* Slides based on Tanenbaum, "Structured Computer Organization" 5e

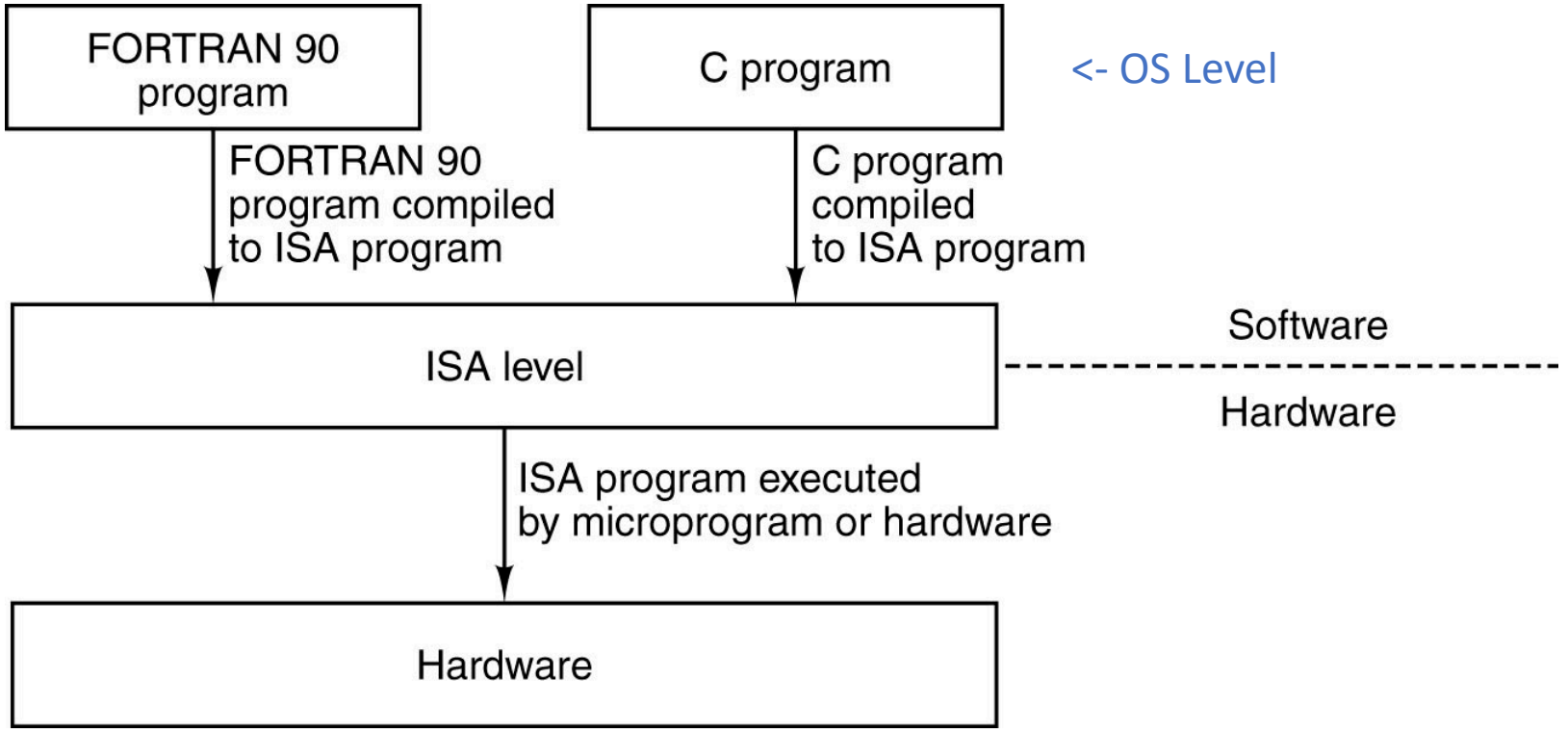
Instructor Set Architecture Level (Chapitre 5)

Francesco Bronzino

ArchiSys L3

ISA Level

The ISA level is the interface between the compilers and the hardware.



ISA Level

- Historically it was the first one to be developed
- Interface between software and hardware
  - Incorrectly referred to as “the architecture” or “assembly language”
  - Still an abstraction!
  - Most computers have to be able to execute programs written in multiple languages
- High-level languages be translated to a common intermediate form

ISA Level

- When a new machine comes along, the first question all the potential customers ask is: “Is it compatible with its predecessor?”
- The second is: “Can I run my old operating system on it?”
- The third is: “Will it run all my existing application programs unmodified?”
- If any of the answers are “no,” the designers will have a lot of explaining to do

Backward compatible!

ISA Level

- What makes a good ISA?
- A good ISA should define a set of instructions that can be implemented efficiently in current and future technologies
- A good ISA should provide a clean target for compiled code

Properties of the ISA

- The ISA level is defined by how the machine appears to a machine-language programmer
  - The ISA-level code is what a compiler outputs
- What the memory model is
- What registers there are
- What data types and instructions are available

What the ISA is not

- Whether the microarchitecture is microprogrammed or not
- Whether it is pipelined or not
- Whether it is superscalar or not
- Etc.

Not 100% true…

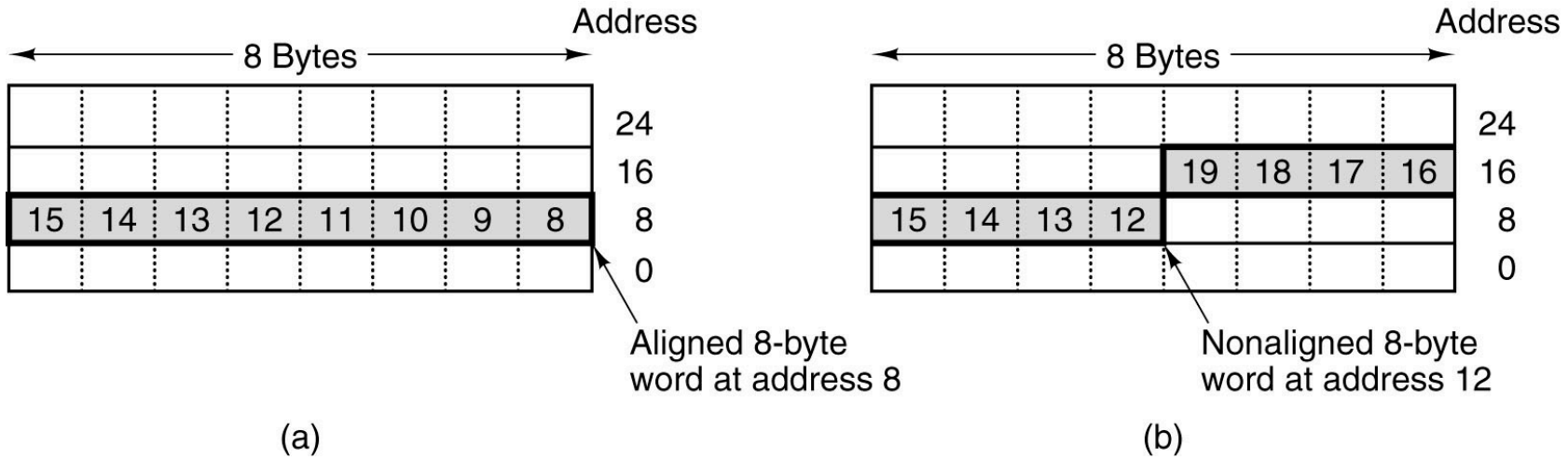
ISA Level

- Normative sections
  - Requirements
- Informative sections
  - Help the reader

Operating modes

- Kernel mode
  - Intended to run the operating system and allows all instructions to be executed
  - Example: access to cache
- User mode
  - Intended to run application programs and does not permit certain sensitive instructions
- We focus on user mode

Memory Models



An 8-byte word in a little-endian memory.
(a) Aligned. (b) Not aligned.

Memory Models: Potential Problems

- Cost of different reference sizes
  - 1 byte vs 4 bytes
- Alignment
- Instructions ordering
  - What happens if you run a STORE followed by a LOAD?
    - Forced ordering
    - SYNC instruction

Registers

- Some registers are visible at the ISA level
- Which ones?
- Registers visible at the microarchitecture level are not visible at the ISA level
  - TOS and MAR
- Special-purpose registers vs general-purpose registers

Instructions

- The main feature of the ISA level is its set of machine instructions.
- Data movement instructions
  - LOAD, STORE, MOVE
- Arithmetic instructions
- Boolean instructions
- Comparative instructions

