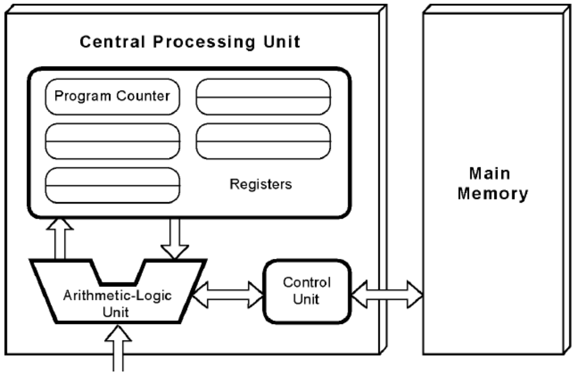
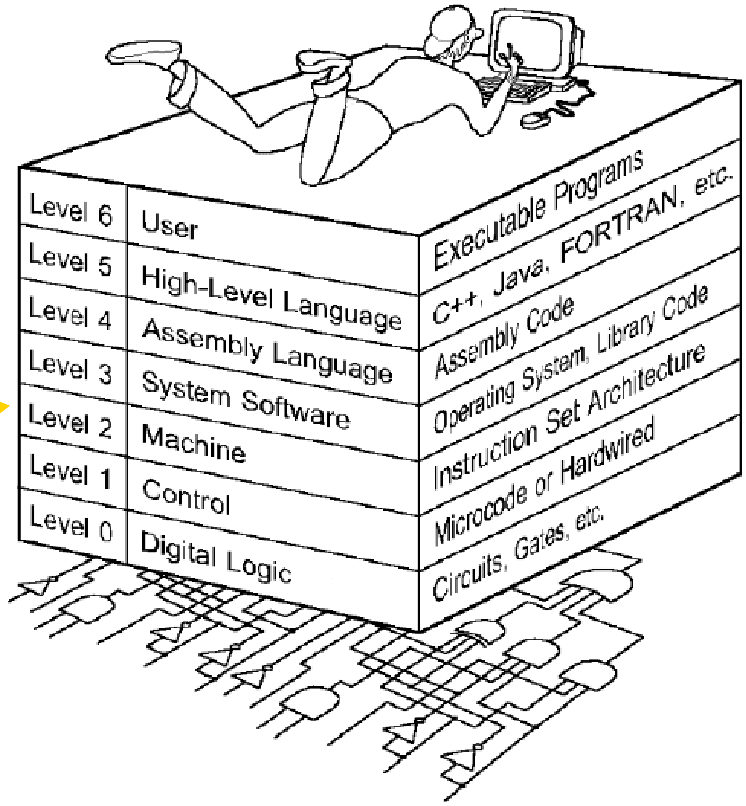
Fetch-Decode-Execution CYCLE





Level 6	User	Executable Programs
Level 5	High-Level Language	C++, Java, FORTRAN, etc.
Level 4	Assembly Language	Assembly Code
Level 3	System Software	Operating System, Library Code
Level 2	Machine	Instruction Set Architecture
Level 1	Control	Microcode or Hardwired
Level 0	Digital Logic	Circuits, Gates, etc.

Chapter 5 – A Closer Look at Instruction Set Architecture (ISA)

LO-3:

- Describe and explain the **organization** of the classical **von Neumann computer** and its **major functional units**.
- Describe the functioning of a **single cycle CPU** and its internal operations.

>>> Quiz-3 and **Test-3** (Chs-4 & 5)

Chapter 5 Objectives

- □ Understand the factors involved in **ISA design**.
- □ Gain familiarity with memory addressing modes.
- □ Understand the concepts of instruction-level pipelining and its affect upon execution **performance**.

Outline

- □Instruction formats and types
- □Operand types
- □Addressing (memory access)
- □Instruction pipelining
- □Real-world examples of ISAs

An Instruction: **OpCode** Operand1 Addr. Operand2 Addr. Operand3 Addr.

5.2 Instruction Formats

Instruction sets are differentiated by the following:

- Number of bits per instruction.
  - Whether short, long, or variable.
- Types of operations.
- Stack-based or register-based.
- Number of explicit operands per instruction.
- Operand location.
  - Number of addressable registers.
  - Memory byte- or word- addressable.
  - Memory Addressing modes.
- Type and size of operands.

Primarily, based on the operator and operand types, size, location

5.2 Instruction Formats

Instruction set architectures are measured according to:

- Main memory **space** occupied by a program.
- Instruction **complexity**.
- Instruction **length** (in bits).
- **Total** number of **instructions** in the instruction set.

5.2 Instruction Formats

- Byte ordering, or *endianness*, is another major architectural consideration.
- If we have a two-byte integer, the integer may be stored so that the least significant byte is followed by the most significant byte or vice versa.
  - In *little endian* machines, the least significant byte is followed by the most significant byte.
  - *Big endian* machines store the most significant byte first (at the lower address).

