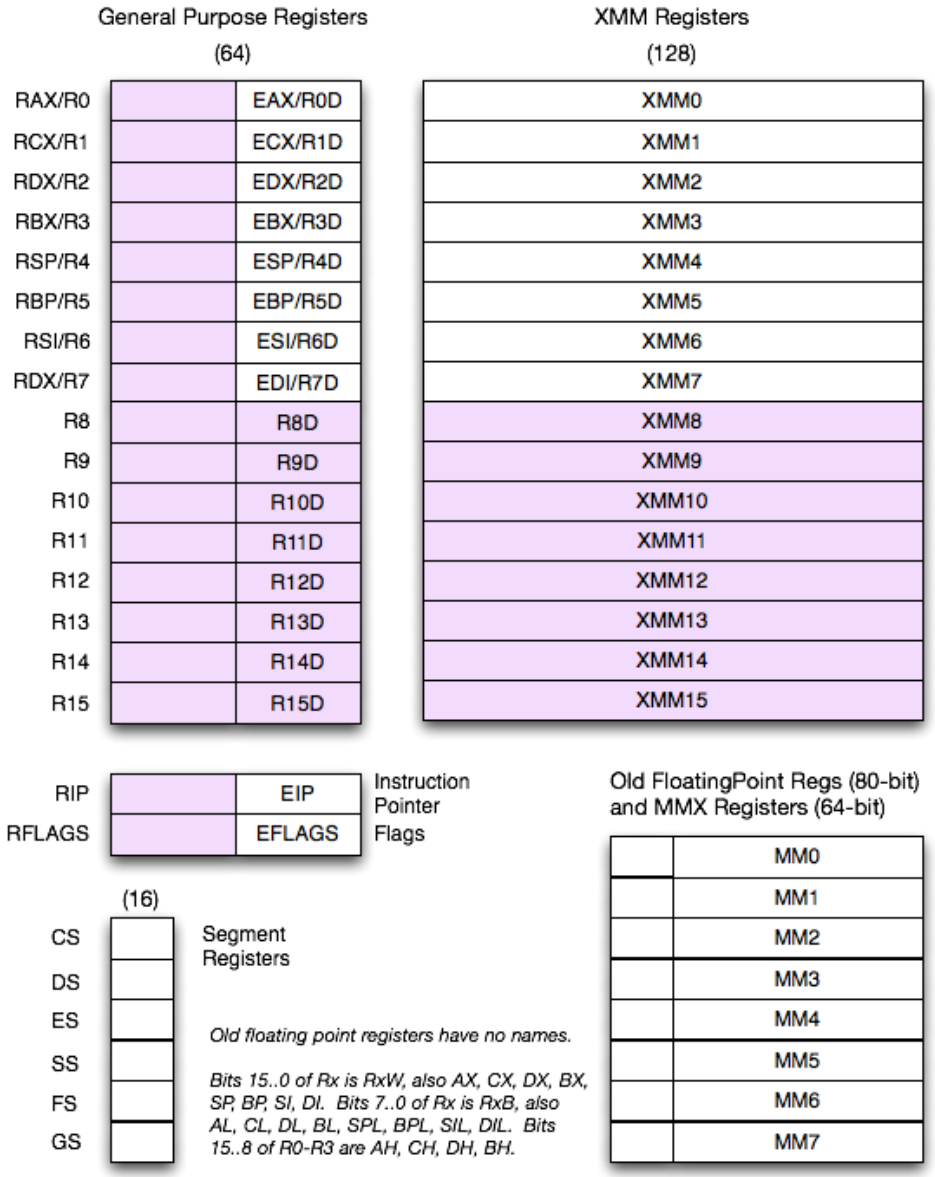
- **Compatibility Mode**: Lets "old binaries" run without being recompiled.
- **64-bit Mode**: Full 64-bit capabilities.

x86 Architecture

Register Set

Application programmers can remain oblivious of the rest of the registers:

- The 8 32-bit **processor control registers**: CR0, CR1, CR2, CR3, CR4, CR5, CR6, CR7. The lower 16 bits of CR0 is called the Machine Status Word (MSW).
- The 4 16-bit **table registers**: GDTR, IDTR, LDTR and TR.
- The 8 32-bit **debug registers**: DR0, DR1, DR2, DR3, DR4, DR5, DR6 and DR7.
- The 5 test registers: TR3, TR4, TR5, TR6 and TR7.
- The **memory type range registers**
- The **machine specific registers**
- The **machine check registers**



x86 Architecture

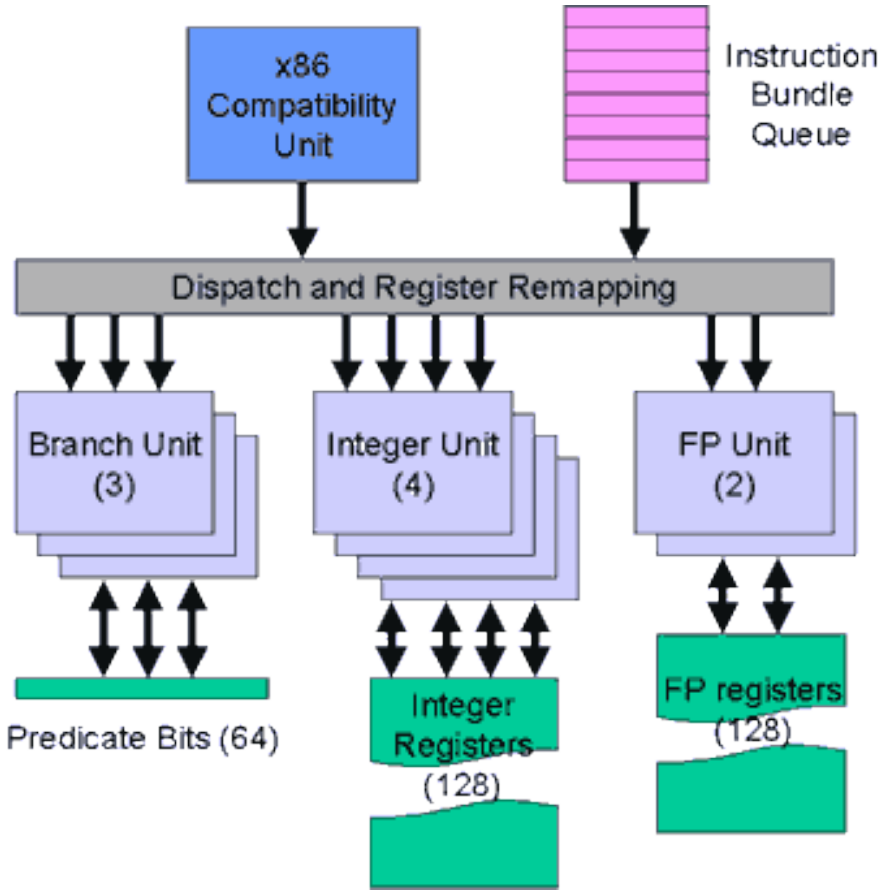
Instruction Set

SYSTEM	MMX	SSE3	SSE4
LGDT | SGDT LLDT | SLDT LTR | STR LIDT | SIDT LMSW | SMSW CLTS ARPL LAR LSL VERR | VERW INVD | WBINVD INVLPG LOCK HLT RSM RDMSR | WRMSR RDPMC RDTSC SYSENTER SYSEXIT	MOVD MOVQ PACKSS(WB|DW) PACKUSWB PUNPCK(H|L)(BW|WD|DQ) PADD(B|W|D) PADD(S|US)(B|W) PSUB(B|W|D) PSUB(S|US)(B|W) PMUL(H|L)W PMADDWD PCMP(EQ|GT)(B|W|D) PAND PANDN POR PXOR PS(L|R)L(W|D|Q) PSRA(W|D) EMMS	FISTTP LDDQU ADDSUBP(S|D) HADDP(S|D) HSUBP(S|D) MOVS(H|L)DUP MOVDDUP MONITOR MWAIT	PMUL(LD|DQ) DPP(D|S) MOVNTDQA BLEND[V](PD|PS) PBLENDVB|W) PMIN(UW|UD|SB|SD) PMAX(UW|UD|SB|SD) ROUND(P|S)(S|D) EXTRACTPS INSERTPS PINSR(B|D|Q) PEXTR(B|W|D|Q) PMOV(S|Z)X(BW|BD|BQ|WD|WQ|DQ) MPSADBW PHMINPOSUW PTEST PCMPEQQ PACKUSDW PCMP(E|I)STR(I|M) PCMPGTQ CRC32 POPCNT
64-BIT MODE	**VIRTUAL MACHINE**	**SSSE3**	**AESNI**
CDQE CMPSQ CMPXCHG16B LODSQ MOVSQ MOVZX STOSQ SWAPGS SYSCALL SYSRET	VMPTRLD VPTRST VMCLEAR VMREAD VMWRITE VMCALL VMLAUNCH VMRESUME VMXOFF VMXON INVEPT INVVPID	PHADD(W|SW|D) PHSUB(W|SW|D) PABS(B|W|D) PMADDUBSW PMULHRSW PSHUFB PSIGN(B|W|D) PALIGNR	AESDEC[LAST] AESENC[LAST] AESIMC AESKEYGENASSIST PCLMULQDQ