ISA discussed in the chapter

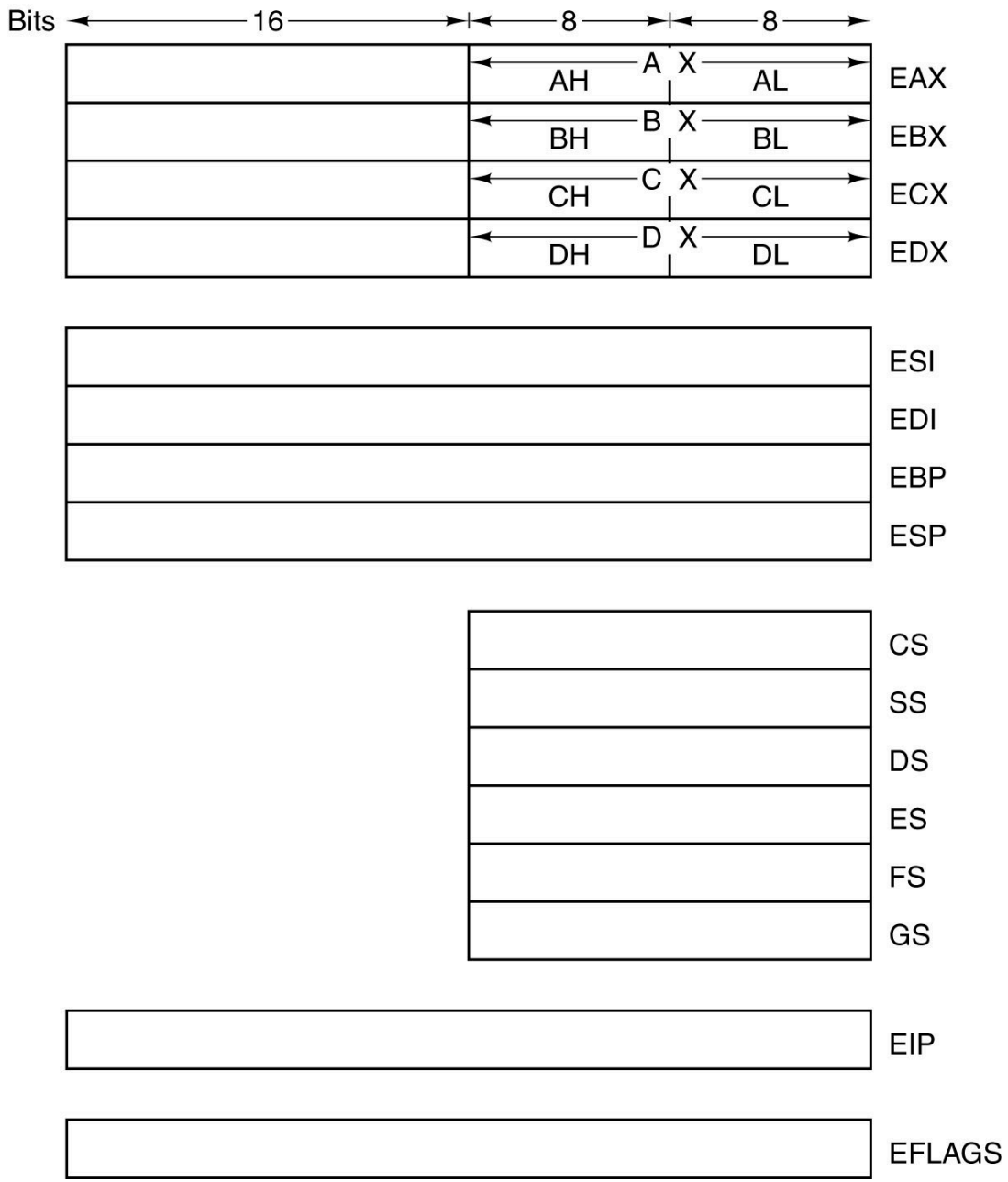
- Core i7
- ARM v7
- AVR 8-bit

Core i7 operating modes

- Real mode
- Virtual 8086 mode
- Protected mode

Overview of the Core i7 ISA Level

The Core i7's primary registers.



Overview of the 7 ARM

Register	Alt. name	Function
R0–R3	A1–A4	Holds parameters to the procedure being called
R4–R11	V1–V8	Holds local variables for the current procedure
R12	IP	Intraprocedure call register (for 32-bit calls)
R13	SP	Stack pointer
R14	LR	Link register (return address for current function)
R15	PC	Program counter

Data Types

- A variety of different data types are used
- Key issue: hardware support for a particular data type?
- Use case: verify the federal debt (how much the U.S. government owes everyone)
- 32-bit arithmetic would not work here because the numbers involved are larger than 232 (about 4 billion)
- One solution is to use two 32-bit integers to represent each number, giving 64 bits in all.
- If the machine does not support double-precision numbers, all arithmetic on them will have to be done in software

Data Types on the Core i7

Type	8 Bits	16 Bits	32 Bits	64 Bits
Signed integer	×	×	×	× (64-bit)
Unsigned integer	×	×	×	× (64-bit)
Binary coded decimal integer	×			
Floating point			×	×

The Core i7 numeric data types.

Supported types are marked with ×

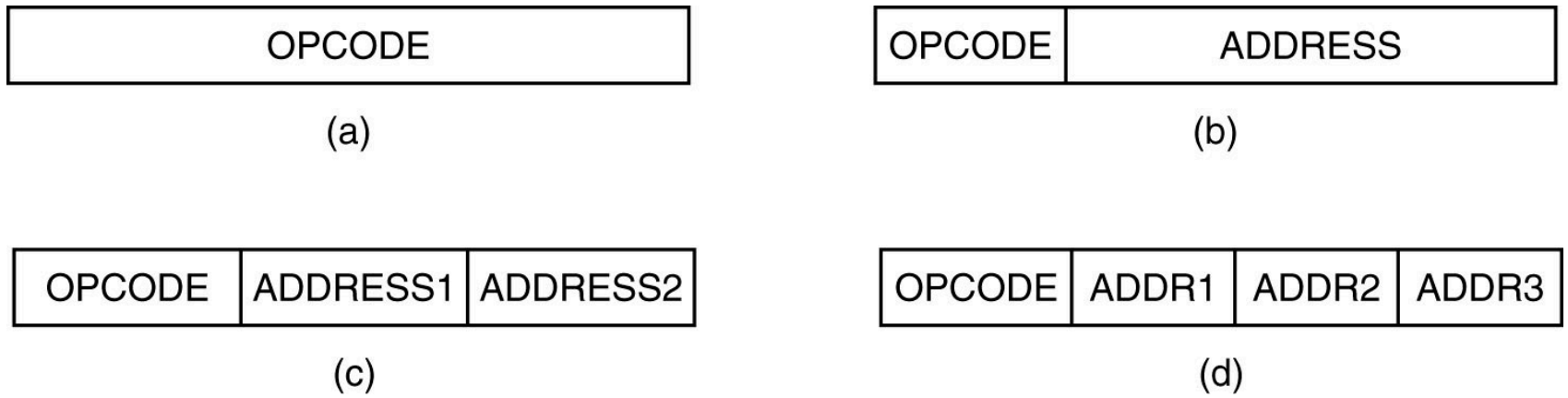
How many types to support?

- Some programming languages (COBOL) allow decimal numbers as a data type
- Machines that wish to be COBOL-friendly support decimal numbers in hardware
  - Encode a decimal digit in 4 bits and then packing two decimal digits per byte
- Binary arithmetic does not work correctly on packed decimal numbers
  - Special decimal arithmetic-correction instructions are needed

Non-numeric data types

- Example?
- Characters
- Boolean
- Not uncommon for the ISA level to have special instruction

Instruction Formats

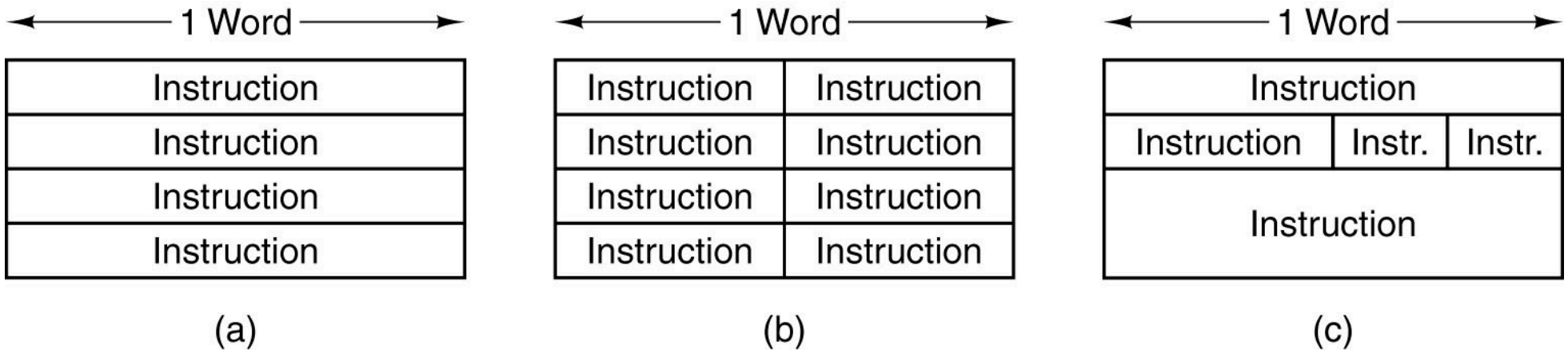


Four common instruction formats:
(a) Zero-address instruction. (b) One-address instruction
(c) Two-address instruction. (d) Three-address instruction.