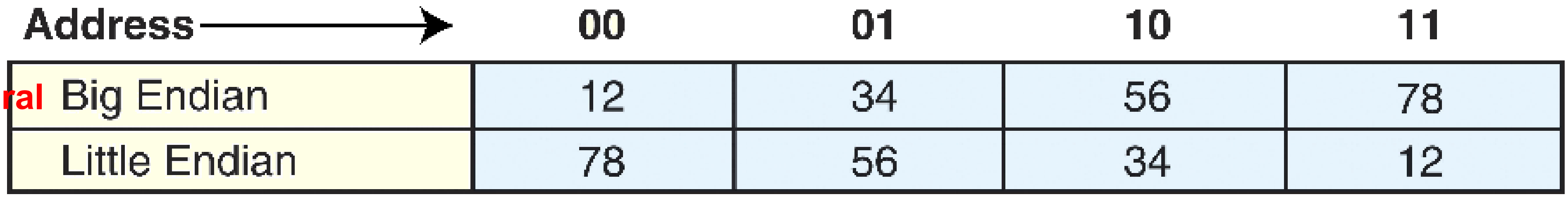
e.g. IA x86, MIPS, ARM*
Windows
* configurable

e.g. RISC, IBM
Motorolla

Unix, Linux, Mac

5.2 Instruction Formats

- As an example, suppose we have the hexadecimal number 0x12345678.
- The big endian and small endian arrangements of the bytes are shown below.

