Instruction Set Architecture (Week#6)

(Interface between hardware and software)

Intel x86 and ARM Operation Types

Call/return instructions: x86 provides four instructions to support procedure call/return: CALL, ENTER, LEAVE, RETURN. It will be instructive to look at the support provided by these instructions.

Figure 12.10 shows stack frames. When new procedure is called, the following must be performed upon entry to the new procedure:

- ✔ Push the return point on the stack.
- ✔ Push the current frame pointer on the stack.
- ✔ Copy the stack pointer as the new value of the frame pointer.
- ✔ Adjust the stack pointer to allocate a frame.

CALL instruction pushes the current instruction pointer value onto the stack and causes a jump to the entry point of the procedure by placing the address of the entry point in the instruction pointer.

Intel x86 and ARM Operation Types

In the 8088 and 8086 machines, the typical procedure began with the sequence.

```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, space_for_locals
```

EBP is the frame pointer and ESP is the stack pointer. ENTER instruction was added to the instruction set to provide direct support for the compiler.

ENTER instruction saves a few bytes of memory compared with the PUSH, MOV, SUB sequence (4 bytes versus 6 bytes), it takes longer to execute (10 clock cycles versus 6 clock cycles).

RISC approach to processor design would avoid complex instructions such as ENTER and might produce a more efficient implementation with a sequence of simpler instructions.

RISC computer might require more instructions (more code) in order to accomplish a task because the individual instructions are written in simpler code. The goal is to offset the need to process more instructions by increasing the speed of each instruction, in particular by implementing an instruction pipeline, which may be simpler to achieve given simpler instructions.

Intel x86 and ARM Operation Types

Memory Management: Another set of specialized instructions deals with memory segmentation. These are privileged instructions that can only be executed from operating system.

Allow local and global segment tables (called descriptor tables) to be loaded and read, and for the privilege level of a segment to be checked and altered.

Status Flags and Condition Codes:

Status flags are bits in special registers that may be set by certain operations and used in conditional branch instructions.

Condition code refers to the settings of one or more status flags. In the x86 and many other architectures, status flags are set by arithmetic and compare operations.

Compare operation in most languages subtracts two operands, as does a subtract operation. The difference is that a compare operation only sets status flags, whereas a subtract operation also stores the result of the subtraction in the destination operand. Some architectures also set status flags for data transfer instructions.

Intel x86 and ARM Operation Types

Table 12.8 lists the status flags used on the x86. Each flag, or combinations of these flags can be tested for a conditional jump.

Table 12.8 x86 Status Flags

Status Bit	Name	Description
C	Carry	Indicates carrying or borrowing out of the left-most bit position following an arithmetic operation. Also modified by some of the shift and rotate operations.
P	Parity	Parity of the least-significant byte of the result of an arithmetic or logic operation. 1 indicates even parity; 0 indicates odd parity.
A	Auxiliary Carry	Represents carrying or borrowing between half-bytes of an 8-bit arithmetic or logic operation. Used in binary-coded decimal arithmetic.
Z	Zero	Indicates that the result of an arithmetic or logic operation is 0.
S	Sign	Indicates the sign of the result of an arithmetic or logic operation.
O	Overflow	Indicates an arithmetic overflow after an addition or subtraction for twos complement arithmetic.

Intel x86 and ARM Operation Types

Table 12.9 shows the condition codes conditional jump and SETcc Instructions.SETcc stores a byte at the destination specified by the effective address or register if the condition is met, or a 0 byte if the condition is not met.