Instruction Formats

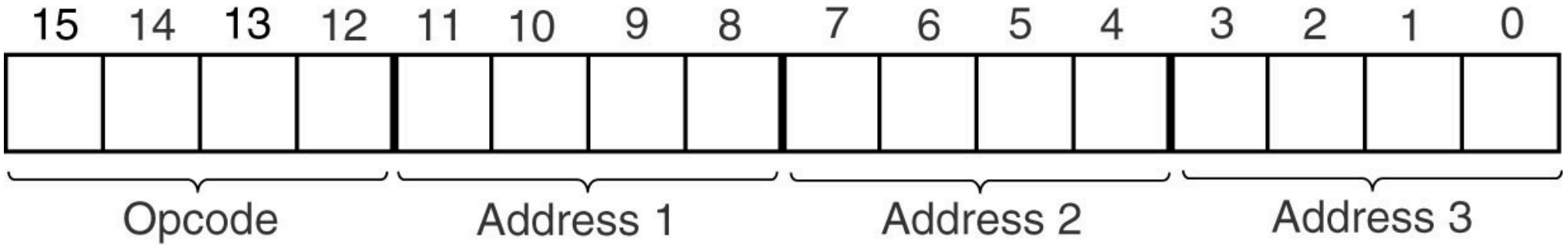
- Is designing instruction formats easy?
- NO!
- Many tradeoffs to be considered
- The efficiency of a particular ISA is highly dependent on the technology with which the computer is to be implemented
  - If memory accesses are fast -> stack- based design
  - If they are slow -> many registers
  - If the bandwidth of an instruction cache is *t* bps and the average instruction length is *r* bits, the cache can deliver at most *t/r* instructions per second

Instruction Formats



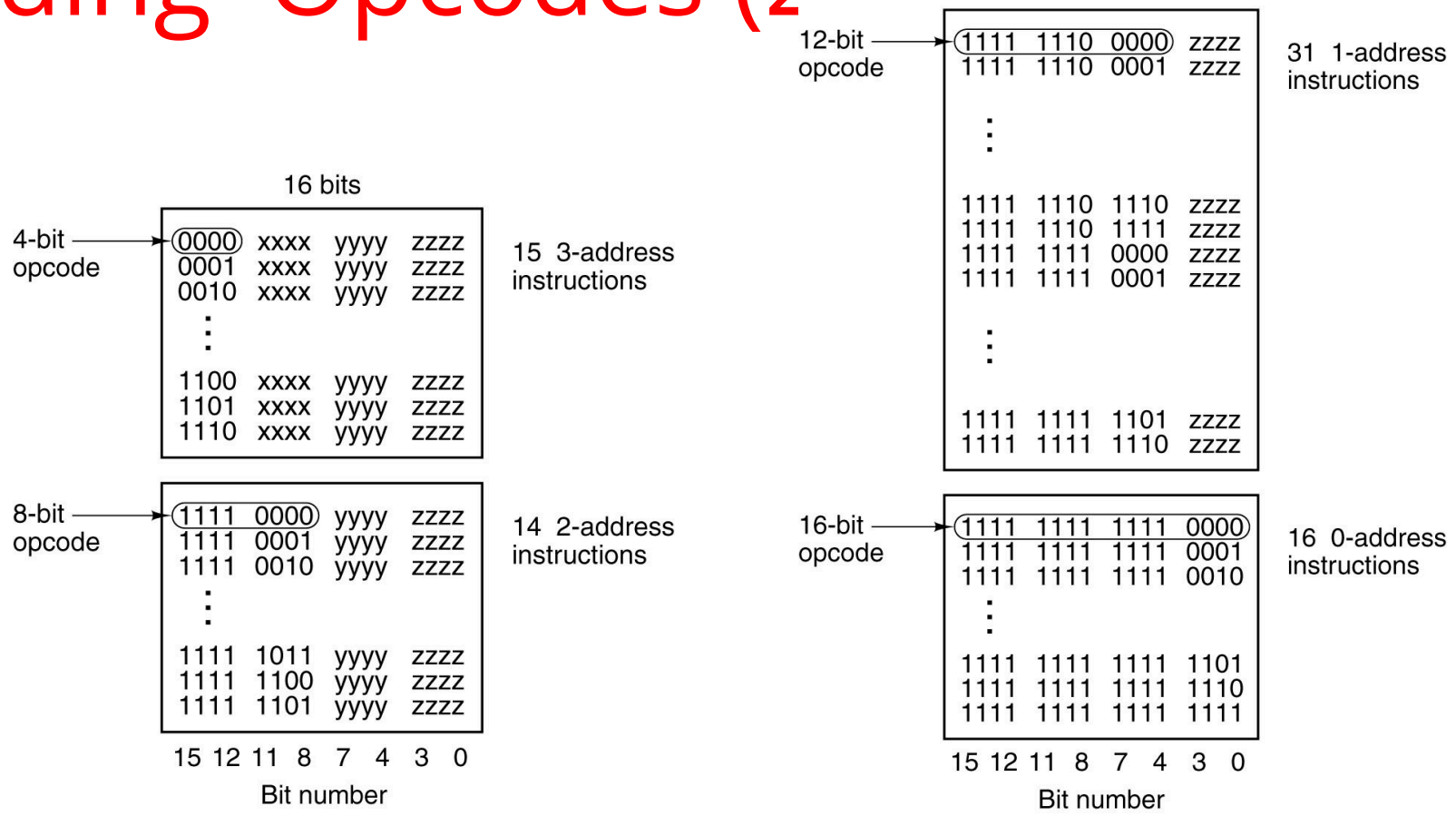
Some possible relationships between instruction and word length.

Expanding Opcodes (1)



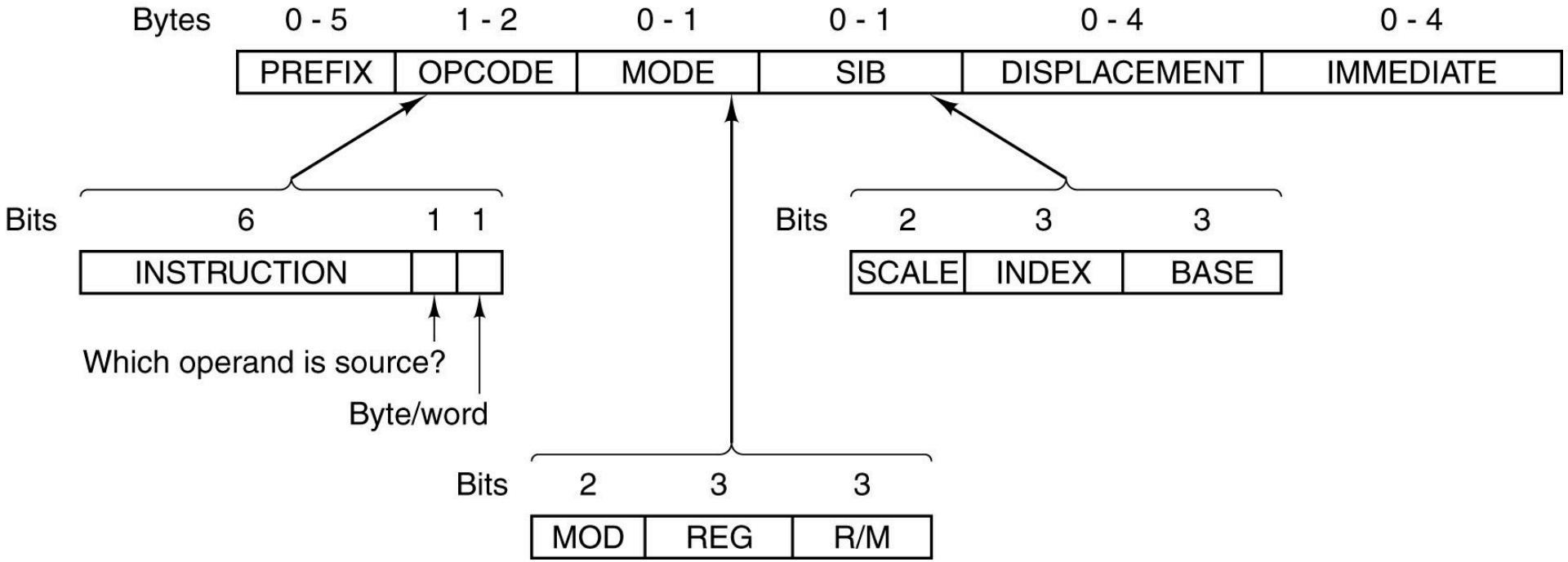
An instruction with a 4-bit opcode and three 4-bit address fields.

Expanding Opcodes (2)



An expanding opcode allowing 15 three-address instructions, 14 two-address instructions, 31 one-address instructions, and 16 zero-address instructions. The fields marked *xxxx*, *yyyy*, and *zzzz* are 4-bit address fields.

The Core i7 Instruction Formats



The Core i7 instruction formats.

Addressing Modes

- Most instructions have operands, so some way is needed to specify where they are
- Immediate Addressing
- Direct Addressing
- Register Addressing
- Register Indirect Addressing
- Indexed Addressing
- Based-Indexed Addressing
- Stack Addressing

Addressing

An immediate instruction for loading 4 into register 1.

```
        MOV R1,#0          ; accumulate the sum in R1, initially 0
        MOV R2,#A          ; R2 = address of the array A
        MOV R3,#A+4096     ; R3 = address of the first word beyond A
LOOP:   ADD R1,(R2)        ; register indirect through R2 to get operand
        ADD R2,#4          ; increment R2 by one word (4 bytes)
        CMP R2,R3          ; are we done yet?
        BLT LOOP           ; if R2 < R3, we are not done, so continue
```

Register Indirect Addressing: a generic assembly program for computing the sum of the elements of an array.