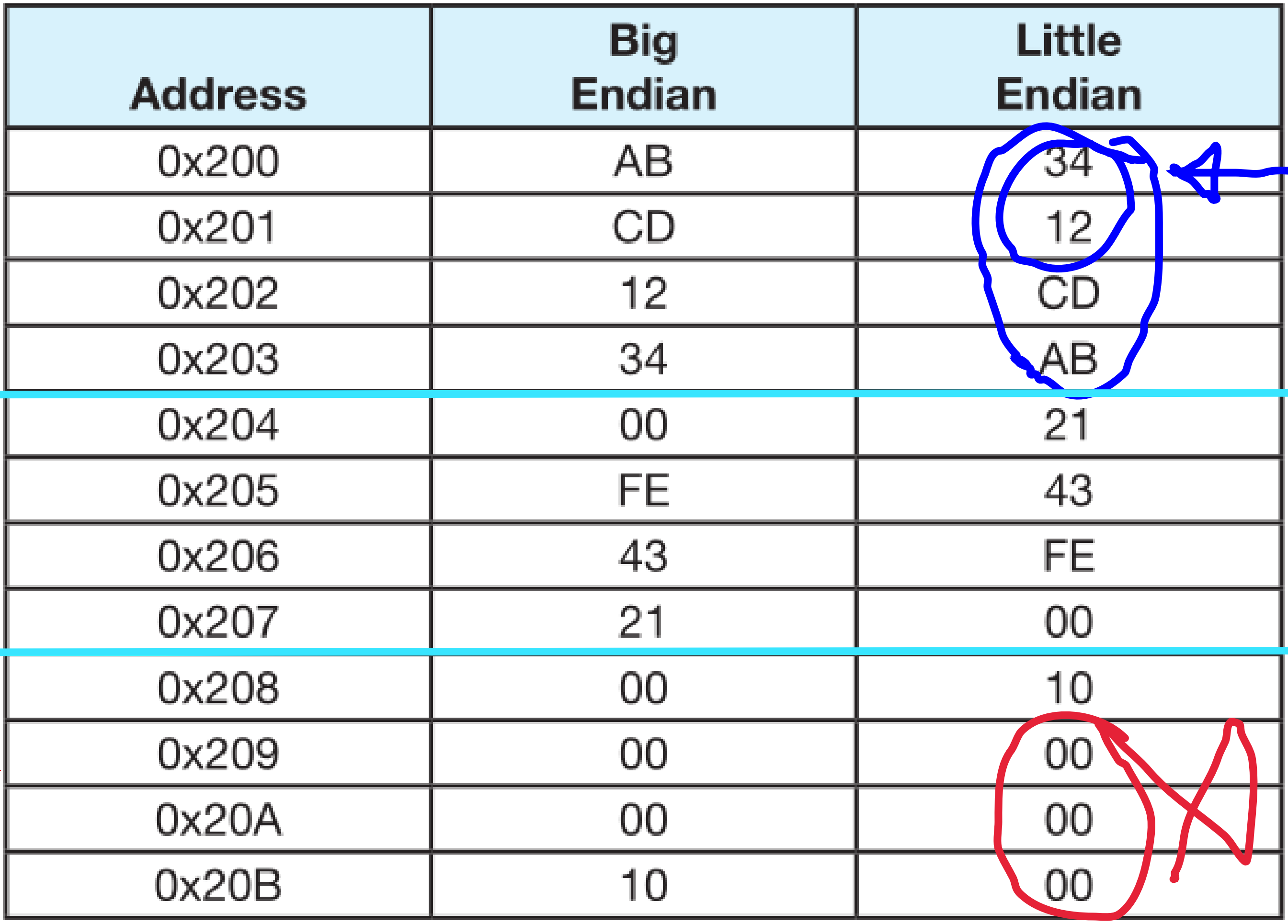


Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

natural

5.2 Instruction Formats

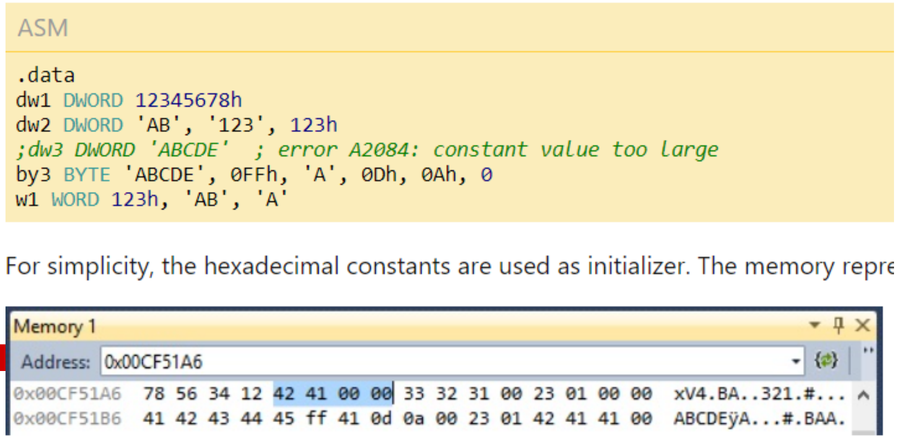
- A larger example: A computer uses 32-bit integers. The values 0xABCD1234, 0x00FE4321, and 0x10 would be stored sequentially in memory, starting at address 0x200 as below.



Address	Big Endian	Little Endian
0x200	AB	34
0x201	CD	12
0x202	12	CD
0x203	34	AB
0x204	00	21
0x205	FE	43
0x206	43	FE
0x207	21	00
0x208	00	10
0x209	00	00
0x20A	00	00
0x20B	10	00

32 bit word boundary

5.2 Instruction Formats



ASM

```
.data
dw1 DWORD 12345678h
dw2 DWORD 'AB', '123', 123h
;dw3 DWORD 'ABCDE'  ; error A2084: constant value too large
by3 BYTE 'ABCDE', 0FFh, 'A', 0Dh, 0Ah, 0
w1 WORD 123h, 'AB', 'A'
```

For simplicity, the hexadecimal constants are used as initializer. The memory repre

Memory 1

Address: 0x00CF51A6

```
0x00CF51A6  78 56 34 12 42 41 00 00 33 32 31 00 23 01 00 00  xV4.BA..321.#...
0x00CF51B6  41 42 43 44 45 ff 41 0d 0a 00 23 01 42 41 41 00  ABCDEÿA...#.BAA.
```

- Big endian:
  - Is more natural.
  - The sign of the number can be determined by looking at the byte at address offset 0.
  - Strings and integers are stored in the same order.
- Little endian:
  - Makes it easier to place values on non-word boundaries.
  - Conversion from a 16-bit integer address to a 32-bit integer address does not require any arithmetic.

Tradeoffs:

- simplicity, speed, cost, ease of use

Where are operands?

Memory-memory: 2+ in memory.
Register-memory: 1+ in a register.
Load-store: no operand in memory.

5.2 Instruction Formats

- □ In a stack architecture, instructions and operands are implicitly taken from the stack.
  - ■ A stack cannot be accessed randomly!

>>> Java

- □ In an accumulator architecture, one operand of a binary operation is implicitly in the accumulator.
  - ■ Other operand is in memory, creating lots of bus traffic.

>>> ~MARIE

- □ In a general purpose register (GPR) architecture, most used today, registers can be used instead of memory.
  - ■ Faster than accumulator architecture.
  - ■ Efficient implementation for compilers.
  - ■ Results in longer instructions.

>>> IA x86

A pretext to Cache memory in more advanced and contemporary architectures and to facilitate pipelining (upcoming)!