Table 12.9 x86 Condition Codes for Conditional Jump and SETcc Instructions

Symbol	Condition Tested	Comment
A, NBE	C = 0 AND Z = 0	Above; Not below or equal (greater than, unsigned)
AE, NB, NC	C = 0	Above or equal; Not below (greater than or equal, unsigned); Not carry
B, NAE, C	C = 1	Below; Not above or equal (less than, unsigned); Carry set
BE, NA	C = 1 OR Z = 1	Below or equal; Not above (less than or equal, unsigned)
E, Z	Z = 1	Equal; Zero (signed or unsigned)
G, NLE	[(S = 1 AND O = 1) OR (S = 0 AND O = 0)]AND[Z = 0]	Greater than; Not less than or equal (signed)
GE, NL	(S = 1 AND O = 1) OR (S = 0 AND O = 0)	Greater than or equal; Not less than (signed)
L, NGE	(S = 1 AND O = 0) OR (S = 0 AND O = 0)	Less than; Not greater than or equal (signed)
LE, NG	(S = 1 AND O = 0) OR (S = 0 AND O = 1) OR (Z = 1)	Less than or equal; Not greater than (signed)
NE, NZ	Z = 0	Not equal; Not zero (signed or unsigned)
NO	O = 0	No overflow
NS	S = 0	Not sign (not negative)
NP, PO	P = 0	Not parity; Parity odd
O	O = 1	Overflow
P	P = 1	Parity; Parity even
S	S = 1	Sign (negative)

Intel x86 and ARM Operation Types

8-bit number 11111111 is bigger than 00000000 if the two numbers are interpreted as unsigned integers (255 > 0) but is less if they are considered as 8-bit twos complement numbers (- 1 < 0).

Many assembly languages introduce two sets of terms to distinguish the two cases:

If we are comparing two numbers as signed integers, we use the terms less than and greater than; if we are comparing them as unsigned integers, we use the terms below and above.

A signed result is greater than or equal to zero if

(1) the sign bit is zero and there is no overflow (S = 0 AND O = 0),

(2) the sign bit is one and there is an overflow.

Intel x86 and ARM Operation Types

x86 SIMD instructions: Intel introduced MMX technology into its Pentium product line. MMX is set of highly optimized instructions for multimedia tasks.

There are 57 new instructions that treat data in a SIMD (single-instruction, multiple-data) fashion, which makes it possible to perform the same operation, such as addition or multiplication, on multiple data elements at once.

Each instruction typically takes a single clock cycle to execute.

For the proper application, these fast parallel operations can yield a speedup of two to eight times over comparable algorithms that do not use the MMX instructions.

With the introduction of 64-bit x86 architecture, Intel has expanded this extension to include double quadword (128 bits) operands and floating-point operations.

Intel x86 and ARM Operation Types

MMX Feature: MMX focus is multimedia programming. Video and audio data are composed of large arrays of small data types, such as 8 or 16 bits, whereas conventional instructions are tailored to operate on 32- or 64-bit data.

Examples: In graphics and video, a single scene consists of an array of pixels and there are 8 bits for each pixel or 8 bits for each pixel color component (red, green, blue).

Audio samples are quantized using 16 bits. For some 3D graphics algorithms, 32 bits are common for basic data types.

For parallel operation on these data lengths, three new data types are defined in MMX. Each data type is 64 bits in length and consists of multiple smaller data fields, each of which holds a fixed-point integer. The types are as follows:

- ✔ **Packed byte:** Eight bytes packed into one 64-bit quantity.
- ✔ **Packed word:** Four 16-bit words packed into 64 bits.
- ✔ **Packed doubleword:** Two 32-bit doublewords packed into 64 bits.