Addressing

- How about direct addressing?
- Global variables

Indexed Addressing

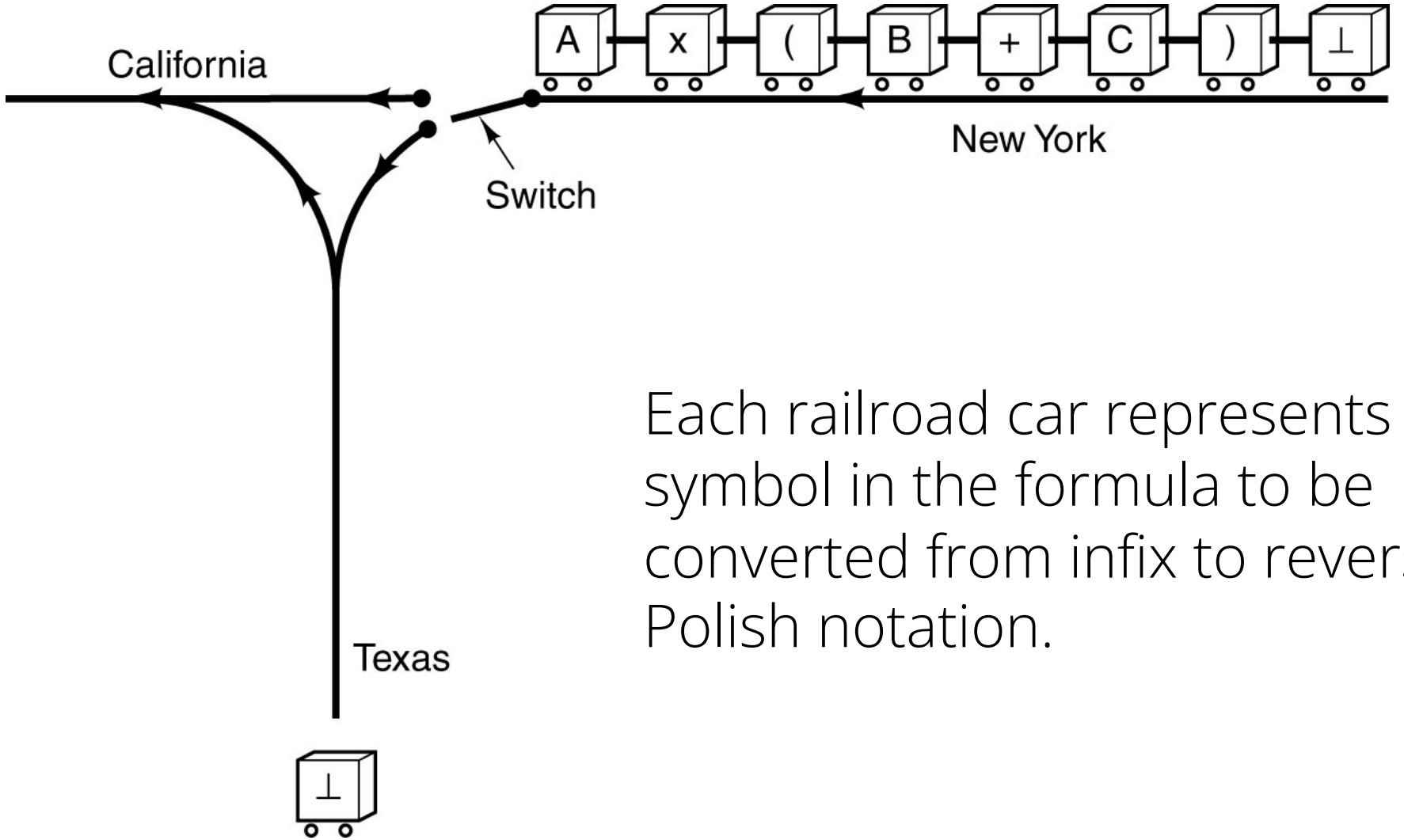
```
        MOV R1,#0          ; accumulate the OR in R1, initially 0
        MOV R2,#0          ; R2 = index, i, of current product: A[i] AND B[i]
        MOV R3,#4096       ; R3 = first index value not to use
LOOP:   MOV R4,A(R2)       ; R4 = A[i]
        AND R4,B(R2)       ; R4 = A[i] AND B[i]
        OR R1,R4           ; OR all the Boolean products into R1
        ADD R2,#4          ; i = i + 4 (step in units of 1 word = 4 bytes)
        CMP R2,R3          ; are we done yet?
        BLT LOOP           ; if R2 < R3, we are not done, so continue
```

A generic assembly program for computing the OR of A_i AND B_i for two 1024-element arrays.

Indexed Addressing

A possible representation of MOV R4,A(R2).

Reverse Polish Notation (1)



Each railroad car represents one symbol in the formula to be converted from infix to reverse Polish notation.

Reverse Polish Notation (2)

Most recently arrived car on the Texas line \ Car at the switch	⊥	+	−	x	/	(	)
⊥	4	1	1	1	1	1	5
+	2	2	2	1	1	1	2
−	2	2	2	1	1	1	2
x	2	2	2	2	2	1	2
/	2	2	2	2	2	1	2
(	5	1	1	1	1	1	3

Decision table used by the infix-to-reverse Polish notation algorithm

Reverse Polish Notation (3)

Infix	Reverse Polish notation
A + B × C	A B C × +
A × B + C	A B × C +
A × B + C × D	A B × C D × +
(A + B) / (C – D)	A B + C D – /
A × B / C	A B × C /
((A + B) × C + D)/(E + F + G)	A B + C × D + E F + G + /

Some examples of infix expressions and their reverse Polish notation equivalents.

Evaluation of Reverse Polish notation Formulas

Step	Remaining string	Instruction	Stack
1	8 2 5 × + 1 3 2 × + 4 – /	BIPUSH 8	8
2	2 5 × + 1 3 2 × + 4 – /	BIPUSH 2	8, 2
3	5 × + 1 3 2 × + 4 – /	BIPUSH 5	8, 2, 5
4	× + 1 3 2 × + 4 – /	IMUL	8, 10
5	+ 1 3 2 × + 4 – /	IADD	18
6	1 3 2 × + 4 – /	BIPUSH 1	18, 1
7	3 2 × + 4 – /	BIPUSH 3	18, 1, 3
8	2 × + 4 – /	BIPUSH 2	18, 1, 3, 2
9	× + 4 – /	IMUL	18, 1, 6
10	+ 4 – /	IADD	18, 7
11	4 – /	BIPUSH 4	18, 7, 4
12	– /	ISUB	18, 3
13	/	IDIV	6

Use of a stack to evaluate a reverse Polish notation formula.

The Core i7 Addressing Modes

	MOD			
R/M	**00**	**01**	**10**	**11**
000	M[EAX]	M[EAX + OFFSET8]	M[EAX + OFFSET32]	EAX or AL
001	M[ECX]	M[ECX + OFFSET8]	M[ECX + OFFSET32]	ECX or CL
010	M[EDX]	M[EDX + OFFSET8]	M[EDX + OFFSET32]	EDX or DL
011	M[EBX]	M[EBX + OFFSET8]	M[EBX + OFFSET32]	EBX or BL
100	SIB	SIB with OFFSET8	SIB with OFFSET32	ESP or AH
101	Direct	M[EBP + OFFSET8]	M[EBP + OFFSET32]	EBP or CH
110	M[ESI]	M[ESI + OFFSET8]	M[ESI + OFFSET32]	ESI or DH
111	M[EDI]	M[EDI + OFFSET8]	M[EDI + OFFSET32]	EDI or BH

The Core i7 32-bit addressing modes. M[*x*] is the memory word at *x*.

Loop Control

```
        i = 1;

L1:     first-statement;
        .
        .
        .
        last-statement;
        i = i + 1;
        if (i < n) goto L1;
```

(a)

```
        i = 1;
L1:     if (i > n) goto L2;
        first-statement;
        .
        .
        .
        last-statement
        i = i + 1;
        goto L1;
L2:
```

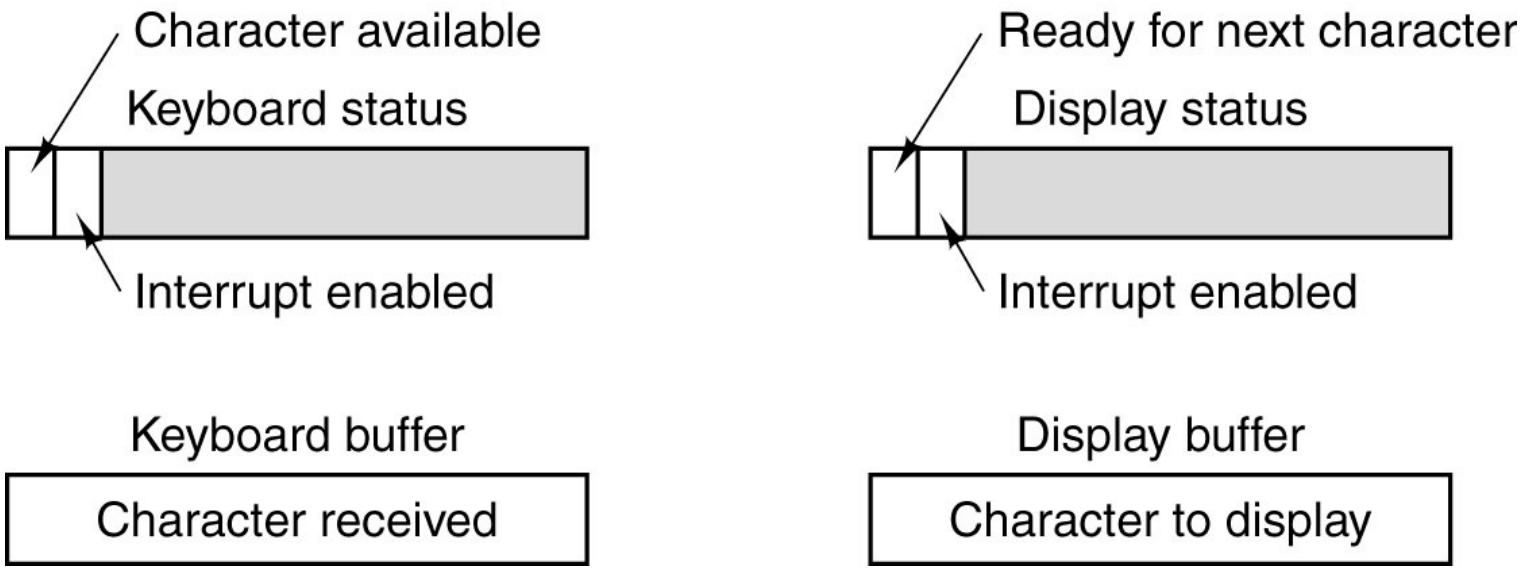
(b)

(a) Test-at-the-end loop.
(b) Test-at-the-beginning loop.