5.2 Instruction Formats

- Let's see how to evaluate an **infix expression** using different instruction formats.
- With a three-address ISA, (e.g.,mainframes), the infix expression, e.g. RISC, ARM

$$Z = X \times Y + W \times U$$

<== A commonly used notation in arithmetical and logical formulae and statements.

might look like this:

```
MULT R1,X,Y
MULT R2,W,U
ADD  Z,R1,R2
```

5.2 Instruction Formats

- In a two-address ISA, (e.g.,Intel, Motorola), the infix expression,

then Atmel (used in Arduino), now Microchip

$$Z = X \times Y + W \times U$$

might look like this:

```
LOAD  R1,X
MULT  R1,Y
LOAD  R2,W
MULT  R2,U
ADD   R1,R2
STORE Z,R1
```

Note: Two-address ISAs usually require one operand to be a register.

5.2 Instruction Formats

- In a one-address ISA, like MARIE, the infix expression,

  Z = X × Y + W × U

  looks like this:

```
LOAD X
MULT Y
STORE TEMP
LOAD W
MULT U
ADD TEMP
STORE Z
```

Memory references (X, Y, ...) come from the Symbol Table

5.2 Instruction Formats

- In a stack ISA, the **postfix** expression,

  Z = X Y × W U × +

  might look like this:

```
PUSH X
PUSH Y
MULT
PUSH W
PUSH U
MULT
ADD
POP Z
```

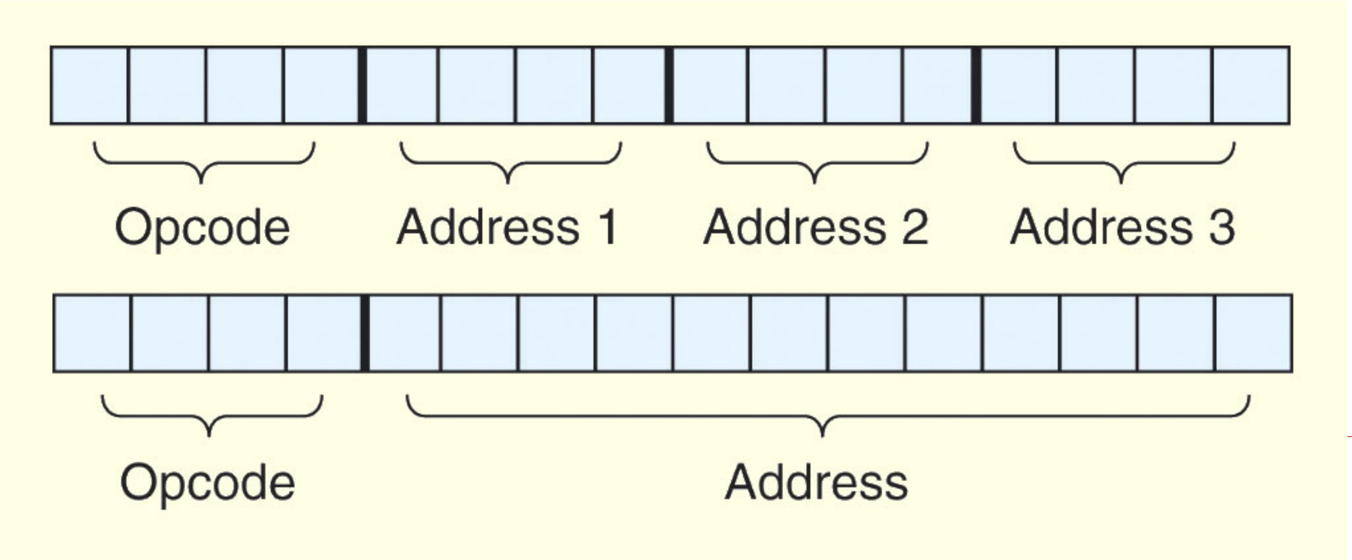
In sum, the ISA with lesser* number of operands results in bigger/longer program that will have more number of instructions to be executed.

* implicit registers may be involved.

Expanding opcodes are used to save program space wastage when fixed-length instructions are used.