Table 12.10 MMX Instruction Set

Category	Instruction	Description
Arithmetic	PADD [B, W, D]	Parallel add of packed eight bytes, four 16-bit words, or two 32-bit doublewords, with wraparound.
	PADDS [B, W]	Add with saturation.
	PADDUS [B, W]	Add unsigned with saturation.
	PSUB [B, W, D]	Subtract with wraparound.
	PSUBS [B, W]	Subtract with saturation.
	PSUBUS [B, W]	Subtract unsigned with saturation.
	PMULHW	Parallel multiply of four signed 16-bit words, with high-order 16 bits of 32-bit result chosen.
	PMULLW	Parallel multiply of four signed 16-bit words, with low-order 16 bits of 32-bit result chosen.
	PMADDWD	Parallel multiply of four signed 16-bit words; add together adjacent pairs of 32-bit results.
Comparison	PCMPEQ [B, W, D]	Parallel compare for equality; result is mask of 1s if true or 0s if false.
	PCMPGT [B, W, D]	Parallel compare for greater than; result is mask of 1s if true or 0s if false.
Conversion	PACKUSWB	Pack words into bytes with unsigned saturation.
	PACKSS [WB, DW]	Pack words into bytes, or doublewords into words, with signed saturation.
	PUNPCKH [BW, WD, DQ]	Parallel unpack (interleaved merge) high-order bytes, words, or doublewords from MMX register.
	PUNPCKL [BW, WD, DQ]	Parallel unpack (interleaved merge) low-order bytes, words, or doublewords from MMX register.
Logical	PAND	64-bit bitwise logical AND
	PNDN	64-bit bitwise logical AND NOT
	POR	64-bit bitwise logical OR
	PXOR	64-bit bitwise logical XOR
Shift	PSLL [W, D, Q]	Parallel logical left shift of packed words, doublewords, or quadword by amount specified in MMX register or immediate value.
