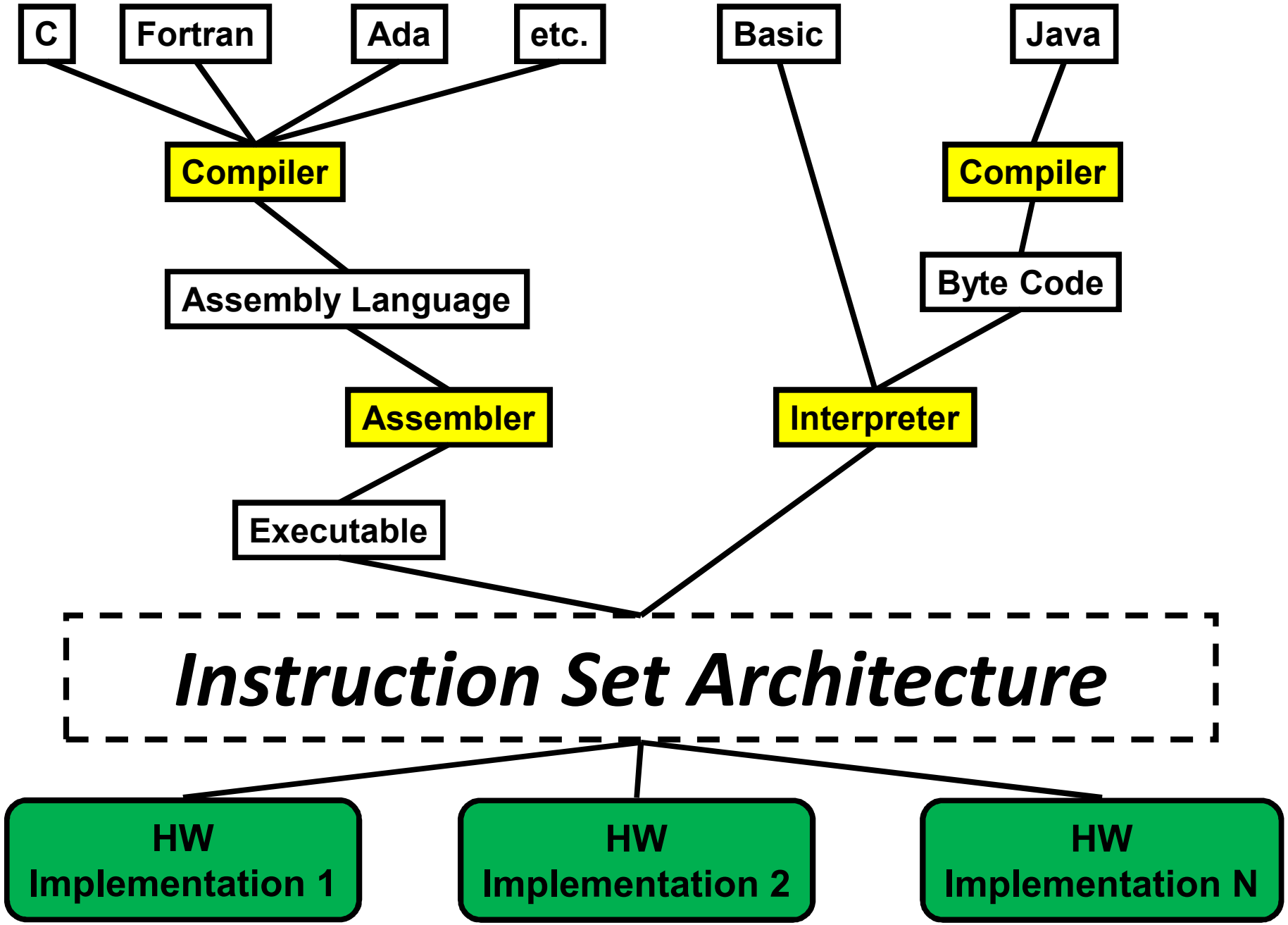
William Stallings
Computer Organization and Architecture
6th Edition

Chapter 10
Instruction Sets: Characteristics and Functions

What is an instruction set?

- The complete collection of instructions that are understood by a CPU
- Can be considered as a functional spec for a CPU
  - —Implementing the CPU in large part is implementing the machine instruction set
- Machine Code is rarely used by humans
  - —Binary numbers / bits
  - —Machine code is usually represented by human readable assembly codes
  - —In general, one assembler instruction equals one machine instruction


C
Fortran
Ada
etc.
Basic
Java
Compiler
Compiler
Assembly Language
Byte Code
Assembler
Interpreter
Executable
Instruction Set Architecture
HW Implementation 1
HW Implementation 2
HW Implementation N

High Level language Constructs

- **High Level Language Constructs**

```
a = b + c; /* add b and c and place in a */
d = e - f; /* subtract f from e and place in d */
x = y & z; /* AND y and z and place in x */
```

- **Assembly Language Constructs**

```
add a, b, c;  a ← b + c
sub d, e, f;  d ← e - f
and x, y, z;  x ← y & z
```

Elements of an Instruction

- Operation code (Op code)
  - —Do this
- Source Operand reference
  - —To this
- Result Operand reference
  - —Put the answer here
- Next Instruction Reference
  - —When you have done that, do this...
  - —Next instruction reference often implicit (sequential execution)

