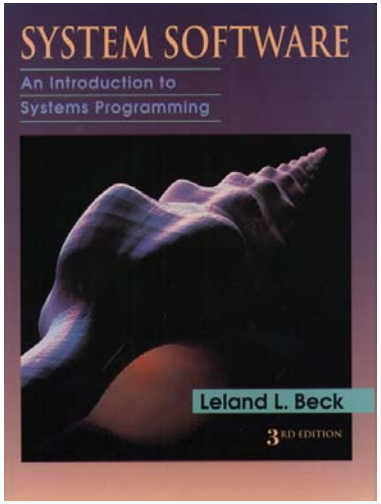
System Programming 系統程式

Yuan-Hao Chang (張原豪)
johnsonchang@ntut.edu.tw
Department of Electronic Engineering
National Taipei University of Technology

Course Information



- **Lecturer:**
  - Yuan-Hao Chang
  - Office: Room 207-2
  - Phone: +886-2-2771-2171 ext. 2288
- **Lecturing hours:**
  - 2:10 pm ~ 5:00 pm, Friday
- **Classroom:**
  - Room 309, Third Education Building (三教 309)
- **Textbook:**
  - System Software - An Introduction to Systems Programming, 3rd Edition, 1997
  - Author: Leland L. Beck
    Publisher: Addison Wesley (台北圖書代理), ISBN: 0-321-21177-4
- **Course webpage:**
  - http://www.ntut.edu.tw/~johnsonchang/courses/Algorithm201008/
- **Grading：(subject to changes)**
  - Homework: (30%), Midterm exam (30%), Final exam (30%), In-class performance (10%)

Outline of the Course

- Background
- Assemblers
- Loaders and linkers
- Macro processors
- Compilers
- Operating systems
- Other system software
- Software engineering issues

Chapter 1 Background

Outline

- Introduction
- The Simplified Instructional Computer (SIC)
- CISC Machines
- RISC Machines
- References to Addressing Modes and Instruction Set

Introduction

Objectives of This Course

- This course is an introduction to the design and implementation of system software.
  - System software consists of a variety of programs that support the operation of a computer.
  - System software makes it possible for users to focus on an application or a problem to be solved, without knowing how the machine works internally.
- By understanding the system software, you will
  - Gain a deeper understanding of how computers really work.
  - Learn the relationship between system software and machine architecture.

Examples of System Software

- ***Assembler***
  - Translate assembler-language programs into machine language, and may include ***macro processor***.
- ***Loader*** or ***Linker***
  - Load the resulting machine-language program into memory and prepared for execution.
- ***Compiler***
  - Translate programs in a high-level language (that are edited by ***text editor***) into machine language.
- ***Operating system***
  - Control all of the processes running on the machine and take care of all the machine-level details.
- ***Debugger***
  - Help detect errors in programs.

System Software and Machine Architecture

- ***Machine dependency*** is the main difference between *system software* and *application software*.
  - Application programs are concerned with the solution of some problem, using the computer as a tool.
  - System programs are intended to support the operation and use of the computer itself. For example:
    - ***Assemblers*** translate mnemonic instructions into machine code.
    - ***Compilers*** generate machine language code, taking hardware characteristics into account.
    - ***Operating systems*** are directly concerned with the management of nearly all of the resources of a computing system.
- There are some aspects of system software that do not directly depend upon the computing system. For example:
  - The general design and logic of an assembler.
  - Some code optimization techniques used by compilers.

The Simplified Instructional Computer (SIC)

Simplified Instructional Computer (SIC)

- It is difficult to distinguish whether the features of software are truly fundamental or solely depend on a particular machine.
  - To avoid this problem, the fundamental functions of system software are discussed through a Simplified Instructional Computer (SIC).
  - SIC is a hypothetical (假定的) computer that is carefully designed
    - To include the hardware features most often found on real computers.
    - To avoid unusual or irrelevant complexities.
  - SIC provides the reader with a starting point to begin the design of system software for a new computer.
- SIC comes in two versions:
  - The standard model: SIC
  - The extra equipment version: SIC/XE

SIC Machine Architecture

- ***Memory***
  - 8 bits forms a *byte*.
  - 3 consecutive bytes form a *word* (24 bits).
  - All addresses on SIC are byte addresses.
  - Words are addressed by the location of their lowest numbered byte.
  - There are *32,768* (2^{15}) bytes in the computer memory.

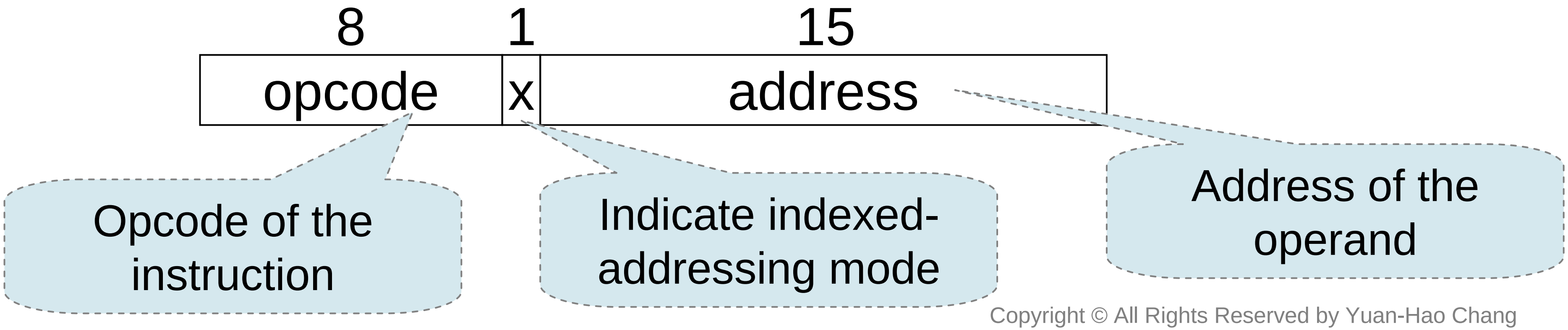
SIC Machine Architecture (Cont.)

- ***Registers***
  - Five registers, all of each have special uses.
  - 24 bits long in each register.

Mnemonic (助記符號)	Number	Special use
A	0	***Accumulator***; used for **arithmetic operations**
X	1	***IndeX Register***; used for **(indexed) addressing**
L	2	***Linkage register***; the Jump to Subroutine (***JSUB***) instruction stores **the return address** in this register
PC	8	***Program Counter***; contains the address of **the next instruction** to be fetched for execution
SW	9	***Status Word***; contains a variety of information, including a **Condition Code (CC)**

SIC Machine Architecture (Cont.)

- ***Data formats***
  - ***Integers*** are stored as 24-bit binary numbers.
  - ***Negative values*** are represented by *2's complement*.
  - ***Characters*** are stored using 8-bit *ASCII codes*.
  - ***Floating-point*** hardware is not supported.
- ***Instruction formats***



SIC Machine Architecture (Cont.)

- ***Addressing Modes***
  - Indicated by the setting of the *x* bit in the instruction.
  - The *target address* is calculated from the address given in the instruction.
    - Target address of an instruction is ***the address (of the operand)*** that will be accessed by the instruction.
  - ***Parentheses*** are used to indicate ***the contents of a register or a memory location***.
    - E.g., **(X)** represents the contents of register **X**.

Mode	Indication	Target address (TA) calculation
Direct	x = 0	TA = address
Indexed	x = 1	TA = address + (X)

SIC Machine Architecture (Cont.)

A complete list of instructions are listed in the appendix / references.

- ***Instruction Set***
  - ***Instructions that load and store registers***
    - *E.g.*, ***LDA***, ***LDX***, ***STA***, ***STX***, etc.
  - ***Integer arithmetic operations***
    - All arithmetic operations involve register **A** and a word in memory with the result left in register **A**.
    - *E.g.*, ***ADD***, ***SUB***, ***MUL***, ***DIV***
  - ***Comparison instructions*** (***COMP***)
    - COMP compares the value in register **A** with a word in memory, and sets a ***condition code*** (***CC***) to indicate the result (<, =, or >).
  - ***Conditional jump instructions*** (***JLT***, ***JEQ***, ***JGT***)
    - Test the setting of CC, and jump accordingly.
  - ***Subroutine linkage***
    - ***JSUB*** jumps to the subroutine, placing the return address in register **L**.
    - ***RSUB*** returns by jumping to the address contained in register **L**.

SIC Machine Architecture (Cont.)

- ***Input and Output***
  - Input and output are performed by transferring *1 byte* at a time to or from *the rightmost 8 bits of register A*.
  - Each device is assigned a unique 8-bit code.
  - There are three I/O instructions, each of which specifies the *device code* as an operand:
    - ***Test Device (TD) :***
      - Test whether the addressed device is ready to send or received a byte of data.
      - The condition code is set to indicate the result of this test.
        - < means the device is ready.
        - = means the device is not ready.
      - A program cannot transfer data until the device is ready.
    - ***Read Data (RD):*** Read a byte from the device
    - ***Write Data (WD):*** Write a byte to the device

SIC/XE Machine Architecture

- ***Memory***
  - The memory structure for SIC/XE is the same as SIC.
  - The maximum memory available on a SIC/XE system is *1 megabytes* (*2^{20} bytes*).
  - This increase leads to a change in *instruction formats* and *addressing modes.*

- ***Registers***
  - Four additional registers (compared to SIC) are provided by SIC/XE:

Mnemonic	Number	Special use
B	3	***Base register***; used for **(base-register relative) addressing**
S	4	***General working register***; no special use
T	5	***General working register***; no special use
F	6	***Floating-point accumulator (48 bits)***

SIC/XE Machine Architecture (Cont.)

- ***Data formats***
  - SIC/XE provides the same data formats as the standard version.
  - SIC/XE provides a 48-bit floating-point data type:
    - The ***fraction*** (***f***) is interpreted as a value between *0 and 1*.
      - The assumed binary point is immediately before the high-order bit.
      - For normalized floating-point, the high-order bit of the fraction must be 1.
    - The ***exponent*** (**e**) is interpreted as an unsigned binary number between *0 and 2047*.
    - The ***sign*** of the float-point number is indicated by the value of ***s***
      - 0 = positive, 1 = negative.
    - A value of zero is represented by setting all bits to 0.

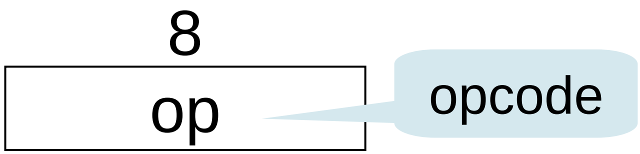
1	11	36
s	exponent (**e**)	fraction (**f**)

The floating-point value = f * $2^{(e-1024)}$

SIC/XE Machine Architecture (Cont.)

- ***Instruction formats***
  - The *15-bit address* field in SIC is no longer suitable due to the larger memory in SIC/XE.
  - Two possible options:
    - Use some form of ***relative addressing*** or
    - Extend the address field to ***20 bits***.
  - Four instructions formats:
    - Formats 1 and 2 do not reference memory at all.

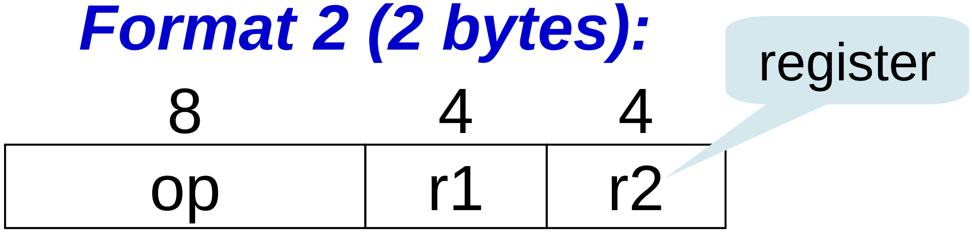
Format 1 (1 byte):



8
op

opcode

Format 2 (2 bytes):



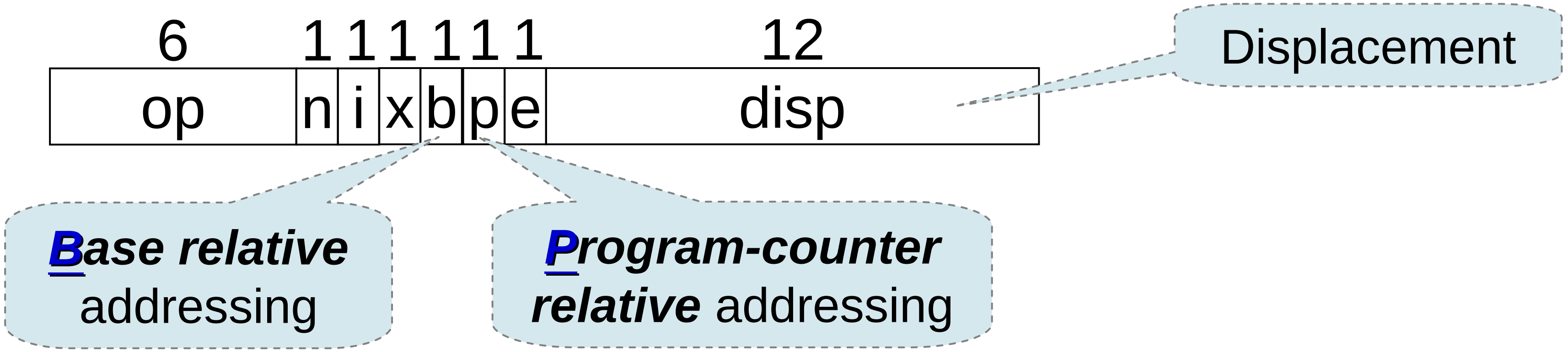
8	4	4
op	r1	r2

register

SIC/XE Machine Architecture (Cont.)

- ***Instruction formats (Cont.)***

Format 3 (3 bytes):



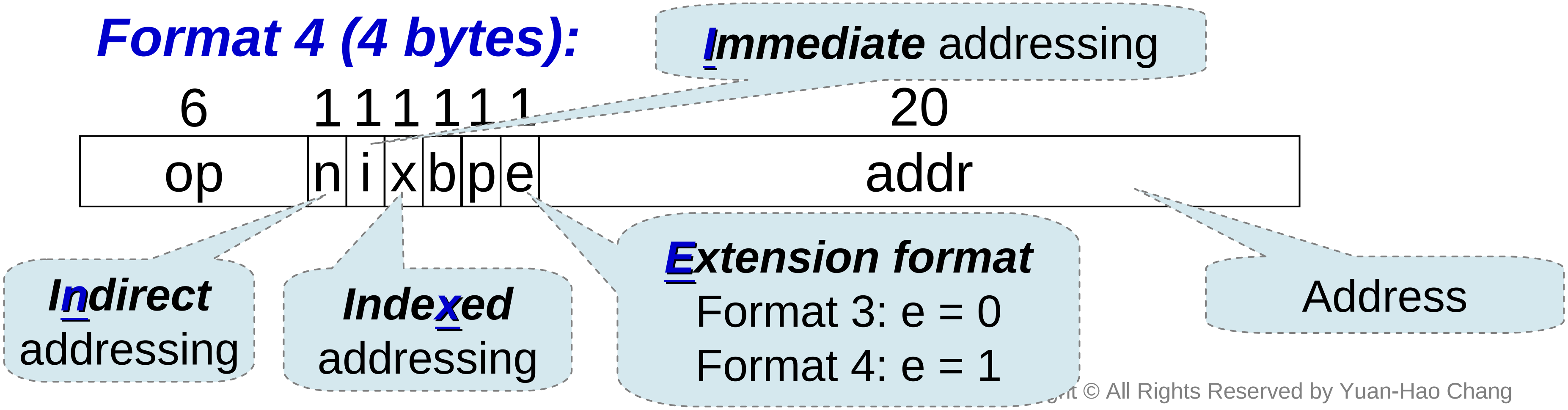
6	1	1	1	1	1	1	12
op	n	i	x	b	p	e	disp

Displacement

Base relative addressing

Program-counter relative addressing

Format 4 (4 bytes):



Immediate addressing

6	1	1	1	1	1	1	20
op	n	i	x	b	p	e	addr

Indirect addressing

Indexed addressing

Extension format
Format 3: e = 0
Format 4: e = 1

Address

SIC/XE Machine Architecture (Cont.)

- ***Addressing Modes – Target Address Calculation***

(PC), **(B)**, and **(X)** represent the contents of register **PC**, **B**, and **X**, respectively.