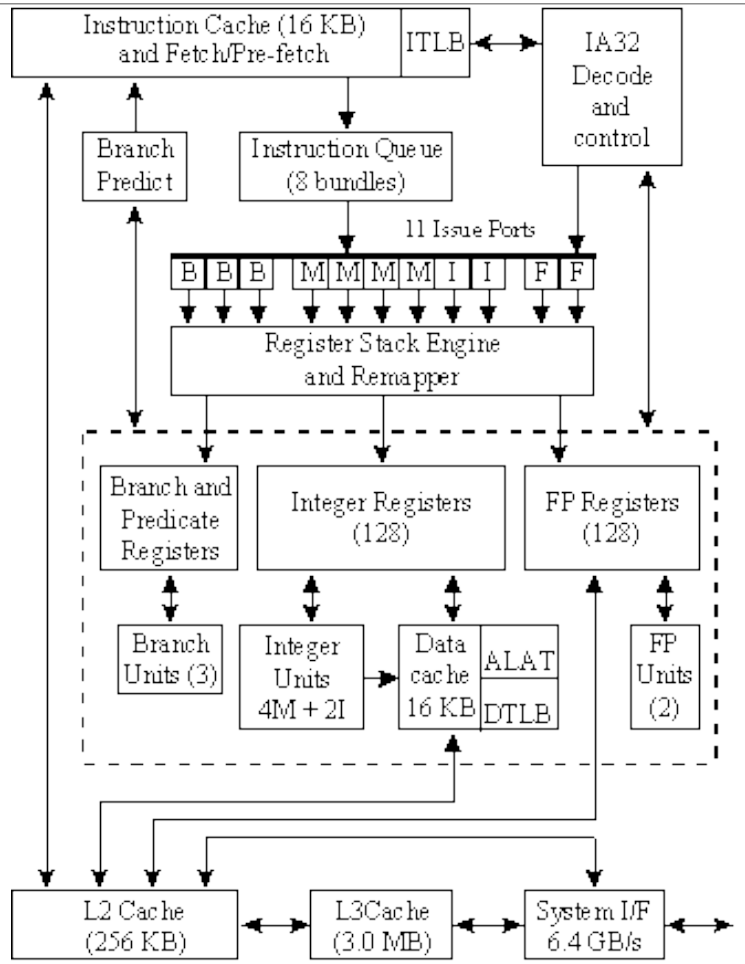
INTEGER	FPU	SSE	SSE2
MOV CMOV[N]((L|G|A|B)[E]|E|Z|S|C|O|P) XCHG BSWAP XADD CMPXCHG[8B] PUSH[A(D)] | POP[A(D)] IN | OUT CBW | CWDE | CWD | CDQ MOVSX | MOVZX ADD | ADC SUB | SBB [I]MUL [I]DIV INC | DEC NEG CMP DAA | DAS AAA | AAS | AAM | AAD AND | OR | XOR | NOT SH(L|R)[D] RA(L|R) RO(L|R) RC(L|R) BT[S|R|C] BS(F|R) SET[N]((L|G|A|B)[E]|E|Z|S|C|O|P) TEST JMP J[N]((L|G|A|B)[E]|E|Z|S|C|O|P) J[E]CXZ LOOP[N][Z|E] CALL | RET INT[O] | IRET ENTER | LEAVE BOUND MOVS[B|W|D] CMPS[B|W|D] SCAS[B|W|D] LODS[B|W|D] STOS[B|W|D] INS[B|W|D] OUTS[B|W|D] REP[N][Z|E] STC | CLC | CMC STD | CLD STI | CLI LAHF | SAHF PUSHF[D] | POPF[D] LDS | LES | LFS | LGS | LSS LEA NOP UD2 XLAT[B] CPUID	F[I]LD F[I]ST[P] FBLD FBSTP FXCH FCMOV[N](E|B|BE|U) FADD[P] FIADD FSUB[R][P] FISUB[R] FMUL[P] FIMUL FDIV[R][P] FIDIV[R] FPREM[1] FABS FCHS FRNDINT FSCALE FSQRT FXTRACT F[U]COM[P][P] FICOM[P] F[U]COMI[P] FTST FXAM FSIN FCOS FSINCOS FPTAN FPATAN F2XM1 FYL2X FYL2XP1 