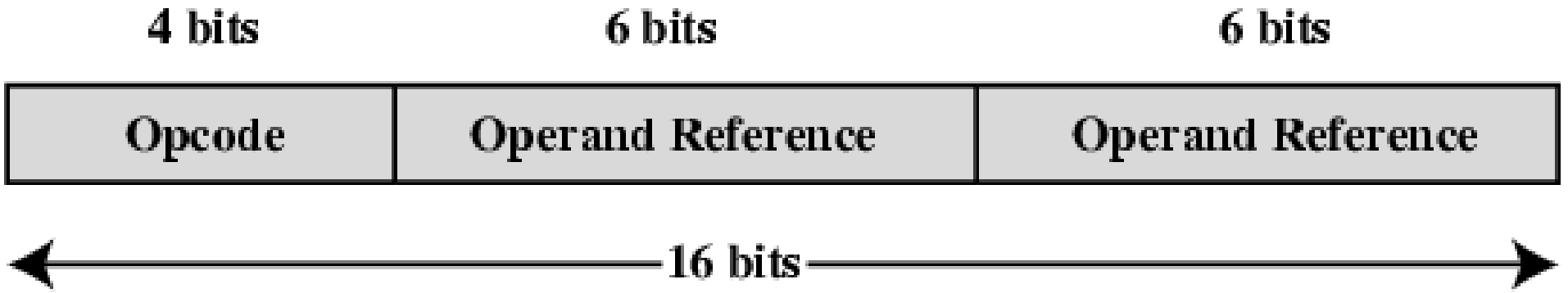
the Operands?

- Main memory (or virtual memory or cache)
  - —Requires address
- CPU register
  - —Can be an implicit reference (e.g., x87 FADD) or explicit operands (add eax, ecx)
- I/O device
  - —Several forms:
  - —Specify I/O module and device
  - —Specify address in I/O space
  - —Memory-mapped I/O just another memory address

Instruction Representation

- In machine code each instruction has a unique bit pattern
- For human consumption (well, programmers anyway) a symbolic representation is used
  - —e.g. ADD, SUB, LOAD
  - —Called ***mnemonics***
- Operands can also be represented in this way
  - —ADD A,B

Simple Instruction Format



Instruction Types (for OpCode)

- Data processing
  - —Arithmetic and logical instructions
- Data storage (main memory)
- Data movement (I/O)
- Program flow control
  - —Conditional and unconditional branches
  - —Call and Return

Design Issues: still in dispute

- Operation Repertoire
  - —How many operations and how complex should they be? Few operations = simple silicon; many ops makes it easier to program
- Data types
- Instruction Format and Encoding
  - —Fixed or variable length? How many bits? How many addresses? This has some bearing on data types
- Registers
  - —How many can be addressed and how are they used?
- Addressing Modes
  - —Many of the same issues as Operation Repertoire

Number of Operands?

Architecture Styles:

- Stack oriented
  - Burroughs
- Memory oriented
  - IBM s/360 et al
- Register oriented
  - MIPS, Alpha, ARM
- Hybrid
  - Intel x86, Power PC

Number of Operands: Instruction Format

- Zero Operand Instructions
  - Halt, NOP
  - Stack machines: Add
- One Operand Instructions
  - Inc, Dec, Neg, Not
  - Accumulator machines: Load M, Add M
- Two Operand Instructions
  - Add r1, r2 (i.e. r1 = r1 + r2)
  - Mov r1, r2
- Three Operand Instructions
  - Add r1, r2, r3
  - Load rd, rb, offset

Number of Addresses (a)

- 3 addresses
  - —Operand 1, Operand 2, Result
  - —a = b + c;
  - —add ax, bx, cx
  - —May be a forth - next instruction (usually implicit)
  - —Not common
  - —Needs very long words to hold everything
- 3 address format rarely used
  - —Instructions are long because 3 or more operands have to be specified

Number of Addresses (b)

- 2 addresses
  - —One address doubles as operand and result
  - —a = a + b
  - —add ax, bx
  - —Reduces length of instruction over 3-address format
  - —Requires some extra work by processor
    - – Temporary storage to hold some results

Number of Addresses (c)

- 1 address
  - —Implicit second address
  - —Usually a register (accumulator)
  - —Common on early machines

- Used in some Intel x86 instructions with implied operands
  - — mul ax

Number of Addresses (d)

- 0 (zero) addresses
  - —All addresses implicit
  - —Uses a stack
  - —Example: c = a + b

```
push a
push b
add             ;st(1) <- a+b,
pop c
```

Computation of Y = (a-b) / (c + (d * e))

- Three address instructions

  sub y,a,b

  mul t,d,e

  add t,t,c

  div y,y,t

- Two address instructions

  mov y,a

  sub y,b

  mov t,d

  mul t,e

  add t,c

  div y,t

Computation of Y = (a-b) / (c + (d * e))

- One address instructions

load d

mul e

add c

store y

load a

sub b

div y

store y

How Many Addresses

- More addresses
  - —More complex (powerful?) instructions
  - —More registers
    - Inter-register operations are quicker
  - —Fewer instructions per program
- Fewer addresses
  - —Less complex (powerful?) instructions
  - —More instructions per program
  - —Faster fetch/execution of instructions
  - —Less complexity in processor
  - —One address format however limits you to one register

Fixed/Var-Length Instruction Format

Fixed Length Instructions

- Pros
  - Simplifies implementation
  - Can start interpreting
- Cons
  - May waste space
  - May need additional logic in datapath
  - Limits instruction set designer