Format 3 (e=0)

TA Calculation Mode	Indication	Target address calculation
Program-counter relative addressing	x=0, b=0, *p=1*	TA = (PC) + disp (-2048 ≤ disp ≤ 2047)
	x=1, b=0, *p=1*	TA = (PC) + disp + (X) (-2048 ≤ disp ≤ 2047)
Base relative addressing	x=0, *b=1*, p=0	TA = (B) + disp (0 ≤ disp ≤ 4095)
	x=1, *b=1*, p=0	TA = (B) + disp + (X) (0 ≤ disp ≤ 4095)
Direct addressing	x=0, b=0, p=0	TA = disp
	x=1, b=0, p=0	TA = disp + (X)

Signed integer

Unsigned integer

– For a ***Format 4 (e=1)*** instruction, bits *b* and *p* are normally set to 0.

TA Calculation Mode	Indication	Target address calculation
Direct addressing	x=0, b=0, p=0	TA = addr
	x=1, b=0, p=0	TA = addr + (X)

SIC/XE Machine Architecture (Cont.)

- ***Addressing Types – How The Target Address Is Used***
  - *Simple addressing*: The target address is taken as the ***address of the operand value***.
  - *Immediate addressing*: The target address is used as the ***operand value***.
  - *Indirect addressing*: The value of the word at the location given by the target address is the ***address of the operand value***.

TA Addressing Type	Indication	Operand value
Simple addressing	n=0, i=0	(TA) → *for SIC machine, because the 8-bit binary opcodes of SIC instructions end in 00.* ***b/p/e/disp = 15-bit address field of SIC instruction***
	n=1, i=1	(TA) → *for SIC/XE machine*
Immediate addressing	n=0, *i=1*, x=0	TA
Indirect addressing	*n=1*, i=0, x=0	((TA))

Indexed addressing cannot be used with immediate or indirect addressing modes. (x=0)

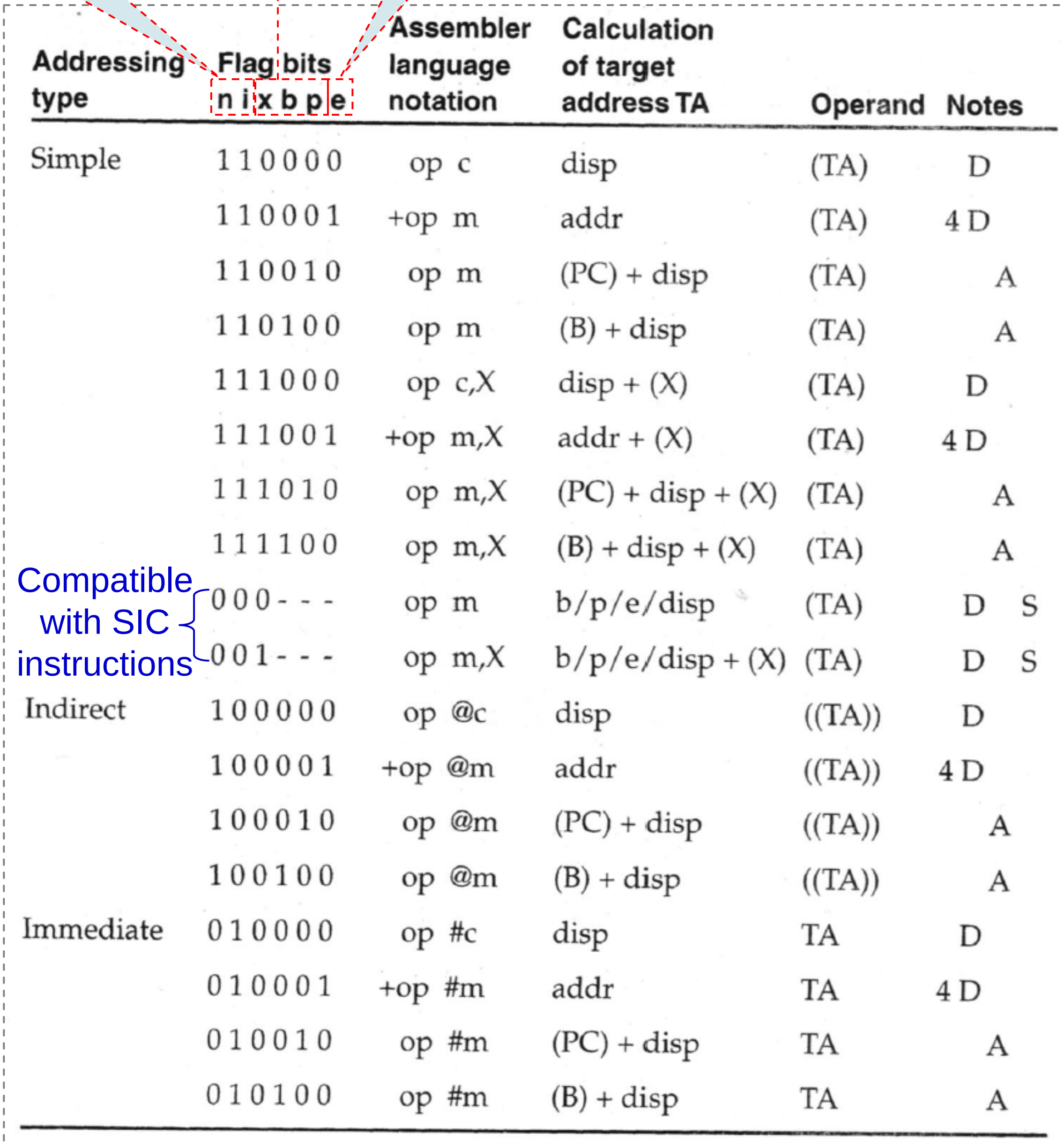
SIC/XE Machine Architecture (Cont.)

- Addressing modes for Format 3 and Format 4 instructions.
- Combinations of addressing bits not in this table are errors.
- Assembler language notation:
  - ***c*** indicates a constant between 0 and 4095.
  - ***m*** indicates a ***memory address*** or a ***constant value*** larger than 4095.
  - **+** indicates extended format (i.e., ***Format 4***)
  - **@** indicates a ***indirect addressing***
  - **#** indicates an ***immediate addressing***
  - ***X*** indicates register X (using indexed addressing)
- Letters in Notes
  - **4** Format 4 instruction
  - **D** Direct-addressing instruction
  - **A** Program-counter relative or base relative mode
  - **S** Compatible with standard SIC instructions. Operand value can be between 0 and 32,767.

Addressing type

Addressing mode

Extension



Addressing type	Flag bits n i x b p e	Assembler language notation	Calculation of target address TA	Operand	Notes
Simple	110000	op c	disp	(TA)	D
	110001	+op m	addr	(TA)	4 D
	110010	op m	(PC) + disp	(TA)	A
	110100	op m	(B) + disp	(TA)	A
	111000	op c,X	disp + (X)	(TA)	D
	111001	+op m,X	addr + (X)	(TA)	4 D
	111010	op m,X	(PC) + disp + (X)	(TA)	A
	111100	op m,X	(B) + disp + (X)	(TA)	A
	000---	op m	b/p/e/disp	(TA)	D S
	001---	op m,X	b/p/e/disp + (X)	(TA)	D S
Indirect	100000	op @c	disp	((TA))	D
	100001	+op @m	addr	((TA))	4 D
	100010	op @m	(PC) + disp	((TA))	A
	100100	op @m	(B) + disp	((TA))	A
Immediate	010000	op #c	disp	TA	D
	010001	+op #m	addr	TA	4 D
	010010	op #m	(PC) + disp	TA	A
	010100	op #m	(B) + disp	TA	A

Compatible with SIC instructions

SIC/XE Machine Architecture (Cont.)

(B) = 006000

(PC) = 003000

(X) = 000090

. . . .	3030	. . .	3600	. . . . .	6390	. . . . . .	C303	. . . .
. . . .	003600	. . .	103000	. . . . .	00C303	. . . . . .	003030	. . . .

Machine instruction										Target address	Value loaded into register A
Hex	Binary										
	op	n	i	x	b	p	e	disp/address			
032600	000000	1	1	0	0	1	0	0110 0000 0000	TA = (PC)+disp	3600	(TA)= 103000
03C300	000000	1	1	1	1	0	0	0011 0000 0000	TA = (B)+disp+(X)	6390	(TA)= 00C303
022030	000000	1	0	0	0	1	0	0000 0011 0000	TA = (PC)+disp	3030	((TA))= 103000
010030	000000	0	1	0	0	0	0	0000 0011 0000	TA = disp	30	TA= 000030
003600	000000	0	0	0	0	1	1	0110 0000 0000	TA = (PC)+disp	3600	(TA)= 103000
0310C303	000000	1	1	0	0	0	1	0000 1100 0011 0000 0011		C303	(TA)=003030

Indirect addressing

Immediate addressing

Base-relative addressing

Indexed addressing

PC-relative addressing

Format 4

Addressing mode

Addressing type

TA = addr

LDA

Simple addressing

SIC

SIC/XE Machine Architecture (Cont.)

- ***Instruction Set***
  - SIC/XE provides all of the instructions that are available on SIC.
  - Some extended instructions are supported.
    - Load and store new registers
      - E.g., ***LDB***, ***STB***, etc.
    - Floating-point arithmetic operations
      - E.g., ***ADDF***, ***SUBF***, ***MULF***, ***DIVF***
    - Register-to-register arithmetic operations
      - E.g., ***RMO (register move operation)***, ***ADDR***, ***SUBR***, ***MULR***, ***DIVR***
    - Special *supervisor call* instruction (***SVC***)
      - This instruction generates an interrupt that can be used for communication with the operating system.

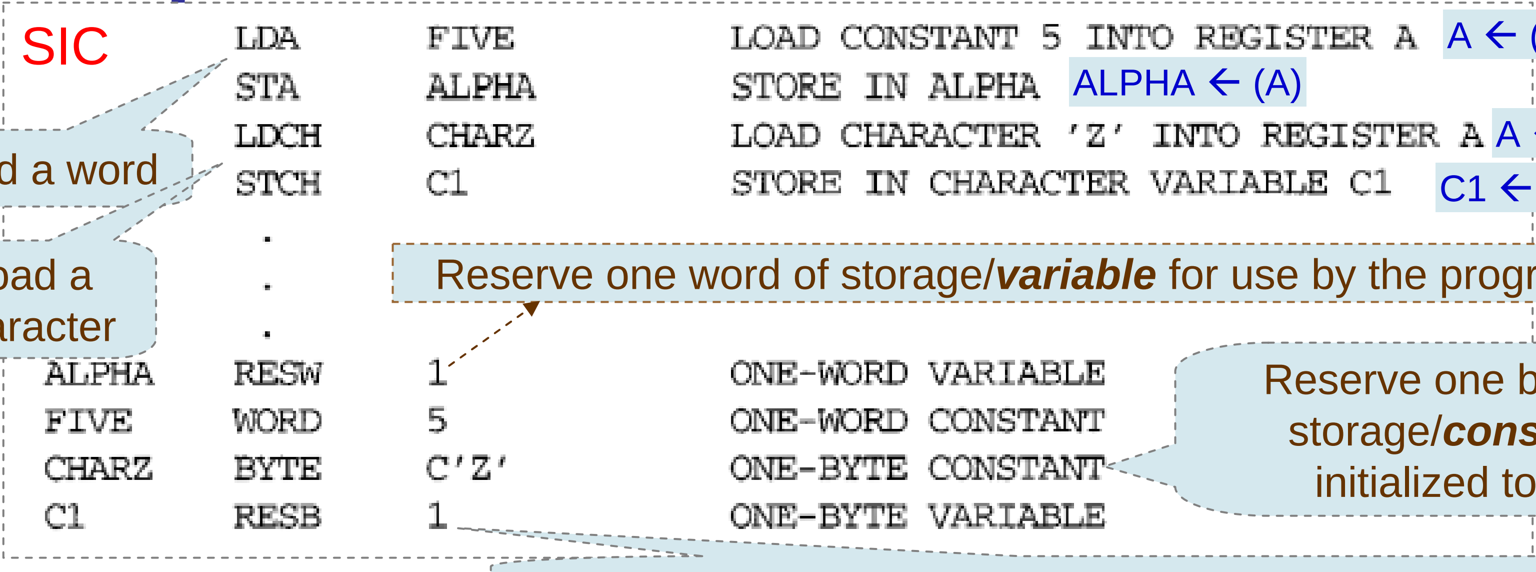
SIC/XE Machine Architecture (Cont.)

- ***Input and Output***
  - The I/O instructions supported by SIC are also available on SIC/XE.
  - I/O channels that can be used to perform input and output while the CPU is executing other instructions.
    - This allows overlap of computing and I/O.
      - ***SIO***, ***TIO***, and ***HIO*** are used to *start*, *test*, and *halt* the operation of I/O channels.

Expression Convention

- ***Parentheses*** are used to denote ***the contents of a register or memory location***.
  - ***A←(m..m+2)*** specifies that the contents of the memory locations *m* through *m+2* are loaded into register *A*.
  - ***m..m+2←(A)*** specifies that the contents of register *A* are stored in the word that begins at address *m*.
  - ***(A) : (m..m+2)*** specifies that the contents of register *A* and memory location *m* through *m+2* are compared.

Sample Data Movement



SIC

```
        LDA     FIVE            LOAD CONSTANT 5 INTO REGISTER A
        STA     ALPHA           STORE IN ALPHA
        LDCH    CHARZ           LOAD CHARACTER 'Z' INTO REGISTER A
        STCH    C1              STORE IN CHARACTER VARIABLE C1
         .
         .
         .
ALPHA   RESW    1               ONE-WORD VARIABLE
FIVE    WORD    5               ONE-WORD CONSTANT
CHARZ   BYTE    C'Z'            ONE-BYTE CONSTANT
C1      RESB    1               ONE-BYTE VARIABLE
```

A ← (FIVE)

ALPHA ← (A)

A ← (CHARZ)

C1 ← (A)

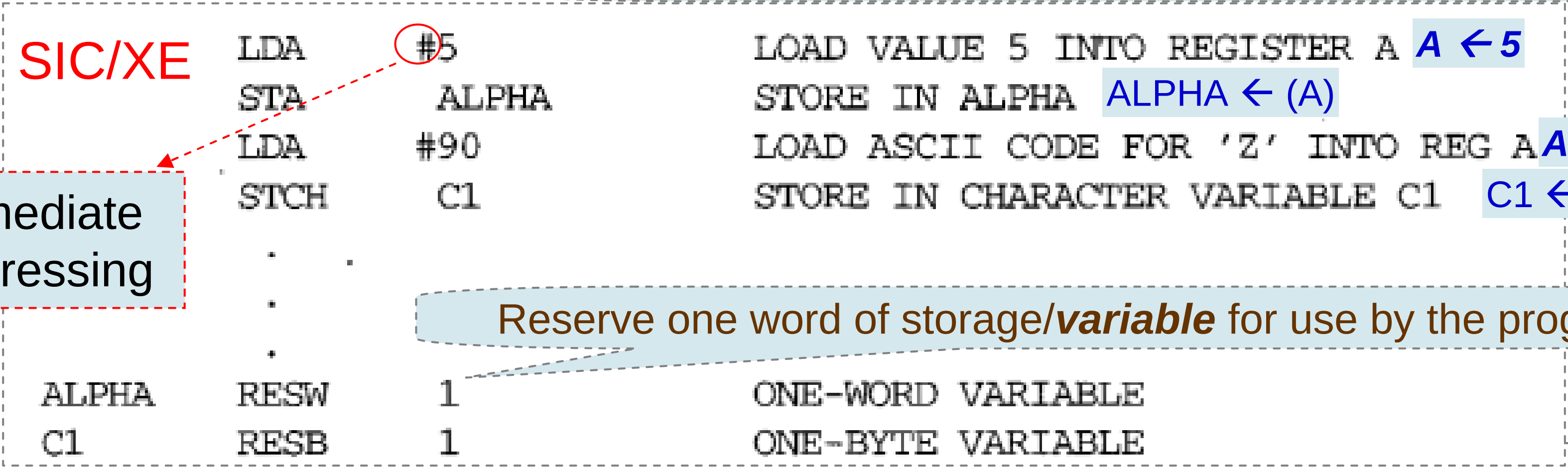
Load a word

Load a character

Reserve one word of storage/***variable*** for use by the program

Reserve one byte of storage/***constant*** initialized to 'Z'

Reserve one byte of storage/***variable*** for use by the program



SIC/XE

```
        LDA     #5              LOAD VALUE 5 INTO REGISTER A
        STA      ALPHA          STORE IN ALPHA
        LDA     #90             LOAD ASCII CODE FOR 'Z' INTO REG A
        STCH     C1             STORE IN CHARACTER VARIABLE C1
         .   .
         .
         .
ALPHA   RESW     1              ONE-WORD VARIABLE
C1      RESB     1              ONE-BYTE VARIABLE
```

A ← 5

ALPHA ← (A)

A ← 90

C1 ← (A)