5.2 Instruction Formats

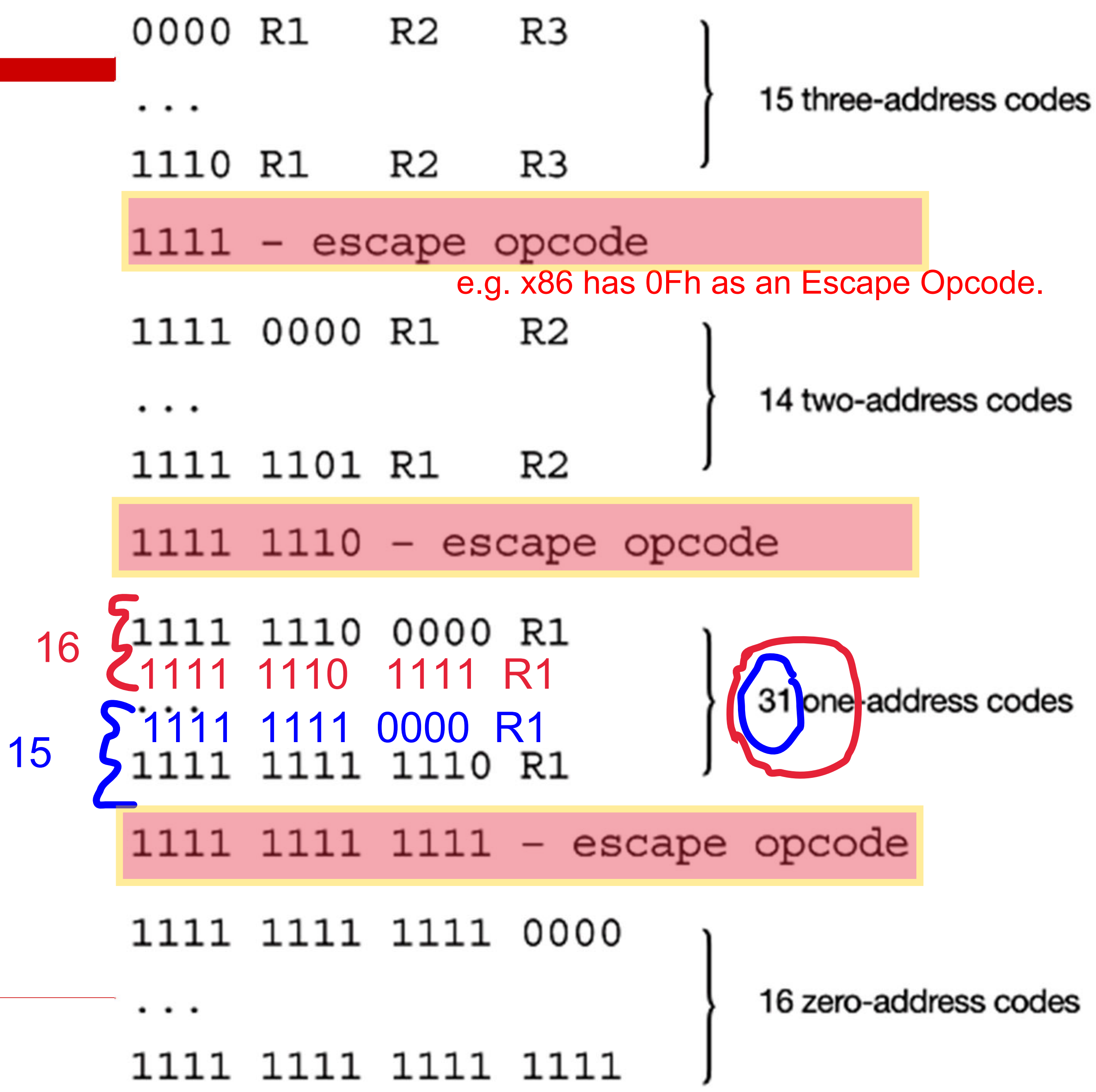
- A system has 16 registers and 4K of memory.
- We need 4 bits to access one of the registers. We also need 12 bits for a memory address.
- If the system is to have 16-bit instructions, we have two choices for our instructions:



5.2 Instruction Formats

Designing a rich ISA
with 76 operations/codes:

- □ If we allow the length of the opcode to vary, we could create a very rich instruction set:



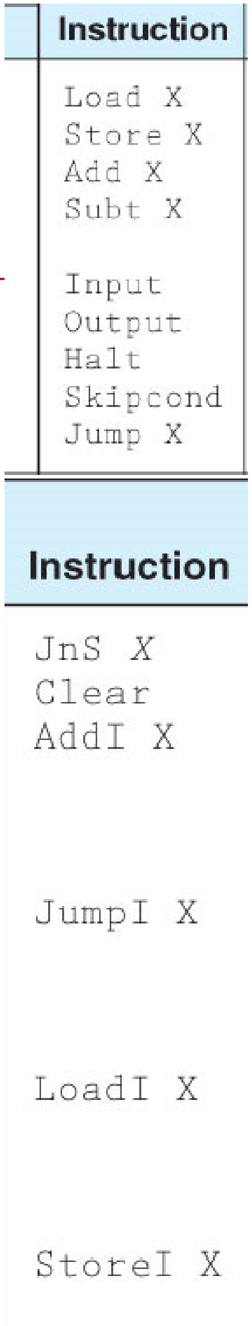
```
     0000 R1   R2   R3    }
     ...                  } 15 three-address codes
     1110 R1   R2   R3    }
     1111 - escape opcode
                    e.g. x86 has 0Fh as an Escape Opcode.
     1111 0000 R1   R2    }
     ...                  } 14 two-address codes
     1111 1101 R1   R2    }
     1111 1110 - escape opcode
16 { 1111 1110 0000 R1    }
   { 1111 1110 1111 R1    }
15 { 1111 1111 0000 R1    } 31 one-address codes
   { 1111 1111 1110 R1    }
     1111 1111 1111 - escape opcode
     1111 1111 1111 0000  }
     ...                  } 16 zero-address codes
     1111 1111 1111 1111  }
```

5.3 Instruction types

Instructions fall into several broad categories that you should be familiar with:

- Data movement.
- Arithmetic.
- Boolean.
- Bit manipulation.
- I/O.
- Control transfer.
- Special purpose.