	PSRL [W, D, Q]	Parallel logical right shift of packed words, doublewords, or quadword.
	PSRA [W, D]	Parallel arithmetic right shift of packed words, doublewords, or quadword.
Data transfer	MOV [D, Q]	Move doubleword or quadword to/from MMX register.
Statemgt	EMMS	Empty MMX state (empty FP registers tag bits).

Note: If an instruction supports multiple data types [byte (B), word (W), doubleword (D), quadword (Q)], the data types are indicated in brackets.

Intel x86 and ARM Operation Types

Example:

PSLLW instruction performs a left logical shift separately on each of the four words in the packed word operand.

PADDB instruction takes packed byte operands as input and performs parallel additions on each byte position independently to produce a packed byte output.

Saturation Arithmetic: Feature of new instruction set is saturation arithmetic for byte and 16-bit word operands.

With ordinary unsigned arithmetic, when an operation overflows (carry out of MSB), the extra bit is truncated. This is referred to as wraparound.

Example:Consider the addition of the two words, in hexadecimal, F000h and 3000h.....

Saturation arithmetic also enables overflow of additions and multiplications to be detected consistently without an overflow bit or excessive computation, by simple comparison with the maximum or minimum value .

Intel x86 and ARM Operation Types

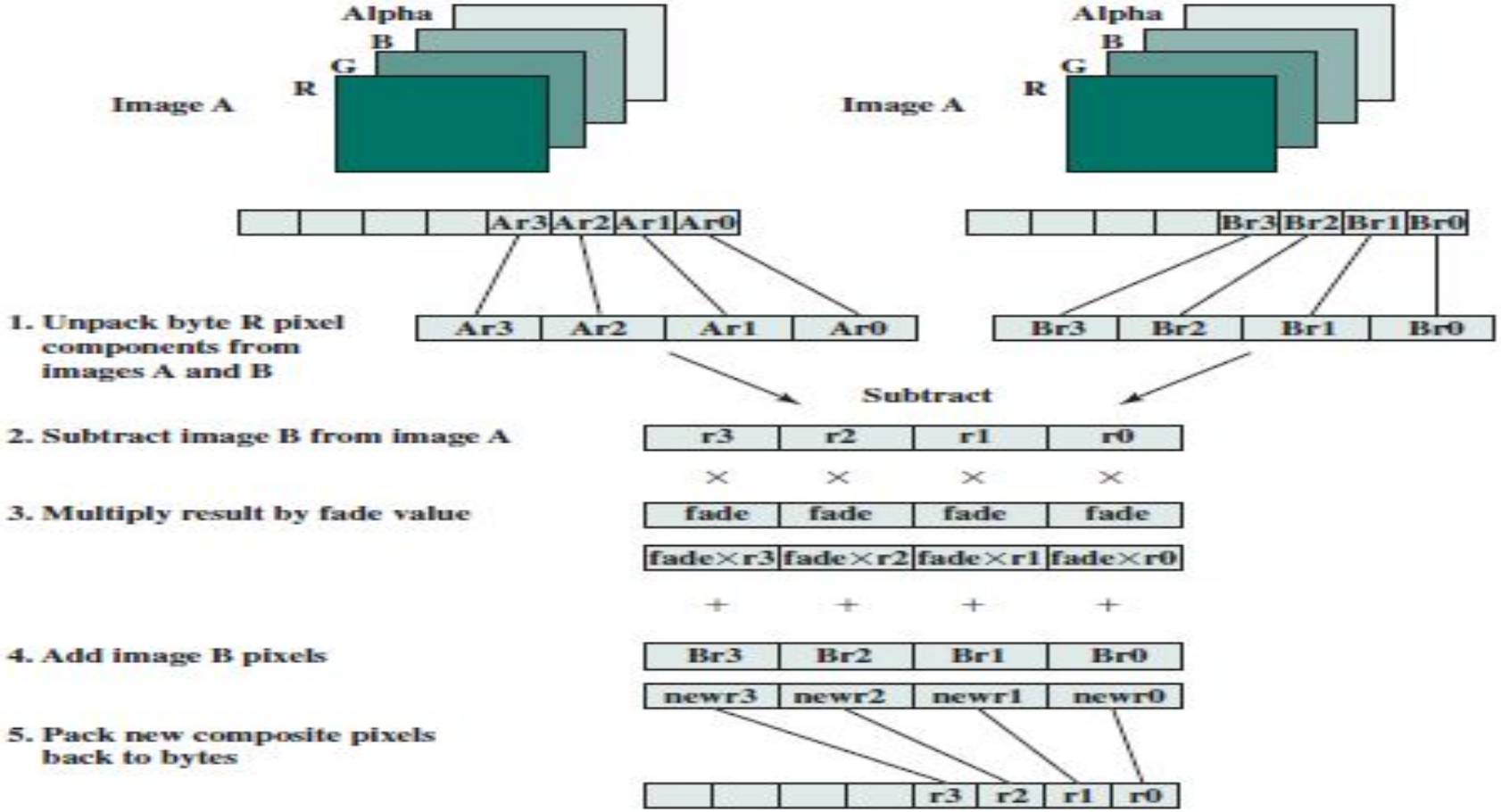
To provide a feel for the use of MMX instructions, video application is the fade-out, fade-in effect, in which one scene gradually dissolves into another. Two images are combined with a weighted average:

$$\text{Result\_pixel} = \text{A\_pixel} \times \text{fade} + \text{B\_pixel} \times (1 - \text{fade})$$

This calculation is performed on each pixel position in A and B. If a series of video frames is produced while gradually changing the fade value from 1 to 0 (scaled appropriately for an 8-bit integer),result is to fade from image A to image B.

Figure 12.11 shows the sequence of steps required for one set of pixels. 8-bit pixel components are converted to 16-bit elements to accommodate the MMX 16-bit multiply capability.

If these images use 640 * 480 resolution, and the dissolve technique uses all 255 possible values of the fade value, then the total number of instructions executed using MMX is 535 million. The same calculation, performed without the MMX instructions, requires 1.4 billion instruction executions.



MMX code sequence performing this operation:

```
pxor        mm7, mm7      ;zero out mm7
movq        mm3, fad_val  ;load fade value replicated 4 times
movd        mm0, imageA   ;load 4 red pixel components from image A
movd        mm1, imageB   ;load 4 red pixel components from image B
punpckblw   mm0, mm7      ;unpack 4 pixels to 16 bits
punpckblw   mm1, mm7      ;unpack 4 pixels to 16 bits
psubw       mm0, mm1      ;subtract image B from image A
pmulhw      mm0, mm3      ;multiply the subtract result by fade values
padddw      mm0, mm1      ;add result to image B
packuswb    mm0, mm7      ;pack 16-bit results back to bytes
```

Figure 12.11 Image Compositing on Color Plane Representation

MMX code sequence performing this operation:

```
pxor      mm7, mm7     ;zero out mm7
movq      mm3, fad_val ;load fade value replicated 4 times
movd      mm0, imageA  ;load 4 red pixel components from image A
movd      mm1, imageB  ;load 4 red pixel components from image B
punpckblw mm0, mm7     ;unpack 4 pixels to 16 bits
punpckblw mm1, mm7     ;unpack 4 pixels to 16 bits
psubw     mm0, mm1     ;subtract image B from image A
pmulhw    mm0, mm3     ;multiply the subtract result by fade values
padddw    mm0, mm1     ;add result to image B
packuswb  mm0, mm7     ;pack 16-bit results back to bytes
```

Intel x86 and ARM Operation Types

ARM Operation Types: ARM architecture provides a large collection of operation types. The following are the principal categories:

Load and store instructions: In the ARM architecture, only load and store instructions access memory locations; arithmetic and logical instructions are performed only on registers and immediate values encoded in the instruction.