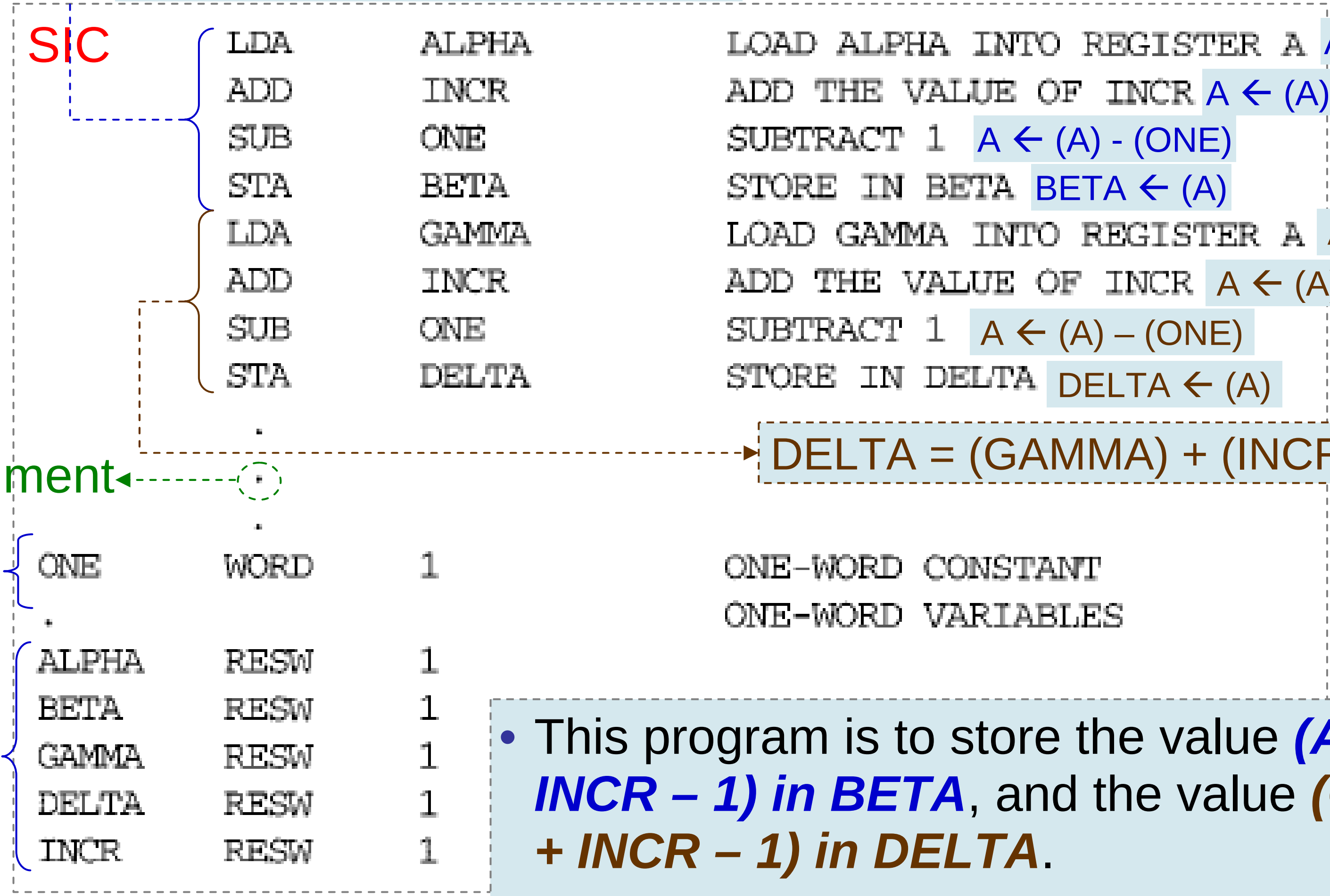
Immediate addressing

Reserve one word of storage/***variable*** for use by the program

Sample Arithmetic Operations

BETA = (ALPHA) + (INCR) - 1

SIC



```
          LDA     ALPHA          LOAD ALPHA INTO REGISTER A
          ADD     INCR           ADD THE VALUE OF INCR
          SUB     ONE            SUBTRACT 1
          STA     BETA           STORE IN BETA
          LDA     GAMMA          LOAD GAMMA INTO REGISTER A
          ADD     INCR           ADD THE VALUE OF INCR
          SUB     ONE            SUBTRACT 1
          STA     DELTA          STORE IN DELTA
           .
           .
           .
ONE       WORD    1              ONE-WORD CONSTANT
.                                ONE-WORD VARIABLES
ALPHA     RESW    1
BETA      RESW    1
GAMMA     RESW    1
DELTA     RESW    1
INCR      RESW    1
```

A ← (ALPHA)

A ← (A) + (INCR)

A ← (A) - (ONE)

BETA ← (A)

A ← (GAMMA)

A ← (A) + (INCR)

A ← (A) – (ONE)

DELTA ← (A)

DELTA = (GAMMA) + (INCR) - 1

comment

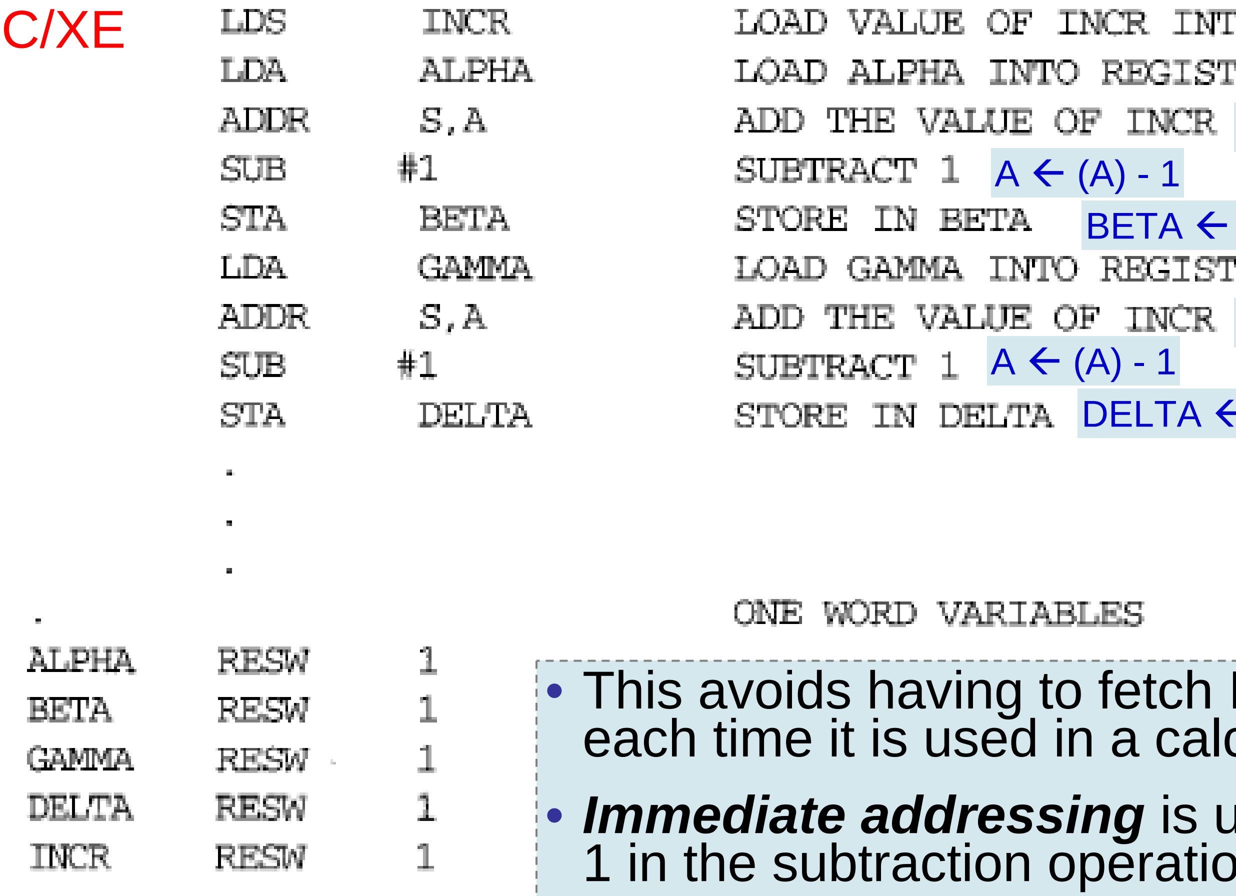
constant

variable

- This program is to store the value ***(ALPHA + INCR – 1) in BETA***, and the value ***(GAMMA + INCR – 1) in DELTA***.

Sample Arithmetic Operations (Cont.)

SIC/XE



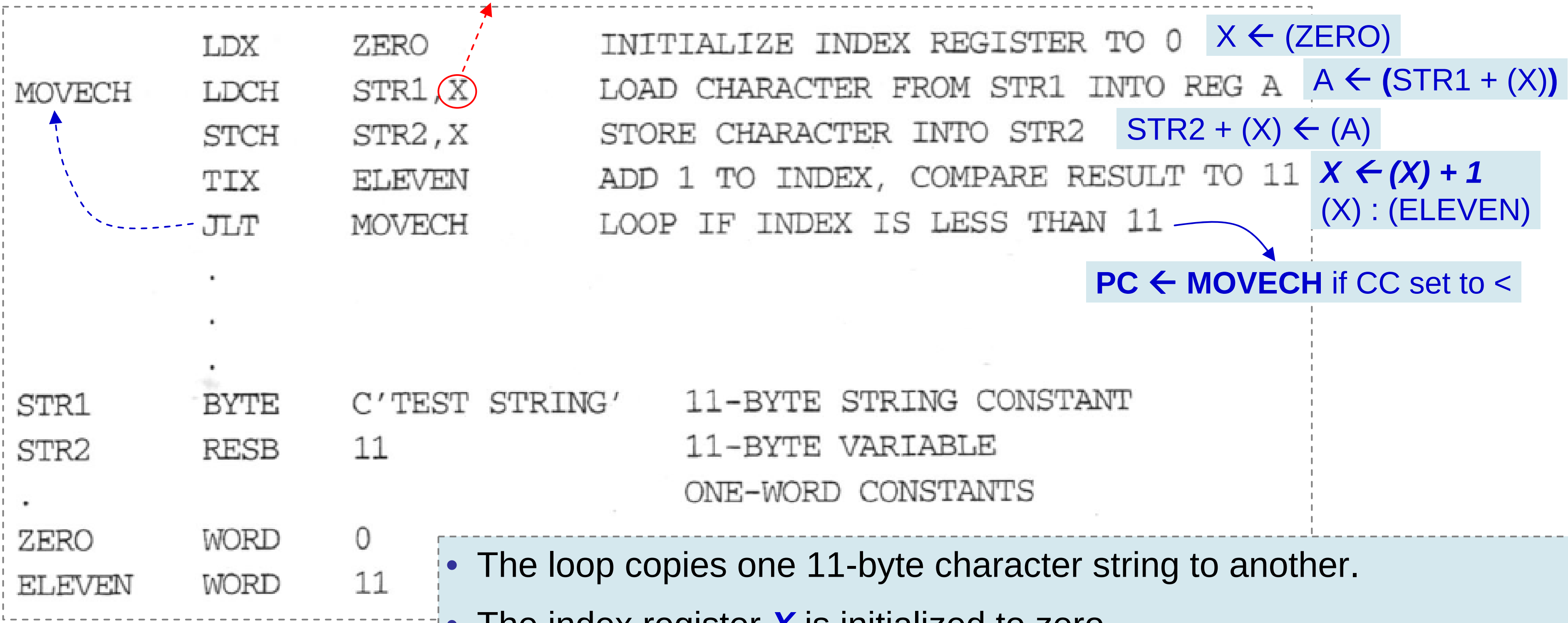
```
         LDS     INCR         LOAD VALUE OF INCR INTO REGISTER S   S ← (INCR)
         LDA     ALPHA        LOAD ALPHA INTO REGISTER A   A ← (ALPHA)
         ADDR    S,A          ADD THE VALUE OF INCR   A ← (A) + (S)
         SUB     #1           SUBTRACT 1   A ← (A) - 1
         STA     BETA         STORE IN BETA   BETA ← (A)
         LDA     GAMMA        LOAD GAMMA INTO REGISTER A   A ← (GAMMA)
         ADDR    S,A          ADD THE VALUE OF INCR   A ← (A) + (S)
         SUB     #1           SUBTRACT 1   A ← (A) - 1
         STA     DELTA        STORE IN DELTA   DELTA ← (A)
         .
         .
         .
.                             ONE WORD VARIABLES
ALPHA    RESW    1
BETA     RESW    1
GAMMA    RESW    1
DELTA    RESW    1
INCR     RESW    1
```

- This avoids having to fetch INCR from memory each time it is used in a calculation.
- ***Immediate addressing*** is used for the constant 1 in the subtraction operations.

Sample Looping and Indexing

SIC

Indexed addressing



```
         LDX      ZERO           INITIALIZE INDEX REGISTER TO 0
MOVECH   LDCH     STR1,X         LOAD CHARACTER FROM STR1 INTO REG A
         STCH     STR2,X         STORE CHARACTER INTO STR2
         TIX      ELEVEN         ADD 1 TO INDEX, COMPARE RESULT TO 11
         JLT      MOVECH         LOOP IF INDEX IS LESS THAN 11
         .
         .
         .
STR1     BYTE     C'TEST STRING'    11-BYTE STRING CONSTANT
STR2     RESB     11                11-BYTE VARIABLE
.                                   ONE-WORD CONSTANTS
ZERO     WORD     0
ELEVEN   WORD     11
```

X ← (ZERO)

A ← **(**STR1 + (X)**)**

STR2 + (X) ← (A)

X ← (X) + 1

(X) : (ELEVEN)

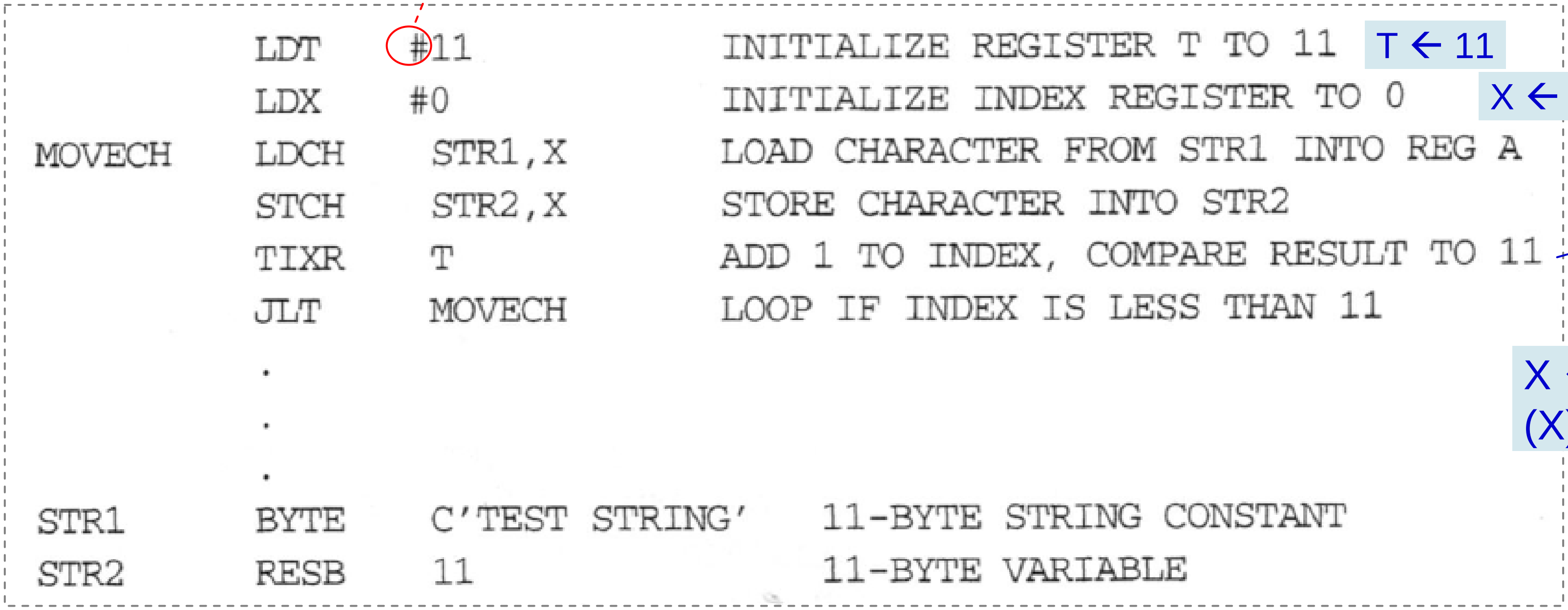
PC ← MOVECH if CC set to <

- The loop copies one 11-byte character string to another.
- The index register ***X*** is initialized to zero.
- ***TIX*** performs two functions:
  - First it adds 1 to the value in register X.
  - Then it compares the new value of register X to the value of the operand. The condition code is set according to the compared result (<, =, >)

Sample Looping and Indexing (Cont.)