FLD1 FLDZ FLDPI FLDL2E FLDLN2 FLDL2T FLDLG2 FINCSTP FDECSTP FFREE F[N]INIT F[N]CLEX F[N]STCW FLDCW F[N]STENV FLDENV F[N]SAVE FRSTOR F[N]STSW FWAIT | WAIT FNOP FXSAVE FXRSTOR	MOV(A|U)PS MOV(H|HL|L|LH)PS MOVSS MOVMSKPS ADD(P|S)S SUB(P|S)S MUL(P|S)S DIV(P|S)S RCP(P|S)S SQRT(P|S)S RSQRT(P|S)S MAX(P|S)S MIN(P|S)S CMP(P|S)S [U]COMISS ANDPS ANDNPS ORPS XORPS SHUFPS UNPCK(H|L)PS CVTPI2PS CVT[T]PS2PI CVTSI2SS CVT[T]SS2SI	MOV(A|U)PD MOV(H|L)PD MOVSD MOVMSKPD ADD(P|S)D SUB(P|S)D MUL(P|S)D DIV(P|S)D SQRT(P|S)D MAX(P|S)D MIN(P|S)D CMP(P|S)D [U]COMISD ANDPD ANDNPD ORPD XORPD SHUFPD UNPCK(H|L)PD CVT(PI|DQ)2PD CVT[T]PD2(PI|DQ) CVTSI2SD CVT[T]SD2SI CVTPS2PD CVTPD2PS CVTDQ2PS CVT[T]PS2DQ CVTSS2SD CVTSD2SS MOVDQ(A|U) MOVQ2DQ MOVDQ2Q PUNPCK(H|L)QDQ PADDQ PSUBQ PMULUDQ PSHUF(LW|HW|D) PS(L|R)LDQ MASKMOVDQU MOVNT(PD|DQ|I) CLFLUSH LFENCE MFENCE PAUSE
		PEXTRW PINSRW P(MIN|MAX)(UB|SW) PMOVMSKB PMULHUW PSADBW PSHUFW LDMXCSR STMXCSR MASKMOVQ MOVNT(Q|PS) PREFETCHT(0|1|2) PREFETCHNTA SFENCE	