Can you think of some examples of each of these?



Instruction
Load X Store X Add X Subt X
Input Output Halt Skipcond Jump X

Instruction
JnS X Clear AddI X
JumpI X
LoadI X
StoreI X

5.4 Addressing

- where an operand is located?

- operand's actual location is its **effective address.**

- □ *Immediate addressing* is where the data is part of the instruction. #
- □ *Direct addressing* is where the address of the data is given in the instruction. X
- □ *Register addressing* is where the data is located in a register. R1
- □ *Indirect addressing* gives the address of the address of the data in the instruction. e.g. pointers, IVT *x
- □ *Register indirect addressing* uses a register to store the address of the data. e.g. MAR in MARIE *r1

5.4 Addressing

- □ *Indexed addressing* uses a **register** (implicitly or explicitly) as an **offset**, which is **added** to the **address in the operand to determine the effective address** of the data. e.g. arrays

- □ *Base addressing* is similar except that a base register is used instead of an index register.

- □ The difference between these two is that an index register holds an **offset addition** to the address given in the instruction, a base register holds a base address where the address field represents a **displacement from this base**. e.g. JVM, segmentation

5.4 Addressing

- In *stack addressing* the operand is assumed to be on top of the stack.
- There are many variations to these addressing modes including:
  - Indirect indexed.
  - Base/offset.
  - Self-relative
  - Auto increment - decrement.
- We won’t cover these in detail.

Let’s look at an example of the principal addressing modes.

5.4 Addressing

Memory	
0x800	0x900
...	
0x900	0x1000
...	
0x A 00	0x500
...	
0x B 00	0x600
...	
0x1000	0x700

```
LOAD X 800
```

Load R1

Reg. Dir. X 800
Reg. Indir. X 900

Mode	Value Loaded into AC
Immediate	0x800
Direct	0x900
Indirect	0x1000
Indexed	0x700

R1	0x800

R1=Index Register

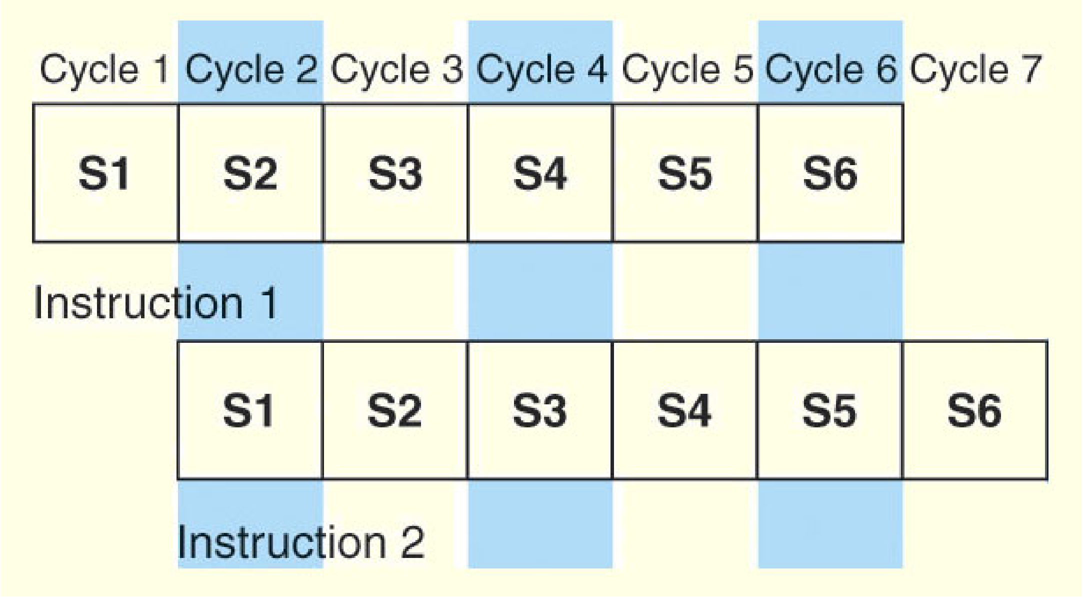
CU parallelizes microinstructions of FDE cycle to increase throughput.