SIC/XE

Immediate addressing



```
          LDT     #11            INITIALIZE REGISTER T TO 11
          LDX     #0             INITIALIZE INDEX REGISTER TO 0
MOVECH    LDCH     STR1,X        LOAD CHARACTER FROM STR1 INTO REG A
          STCH     STR2,X        STORE CHARACTER INTO STR2
          TIXR     T             ADD 1 TO INDEX, COMPARE RESULT TO 11
          JLT      MOVECH        LOOP IF INDEX IS LESS THAN 11
          .
          .
          .
STR1      BYTE     C'TEST STRING'    11-BYTE STRING CONSTANT
STR2      RESB     11                11-BYTE VARIABLE
```

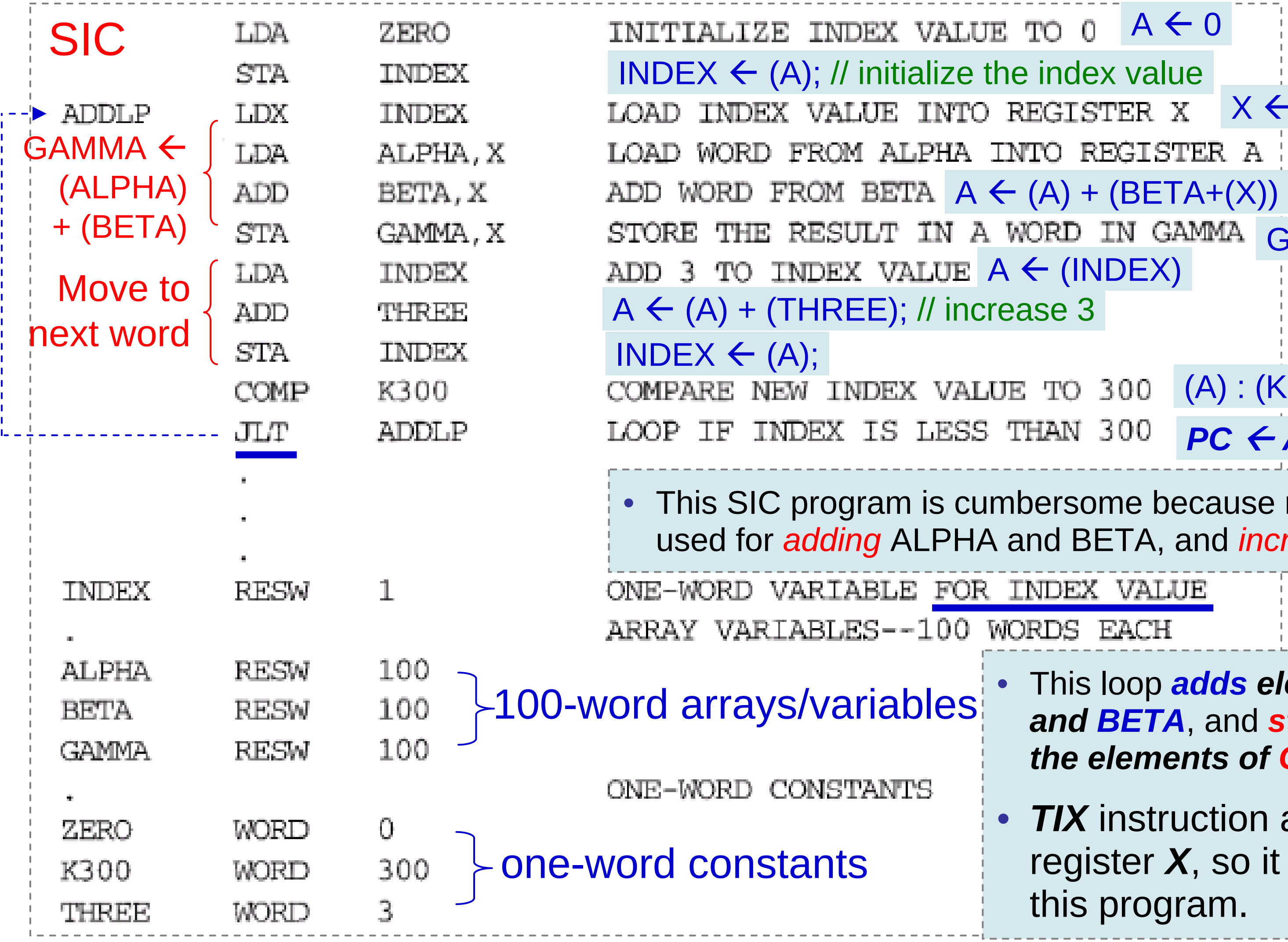
T ← 11

X ← 0

X ← (X) + 1
(X) : (T)

- ***TIXR*** allows to compare index register ***X*** with another register.
  - This is efficient because the value does not have to be fetched from memory each time the loop is executed.

Another Sample Indexing and Looping



SIC

```
         LDA    ZERO       INITIALIZE INDEX VALUE TO 0
         STA    INDEX
ADDLP    LDX    INDEX      LOAD INDEX VALUE INTO REGISTER X
         LDA    ALPHA,X    LOAD WORD FROM ALPHA INTO REGISTER A
         ADD    BETA,X     ADD WORD FROM BETA
         STA    GAMMA,X    STORE THE RESULT IN A WORD IN GAMMA
         LDA    INDEX      ADD 3 TO INDEX VALUE
         ADD    THREE
         STA    INDEX
         COMP   K300       COMPARE NEW INDEX VALUE TO 300
         JLT    ADDLP      LOOP IF INDEX IS LESS THAN 300
         .
         .
         .
INDEX    RESW   1          ONE-WORD VARIABLE FOR INDEX VALUE
.                          ARRAY VARIABLES--100 WORDS EACH
ALPHA    RESW   100
BETA     RESW   100
GAMMA    RESW   100
.                          ONE-WORD CONSTANTS
ZERO     WORD   0
K300     WORD   300
THREE    WORD   3
```

A ← 0

INDEX ← (A); // initialize the index value

X ← (INDEX)

A ← (ALPHA + (X))

A ← (A) + (BETA+(X))

GAMMA+(X) ← (A)

A ← (INDEX)

A ← (A) + (THREE); // increase 3

INDEX ← (A);

(A) : (K300)

PC* ← *ADDLP if CC set to <

GAMMA ← (ALPHA) + (BETA)

Move to next word

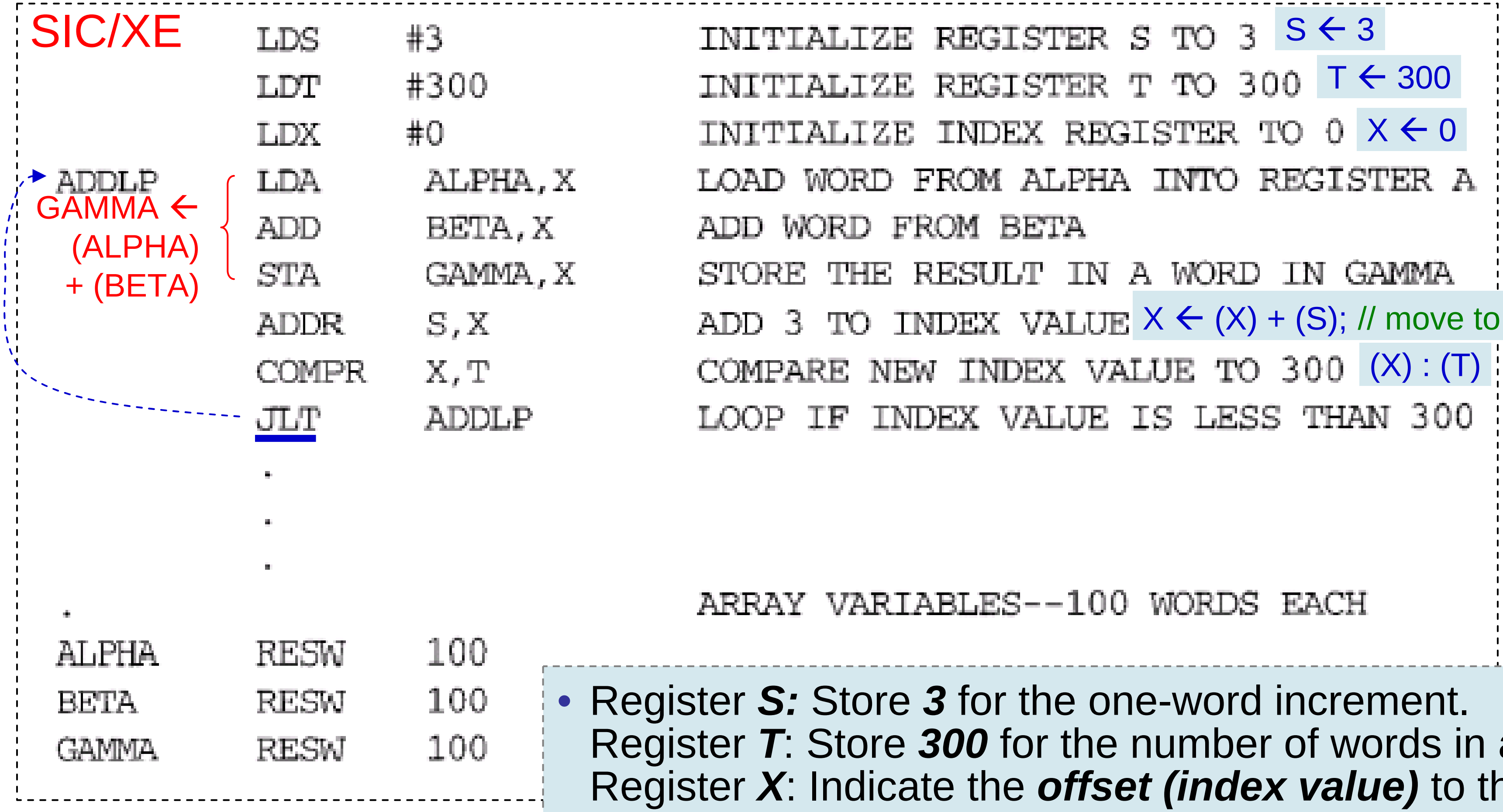
100-word arrays/variables

one-word constants

- This SIC program is cumbersome because register A must be used for *adding* ALPHA and BETA, and *incrementing* INDEX.

- This loop ***adds elements of ALPHA and BETA***, and ***stores the results in the elements of GAMMA***.
- ***TIX*** instruction always adds 1 to register ***X***, so it is not suitable to this program.

Another Sample Indexing and Looping (Cont.)



SIC/XE

```
         LDS     #3              INITIALIZE REGISTER S TO 3
         LDT     #300            INITIALIZE REGISTER T TO 300
         LDX     #0              INITIALIZE INDEX REGISTER TO 0
ADDLP    LDA     ALPHA,X         LOAD WORD FROM ALPHA INTO REGISTER A
         ADD     BETA,X          ADD WORD FROM BETA
         STA     GAMMA,X         STORE THE RESULT IN A WORD IN GAMMA
         ADDR    S,X             ADD 3 TO INDEX VALUE
         COMPR   X,T             COMPARE NEW INDEX VALUE TO 300
         JLT     ADDLP           LOOP IF INDEX VALUE IS LESS THAN 300
         .
         .
         .
.                                ARRAY VARIABLES--100 WORDS EACH
ALPHA    RESW    100
BETA     RESW    100
GAMMA    RESW    100
```

S ← 3

T ← 300

X ← 0

GAMMA ← (ALPHA) + (BETA)

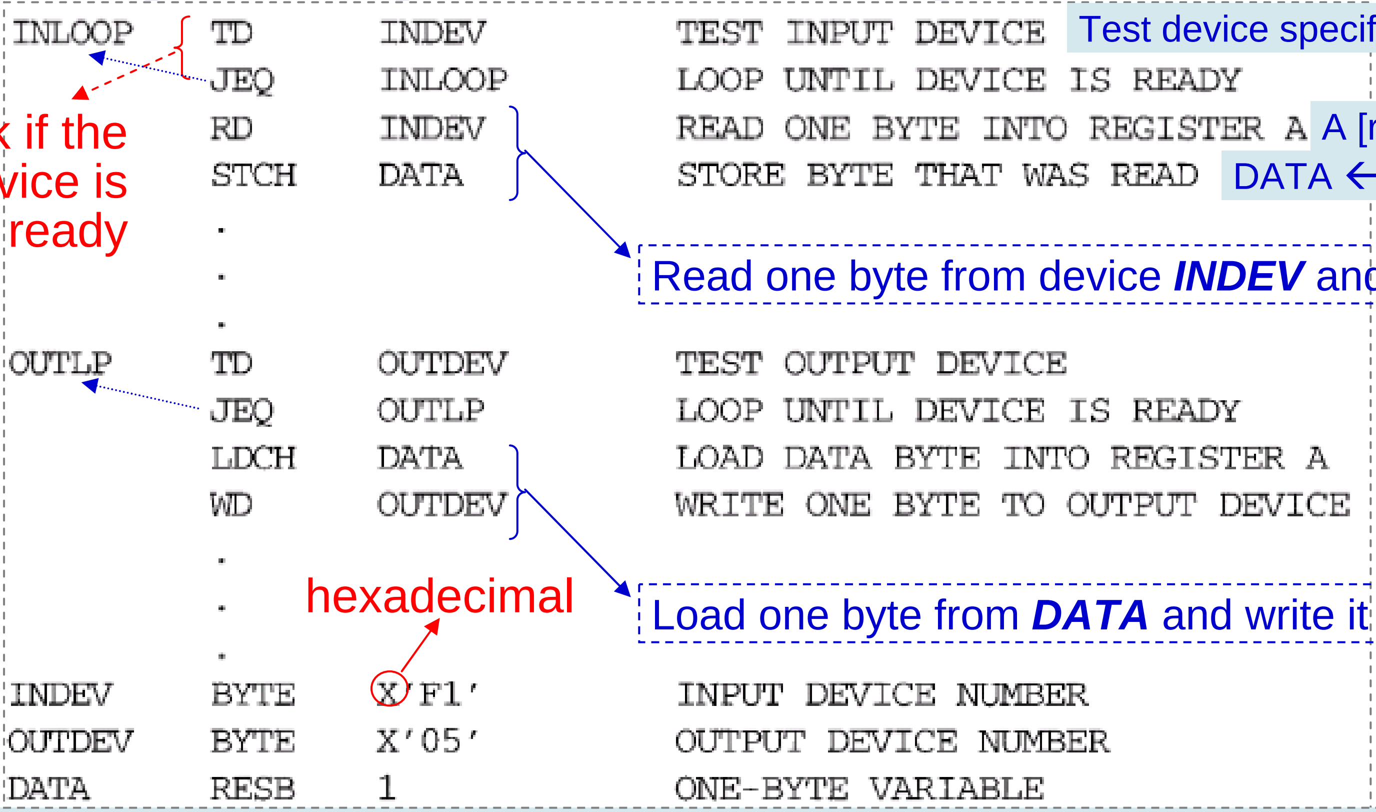
X ← (X) + (S); // move to next word

(X) : (T)

- Register ***S:*** Store ***3*** for the one-word increment.
  Register ***T***: Store ***300*** for the number of words in an array.
  Register ***X***: Indicate the ***offset (index value)*** to the starting of an array.