x86 Architecture

Instruction Set



Block Diagram of the IA64 McKinley



X86 Architecture

Addressing Memory

- In protected mode, applications can choose a flat or segmented memory model (see the SDM Volume 1, Chapter 3 for details); in real mode only a 16-bit segmented model is available. Most programmers will only use protected mode and a flat-memory model, so that's all we'll discuss here.

- A memory reference has four parts and is often written as

[SELECTOR : BASE + INDEX * SCALE + OFFSET]

- The selector is one of the six segment registers; the base is one of the eight general purpose registers; the index is any of the general purpose registers except ESP; the scale is 1, 2, 4, or 8; and the offset is any 32-bit number. (Example: [fs:ecx+esi*8+93221].) The minimal reference consists of only a base register or only an offset; a scale can only appear if there is an index present.

- Sometimes the memory reference is written like this:

selector

offset(base,index,scale)

X86 Architecture

Data Types & Little Endian

Type name	Number of bits	Bit indices
Byte	8	7..0
Word	16	15..0
Doubleword	32	32..0
Quadword	64	63..0
Doublequadword	128	127..0

The IA-32 is little endian, meaning the least significant bytes come first in memory. For example:

0	12
1	31
2	CB
3	74
4	67
5	45
6	0B
7	23
8	A4
9	1F
A	36
B	06
C	FE
D	7A
E	12

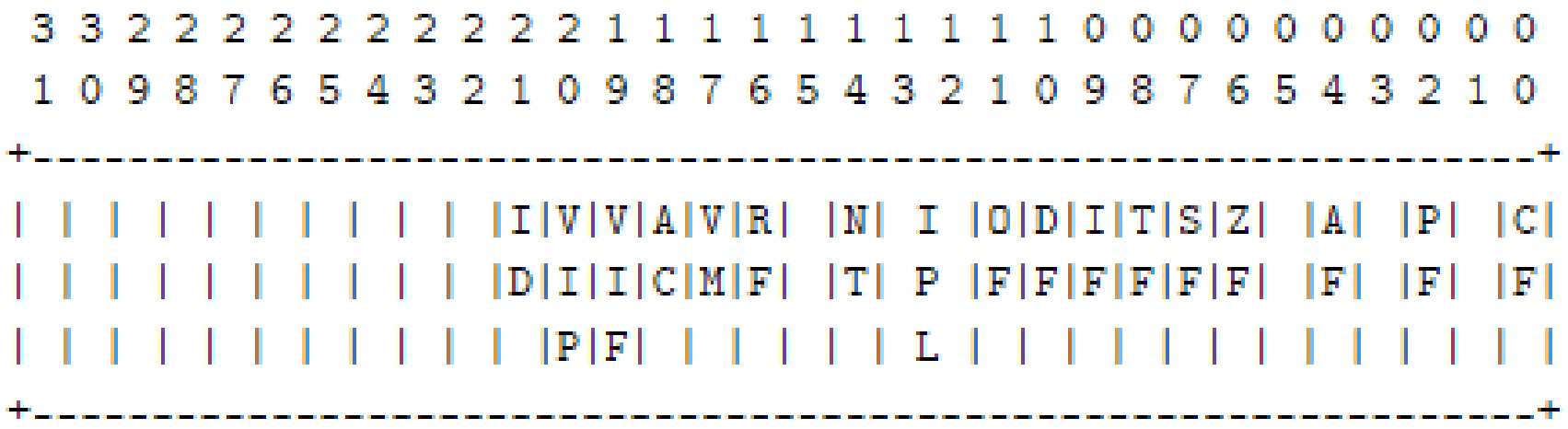
```
byte @ 9 = 1F
word @ B = FE06
word @ 6 = 230B
word @ 1 = CB31
dword @ A = 7AFE0636
qword @ 6 = 7AFE06361FA4230B
word @ 2 = 74CB
qword @ 3 = 361FA4230B456774
dword @ 9 = FE06361F
```

Note that if you draw memory with the lowest bytes at the bottom, then it is easier to read these values!

x86 Architecture

Flags Register

Many instructions cause the flags register to be updated. For example if you execute an add instruction and the sum is too big to fit into the destination register, the Overflow flag is set.



Note that if you draw memory with the lowest bytes at the bottom, then it is easier to read these values!

x86 Architecture

Exceptions (Errors in Lang.)

Sometimes while executing an instruction an exception occurs. There are three types of exceptions.

- Faults - indicate an operation can't be completed
- Traps - software generated interrupts (INT or INTO)
- Aborts - indicate a serious problem with the OS itself.

When exceptions occur, the processor will start executing code in an *exception handler* associated with that exception. The predefined exceptions are:

GENERAL EXCEPTIONS				
0	#DE	Divide Error	fault	DIV or IDIV instruction
1	#DB	Debug	fault trap	...
3	#BP	Breakpoint	trap	INT 3 instruction
4	#OF	Overflow	trap	INTO instruction executed when overflow flag in EFLAGS is set
5	#BR	Bound Range Exceeded	fault	BOUND instruction
6	#UD	Undefined Opcode	fault	UD2 instruction, or attempt to execute an opcode that doesn't correspond to any instruction
7	#NM	Device Not Available	fault	FPU instruction or WAIT instruction on a processor without an FPU that is not linked to a FPU coprocessor
8	#DF	Double Fault	abort	Exception occurred during an exception handler
10	#TS	Invalid TSS	fault	Task switch or implicit TSS access
11	#NP	Not Present	fault	Loading segment registers or accessing system segments
12	#SS	Stack Segment Fault	fault	Stack operations and SS register loads
13	#GP	General Protection Fault	fault	Any memory reference and other protection checks
14	#PF	Page Fault	fault	Any memory reference
16	#MF	FPU Math Fault	fault	Any FPU instruction #IS - FPU stack overflow #IA - Invalid arithmetic operation #Z - Divide by zero #D - Source operand is a denormal number #O - Overflow in result #U - Underflow in result #P - Inexact result
17	#AC	Alignment Check	fault	Any data reference in memory
18	#MC	Machine Fault	abort	Internal Error or bus error
19	#XF	SIMD Math Fault	fault	Any SIMD instruction #I - Invalid arithmetic operation or source operand #Z - Divide by zero #D - Source operand is a denormal number #O - Overflow in result #U - Underflow in result #P - Inexact result