5.5 Instruction Pipelining

==> provides instruction level parallelism (ILP)

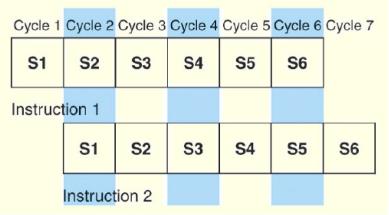
- For every clock cycle, one small step is carried out, and the stages are overlapped.

6-stage pipeline



S1. Fetch instruction.
S2. Decode opcode.
S3. Calculate effective address of operands.

S4. Fetch operands.
S5. Execute.
S6. Store result.



From e.g. above (copied to the left):
- tp = 1 clock cycle per stage
- k=6 stage-pipeline
per instruction T
- k x tp = 6 cycles per instruction
- remaining (n-1) T take tp=1 clock
cycle per task !
==> 7 clock cycles for 2 ins./Tasks

5.5 Instruction Pipelining

□ The theoretical speedup offered by a pipeline can be determined as follows:

Let $\boldsymbol{t_p}$ be the **time per stage**. **Each instruction represents a task, *T***, in the pipeline.

The first task (instruction) **requires *k* x $\boldsymbol{t_p}$** time to complete in a ***k*-stage** pipeline. The remaining (n - 1) tasks emerge from the pipeline one per cycle. So the total time to complete the remaining tasks is $(n - 1)t_p$.

Thus, to complete *n* tasks using a *k*-stage pipeline requires:

$$(k \times t_p) + (n - 1)t_p = (k + n - 1)t_p.$$

5.5 Instruction Pipelining

- If we take the time required to complete *n* tasks without a pipeline and divide it by the time it takes to complete *n* tasks using a pipeline, we find:

$$\text{Speedup } S = \frac{n t_n}{(k + n - 1)\, t_p}$$

here, let's assume:
tn=time per task
= k tp

- If we take the limit as *n* approaches infinity, (*k* + *n* - 1) approaches *n*, which results in a theoretical speedup of:

$$\text{Speedup } S = \frac{k t_p}{t_p} = k$$

$$= \frac{k\ n\ tp}{(k+n-1)\ tp}$$

Assumptions in the equation:

1) architecture supports fetching instructions and data in parallel.
2) pipeline can be kept filled at all times. Pipeline hazards arise that cause pipeline conflicts and stalls.

5.5 Instruction Pipelining

- An instruction pipeline may **stall**, or be **flushed** for any of the following reasons:
  - Resource conflicts.
  - Data dependencies.
  - Conditional branching.
- Measures can be taken at the **software level** as well as at the **hardware level** to reduce the effects of these hazards, but they cannot be totally **eliminated**.

- Pipelining, performance, capability:
- multi-tasking:
    - concurrency (multi-threading)
    - simultaneously (xCores, multi-processing)

5.6 Real-World Examples of ISAs: Intel

□Pentium-I introduced two 5-stage **pipelines**. Pentium IV had a 24-stage, Itanium (IA-64) had a 10-stage pipeline.

□ The original **8086 provided 17** address modes, Pentium supported them for backward compatibility.

□ The Itanium, having a RISC core, supports only one: register indirect addressing, with optional post increment.

5.6 Real-World Examples of ISAs: MIPS

□ ***Microprocessor Without Interlocked Pipeline Stages***.

□ **Little endian** and **word-addressable** with **three-address**, **fixed-length instructions**.

□ R2000/R3000 5-stage, R4000/R4400 8-stage pipelines

- R10000 instructions: 5-stage for integer, 7-stage for FP, 6-stage for LOAD/STORE.
- only load/store can access memory.
- uses only base addressing mode.

* MIPS is a family of RISC ISA developed by MIPS Computer Systems (now MIPS Technologies) from Stanford Uni's work. Interlocks were stalling the pipeline when hazard arises and defeats the purpose of pipelining.

5.6 Real-World Examples of ISAs: Java4Any!

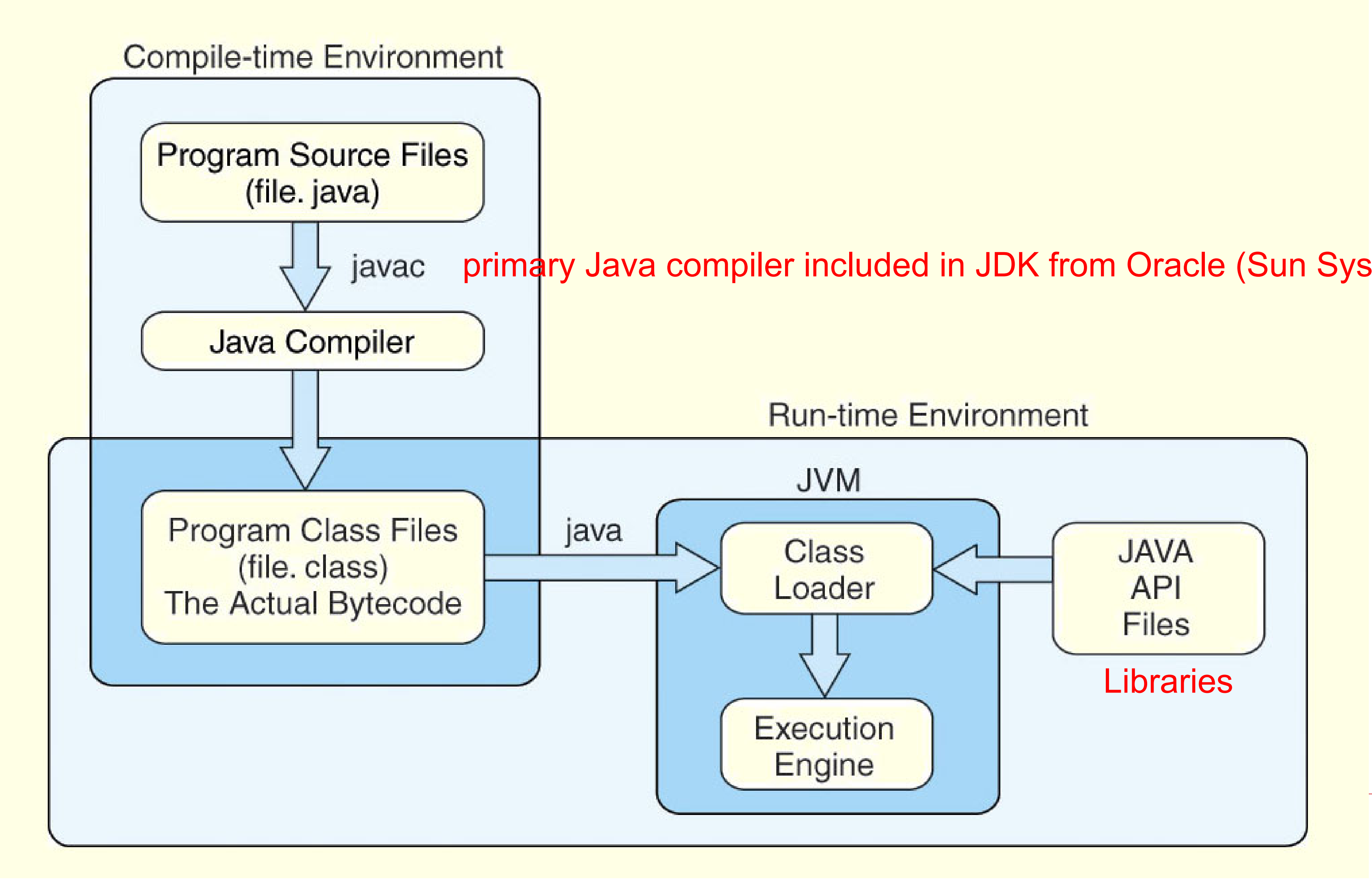
- interpreted language that **runs in a software machine** called the *Java Virtual Machine* (JVM).

□ A JVM is **written in a native language** for a wide array of processors, including MIPS and Intel.

□ **has a stack-based ISA** all of its own, called **bytecode**.

- zero address instructions.The JVM has four registers that provide access to five regions of main memory.
- All references to memory are offsets from these registers. Java uses no pointers or absolute memory

5.6 Real-World Examples of ISAs: Java



primary Java compiler included in JDK from Oracle (Sun Sys)

Libraries

5.6 Real-World Examples of ISAs: ARM

- Most widely used 32-bit instruction architecture:
  - 95%+ smartphones, 80%+ cameras, 40%+ TVs
- Founded in 1990, by Apple and others, ARM (Adv. RISC Machine) is a British firm, ARM Holdings, does not manufacture, but licenses to manufacture.
- Fixed-length, three-operand instructions, load/store arch., simple addressing modes, 3-stage pipeline.
- ARM8+ implementations has 13-stage integer pipeline
- 37 registers, simultaneously load/store any of 16 GPR
- Most instructions execute in a single cycle, if no pipeline hazards or memory accesses.

ISA's Capability: Summary

- Instruction:
  - length, numbers,
    - format, operator/opcode expanding, operand
    - number, location, type, size
  - execution/CPU: stack/Accumulator/GPR, #addr.
  - memory addressing
    - immi, dir, indir, reg, regind, index, base, stack
  - instruction complexity: k-stage pipelining, ILP speedup
- program: length, density
- Adv Arch.
  - Intel: addr modes, pipelining, --> MIPS
  - JVM: stack 0-addr/operand, xPlatform, Referenced
  - ARM: load/store 16 GPRs to 37, fixed 3 ins, pipelined.