Input/Output

- Programmed I/O with busy waiting.
- Interrupt-driven I/O.
- DMA I/O.

Input/Output



Device registers for a simple terminal.

Input/Output

```
// Output a block of data to the device
int status, i, ready;

for (i = 0; i < count; i++) {
    do {
        status = in(display_status_reg);        // get status
        ready = (status >> 7) & 0x01;           // isolate ready bit
    } while (ready != 1);
    out(display_buffer_reg, buf[i]);
}
}
```

An example of programmed I/O.

The Core i7 Instructions

Moves

MOV DST,SRC	Move SRC to DST
PUSH SRC	Push SRC onto the stack
POP DST	Pop a word from the stack to DST
XCHG DS1,DS2	Exchange DS1 and DS2
LEA DST,SRC	Load effective addr of SRC into DST
CMOVcc DST,SRC	Conditional move

Arithmetic

ADD DST,SRC	Add SRC to DST
SUB DST,SRC	Subtract SRC from DST
MUL SRC	Multiply EAX by SRC (unsigned)
IMUL SRC	Multiply EAX by SRC (signed)
DIV SRC	Divide EDX:EAX by SRC (unsigned)
IDIV SRC	Divide EDX:EAX by SRC (signed)
ADC DST,SRC	Add SRC to DST, then add carry bit
SBB DST,SRC	Subtract SRC & carry from DST
INC DST	Add 1 to DST
DEC DST	Subtract 1 from DST
NEG DST	Negate DST (subtract it from 0)

A selection of the Core i7 integer instructions.

The Core i7 Instructions

Binary coded decimal

DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

Boolean

AND DST,SRC	Boolean AND SRC into DST
OR DST,SRC	Boolean OR SRC into DST
XOR DST,SRC	Boolean Exclusive OR SRC to DST
NOT DST	Replace DST with 1's complement

Shift/rotate

SAL/SAR DST,#	Shift DST left/right # bits
SHL/SHR DST,#	Logical shift DST left/right # bits
ROL/ROR DST,#	Rotate DST left/right # bits
RCL/RCR DST,#	Rotate DST through carry # bits

A selection of the Core i7 integer instructions.

The Core i7 Instructions

Test/compare

TEST SRC1,SRC2	Boolean AND operands, set flags
CMP SRC1,SRC2	Set flags based on SRC1 - SRC2

Transfer of control

JMP ADDR	Jump to ADDR
Jxx ADDR	Conditional jumps based on flags
CALL ADDR	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT n	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

Strings

LODS	Load string
STOS	Store string
MOVS	Move string
CMPS	Compare two strings
SCAS	Scan Strings

A selection of the Core i7 integer instructions.

The Core i7 Instructions

Condition codes

STC	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS reg
STI	Set interrupt bit in EFLAGS register
CLI	Clear interrupt bit in EFLAGS reg
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
SAHF	Store AH in EFLAGS register

Miscellaneous

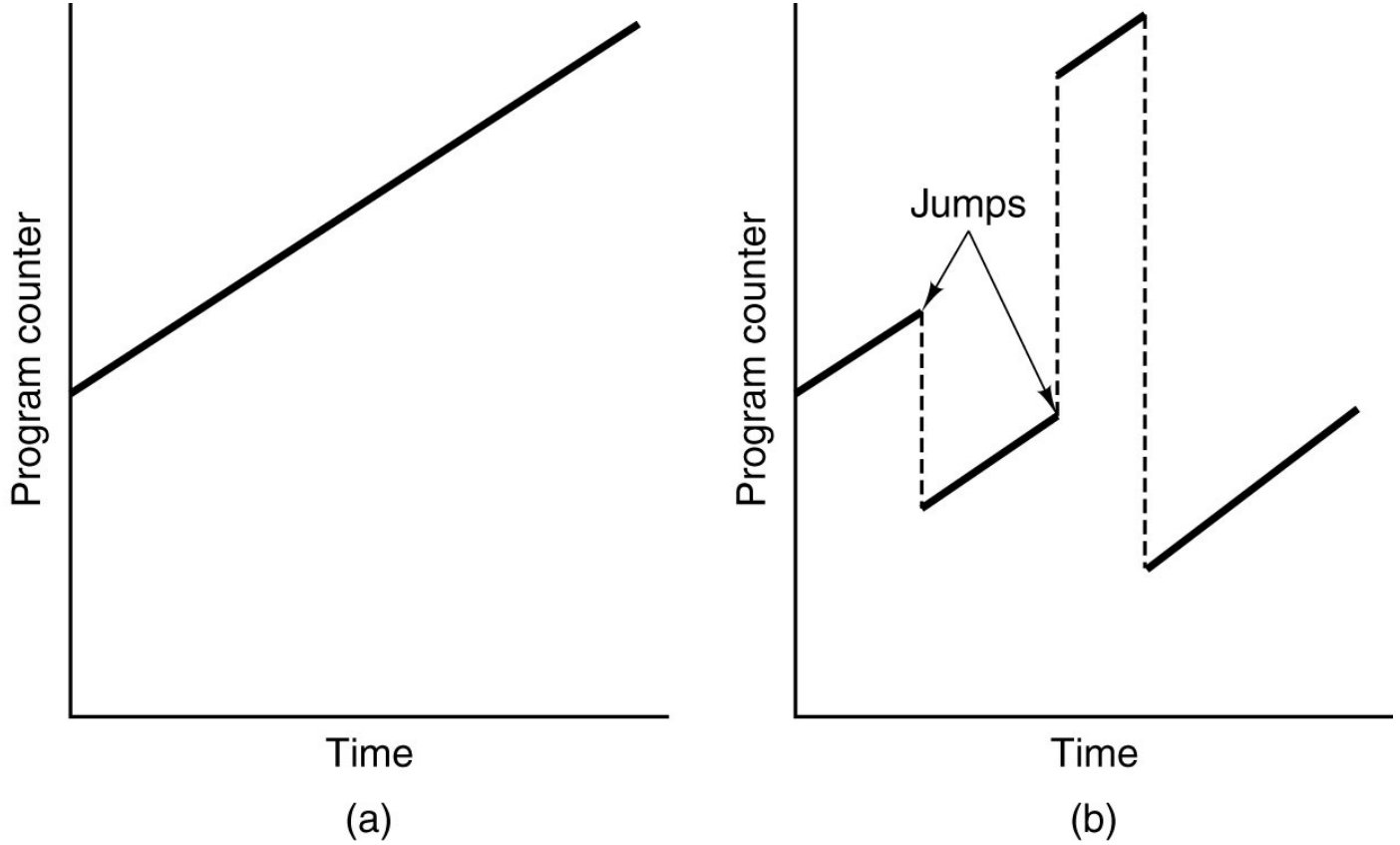
SWAP DST	Change endianness of DST
CWQ	Extend EAX to EDX:EAX for division
CWDE	Extend 16-bit number in AX to EAX
ENTER SIZE,LV	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN AL,PORT	Input a byte from PORT to AL
OUT PORT,AL	Output a byte from AL to PORT
WAIT	Wait for an interrupt