ARM architecture supports two broad types of instruction that load or store the value of a single register, or a pair of registers, from or to memory:

(1) load or store a 32-bit word or an 8-bit unsigned byte

(2) load or store a 16-bit unsigned halfword, and load and sign extend a 16-bit halfword or an 8-bit byte.

Intel x86 and ARM Operation Types

Branch instructions: ARM supports a branch instruction that allows a conditional branch forwards or backwards up to 32 MB.

A subroutine call can be performed by a variant of the standard branch instruction.

Allowing a branch forward or backward up to 32 MB, the Branch with Link (BL) instruction preserves the address of the instruction after the branch (the return address) in the LR (R14).

Branches are determined by a 4-bit condition field in the instruction.

Data-processing instructions: This category includes logical instructions (AND, OR, XOR), add and subtract instructions, and test and compare instructions.

Multiply instructions: The integer multiply instructions operate on word or halfword operands and can produce normal or long results.

For example, there is a multiply instruction that takes two 32-bit operands and produces a 64-bit result.

Intel x86 and ARM Operation Types

Parallel addition and subtraction instructions: In addition to the normal data processing and multiply instructions, there are a set of parallel addition and subtraction instructions, in which portions of two operands are operated on in parallel.

Example, ADD16 adds the top halfwords of two registers to form the top halfword of the result and adds the bottom halfwords of the same two registers to form the bottom halfword of the result.

These instructions are useful in image processing applications similar to x86 MMX instructions.

Extend instructions: There are several instructions for unpacking data by sign or zero extending bytes to halfwords or words, and halfwords to words.

Status register access instructions: ARM provides the ability to read and also to write portions of the status register.

Intel x86 and ARM Operation Types

Condition codes: ARM architecture defines four condition flags that are stored in the program status register: N, Z, C, and V (Negative, Zero, Carry and oVerflow), with meanings essentially the same as the S, Z, C, and V flags in the x86 architecture.

Table 12.11 ARM Conditions for Conditional Instruction Execution

Code	Symbol	Condition Tested	Comment
0000	EQ	Z = 1	Equal
0001	NE	Z = 0	Not equal
0010	CS/HS	C = 1	Carry set/unsigned higher or same
0011	CC/LO	C = 0	Carry clear/unsigned lower
0100	MI	N = 1	Minus/negative
0101	PL	N = 0	Plus/positive or zero
0110	VS	V = 1	Overflow
0111	VC	V = 0	No overflow
1000	HI	C = 1 AND Z = 0	Unsigned higher
1001	LS	C = 0 OR Z = 1	Unsigned lower or same
1010	GE	N = V [(N = 1 AND V = 1) OR (N = 0 AND V = 0)]	Signed greater than or equal
1011	LT	N ≠ V [(N = 1 AND V = 0) OR (N = 0 AND V = 1)]	Signed less than
1100	GT	(Z = 0) AND (N = V)	Signed greater than
1101	LE	(Z = 1) OR (N ≠ V)	Signed less than or equal
1110	AL	—	Always (unconditional)
1111	—	—	This instruction can only be executed unconditionally

Intel x86 and ARM Operation Types

Table 12.11 shows the combination of conditions for which conditional execution is defined.

There are two unusual aspects to the use of condition codes in ARM:

1. All instructions, not just branch instructions, include a condition code field, which means that virtually all instructions may be conditionally executed.

Any combination of flag settings except 1110 or 1111 in an instruction's condition code field signifies that the instruction will be executed only if the condition is met.

2. All data processing instructions (arithmetic, logical) include an S bit that signifies whether the instruction updates the condition flags.

Use of conditional execution and conditional setting of the condition flags helps in the design of shorter programs that use less memory.

All instructions include 4 bits for condition code, so there is a trade-off in that fewer bits in 32-bit instruction are available for opcode and operands.