Variable Length Instructions

- Pros
  - No wasted space
  - Less constraints on designer
  - More flexibility opcodes, addressing modes and operands
- Cons
  - Complicates implementation

Design Decisions (1)

- Operation repertoire
  - —How many ops?
  - —What can they do?
  - —How complex are they?
- Data types
- Instruction formats
  - —Length of op code field
  - —Number of addresses

Design Decisions (2)

- Registers
  - Number of CPU registers available
  - Which operations can be performed on which registers?
- Addressing modes (later…)
- RISC v CISC

Types of Operand

- Addresses
- Numbers
  - —Integer/floating point
  - —Binary / BCD integer representations
- Characters
  - —ASCII etc.
- Logical Data
  - —Bits or flags
- (Aside: Is there any difference between numbers and characters? Ask a C programmer!)

Pentium Data Types

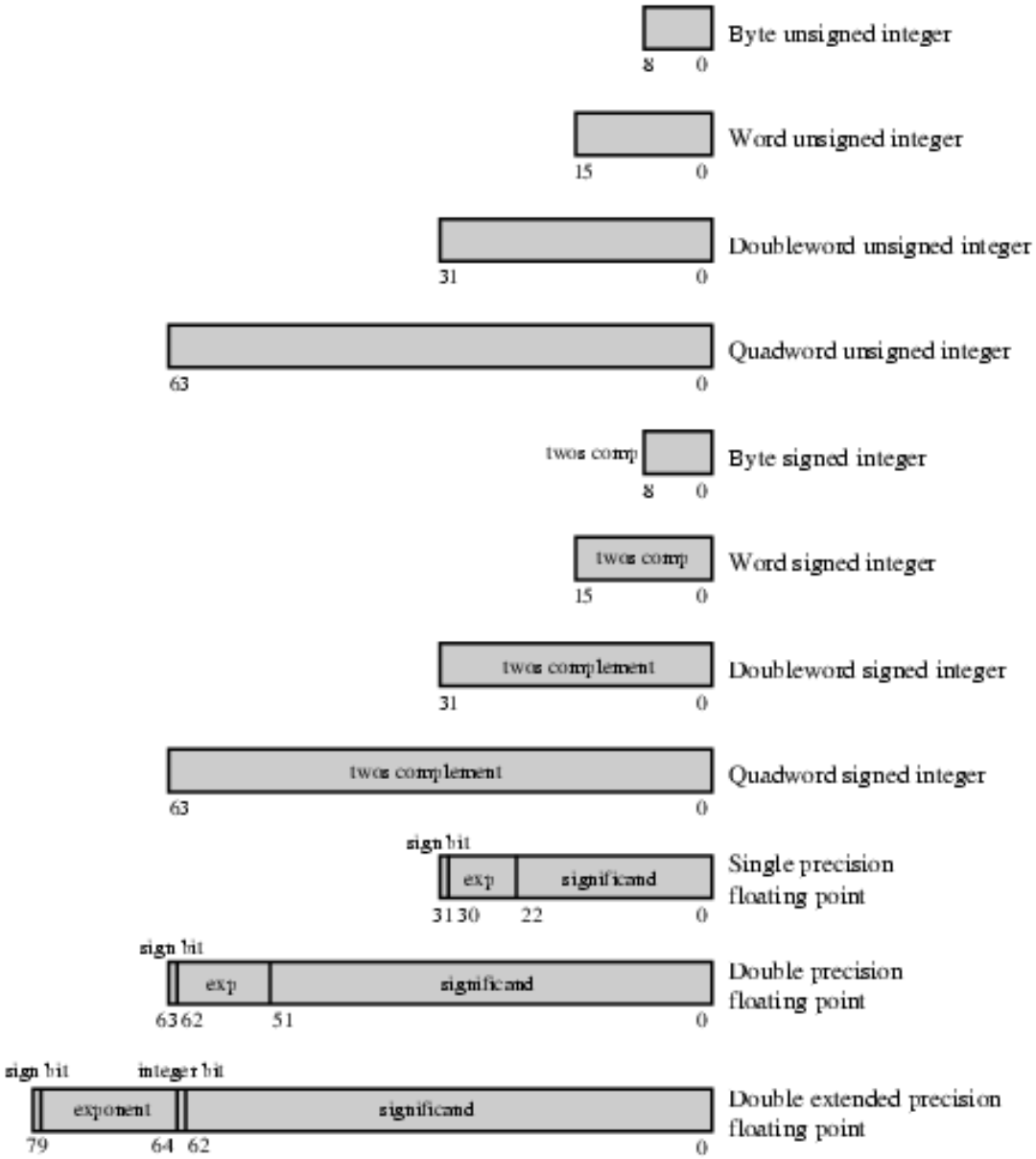
- 8 bit Byte (unsigned or signed)
- 16 bit word (unsigned or signed)
- 32 bit double word (unsigned or signed)
- 64 bit quad word (unsigned or signed)

- Memory is byte-addressable (Addressing is by 8 bit unit)
- A 32 bit double word is read at addresses divisible by 4

Specific Data Types

- General - arbitrary binary contents
- Integer - single binary value
- Ordinal - unsigned integer
- Unpacked BCD - One digit per byte
- Packed BCD - 2 BCD digits per byte
- Near Pointer - 32 bit offset within segment
- Bit field
- Byte String
- Floating Point