SRC = source
DST = destination

# = shift/rotate count
LV = # locals

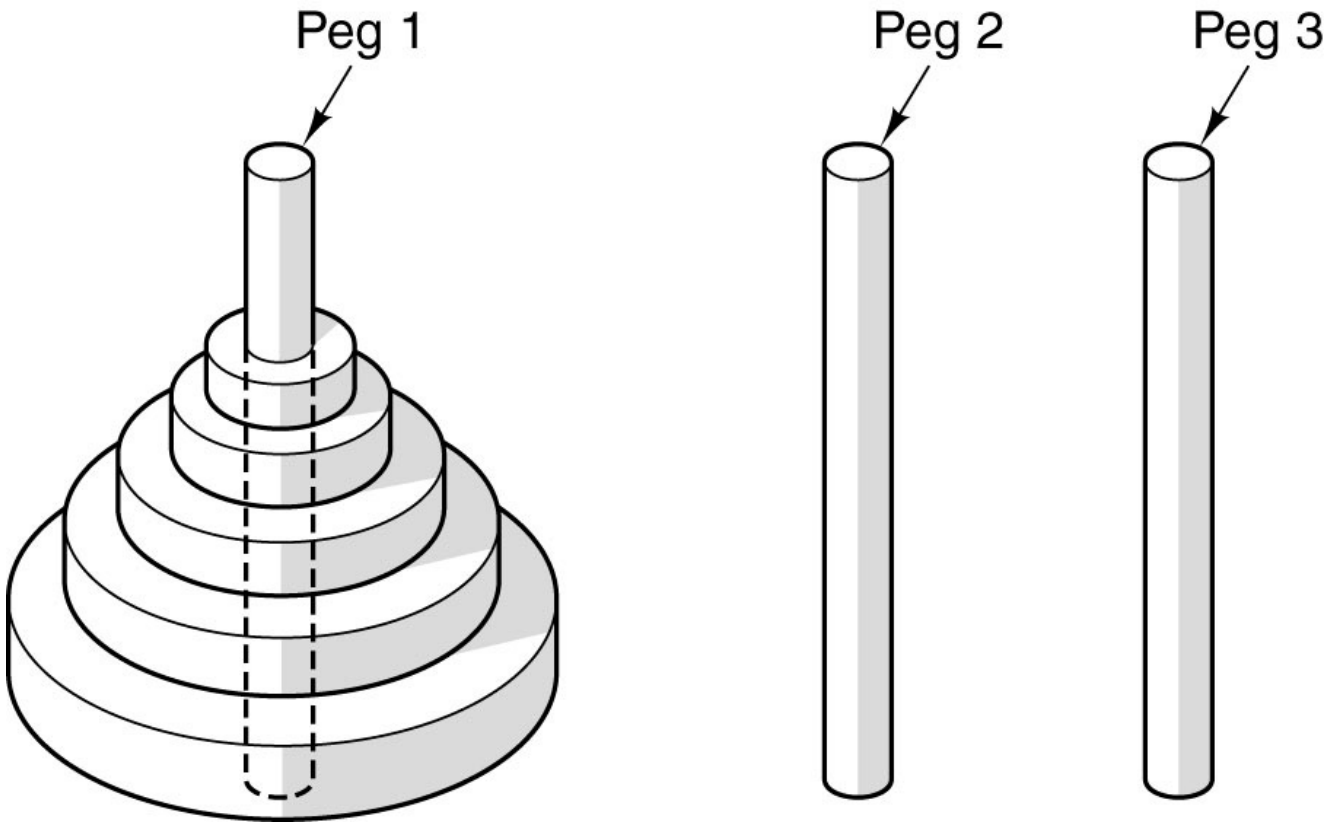
A selection of the Core i7 integer instructions.

Sequential Flow of Control and Branches



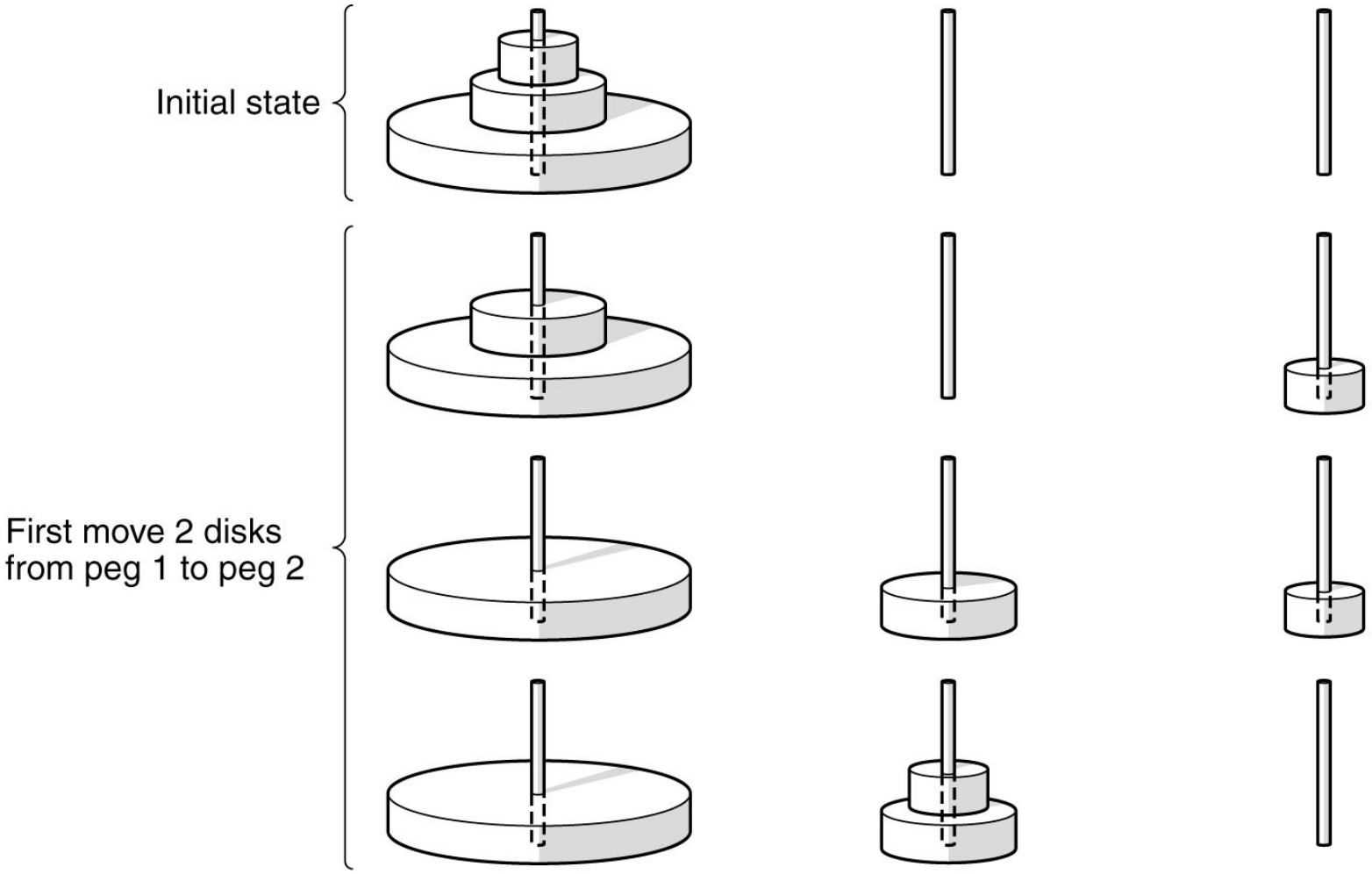
Program counter as a function of time (smoothed).
(a) Without branches. (b) With branches.

Recursive Procedures (1)



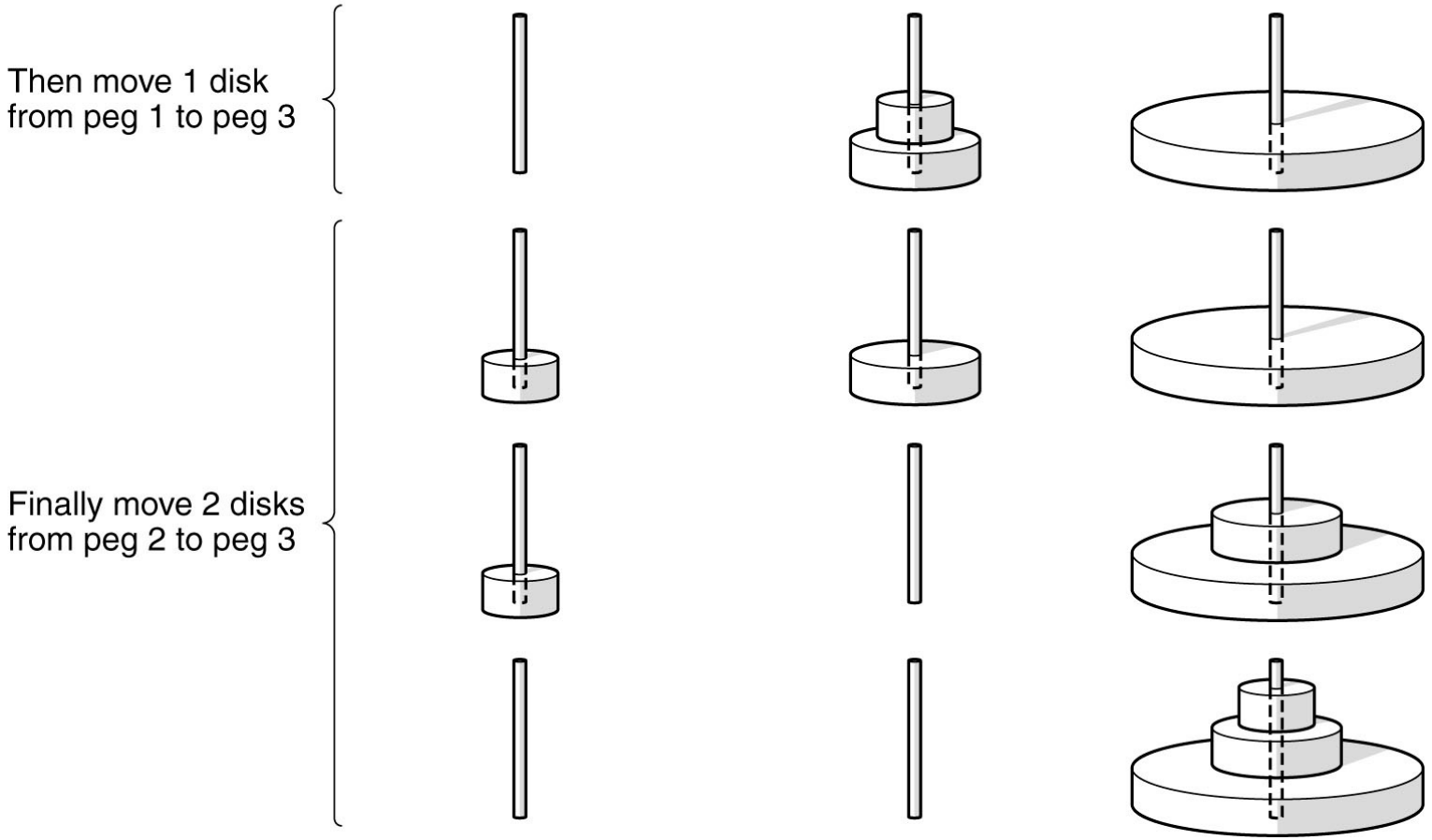
Initial configuration for the Towers of Hanoi problem for five disks.