ARM is a RISC design that relies heavily on register addressing.

Intel x86 and ARM Operation Types

Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Tests special bits set by ALU operations, possibly under program control	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructionsbecause they pass information from one instruction to a branch
Condition register/ limited comparison	Alpha, MIPS	Tests arbitrary register with the result of a simple comparison (equality or zero tests)	Simple	Limited compare may affect critical path or require extra comparison for general condition
Compare and branch	PA-RISC, VAX, RISC-V	Compare is part of the branch. Fairly general compares are allowed (greater then, less then)	One instruction rather than two for a branch	May set critical path for branch instructions

Figure A.16 The major methods for evaluating branch conditions, their advantages, and their disadvantages. Although condition codes can be set by ALU operations that are needed for other purposes, measurements on programs show that this rarely happens. The major implementation problems with condition codes arise when the condition code is set by a large or haphazardly chosen subset of the instructions, rather than being controlled by a bit in the instruction. Computers with compare and branch often limit the set of compares and use a separate operation and register for more complex compares. Often, different techniques are used for branches based on floating-point comparison versus those based on integer comparison. This dichotomy is reasonable because the number of branches that depend on floating-point comparisons is much smaller than the number depending on integer comparisons.

how endianness determines addressing and byte order

Example: Draw big endian and little endian layouts.

```
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;     //0x6162_6364                    word
} s;
```

how endianness determines addressing and byte order.

Big Endianess Example: Figure 12.13 illustrates how endianness determines addressing and byte order.
C structure at the top contains a number of data types.

Memory is depicted as a series of 64-bit rows.

For the big- endian case, memory typically is viewed left to right, top to bottom.

```
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;     //0x6162_6364                    word
} s;
```

Big-endian address mapping

Byte address								
	11	12	13	14				
00	00	01	02	03	04	05	06	07
	21	22	23	24	25	26	27	28
08	08	09	0A	0B	0C	0D	0E	0F
	31	32	33	34	'A'	'B'	'C'	'D'
10	10	11	12	13	14	15	16	17
	'E'	'F'	'G'		51	52		
18	18	19	1A	1B	1C	1D	1E	1F
	61	62	63	64				
20	20	21	22	23				

how endianness determines addressing and byte order.

Little Endianess Example: Figure 12.13 illustrates how endianness determines addressing and byte order.

```
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;     //0x6162_6364                    word
} s;
```

Memory is depicted as a series of 64-bit rows.

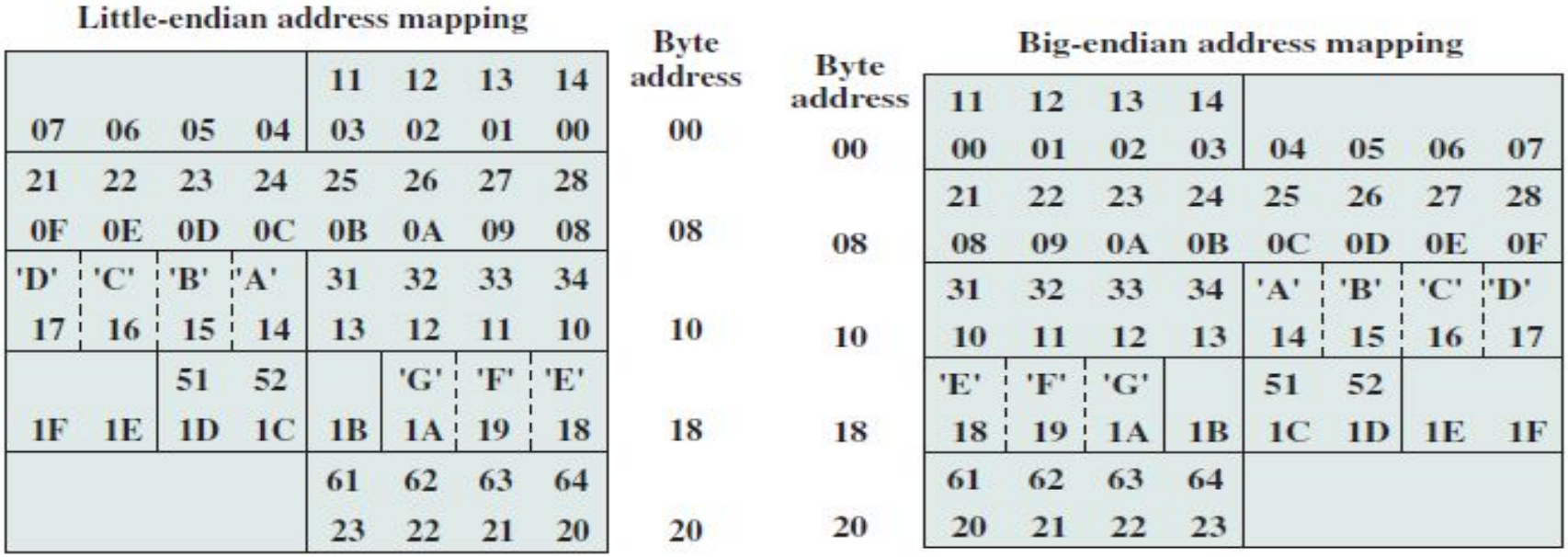
For little endian case, memory typically is viewed as right to left, top to bottom.