Simple Input and Output

SIC



```
INLOOP   TD       INDEV          TEST INPUT DEVICE
         JEQ      INLOOP         LOOP UNTIL DEVICE IS READY
         RD       INDEV          READ ONE BYTE INTO REGISTER A
         STCH     DATA           STORE BYTE THAT WAS READ
         .
         .
         .
OUTLP    TD       OUTDEV         TEST OUTPUT DEVICE
         JEQ      OUTLP          LOOP UNTIL DEVICE IS READY
         LDCH     DATA           LOAD DATA BYTE INTO REGISTER A
         WD       OUTDEV         WRITE ONE BYTE TO OUTPUT DEVICE
         .
         .
         .
INDEV    BYTE     X'F1'          INPUT DEVICE NUMBER
OUTDEV   BYTE     X'05'          OUTPUT DEVICE NUMBER
DATA     RESB     1              ONE-BYTE VARIABLE
```

Test device specified by (m)

Check if the device is ready

A [rightmost byte] ← data

DATA ← (A) [rightmost byte]

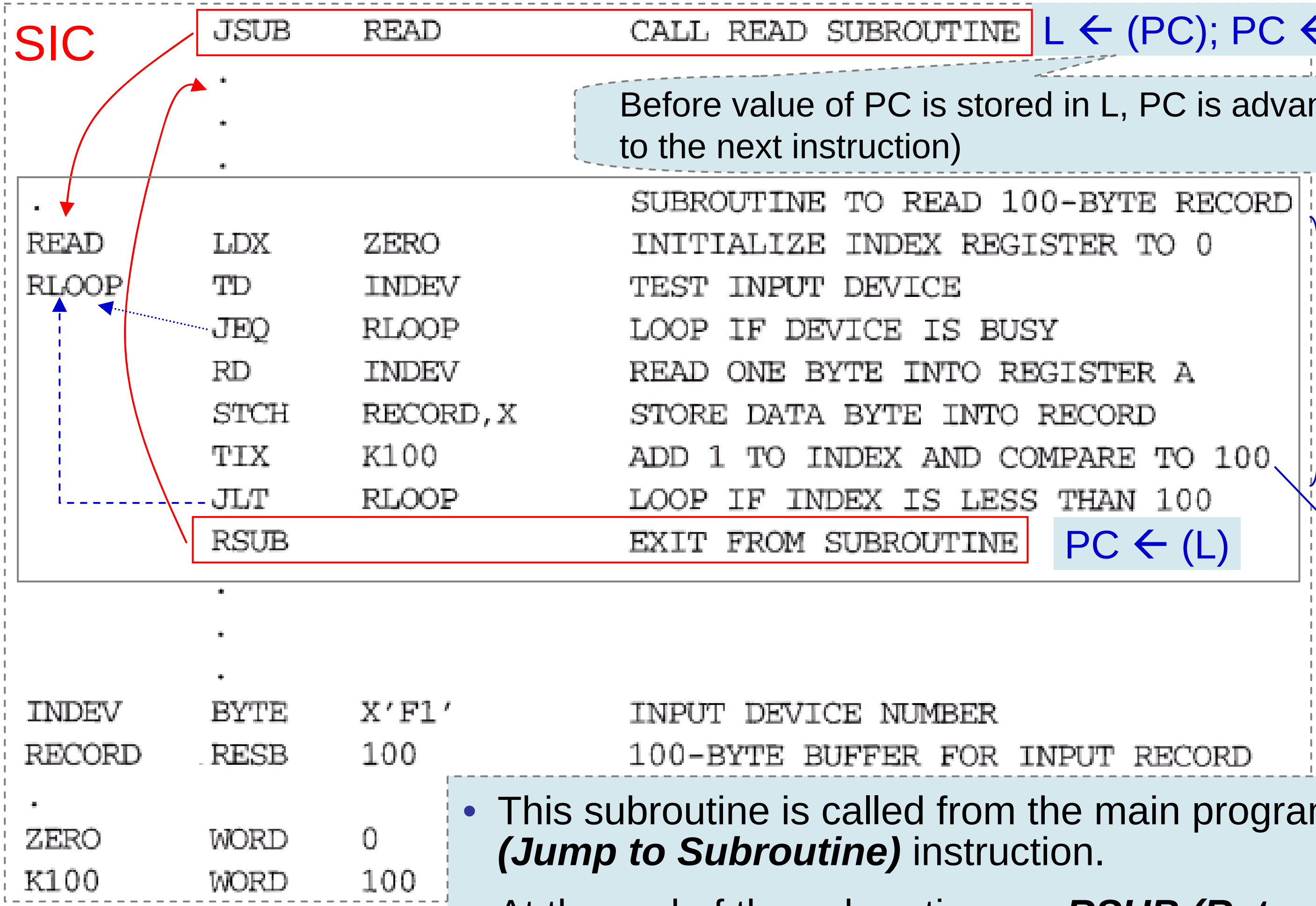
Read one byte from device ***INDEV*** and store it at ***DATA***

Load one byte from ***DATA*** and write it to device ***OUTDEV***

hexadecimal

- This program reads 1 byte of data from ***device F1*** and copies it to ***device 05***.
- ***TD*** instruction is to test whether the device is ready.
  - If the device is ***ready*** to transmit data, the condition code is set to "***less than***."
  - If the device is ***not ready***, the condition is set to "***equal***."

Sample Subroutine Call



SIC

```
          JSUB     READ          CALL READ SUBROUTINE
          .
          .
          .
.                                SUBROUTINE TO READ 100-BYTE RECORD
READ      LDX      ZERO          INITIALIZE INDEX REGISTER TO 0
RLOOP     TD       INDEV         TEST INPUT DEVICE
          JEQ      RLOOP         LOOP IF DEVICE IS BUSY
          RD       INDEV         READ ONE BYTE INTO REGISTER A
          STCH     RECORD,X      STORE DATA BYTE INTO RECORD
          TIX      K100          ADD 1 TO INDEX AND COMPARE TO 100
          JLT      RLOOP         LOOP IF INDEX IS LESS THAN 100
          RSUB                   EXIT FROM SUBROUTINE
          .
          .
          .
INDEV     BYTE     X'F1'         INPUT DEVICE NUMBER
RECORD    RESB     100           100-BYTE BUFFER FOR INPUT RECORD
.
ZERO      WORD     0
K100      WORD     100
```

L ← (PC); PC ← READ

Before value of PC is stored in L, PC is advanced by 3 (pointing to the next instruction)

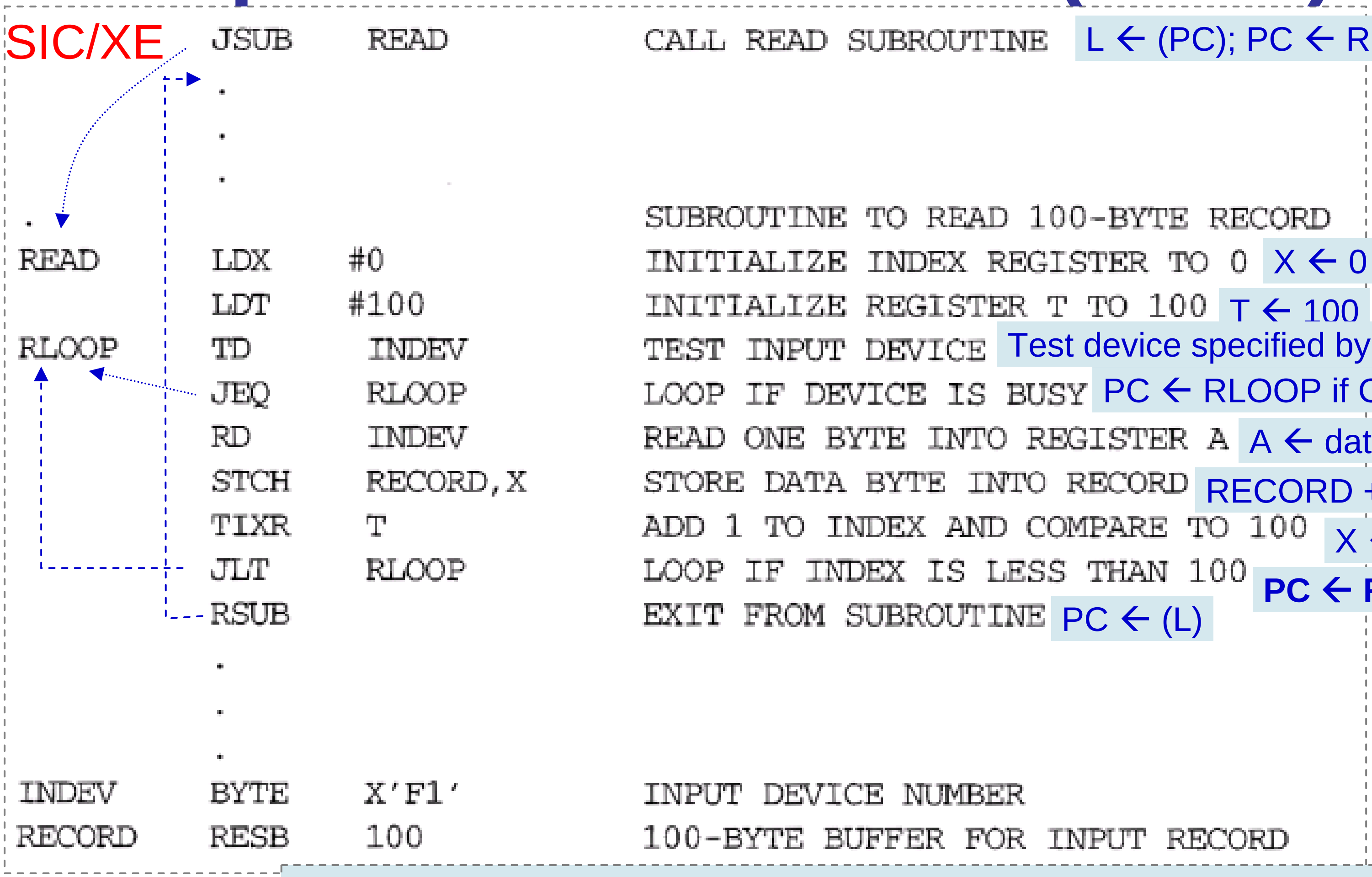
Read 100 bytes from device F1 and store them in a 100-bye buffer area labeled RECORD.

PC ← (L)

X ← (X) + 1
(X) : (K100)

- This subroutine is called from the main program by the ***JSUB (Jump to Subroutine)*** instruction.
- At the end of the subroutine, an ***RSUB (Return from Subroutine)*** instruction returns to the instruction *following the JSUB*.

Sample Subroutine Call (Cont.)



SIC/XE

```
        JSUB    READ            CALL READ SUBROUTINE            L ← (PC); PC ← READ
        .
        .
        .
.                               SUBROUTINE TO READ 100-BYTE RECORD
READ    LDX     #0              INITIALIZE INDEX REGISTER TO 0  X ← 0
        LDT     #100            INITIALIZE REGISTER T TO 100    T ← 100
RLOOP   TD      INDEV           TEST INPUT DEVICE               Test device specified by (INDEV)
        JEQ     RLOOP           LOOP IF DEVICE IS BUSY          PC ← RLOOP if CC set to =
        RD      INDEV           READ ONE BYTE INTO REGISTER A   A ← data
        STCH    RECORD,X        STORE DATA BYTE INTO RECORD     RECORD + (X) ← (A)
        TIXR    T               ADD 1 TO INDEX AND COMPARE TO 100   X ← (X) + 1; (X) : (T)
        JLT     RLOOP           LOOP IF INDEX IS LESS THAN 100  PC ← RLOOP if CC set to <
        RSUB                    EXIT FROM SUBROUTINE            PC ← (L)
        .
        .
        .
INDEV   BYTE    X'F1'           INPUT DEVICE NUMBER
RECORD  RESB    100             100-BYTE BUFFER FOR INPUT RECORD
```

- Register ***T***: Store ***100*** for the number of bytes to read.
- Register ***X***: Indicate the ***offset (index value)*** to the starting of an array.

CISC Machines

CISC Machines

- Complex instruction set computers (CISC) usually include
  - A relatively large and complicated instruction set.
  - Several different instruction formats and lengths.
  - Many different addressing modes.
- The implementation of CISC architecture in hardware tends to be complex.
- Sample CISC machines
  - VAX architecture
  - Pentium architecture

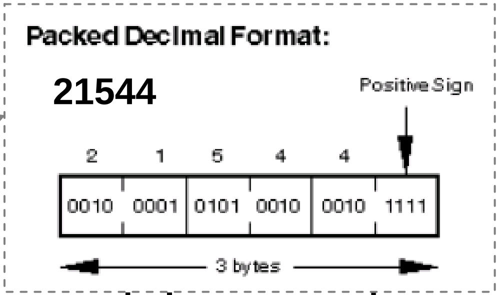
VAX Architecture

- ***Memory***
  - Bytes and words
    - 8 bits form a ***byte***.
    - 2 bytes form a ***word***.
    - 4 bytes form a ***longword***.
    - 8 bytes form a ***quadward***.
    - 16 bytes form a ***octaword***.
  - Some operations are more efficient when ***operands*** are aligned in a particular way.
    - E.g., a *longword* operand that begins at a byte address that is a multiple of 4.
  - Virtual address space: 2^{32} bytes
    - This virtual memory allows program to operate as though they have a large memory.
    - One half of virtual address space is ***system space*** that contains the *operating system* and is *shared by all programs*.
    - The other half of the address space is ***process space*** that is defined *separately for each program*.
      - ***A part of the process space*** contains stacks that are available to the program.

VAX Architecture (Cont.)

- ***Registers***
  - 16 general-purpose registers denoted by ***R0*** through ***R15.***
    - ***R15*** is the ***program counter (PC)***.
      - It is updated during instruction execution to point to the *next instruction*.
    - ***R14*** is the ***stack pointer (SP)***.
      - It points to the *current top of the stack* in the program’s *process space*.
    - ***R13*** is the ***frame pointer (FP)***.
      - VAX *procedure call conventions* build a data structure called a ***stack frame***, and place its address in FP.
    - ***R12*** is the ***argument pointer (AP)***.
      - The procedure call convention uses AP to pass *a list of arguments* associated with the call.
    - ***R6 through R11*** have no special functions.
    - ***R0 through R5*** are likewise available for general use, but are also used by some machine instructions.
  - A ***processor status longword (PSL)*** that contains *state variables and flags* associated with a process.

VAX Architecture (Cont.)



hexadecimal F for positive numbers and hexadecimal D for negative numbers

- ***Data Formats***
  - *Integers* are stored as *binary numbers* in a byte, word, longword, quadword, or octaword.
  - *2's complement representation* is used for *negative values*.
  - *Characters* are stored using their *8-bit ASCII codes*.
  - Four different floating-point data formats, ranging in length from 4 to 16 bytes.
  - VAX processors provide a *packed decimal* data format.
    - Each byte represents two decimal digits, with each digit encoded using 4 bits of the byte. (4 bits form a ***nibble***.)
    - The sign is encoded in the last 4 bits.
  - VAX also provides hardware to support *queues* and *variable-length bit strings*.
    - There are single machine instructions that
      - Insert and remove entries in queues. (support ***atomic*** operations)
      - Perform a variety of operations on bit strings.

VAX Architecture (Cont.)

- ***Instruction formats***
  - VAX machine instructions use a variable-length instruction format.
  - Each instruction consists of
    - An ***operation*** code (1 or 2 bytes)
    - Up to six ***operand specifiers***, depending on the type of instruction.
      - Each operand specifier designates one of the VAX addressing modes and gives any additional information to locate the operand.
- ***Addressing modes***
  - ***Register mode***: The operand itself is in a register.
  - ***Register deferred mode***: The address of an operand is specified by a register.
    - The register contents may be automatically incremented or decremented by the operand length (***autoincrement*** and ***autodecrement*** modes).
  - There are several ***base relative addressing modes***, with displacement fields of different lengths.
    - *E.g.*, ***PC-relative mode*** by using register PC for the displacement.
  - ***Immediate addressing*** is to include operands in the instruction itself.
  - All of the addressing modes could include an ***index register***, and many of them are available in a form that specifies ***indirect addressing***.

VAX Architecture (Cont.)

- ***Instruction set***
  - Instruction ***mnemonics*** are formed by combining the following elements:
    - 1. a ***prefix*** that specifies the type of *operations*,
    - 2. a ***suffix*** that specifies the data type of the *operands*,
    - 3. a ***modifier*** (on some instructions) that gives *the number of operands involved*.
  - E.g.,
    - ***ADDW2***: an add operation with ***two*** operands, and each a word in length.
    - ***MULL3***: a multiply operation with ***three*** longword operands.
    - ***CVTWL***: a conversion from word to longword, with ***two*** operands.
  - Operands may be located in registers, memory, or instruction itself (immediate addressing).
  - VAX provides all of the usual types of instructions for *computation*, *data movement and conversion*, *comparison*, *branching*, etc.

VAX Architecture (Cont.)

- ***Instruction set (Cont.)***
  - There are a number of operations that are much more complex than the machine instructions found on most computers.
    - These operations are hardware realizations of frequently occurring sequences of codes. E.g.,
      - Instructions to load and store *multiple registers*, and manipulate *queues* and *variable-length bit fields*.
      - Powerful instruction to call and return from procedures.
      - Single instruction to
        - Save a designated set of registers,
        - Pass a list of arguments to the procedure,
        - Maintain the stack, frame, and argument pointers, and
        - Set a mask to enable error traps for arithmetic operations.

VAX Architecture (Cont.)

- ***Input and output***
  - Input and output are accomplished by I/O device controllers.
  - Each controller has a set of control/status and data registers, which are assigned locations in the ***physical address space***.
  - The address space mapped by the device controller registers are called ***I/O space***.
    - No special instructions are required to access registers in I/O space.
    - An I/O device driver issues commands to the device controller by storing values into the appropriate *registers*, exactly *as if there were physical memory locations*.
    - The association of an address in I/O space with a physical register in a device controller is handled by the ***memory management routines***.