Pentium Floating Point Data Types



Pentium Byte Strings

- X86 processors have a set of 5 instructions called "string" instructions that can manipulate blocks of memory up to 2^{32} -1 bytes in length
- Blocks of memory can be manipulated as bytes, words, or dwords
- Operations:
  - —CMPS: mem-to-mem compare
  - —MOVS: mem-to-mem copy
  - —SCAS: scan memory for match to value in accumulator
  - —STOS: store accumulator to memory
  - —LODS: load accumulator from memory

Types of Operation

- Data Transfer
- Arithmetic
- Logical
- Conversion
- I/O
- System Control
- Transfer of Control

Data Transfer

- Specify
  - —Source
  - —Destination
  - —Amount of data
- May be different instructions for different movements source, destination, size
  - —e.g. IBM mainframes (370 series)
- Or one instruction and different addresses
  - —e.g. VAX, Pentium (MOV)

Example IBM EAS/390 Operations

Operation Mnemonic	Name	Number of Bits Transferred	Description
L	Load	32	Transfer from memory to register
LH	Load Halfword	16	Transfer from memory to register
LR	Load	32	Transfer from register to register
LER	Load (Short)	32	Transfer from floating-point register to floating-point register
LE	Load (Short)	32	Transfer from memory to floating-point register
LDR	Load (Long)	64	Transfer from floating-point register to floating-point register
LD	Load (Long)	64	Transfer from memory to floating-point register
ST	Store	32	Transfer from register to memory
STH	Store Halfword	16	Transfer from register to memory
STC	Store Character	8	Transfer from register to memory
STE	Store (Short)	32	Transfer from floating-point register to memory
STD	Store (Long)	64	Transfer from floating-point register to memory

Types of data transfer operations

Operation Name	Description
Move (transfer)	Transfer word or block from source to destination
Store	Transfer word from processor to memory
Load (fetch)	Transfer word from memory to processor
Exchange	Swap contents of source and destination
Clear (reset)	Transfer word of 0s to destination
Set	Transfer word of 1s to destination
Push	Transfer word from source to top of stack
Pop	Transfer word from top of stack to destination

Arithmetic

- Add, Subtract, Multiply, Divide
- Signed Integer
- Floating point ?
- May include
  - Increment (a++)
  - Decrement (a--)
  - Negate (-a)

Arithmetic Operations

Add	Compute sum of two operands
Subtract	Compute difference of two operands
Multiply	Compute product of two operands
Divide	Compute quotient of two operands
Absolute	Replace operand by its absolute value
Negate	Change sign of operand
Increment	Add 1 to operand
Decrement	Subtract 1 from operand
Compare	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome

- Comparison is usually considered an arithmetic operation – subtraction without storage of results

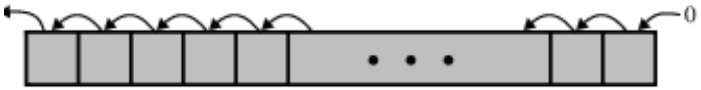
Logical

- Bitwise operations
- AND, OR, NOT

AND	Perform logical AND
OR	Perform logical OR
NOT (complement)	Perform logical NOT
Exclusive-OR	Perform logical XOR
Test	Test specified condition; set flag(s) based on outcome

Shift and Rotate Operations

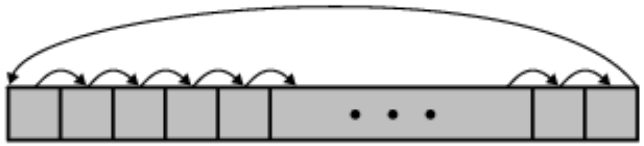
(a) Logical right shift



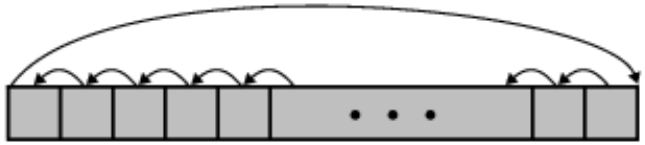
(b) Logical left shift

(c) Arithmetic right shift

(d) Arithmetic left shift



(e) Right rotate



(f) Left rotate

Translation and Conversion

- Conversion: convert one representation to another. Ex: x87 load and store instructions perform conversions from any numeric format to IEEE and vice versa
- Translate: table based translation
  - —S/390 translate instruction TR R1, R2, L
  - —R2 has address of a table of 8-bit codes
  - —L bytes at addr R1 are replaced by byte at table entry in R2 indexed by that byte
  - —Typically used for character code conversions
- Intel XLATE instruction has implied operands
  - —Bx points to the table
  - —AL contains the index and is replaced by BX[al]

Input/Output

- May be specific instructions (Pentium IN and OUT)
- May be done using data movement instructions (memory mapped)
- May be done by a separate controller (DMA)