Recursive Procedures (2)



The steps required to solve the Towers of Hanoi for three disks.

Recursive Procedures (3)



The steps required to solve the Towers of Hanoi for three disks.

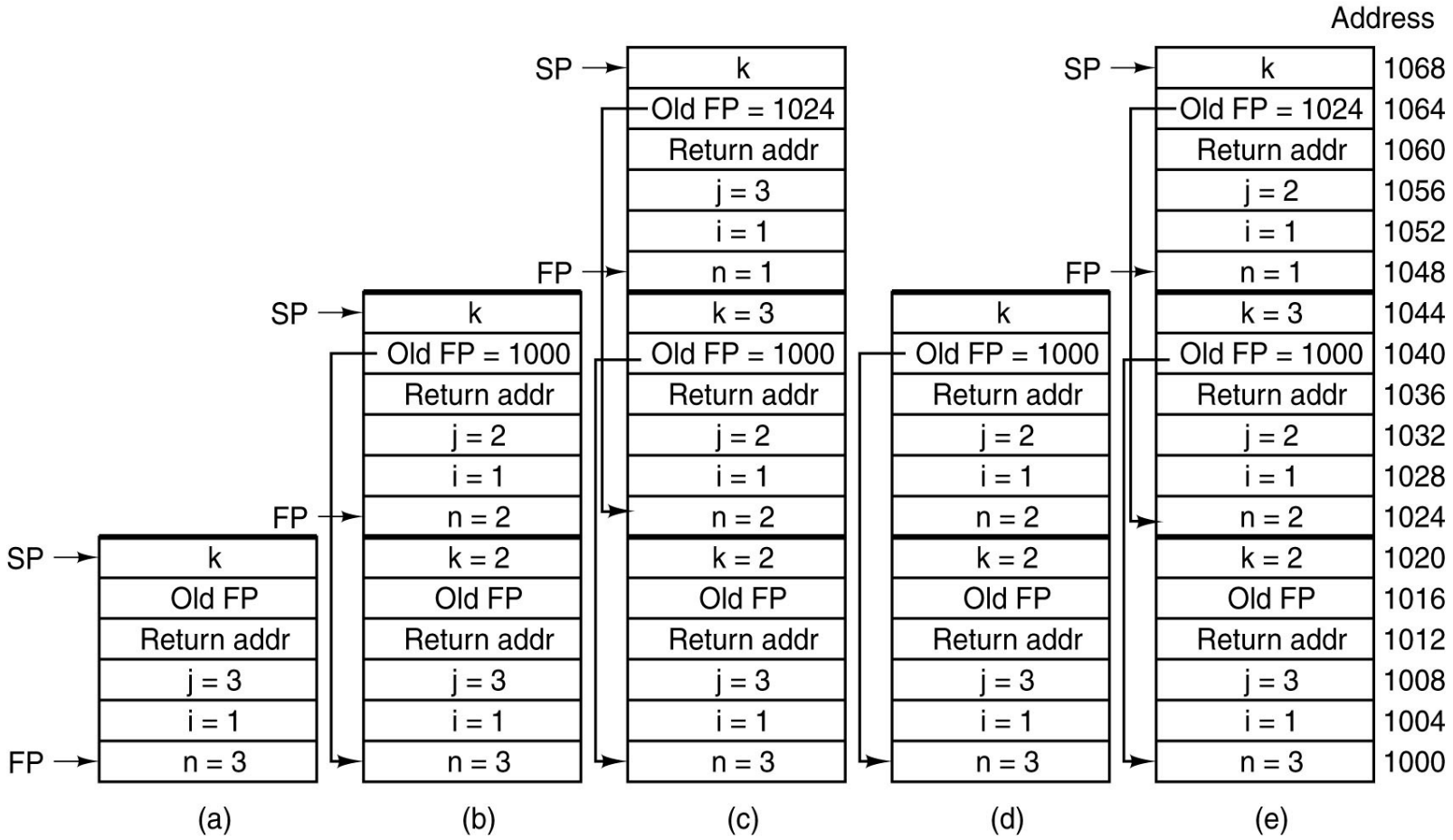
Recursive Procedures (4)

```
public void towers(int n, int i, int j) {
    int k;

    if (n == 1)
        System.out.println("Move a disk from " + i + " to " + j);
    else {
        k = 6 – i – j;
        towers(n – 1, i, k);
        towers(1, i, j);
        towers(n – 1, k, j);
    }
}
```

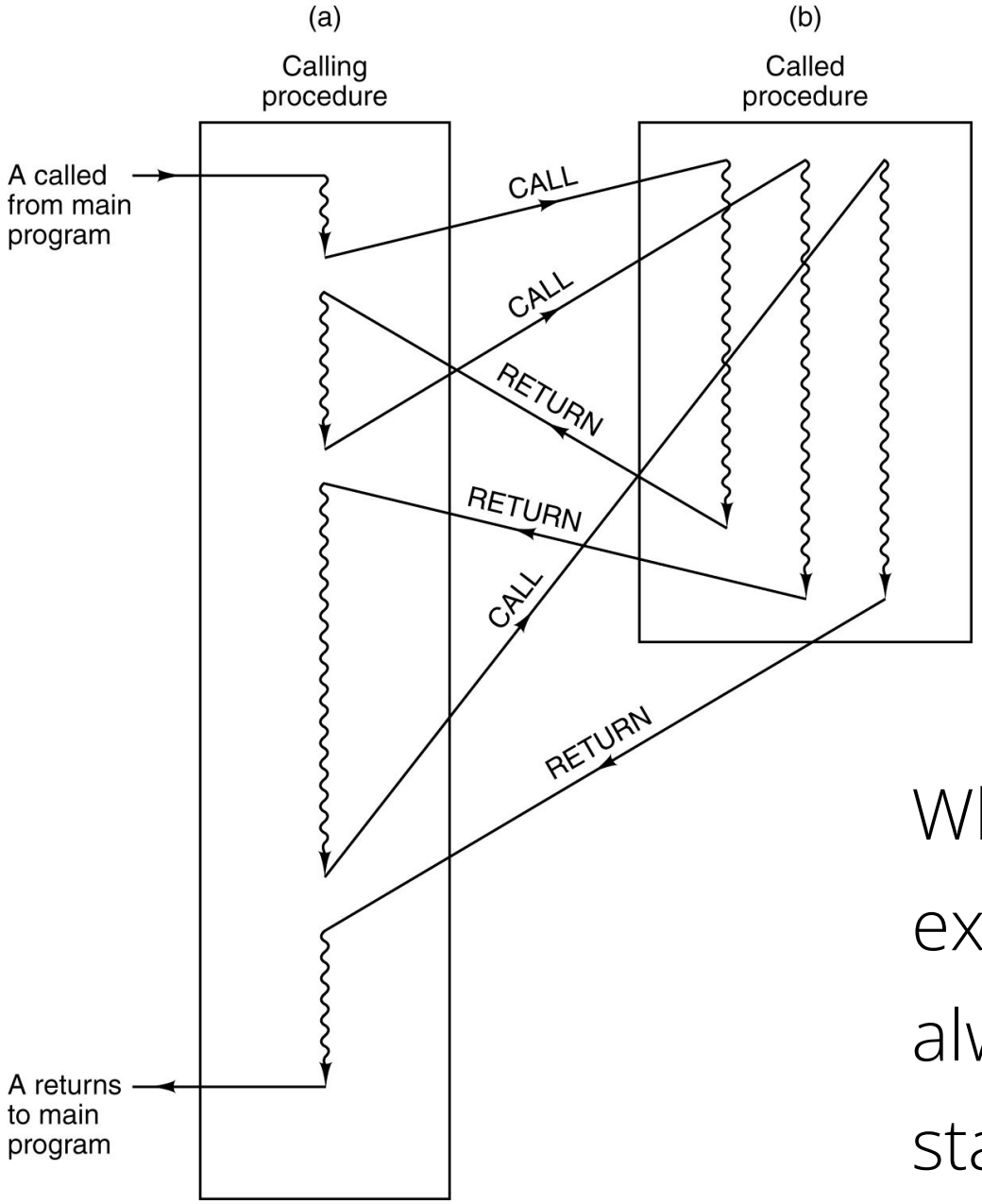
A procedure for solving the Towers of Hanoi.