Pentium Architecture

- ***Memory***
  - At the physical level, memory consists of 8-bit bytes. All addresses used are byte addresses.
  - Programmers view the *x86* memory as ***a collection of segments***.
    - An address consists of two parts: A *segment number* and *an offset*.
      - Segments can be of different sizes, and are often used for different purposes. E.g., a segment for executable instructions or data storage.
      - A segment can be divided into ***pages***. Some of the pages of a segment may be in physical memory, while others may be stored on disk.
    - The segment/offset address specified by the programmer is automatically translated into a physical byte address by the x86 *Memory Management Unit (MMU)*.

Pentium Architecture (Cont.)

- ***Registers***
  - Eight general-purpose registers (32 bits):
    - ***EAX, EBX, ECX, and EDX*** are generally used for data manipulation. It is possible to access individual words or bytes from these registers.
    - ***ESI, EDI, EBP, and ESP*** are usually used to hold addresses.
  - ***EIP*** is a 32-bit register that contains a pointer to the next instruction to be executed. (like program counter (PC))
  - ***FLAGS*** is a 32-bit register that contains many different bit flags.
    - Some indicates the status of the processor.
    - Others are used to record the results of comparisons and arithmetic operations.
  - Six 16-bit ***segment registers*** to locate segments in memory.
    - Segment register ***CS*** contains the address of the ***currently executing code segment***.
    - Segment register ***SS*** contains the address of the ***current stack segment***.
    - ***DS, ES, FS, and GS*** are used to indicate ***the addresses of data segment***.
  - Floating-point computations are performed by a special ***floating-point unit (FPU)***.

Pentium Architecture (Cont.)

- ***Data formats***
  - ***Integers*** are normally stored as 8-, 16-, or 32-bit binary numbers. Negative values are represented in 2’s complement.
  - ***Little-endian*** byte ordering:
    - The least significant part of a numeric value is stored at the lowest-numbered address.
  - Two binary coded decimal (BCD) formats:
    - ***Unpacked BCD format***: Each byte represents one decimal digit.
    - ***Packed BCD format***: Each byte represents two decimal digits.
  - Three different floating-point data formats:
    - Single-precision format: 32 bits long (24 significant bits and 7-bit exponent).
    - Double-precision format: 64 bits long (53 significant bits and 10-bit exponent).
    - Extended-precision format: 80 bits long (64 significant bits and 15-bit exponent).
  - Characters are stored one per byte.

Pentium Architecture (Cont.)

- ***Instruction formats***
  - Basic format:
    - Begin with ***optional prefixes*** to contain flags that modify the operation of the instruction. E.g.,
      - Repetition count for an instruction.
      - Specification to a segment register that is used for addressing an operand (to override the normal default assumptions).
    - Following the prefix (if any) is an ***opcode*** (1 or 2 bytes).
      - The opcode is the only element that is always present in every instruction.
    - Following the opcode are a number of bytes that specify the ***operands*** and ***addressing modes*** to be used.
  - There are a large number of different potential instruction formats, varying in length from 1 byte to 10 bytes or more.

Pentium Architecture (Cont.)

- ***Addressing modes***
  - An operand value may be specified as part of the instruction itself (***immediate mode***), or it may be in a register (***register mode***).
  - Operands stored in memory are often specified by the ***target address calculation***:

    TA = (base register) + (index register) * (scale factor) + displacement

    - Any general-purpose register may be used as a *base register*.
    - Any general-purpose register except ESP can be used as an *index register*.
    - The scale factor may have the value 1, 2, 4, or 8.
    - The displacement may be an 8-, 16-, or 32-bit value.
  - Various combinations of these items may be omitted, resulting in ***eight*** different addressing modes.
  - The address of an operand in memory may be specified as an absolute location (***direct mode***), or as a location relative to the EIO register (***relative mode***).

Pentium Architecture (Cont.)

- ***Instruction set***
  - The x86 architecture has a large and complex instruction set (more than 400 different machine instructions).
  - An instruction may have *zero*, *one*, *two*, or *three* operands.
  - There are *register-to-register instruction*, *register-to-memory instructions*, and a few *memory-to-memory instructions.*
  - Most data movement and integer arithmetic instructions can use operands that are 1, 2, or 4 bytes long.
  - ***String manipulation*** instructions can deal with variable-length strings of bytes, words, or doublewords.
  - Many instructions that
    - Perform ***logical and bit manipulations***, and
    - Support control of the processor and memory-management systems.
  - The x86 architecture includes special-purpose instructions to perform operations frequently required in *high-level programming languages*.
    - E.g., entering and leaving procedures, and checking subscript values against the bounds of an array.

Pentium Architecture (Cont.)

- ***Input and output***
  - Input is performed by instructions that transfer one byte, word, or doubleword at a time ***from an I/O port into register EAX***.
  - Output instructions transfer one byte, word, or doubleword ***from EAX to an I/O port***.
  - ***Repetition prefixes*** allow to transfer an entire string in a single operation.
  - ***Special I/O instructions are needed.***

RISC Machines

RISC Machines

- Reduced Instruction Set Computers (RISC) have a simplified design that results in
  - Faster and less expensive processor development
  - Greater reliability and
  - Faster instruction execution times.
- A RISC system is characterized by
  - A ***fixed instruction length***, and
  - *Single-cycle execution* of most instructions.
- Memory access is usually done by *load and store instructions* only.
- All instructions except for load and store are *register-to-register operations*.
- There are typically a large number of general-purpose registers.
- The number of machine instructions, instruction formats, and addressing modes is relatively small.
- Sample CISC machines are UltraSPARC, PowerPC, and Cray T3E.

UltraSPARC Architecture

- The UltraSPARC processor was announced by Sun.
  - SPARC stands for ***S***calable ***P***rocessor ***Ar***chitecture.
- ***Memory***
  - Bytes and words
    - 8-bit forms a ***byte***.
    - 2 consecutive bytes form a ***halfword***.
    - 4 bytes form a ***word***.
    - 8 bytes form a ***doubleword***.
  - Virtual address space is 2^{64} bytes.
    - This address space is divided into ***pages***.
    - Multiple page sizes are supported.
    - Pages used by a program could be in physical memory or on disk.
    - The virtual address specified by instruction is translated into a physical address by MMU.

UltraSPARC Architecture (Cont.)

- ***Registers***
  - SPARC includes a large *register file* that usually contains more than 100 general-purpose registers.
    - Procedures could only access 32 registers designated to ***r0 through r31***.
      - r0 through r7 are global: They can be accessed by all procedures.
      - The other 24 registers available to a procedure can be visualized as a ***window*** through which part of the ***register file*** can be seen, so some registers in the register file are shared between procedures.
        - » E.g., Registers r8 through r15 of a calling procedure are physically the same registers as r24 through r31 of the called procedure (for parameter passing).
    - If a set of concurrently running procedures need more windows than available, a "***window overflow***" interrupt occurs to trigger saving the contents of some registers in the file (and restore them later).
  - Floating-point computations are performed through FPU.
  - In addition, there are a program counter, condition code registers, and a number of other control registers.

UltraSPARC Architecture (Cont.)

- ***Data formats***
  - It provides integers, floating-point values, and characters.
  - Integers are 8-,16-, 32-, or 64-bit binary numbers. (2's complement for negative values).
  - ***Big-endian*** byte ordering is supported: The most significant part of a numeric value is stored at the lowest-numbered address.
  - Three different floating-point data formats:
    - ***Single-precision format***: 32 bits long (23 significant bits and 8-bit exponent)
    - ***Double-precision format***: 64 bits long (52 significant bits and 11-bit exponent)
    - ***Quad-precision format***: 80 bits long (63 significant bits and 15-bit exponent)
  - Characters are stored one per byte, using 8-bit ASCII codes.

UltraSPARC Architecture (Cont.)

- ***Instruction formats***
  - Every instruction is 32 bits long.
    - The ***first 2 bits*** identify which format is used.
      - ***Format 1*** is used for the ***call*** instruction.
      - ***Format 2*** is used for ***branch*** instructions.
      - ***Format 3*** provides ***register loads and stores***, and ***three-operand arithmetic operations***.
- ***Addressing modes***
  - An operand value is specified as part of the instruction itself (***immediate mode)*** or is in a register (***register direct mode***).
  - Operands in memory are addressed using one of the following mode:

Mode	Target address (TA) calculation
PC-relative	TA = (PC) + displacement{30 bits, signed}
Register indirect with displacement	TA = (register) + displacement{13 bits, signed}
Register indirect indexed	TA = (register-1) + (register-2)

PC-relative mode is only for branch instruction

UltraSPARC Architecture (Cont.)

- ***Instruction set***
  - Fewer than 100 machine instructions (reflecting RISC philosophy).
  - *Load and store* architecture: only load and store instructions access memory.
  - Instruction execution is ***pipelined***.
    - Pipeline means that while one instruction is being executed, the next one is being fetched from memory and decoded.
    - An branch instruction might cause the process to "***stall***."
      - The instruction following the branch would have to be discarded even if it is fetched and decoded.
    - To make pipeline more efficient, SPARC branch instructions are ***delayed branches***.
      - A delayed branch means that the instruction immediately following the branch instruction is actually executed *before* the branch is taken.

```
SUB      %L0, 11, %L1  // %L1 ← (%L0) - 11
BA       NEXT  // PC ← NEXT
MOV      %L1, %O3 // %O3 ← (%L1)
```

In the ***delay slot: MOV*** is executed before the branch ***BA***.

UltraSPARC Architecture (Cont.)

- ***Instruction set (Cont.)***
  - High-bandwidth ***block load and store*** operations.
  - Communication in a multi-processor system is facilitated by "***atomic***" instructions that can execute without allowing other memory accesses to intervene.
  - ***Conditional move instructions*** may allow a compiler to eliminate many branch instruction.
- ***Input and output***
  - Communication with I/O devices is accomplished through memory.
  - A range of memory locations is logically replaced by ***device registers***.
    - Each I/O device has a unique set of addresses assigned to it.
    - When a load or store instruction refers to this device register area of memory, the corresponding device is activated. (***No special I/O instructions are needed.***)

PowerPC Architecture

- POWER is an acronym for ***P***erformance ***O***ptimization ***W***ith ***E***nhanced ***R***ISC.
- ***Memory***
  - Bytes and words
    - 8 bits form a ***byte***.
    - 2 consecutive bytes form a ***halfword***.
    - 4 bytes form a ***word***.
    - 8 bytes form a ***doubleword***.
    - 16 bytes form a ***quadword***.
  - All addresses are byte addresses.
  - If ***operands*** are aligned at a starting address that is *a multiple of their length*.
  - Virtual address space is of 2^{64} bytes.
    - This space is divided into fixed-length ***segments***, which are 256 MBs long.
    - Each segment is divided into ***pages***, which are 4096 bytes long.
    - A page could be in memory or on disk.
    - When an instruction is executed, the hardware and OS make sure the needed page is loaded to physical memory.
    - The virtual address specified by the instruction is automatically translated into a physical address by MMU.

PowerPC Architecture (Cont.)

- ***Registers***
  - 32 general-purpose registers, designated ***GPR0 through GPR31***. Each is 64 bits long.
  - Floating-point computations are performed using a special floating-point unit (***FPU***).
  - A 32-bit ***condition register (CR)*** reflects the result of certain operations such as testing, branching, and arithmetic.
    - This register is divided into *eight 4-bit subfields*, named ***CR0 through CR7***.
  - Other registers:
    - A ***link register (LR)*** and a ***count register (CR)*** for branch instructions.
    - A ***machine status register (MSR)*** and other control and status registers.

PowerPC Architecture (Cont.)

- ***Data formats***
  - ***Integers*** are stored as 8-, 16-, 32-, or 64-bit binary numbers.
  - ***2's complement*** is used to store negative values.
  - ***Big-endian*** ***byte ordering***: The most significant part of a numeric value is stored at the lowest-numbered address. (Little-endian is also supported).
  - There are two float-point data formats:
    - ***Single-precision format***: 32-bit long (23 significant bits and 8-bit exponent)
    - ***Double-precision format***: 64-bit long (52 significant bits and 11-bit exponent)
  - Characters are stored one per byte, using 8-bit ASCI codes.

PowerPC Architecture (Cont.)

- ***Instruction formats***
  - There are seven basic instruction formats (32 bits long).
  - Instructions must be aligned at a word boundary.
    - The first 6 bits specify the ***opcode***, and some have an additional "***extended opcode***" field.
  - The variety and complexity of instruction formats is greater than that on SPARC.
  - The fixed length makes instruction decoding faster and simpler than CISC.

PowerPC Architecture (Cont.)

- ***Addressing modes***
  - An operand value is specified as part of the instruction itself (***Immediate mode)*** or is in a register (***register direct mode***).
  - Only load/store operations and branch operations would access memory.
  - ***Load and store operations*** use one of the following addressing modes (where the register numbers and displacement are encoded in the instruction):

Mode	Target address (TA) calculation
Register indirect	TA = (register)
Register indirect with index	TA = (register-1) + (register-2)
Register indirect with immediate index	TA = (register) + displacement {16 bits, signed}

PowerPC Architecture (Cont.)

- ***Addressing modes***
  - Branch instructions use one of the following addressing modes:

Mode	Target address (TA) calculation
Absolute	TA = actual address
Relative	TA = current instruction address + displacement{25 bits, signed}
Link Register	TA = (LR)
Count Register	TA = (CR)

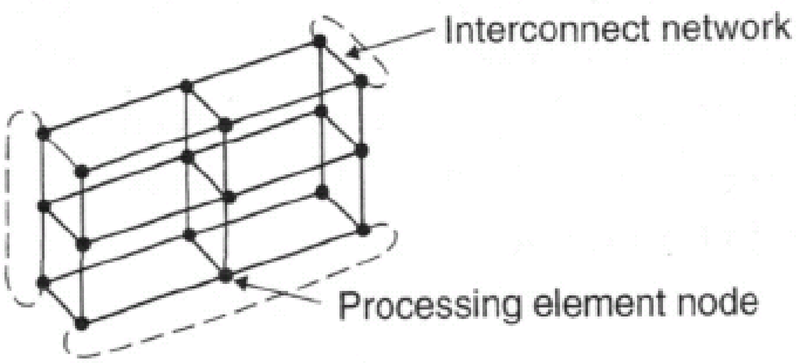
  - The absolute address or displacement is encoded as part of the instruction.

PowerPC Architecture (Cont.)

- ***Instruction set***
  - PowerPC has approximately 200 machine instructions.
    - Some are more complex than those in most RISC systems.
    - E.g.,
      - Load and store instructions may automatically update the index registers.
      - Instructions could perform multiplication and addition in one instruction.
  - Instruction execution on PowerPC is ***pipeline***.
  - ***Branch prediction*** is used to speed execution.
- ***Input and output***
  - Two different methods for performing I/O operations.
    - Segments in the virtual address space are mapped onto ***an external address space***. (called ***direct-store segments)***
    - A reference to an address represents a ***normal virtual memory access***. I/O is performed using the regular virtual memory management hardware and software.

Cray T3E Architecture (Alpha)

- Cray T3E is an architecture of *supercomputers* and is a ***massively parallel processing (MPP)*** system.
- A T3E system contains a large number of processing elements (PE), arranged in a three-dimensional network.
  - The network provides a ***path*** for transferring data between processors.
  - ***Control functions*** are provided to synchronize the operation of the PEs used by a program.
  - The interconnect network is ***circular*** in each dimension.
- Each PE consists of
  - A DEC ***Alpha*** EV5 RISC microprocessor,
  - Local memory, and
  - Performance-accelerating control logic.
- A T3E system may contain from 16 to 2048 PEs.



Cray T3E Architecture (Cont.)

- ***Memory***
  - Each PE has its own local memory with a capacity of from 64 MB to 2 GB.
    - The local memory within each PE is part of a ***physically distributed, logically shared*** memory system.
    - One PE could access the memory of another PE.
  - Bytes and words
    - 8 bits form a ***byte***.
    - 2 consecutive bytes form a ***word***.
    - 4 bytes form a ***longword***.
    - 8 bytes form a ***quadword***.
  - All addresses are byte-addressable.
  - Instruction execution could be more efficient if operands are ***aligned*** at a starting address that is a multiple of their length.

Cray T3E Architecture (Cont.)