Note that these layouts are arbitrary. Either scheme could use either left to right or right to left within a row; this is a matter of depiction, not memory assignment.

Little-endian address mapping

								Byte address
				11	12	13	14	
07	06	05	04	03	02	01	00	00
21	22	23	24	25	26	27	28	
0F	0E	0D	0C	0B	0A	09	08	08
'D'	'C'	'B'	'A'	31	32	33	34	
17	16	15	14	13	12	11	10	10
		51	52		'G'	'F'	'E'	
1F	1E	1D	1C	1B	1A	19	18	18
				61	62	63	64	
				23	22	21	20	20

how endianness determines addressing and byte order.



We can make several observations about this data structure:

-Each data item has the same address in both schemes. For example, the address of the doubleword with hexadecimal value 212223242526272 8 is 08.

-Ordering of bytes in the little- endian structure is the reverse of that for the big- endian structure.

-Endianness does not affect the ordering of data items within a structure.Four character word c exhibits byte reversal, but the seven- character byte array d does not. Address of each individual element of d is the same in both.

Effect of endianness is perhaps more clearly demonstrated when we view memory as a vertical array of bytes, as shown in Figure 12.14.

```
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;    //0x6162_6364                  word
} s;
```

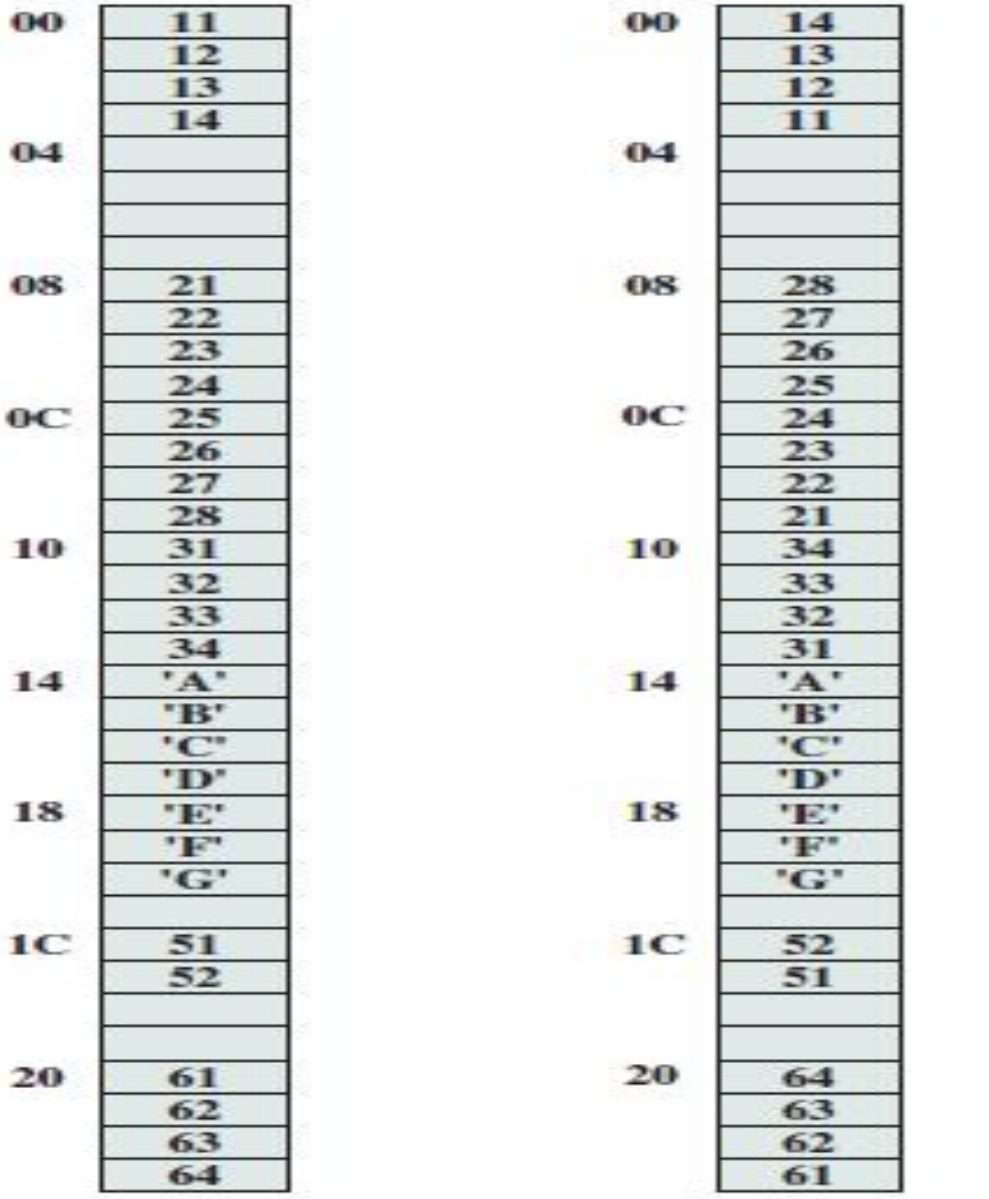


Figure 12.14 Another View of Figure 12.13

Summary Classifying Instruction Set Architecture

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

Figure A.2 The code sequence for C = A+B for four classes of instruction sets. Note that the Add instruction has implicit operands for stack and accumulator architectures and explicit operands for register architectures. It is assumed that A, B, and C all belong in memory and that the values of A and B cannot be destroyed. Figure A.1 shows the Add operation for each class of architecture.

Summary Classifying Instruction Set Architecture

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	ARM, MIPS, PowerPC, SPARC, RISC-V
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

Figure A.3 Typical combinations of memory operands and total operands per typical ALU instruction with examples of computers. Computers with no memory reference per ALU instruction are called load-store or register-register computers. Instructions with multiple memory operands per typical ALU instruction are called register-memory or memory-memory, according to whether they have one or more than one memory operand.

Summary Classifying Instruction Set Architecture

Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see Appendix C)	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density lead to larger programs, which may have some instruction cache effects
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density	Operands are not equivalent because a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

Figure A.4 Advantages and disadvantages of the three most common types of general-purpose register computers. The notation (m, n) means m memory operands and n total operands. In general, computers with fewer alternatives simplify the compiler's task because there are fewer decisions for the compiler to make (see Section A.8). Computers with a wide variety of flexible instruction formats reduce the number of bits required to encode the program. The number of registers also affects the instruction size because you need $\log_2$ (number of registers) for each register specifier in an instruction. Thus, doubling the number of registers takes three extra bits for a register-register architecture, or about 10% of a 32-bit instruction.

Intel x86 and ARM Operation Types

12.18 Mathematical formulas are usually expressed in what is known as infix notation, in which a binary operator appears between the operands. An alternative technique is known as **reverse Polish**, or **postfix**, notation, in which the operator follows its two operands. See Appendix I for more details. Convert the following formulas from reverse Polish to infix:

a. AB + C + D ×
b. AB/CD/ +
c. ABCDE + × × /