Recursive Procedures (5)



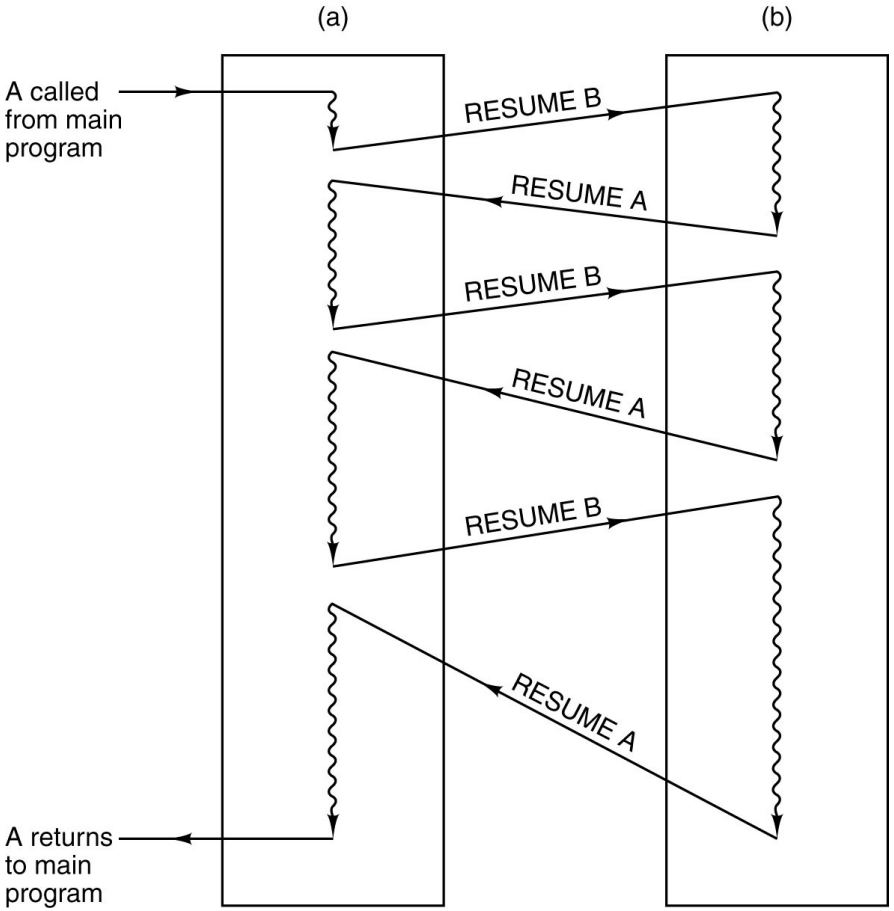
The stack at several points during the execution of Fig. 5-42.

Coroutines (1)



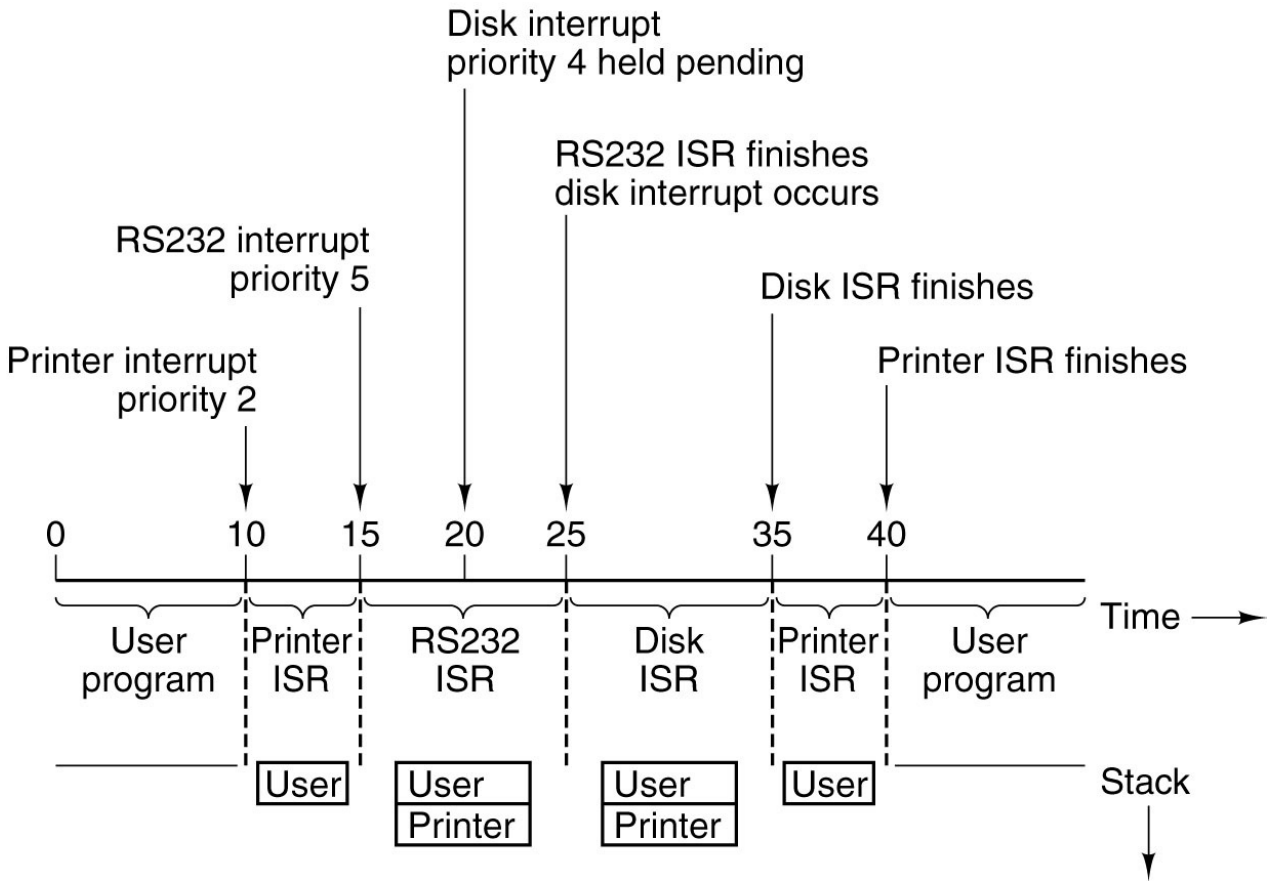
When a procedure is called, execution of the procedure always begins at the first statement of the procedure.

Coroutines (2)



When a coroutine is resumed, execution begins at the statement where it left off the previous time, not at the beginning.

Interrupts



Time sequence of multiple interrupt example.

Towers of Hanoi in Core i7 Assembly Language

```
            .586                        ; compile for Pentium (as opposed to 8088 etc.)
.MODEL FLAT
PUBLIC _towers                          ; export 'towers'
EXTERN _printf:NEAR                     ; import printf
.CODE
 _towers:   PUSH EBP                    ; save EBP (frame pointer) and decrement ESP
            MOV EBP, ESP                ; set new frame pointer above ESP
            CMP [EBP+8], 1              ; if (n == 1)
            JNE L1                      ; branch if n is not 1
            MOV EAX, [EBP+16]           ; printf(" ...", i, j);
            PUSH EAX                    ; note that parameters i, j and the format
            MOV EAX, [EBP+12]           ; string are pushed onto the stack
            PUSH EAX                    ; in reverse order. This is the C calling convention
            PUSH OFFSET FLAT:format     ; offset flat means the address of format
            CALL _printf                ; call printf
            ADD ESP, 12                 ; remove params from the stack
            JMP Done                    ; we are finished
              . . .
```

Towers of Hanoi for Core i7

Towers of Hanoi in Core i7 Assembly Language

```
        . . .
L1:     MOV EAX, 6              ; start k = 6 – i – j
        SUB EAX, [EBP+12]       ; EAX = 6 – i
        SUB EAX, [EBP+16]       ; EAX = 6 – i – j
        MOV [EBP+20], EAX       ; k = EAX
        PUSH EAX                ; start towers(n – 1, i, k)
        MOV EAX, [EBP+12]       ; EAX = i
        PUSH EAX                ; push i
        MOV EAX, [EBP+8]        ; EAX = n
        DEC EAX                 ; EAX = n – 1
        PUSH EAX                ; push n – 1
        CALL _towers            ; call towers(n – 1, i, 6 – i – j)
        ADD ESP, 12             ; remove params from the stack
        MOV EAX, [EBP+16]       ; start towers(1, i, j)
        PUSH EAX                ; push j
        MOV EAX, [EBP+12]       ; EAX = i
        PUSH EAX                ; push i
        PUSH 1                  ; push 1
        CALL _towers            ; call towers(1, i, j)

        . . .
```

Towers of Hanoi for Core i7

Towers of Hanoi in Core i7 Assembly Language

```
        . . .
        ADD ESP, 12                         ; remove params from the stack
        MOV EAX, [EBP+12]                   ; start towers(n – 1, 6 – i – j, i)
        PUSH EAX                            ; push i
        MOV EAX, [EBP+20]                   ; EAX = k
        PUSH EAX                            ; push k
        MOV EAX, [EBP+8]                    ; EAX = n
        DEC EAX                             ; EAX = n–1
        PUSH EAX                            ; push n – 1
        CALL _towers                        ; call towers(n – 1, 6 – i – j, i)
        ADD ESP, 12                         ; adjust stack pointer
Done:   LEAVE                               ; prepare to exit
        RET 0                               ; return to the caller

.DATA
format  DB "Move disk from %d to %d\n"      ; format string
END
```

Towers of Hanoi for Core i7