- ***Registers***
  - There are 32 64-bit general-purpose registers (***R0 through R31***). *R31* always contains the value zero.
  - There are 32 64-bit floating-point registers (***F0 through F31***). ***F31*** always contains the value zero.
  - Other registers: a 64-bit program counter ***PC***, and several ***status and control registers***.
- ***Data formats***
  - ***Integers*** are stored as lognwords or quadwords.
  - ***2's complement*** is used to store negative values.
  - There are two types float-point data formats:
    - One group of three formats is included for compatibility with the VAX architecture.
    - The other group consists of four IEEE standard formats.
  - Characters are stored one per byte, using 8-bit ASCII codes.
    - There are no byte load or store operations in ***Alpha***. Only longwords and quadwords can be transferred between a register and memory.
      → *Characters are usually stored one per longword. (Trade-off between performance and space)*

Cray T3E Architecture (Cont.)

- ***Instruction formats***
  - There are five basic instruction formats in the Alpha architecture.
  - Each instruction is 32 bits long.
    - The first 6 bits specify the ***opcode***.
    - Some instruction have an additional "***function***" field.
- ***Addressing modes***
  - An operand value is specified as part of the instruction itself (***Immediate mode)*** or is in a register (***register direct mode***).
  - Only load/store operations and branch operations would access memory.

For load and store operations and subroutine calls.

Mode	Target address (TA) calculation
PC-relative	TA = (PC) + displacement {23 bits, signed}
Register indirect with displacement	TA = (register) + displacement {16 bits, signed}

For conditional and unconditional branches

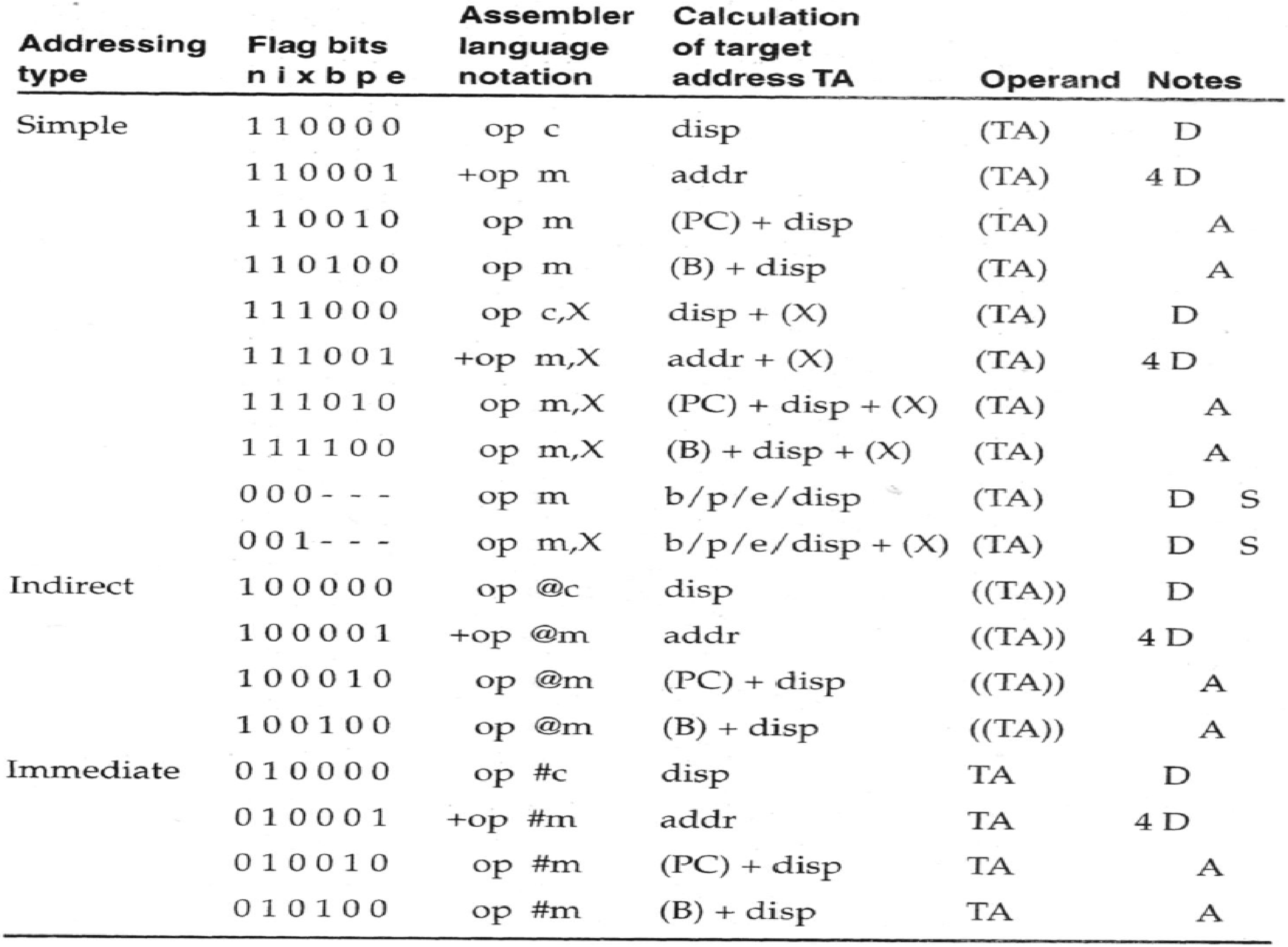
Cray T3E Architecture (Cont.)

- ***Instruction set***
  - There are approximately 130 machine instruction, reflecting its RISC orientation.
  - There are no byte or word load and store instructions.
    - This means that the memory access interface does not need to include ***shift-and-mask*** operations.
- ***Input and output***
  - The T3E system performs I/O through ***multiple ports*** into one or more ***I/O channels***.
  - Channels are integrated into the network that interconnects the processing nodes.
  - A system may be configured with up to one I/O channel for every eight PEs.
  - All channels are accessible and controllable from all PEs.

References to Addressing Modes and Instruction Set

Addressing Modes

- Flag bits
  - Addressing types
    - ***n*** indicates indirect addressing (Addressing types)
    - ***i*** indicates immediate addressing
  - Addressing modes
    - ***b*** indicates base-relative addressing mode (Addressing modes)
    - ***p*** indicates PC-relative addressing mode
    - ***x*** indicates indexed addressing mode
  - Extension
    - ***e*** indicates extended format (i.e., Format 4) (Extension)
- Assembler language notation:
  - ***c*** indicates a constant between 0 and 4095.
  - ***m*** indicates a ***memory address*** or a ***constant value*** larger than 4095.
  - **+** indicates extended format (i.e., ***Format 4***)
  - @ indicates a ***indirect addressing***
  - # indicates a ***immediate addressing***
  - ***X*** indicates register X (using indexed addressing)
- Letters in Notes
  - **4** Format 4 instruction
  - **D** Direct-addressing instruction (***b=0, p=0***)
  - **A** Program-counter relative or base relative mode (***b=1*** *or* ***p=1***)
  - **S** Compatible with standard SIC instructions. Operand value is between 0 and 32,767.



Addressing type	Flag bits n i x b p e	Assembler language notation	Calculation of target address TA	Operand	Notes
Simple	1 1 0 0 0 0	op c	disp	(TA)	D
	1 1 0 0 0 1	+op m	addr	(TA)	4 D
	1 1 0 0 1 0	op m	(PC) + disp	(TA)	A
	1 1 0 1 0 0	op m	(B) + disp	(TA)	A
	1 1 1 0 0 0	op c,X	disp + (X)	(TA)	D
	1 1 1 0 0 1	+op m,X	addr + (X)	(TA)	4 D
	1 1 1 0 1 0	op m,X	(PC) + disp + (X)	(TA)	A
	1 1 1 1 0 0	op m,X	(B) + disp + (X)	(TA)	A
	0 0 0 - - -	op m	b/p/e/disp	(TA)	D S
	0 0 1 - - -	op m,X	b/p/e/disp + (X)	(TA)	D S
Indirect	1 0 0 0 0 0	op @c	disp	((TA))	D
	1 0 0 0 0 1	+op @m	addr	((TA))	4 D
	1 0 0 0 1 0	op @m	(PC) + disp	((TA))	A
	1 0 0 1 0 0	op @m	(B) + disp	((TA))	A
Immediate	0 1 0 0 0 0	op #c	disp	TA	D
	0 1 0 0 0 1	+op #m	addr	TA	4 D
	0 1 0 0 1 0	op #m	(PC) + disp	TA	A
	0 1 0 1 0 0	op #m	(B) + disp	TA	A

Instruct Set of SIC/XE

Format 1 (1 byte):

8
op

Format 2 (2 bytes):

8	4	4
op	r1	r2

register

opcode

Format 3 (3 bytes):

6	1	1	1	1	1	1	12
op	n	i	x	b	p	e	disp

Base relative addressing

Program-counter relative addressing

Displacement

Format 4 (4 bytes):

6	1	1	1	1	1	1	20
op	n	i	x	b	p	e	addr

Immediate addressing

Indirect addressing

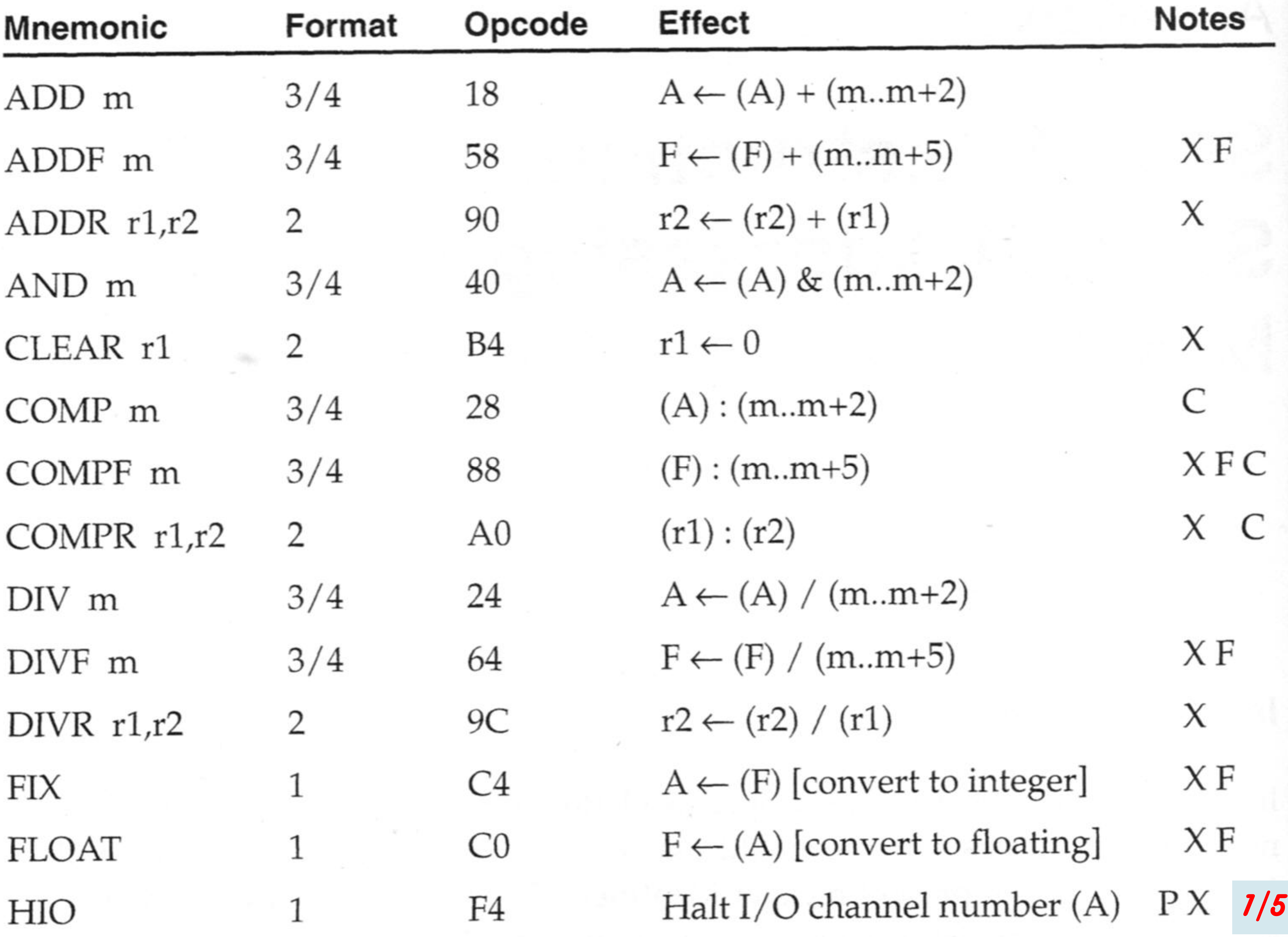
Indexed addressing

Extension format
Format 3: e = 0
Format 4: e = 1

Address

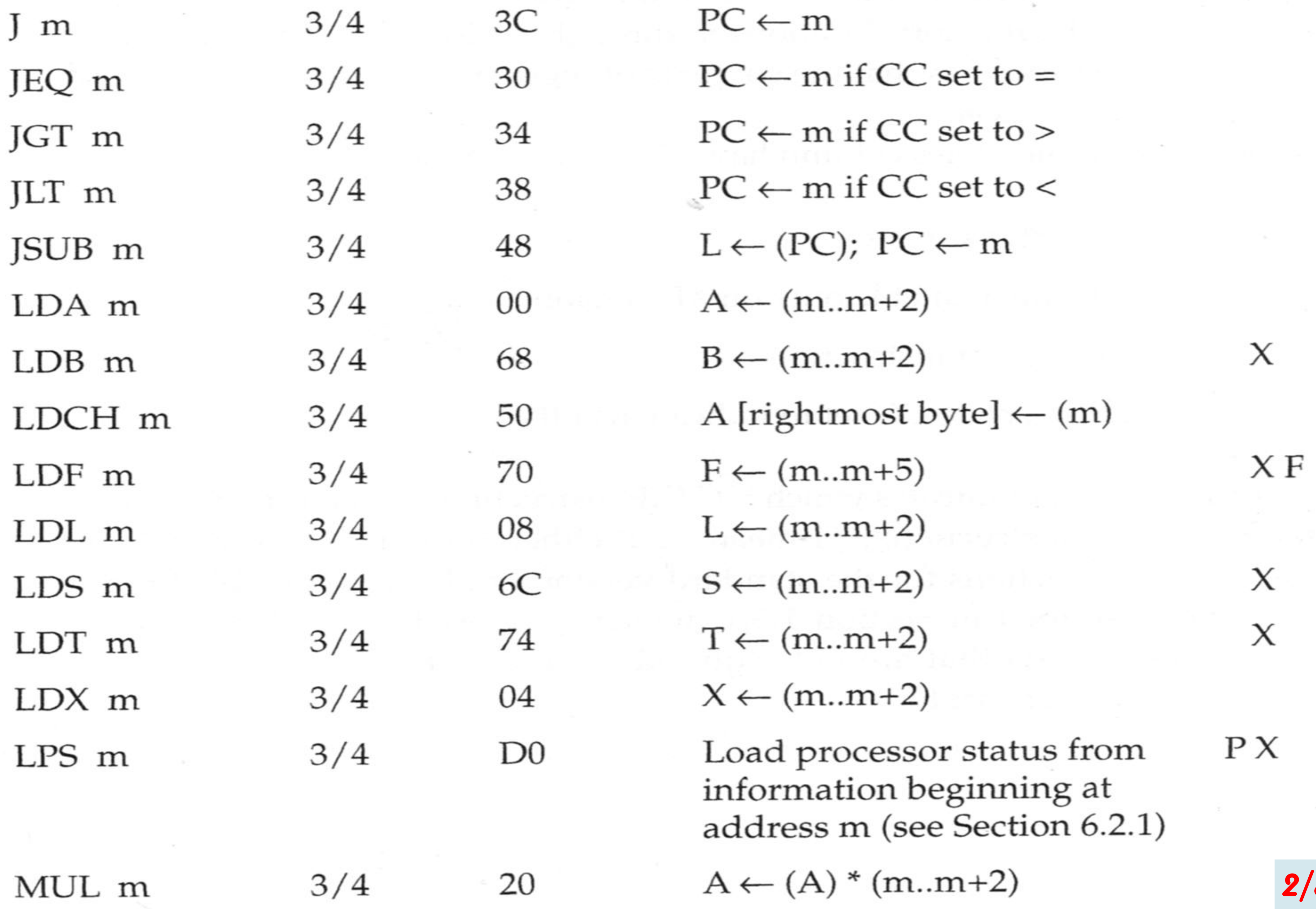
Instruction Set Table of SIC/XE

- Mnemonic
  - ***m*** indicates a memory address.
  - ***n*** indicates an integer between 1 and 16.
  - ***r1, r2*** represent register identifiers.
- Format :
  - Format indicates which SIC/XE instruction format is to be used in assembling each instruction. (***3/4*** : means that either Format 3 or Format 4 can be used.)
  - All instructions for the standard SIC are compatible with *Format 3* of SIC/XE.
  - Instruction subfields that are not required are set to *zero*.
- Effect
  - ***Parentheses*** are used to denote the contents of a register or memory location.
    - ***A←(m..m+2)*** specifies that the contents of the memory locations *m* through *m+2* are loaded into register *A*.
    - ***m..m+2←(A)*** specifies that the contents of register *A* are stored in the word that begins at address *m*.
- Notes
  - ***P*** : ***P****rivileged* instruction
  - ***X*** : Instruction available only on *SIC/****X****E* version
  - ***F*** : ***F****loating-point* instruction
  - ***C*** : ***Condition code CC*** set to indicate result of operation (<, =, or >)