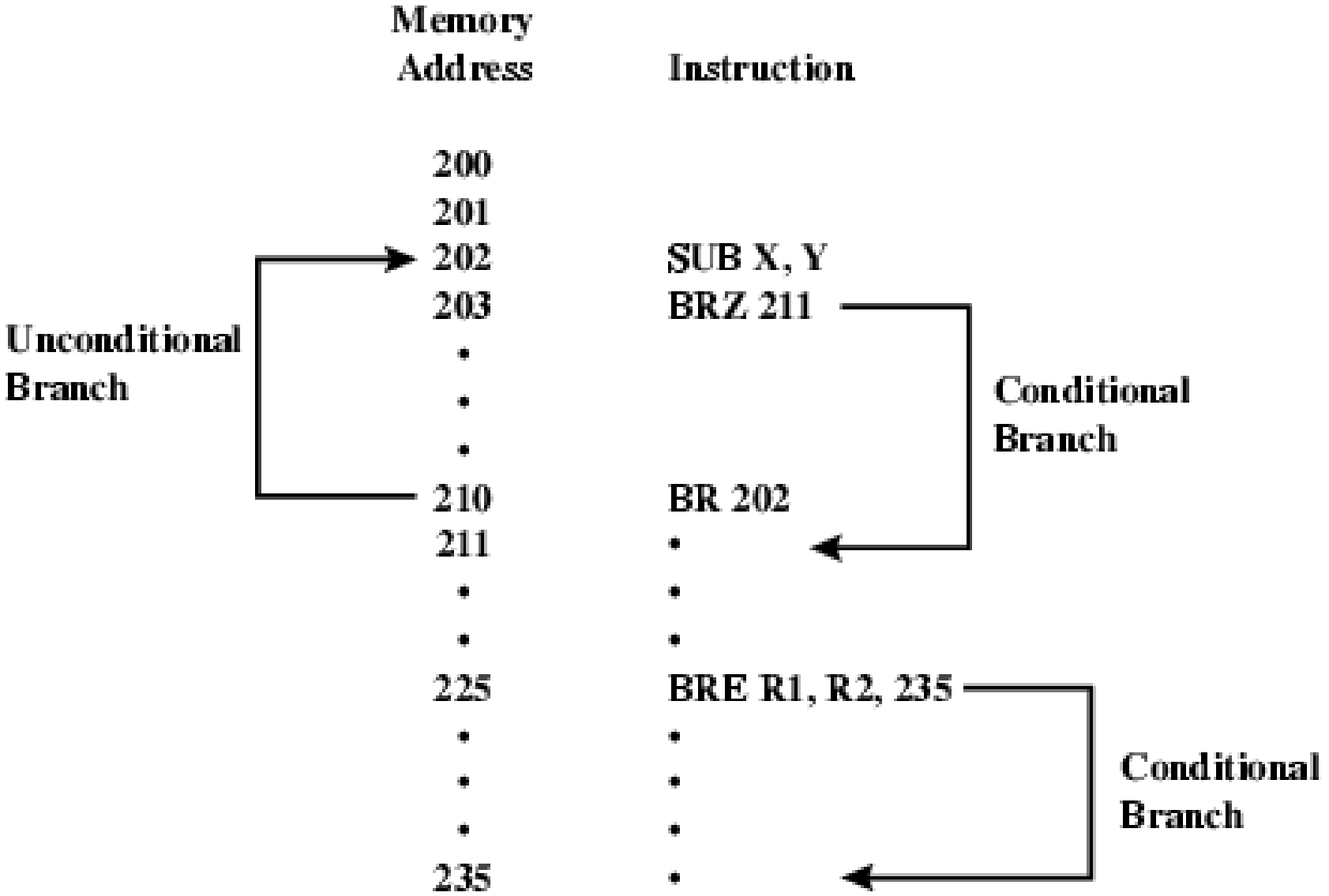
Systems Control

- Privileged instructions
- CPU needs to be in specific state
  - —Ring 0 on 80386+
  - —Kernel mode
- For operating systems use
- Examples: Pentium
  - — LGDTR (Load Global Descriptor Table Register)
  - — LAR (Load Access Rights)
  - — MOV to/from Control Register

Transfer of Control

- Branch (JMP in x86)
  - Unconditional == goto
  - Conditional
    - Can be based on flags (condition codes) or other tests (BRP X, BRN X, BRO X, BRZ X)
    - 3 address format can have instructions such as bre r1,r2, dest ;branch to dest if r1=r2
- Skip instructions
  - Skip the next instruction if condition is true
  - Implied address
  - May include other ops
    - **ISZ R1 ; increment R1 and skip if zero**
  - Not present in x86

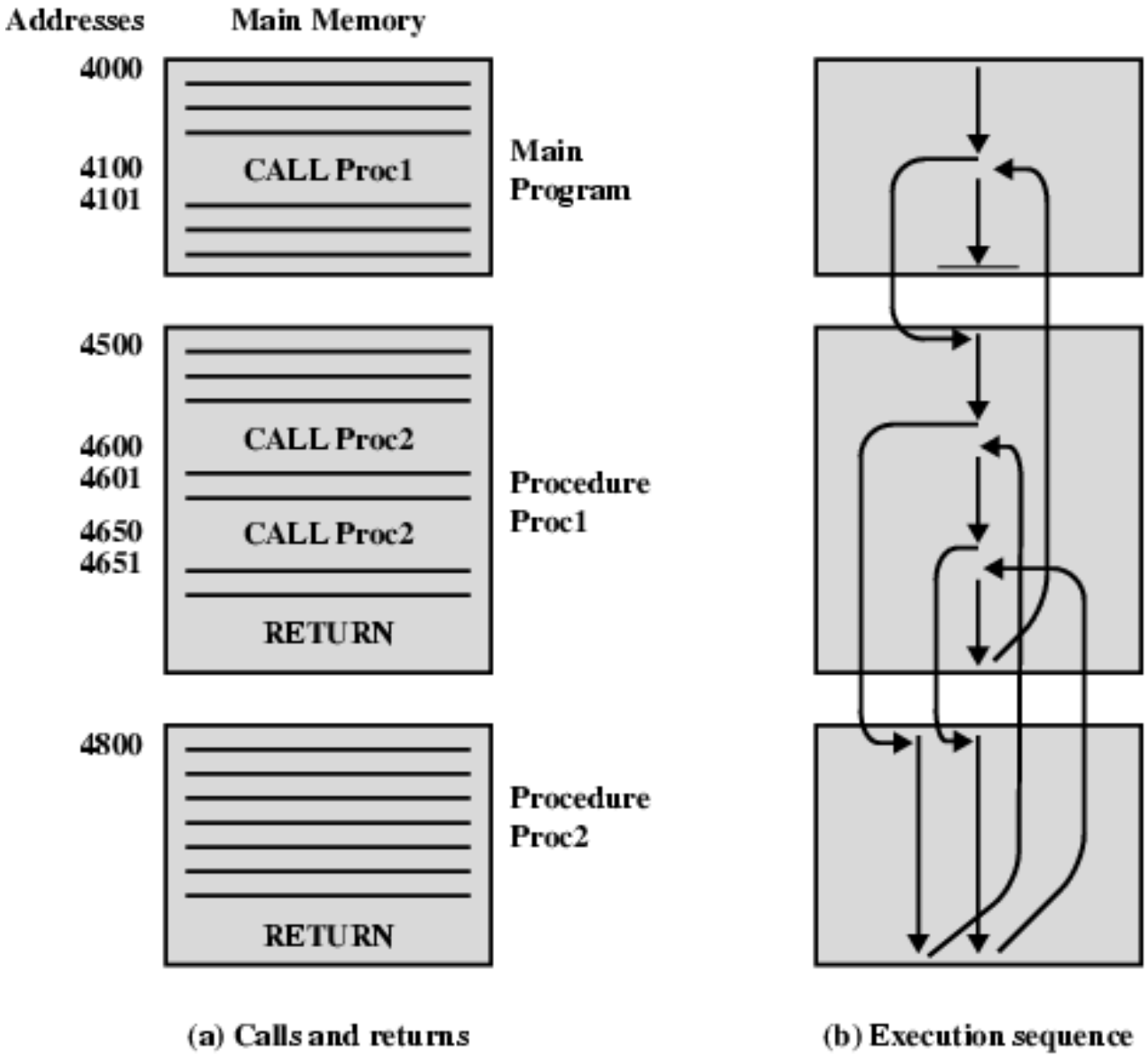
Branch Instruction



Procedure (Function) Calls

- Two instructions: CALL and RETURN
- Most machines use a stack mechanism:
  - CALL pushes return address on stack
  - RETURN pops stack into program counter
- Other approaches:
  - Store return address in special register
    - Rn←PC + d
    - PC ← X
  - Store return address at start of procedure
- These approaches all suffer from problems with re-entrancy (use a stack)

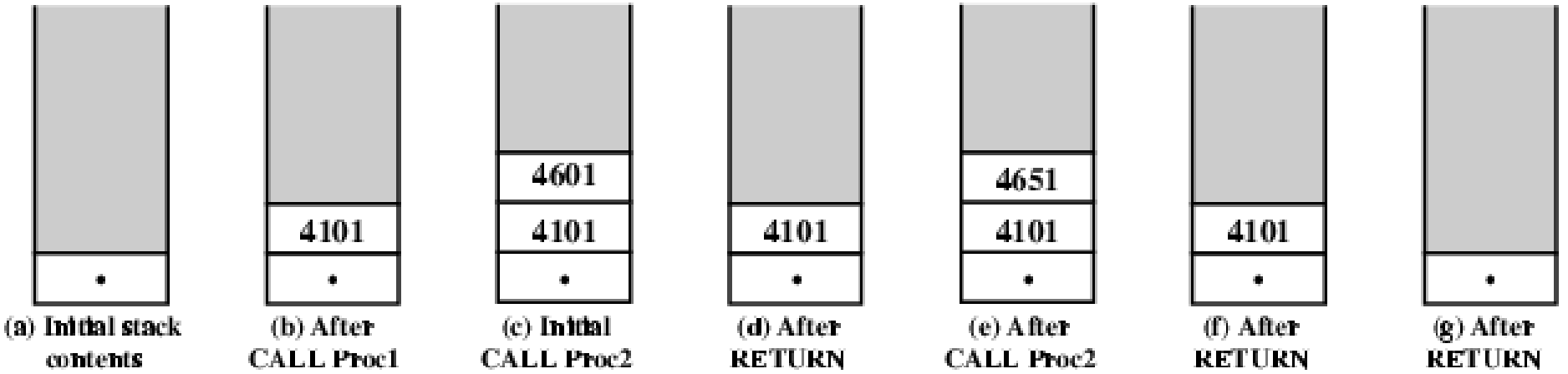
Nested Procedure Calls



Software Interrupts

- Many architectures provide a software interrupt
  - —Intel x86 provides for 224 possible software interrupts
- A software interrupt is an instruction, not an interrupt at all
- Effect is similar to a CALL instruction
- Return address and other data are pushed on the stack
- Address to which transfer is passed is located using the same vectoring mechanism that hardware interrupts use
- Intel x86 provides the IRET instruction for executing a RETURN from an interrupt

Use of Stack


4101
4601
4101
4101
4651
4101
4101
(a) Initial stack contents
(b) After CALL Proc1
(c) Initial CALL Proc2
(d) After RETURN
(e) After CALL Proc2
(f) After RETURN
(g) After RETURN

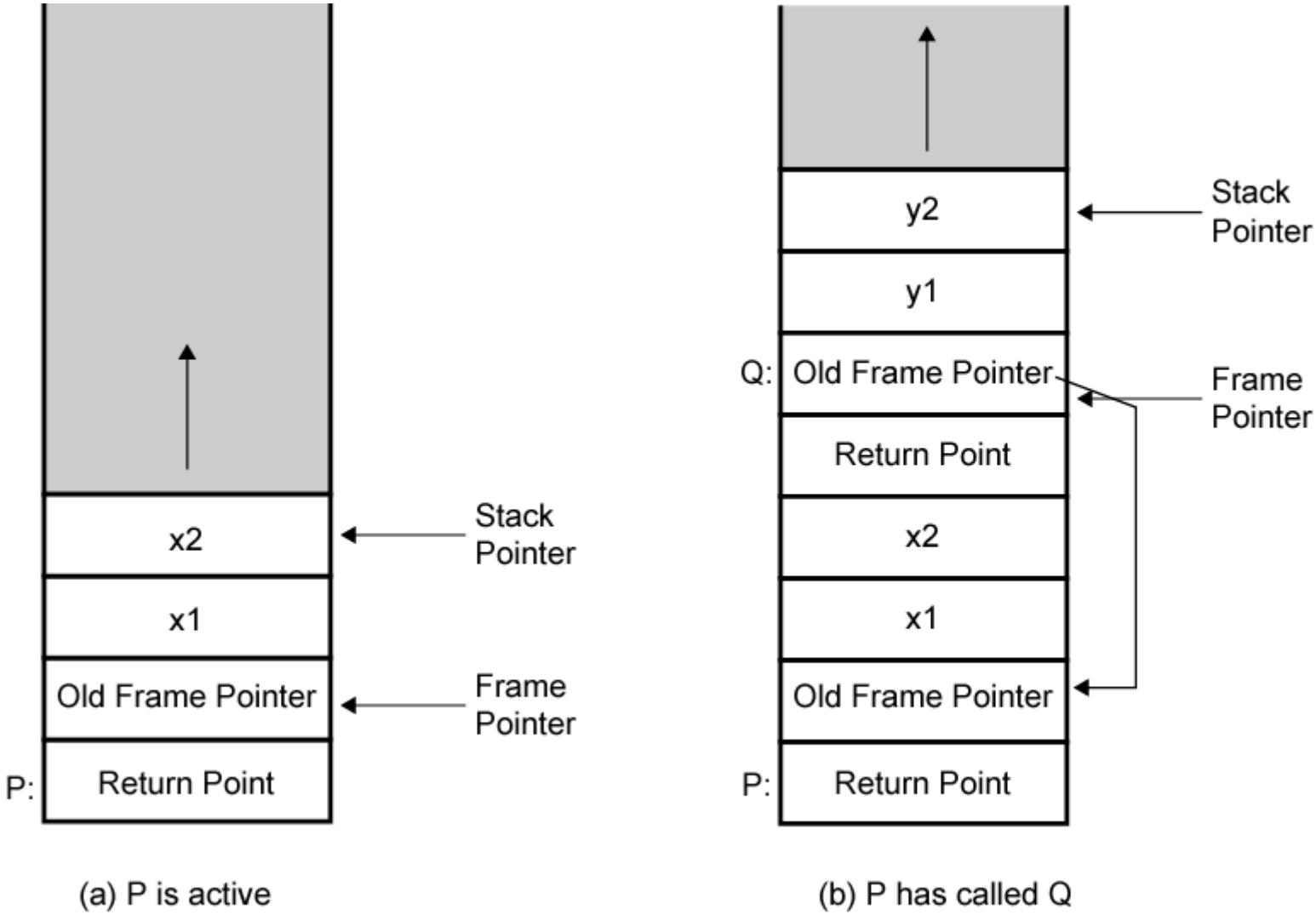
Procedures and Functions

- Most high-level languages make a syntactic distinction between Procedures (no return value) and Functions (returns a value as a result of the call)
- At the machine level this distinction does not exist
- We have only a CALL with RETURN to the instruction after the call

Stack Frames and Parameters

- Stacks provide a very flexible mechanism for parameter passing
- To call a procedure, first push parameters on the stack
- Issue the CALL instruction
- The procedure may also use the stack for storage of local (volatile) variables
- The entire structure (parameters, frame pointer, return address, local storage) is called a Stack Frame

Stack Frame Example (P calling Q)

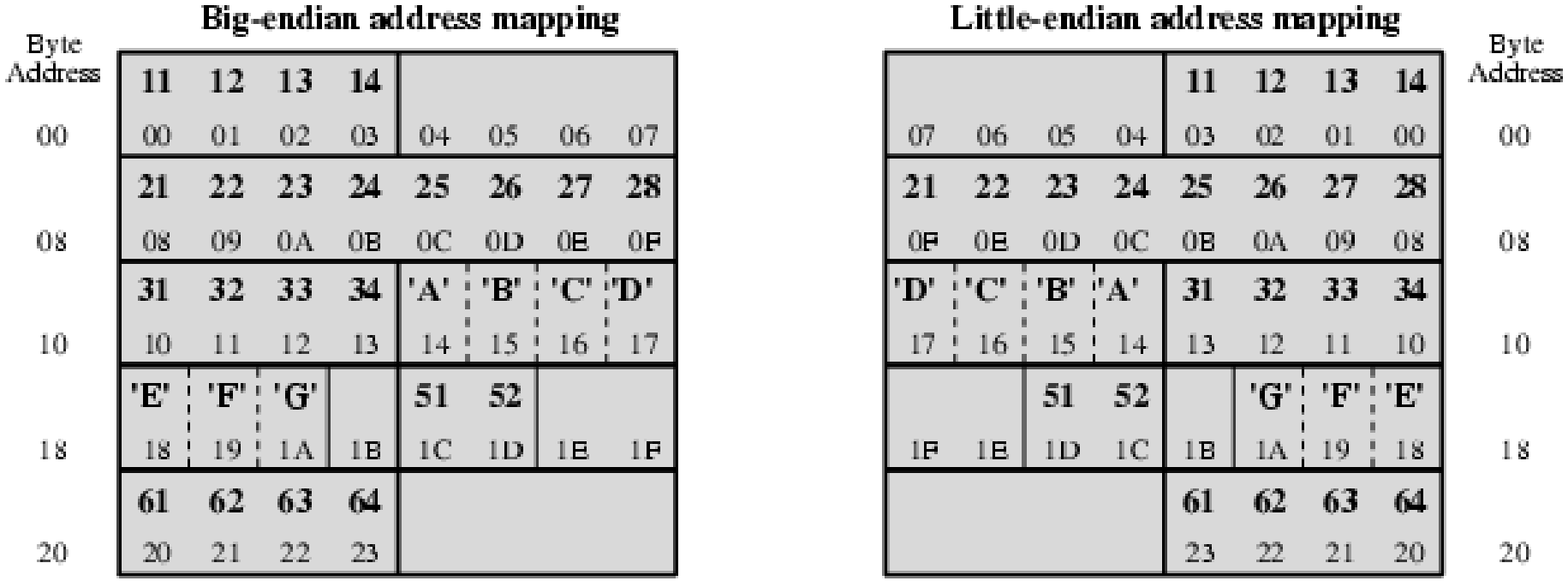


Byte Order Names

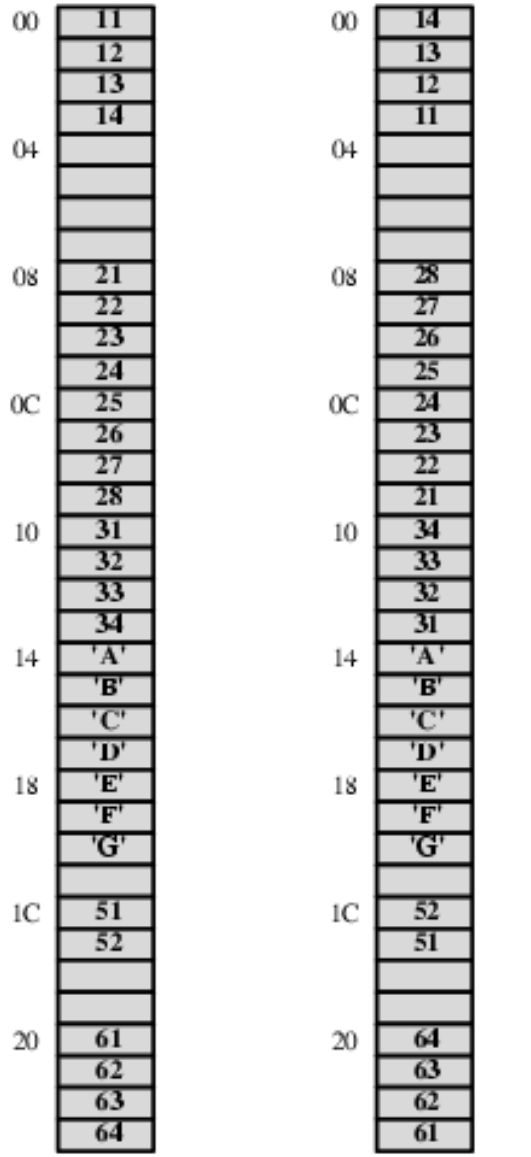
- The problem is called Endian
- The system on the left has the least significant byte in the lowest address
- This is called big-endian
- The system on the right has the least significant byte in the highest address
- This is called little-endian

Example of C Data Structure

```
struct{
   int      a;     //0x1112_1314                    word
   int      pad;   //
   double   b;     //0x2122_2324_2526_2728          doubleword
   char*    c;     //0x3132_3334                    word
   char     d[7];  //'A','B','C','D','E','F','G'   byte array
   short    e;     //0x5152                         halfword
   int      f;     //0x6161_6364                    word
} s;
```



Alternative View of Memory Map



(a) Big-endian (b) Little-endian

Standard...What Standard?

- Pentium (80x86), VAX are little-endian
- IBM 370, Motorola 680x0 (Mac), and most RISC are big-endian
- Internet is big-endian
  - —Makes writing Internet programs on PC more awkward!
  - —WinSock provides htoi and itoh (Host to Internet & Internet to Host) functions to convert