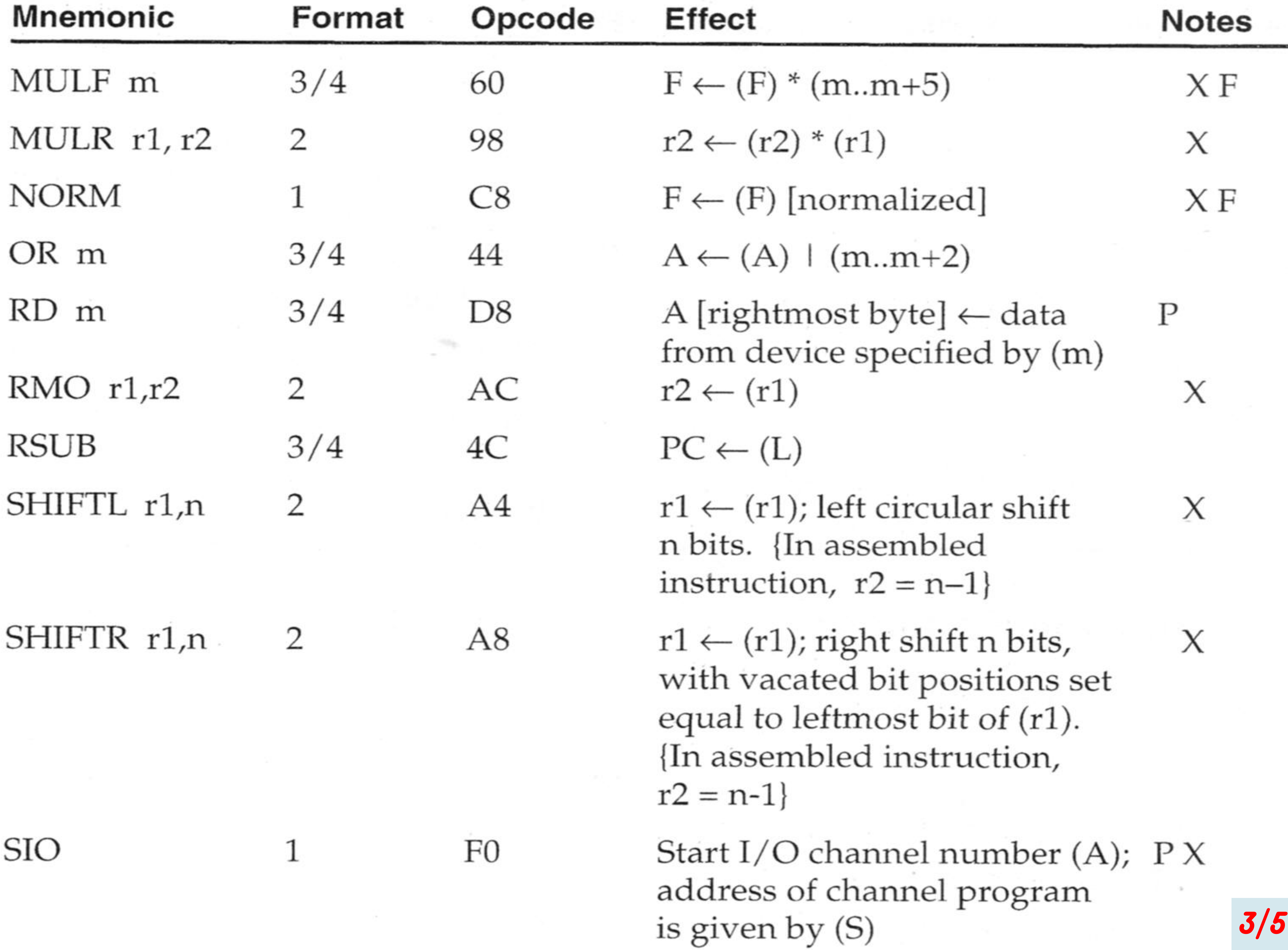


Mnemonic	Format	Opcode	Effect	Notes
ADD m	3/4	18	A ← (A) + (m..m+2)	
ADDF m	3/4	58	F ← (F) + (m..m+5)	X F
ADDR r1,r2	2	90	r2 ← (r2) + (r1)	X
AND m	3/4	40	A ← (A) & (m..m+2)	
CLEAR r1	2	B4	r1 ← 0	X
COMP m	3/4	28	(A) : (m..m+2)	C
COMPF m	3/4	88	(F) : (m..m+5)	X F C
COMPR r1,r2	2	A0	(r1) : (r2)	X C
DIV m	3/4	24	A ← (A) / (m..m+2)	
DIVF m	3/4	64	F ← (F) / (m..m+5)	X F
DIVR r1,r2	2	9C	r2 ← (r2) / (r1)	X
FIX	1	C4	A ← (F) [convert to integer]	X F
FLOAT	1	C0	F ← (A) [convert to floating]	X F
HIO	1	F4	Halt I/O channel number (A)	P X

1/5



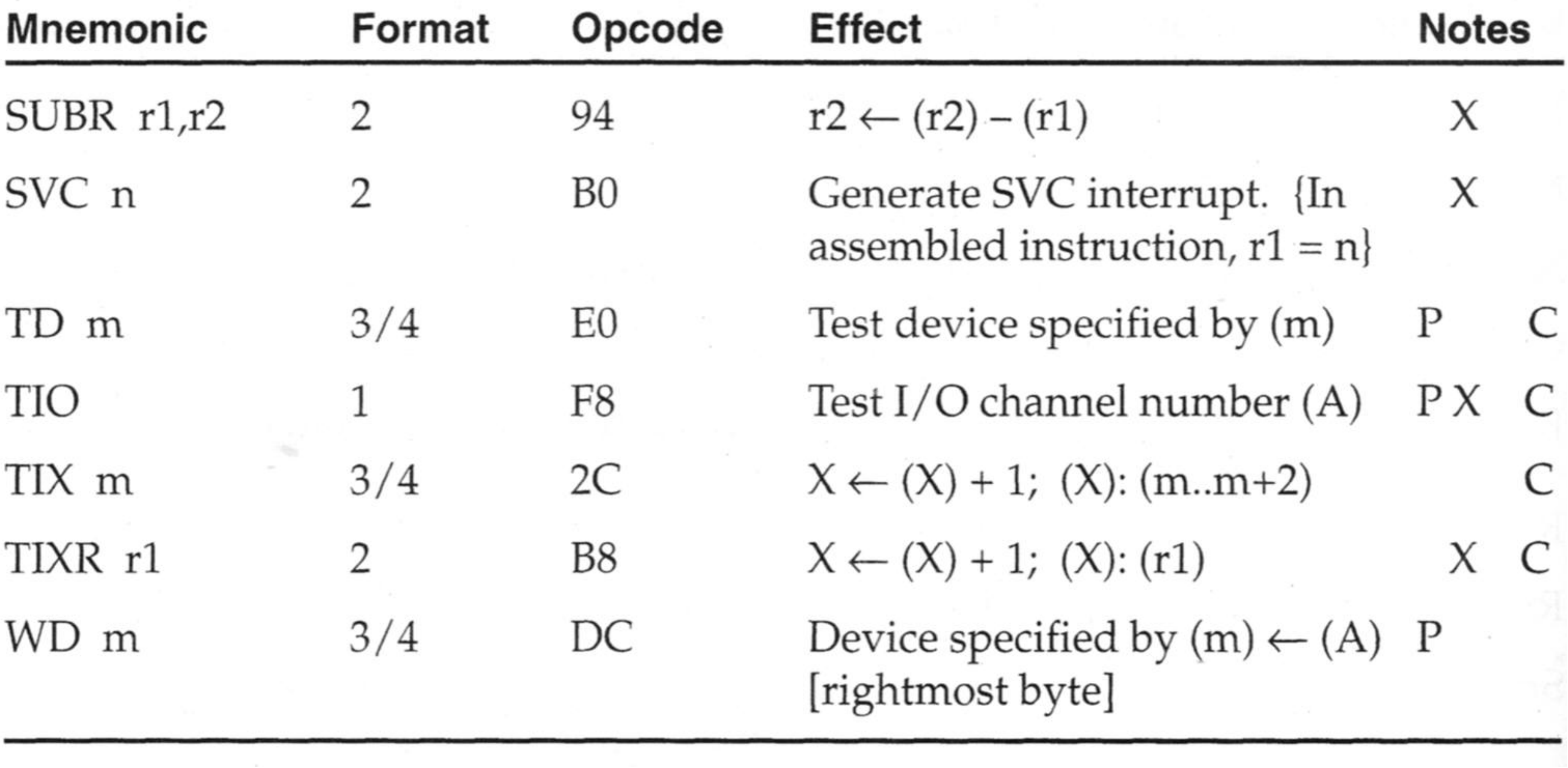
Mnemonic	Format	Opcode	Effect	Notes
J m	3/4	3C	PC ← m	
JEQ m	3/4	30	PC ← m if CC set to =	
JGT m	3/4	34	PC ← m if CC set to >	
JLT m	3/4	38	PC ← m if CC set to <	
JSUB m	3/4	48	L ← (PC); PC ← m	
LDA m	3/4	00	A ← (m..m+2)	
LDB m	3/4	68	B ← (m..m+2)	X
LDCH m	3/4	50	A [rightmost byte] ← (m)	
LDF m	3/4	70	F ← (m..m+5)	X F
LDL m	3/4	08	L ← (m..m+2)	
LDS m	3/4	6C	S ← (m..m+2)	X
LDT m	3/4	74	T ← (m..m+2)	X
LDX m	3/4	04	X ← (m..m+2)	
LPS m	3/4	D0	Load processor status from information beginning at address m (see Section 6.2.1)	P X
MUL m	3/4	20	A ← (A) * (m..m+2)	



Mnemonic	Format	Opcode	Effect	Notes
MULF m	3/4	60	F ← (F) * (m..m+5)	X F
MULR r1, r2	2	98	r2 ← (r2) * (r1)	X
NORM	1	C8	F ← (F) [normalized]	X F
OR m	3/4	44	A ← (A) | (m..m+2)	
RD m	3/4	D8	A [rightmost byte] ← data from device specified by (m)	P
RMO r1,r2	2	AC	r2 ← (r1)	X
RSUB	3/4	4C	PC ← (L)	
SHIFTL r1,n	2	A4	r1 ← (r1); left circular shift n bits. {In assembled instruction, r2 = n–1}	X
SHIFTR r1,n	2	A8	r1 ← (r1); right shift n bits, with vacated bit positions set equal to leftmost bit of (r1). {In assembled instruction, r2 = n-1}	X
SIO	1	F0	Start I/O channel number (A); address of channel program is given by (S)	P X

3/5

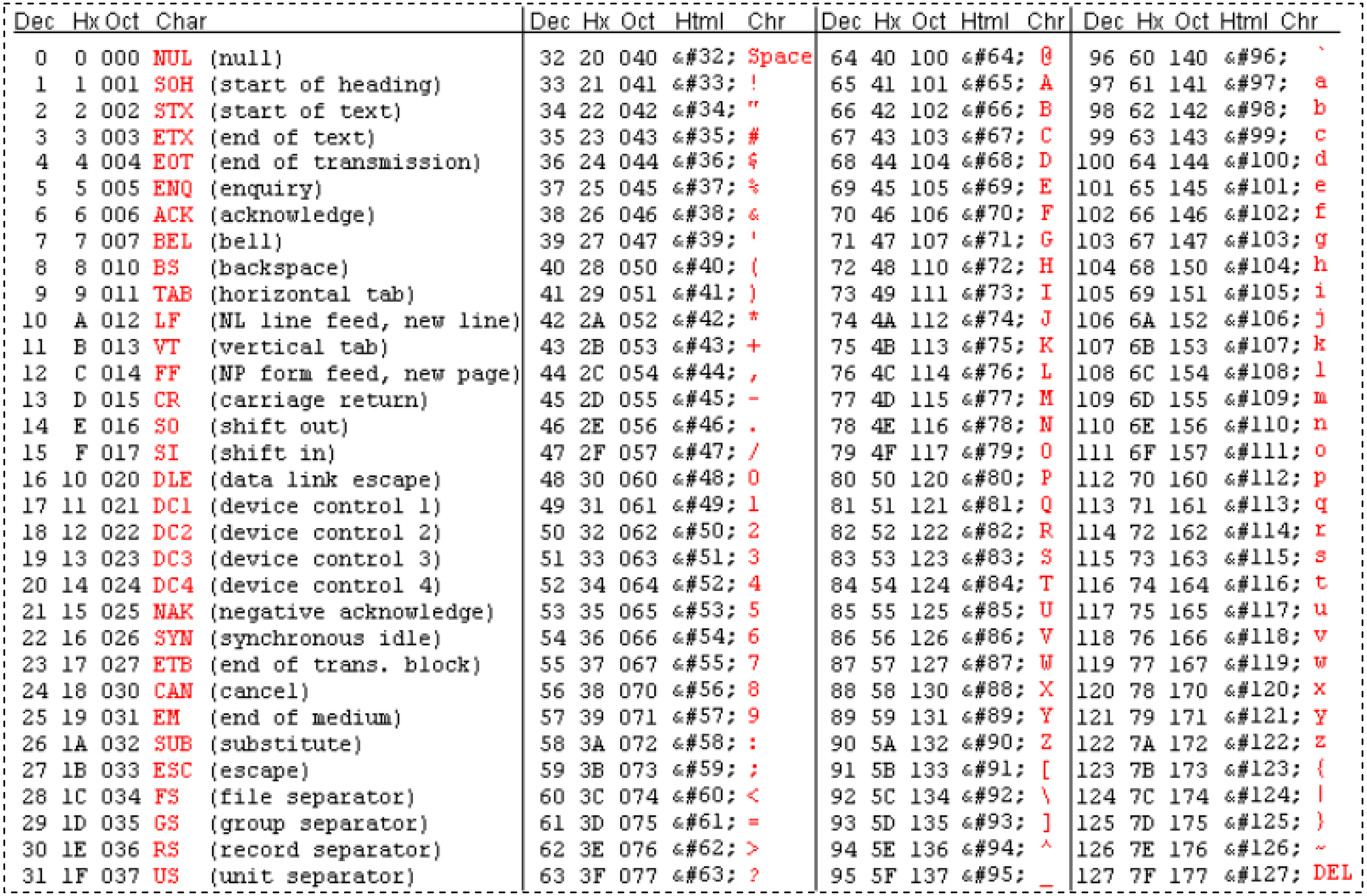
Mnemonic	Format	Opcode	Effect	Notes
SSK m	3/4	EC	Protection key for address m ← (A) (see Section 6.2.4)	P X
STA m	3/4	0C	m..m+2 ← (A)	
STB m	3/4	78	m..m+2 ← (B)	X
STCH m	3/4	54	m ← (A) [rightmost byte]	
STF m	3/4	80	m..m+5 ← (F)	X F
STI m	3/4	D4	Interval timer value ← (m..m+2) (see Section 6.2.1)	P X
STL m	3/4	14	m..m+2 ← (L)	
STS m	3/4	7C	m..m+2 ← (S)	X
STSW m	3/4	E8	m..m+2 ← (SW)	P
STT m	3/4	84	m..m+2 ← (T)	X
STX m	3/4	10	m..m+2 ← (X)	
SUB m	3/4	1C	A ← (A) – (m..m+2)	
SUBF m	3/4	5C	F ← (F) – (m..m+5)	X F



Mnemonic	Format	Opcode	Effect	Notes		
SUBR r1,r2	2	94	r2 ← (r2) − (r1)		X	
SVC n	2	B0	Generate SVC interrupt. {In assembled instruction, r1 = n}		X	
TD m	3/4	E0	Test device specified by (m)	P		C
TIO	1	F8	Test I/O channel number (A)	P	X	C
TIX m	3/4	2C	X ← (X) + 1; (X): (m..m+2)			C
TIXR r1	2	B8	X ← (X) + 1; (X): (r1)		X	C
WD m	3/4	DC	Device specified by (m) ← (A) [rightmost byte]	P		

5/5

ASCII Character Codes



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL