System Programming

(C. Sc. 565)

Text Book – System Software: An Introduction to System Programming by Leyland L. Beck, Pearson

Anand Kumar Sah

- **Course Description**

  This course will introduce the design and implementation of machine dependent, as well as machine independent aspects of assembler, loader, linker, microprocessor and some aspects of compiler. A project involving implementation of an assembler, a linker, a loader, and a compiler will form an integral part of the course.

- **Course Objectives**

  The purpose of this course is to present the basic structure and design of a micro-assembler, a linker, a loader, and a compiler. Since software components are best learned by implementation, each student will complete a project independently which will involve the design and implementation of these three software components.

- **Prerequisites**

  Logic Design, Data Structure and algorithm, Programming language, Familiarity with assembly language programming

Unit 1: SIC and SIC/XE Machine Structure [4 hrs]
1.1: Introduction

- *System software consists of a variety of programs that support the operation* of a computer. This software makes it possible for the user to focus on an application or other problem to be solved, without needing to know the details of how the machine works internally.
- You probably wrote programs in a high-level language like C++ or Pascal, using a *text editor to create and modify* the program. You translated these programs into machine language using a *compiler. The resulting machine language program was loaded into memory* and prepared for execution by a *loader or linker. You may have used a debugger* to help detect errors in the program.

- You may have used macro instructions in these programs to read and write data, or to perform other higher-level functions. You used an *assembler, which probably* included a *macro processor, to translate these programs into machine language.* The translated programs were prepared for execution by the loader or linker, and may have been tested using the debugger.

- As you read this subject, you will learn about several important types of system software. You will come to understand the processes that were going on 'behind the scenes" as you used the computer in previous courses. By understanding the system software, you will gain a deeper understanding of how computers actually work.

1.2: System software and machine architecture

- One characteristic in which most system software differs from application software is machine dependency.
- An application program is primarily concerned with the solution of some problem, using the computer as a tool. The focus is on the application, not on the computing system.
- System programs, on the other hand, are intended to support the operation and use of the computer itself, rather than any particular application.
- For this reason, they are usually related to the architecture of the machine on which they are to run. For example, assemblers translate mnemonic instructions into machine code; the instruction formats, addressing modes, etc., are of direct concern in assembler design.
- Similarly, compilers must generate machine language code, taking into account such hardware characteristics as the number and type of registers and the machine instructions available.
- 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 type of computing system being supported. For example, the general design and logic of an assembler is basically the same on most computers.
- Some of the code optimization techniques used by compilers are independent of the target machine (although there are also machine dependent optimizations).
- Likewise, the process of linking together independently assembled subprograms does not usually depend on the computer being used.
- Because most system software is machine-dependent, we must include real machines and real pieces of software in our study.
- However, most real computers have certain characteristics that are unusual or even unique. It can be difficult to distinguish between those features of the software that are truly fundamental and those that depend solely on the idiosyncrasies of a particular machine.

- To avoid this problem, we present the fundamental functions of each piece of software through discussion of a Simplified Instructional Computer (SIC).
- SIC is a hypothetical computer that has been carefully designed to include the hardware features most often found on real machines, while avoiding unusual or irrelevant complexities.
- In this way, the central concepts of a piece of system software can be clearly separated from the implementation details associated with a particular machine.
- This approach provides the reader with a starting point from which to begin the design of system software for a new or unfamiliar computer.

1.3: Simplified Instructional computers SIC, SIC/XE architecture

- In this section we describe the architecture of our Simplified Instructional Computer (SIC).
- This machine has been designed to illustrate the most commonly encountered hardware features and concepts, while avoiding most of the idiosyncrasies that are often found in real machines.
- Like many other products, SIC comes in two versions: the standard model and an XE version (XE stands for "extra equipment," or perhaps "extra expensive").
- The two versions have been designed to be *upward compatible-that is,* an object program for the standard SIC machine will also execute properly on a SIC/XE system. (Such upward compatibility is often found on real computers that are closely related to one another.)

SIC Machine Architecture

Memory

- Memory consists of 8-bit bytes; any 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 a total of 32,768 (2^{15}) bytes in the computer memory.

Registers

- There are five registers, all of which have special uses. Each register is 24 bits in length. The following table indicates the numbers, mnemonics, and uses of these registers. (The numbering scheme has been chosen for compatibility with the XE version of SIC.)

Mnemonic	Number	Special use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for 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)

Data Formats

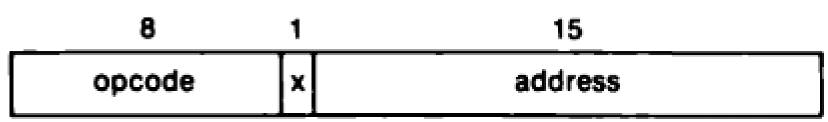
Integers are stored as 24-bit binary numbers; 2's complement representation is used for negative values.

Characters are stored using their 8-bit ASCII codes

There is no floating-point hardware on the standard version of SIC.

Instruction Formats

All machine instructions on the standard version of SIC have the following 24-bit format:



The flag bit x is used to indicate indexed-addressing mode.

Addressing Modes

There are two addressing modes available, indicated by the setting of the x bit in the instruction. The following table describes how the *target address* is calculated from the address given in the instruction. Parentheses are used to indicate the contents of a register or a memory location. For example, (X) represents the contents of register X.

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

Instruction Set

- SIC provides a basic set of instructions that are sufficient for most simple tasks.
- These include instructions that load and store registers (LDA, LDX, STA, STX, etc.), as well as integer arithmetic operations (ADD, SUB, MUL, DIV).
- All arithmetic operations involve register A and a word in memory, with the result being left in the register.
- There is an instruction (COMP) that compares the value in register A with a word in memory; this instruction sets a *condition code* CC to indicate the result (<, =, or >).
- Conditional jump instructions (JLT, JEQ, JGT) can test the setting of CC, and jump accordingly.
- Two instructions are provided for 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.
- **Appendix A gives a complete list of all SIC (and SIC/XE) instructions, with their operation codes and a specification of the function performed by each.**

Input and Output

- On the standard version of SIC, 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.
- The Test Device (TD) instruction tests whether the addressed device is ready to send or receive a byte of data.
- The condition code is set to indicate the result of this test. (A setting of < means the device is ready to send or receive, and = means the device is not ready.)
- A program needing to transfer data must wait until the device is ready, then execute a Read Data (RD) or Write Data (WD).
- This sequence must be repeated for each byte of data to be read or written.
- **The program will be shown in Fig. 2.1 (Chapter 2) illustrates this technique for performing I/O.**

SIC/XE Machine Architecture

Memory

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

Registers

- The following additional registers are provided by SIC/XE:

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

Data Formats

SIC/XE provides the same data formats as the standard version. In addition, there is a 48-bit floating-point data type with the following format:

1	11	36
s	exponent	fraction

The fraction is interpreted as a value between 0 and 1; that is, the assumed binary point is immediately before the high-order bit. For normalized floating-point numbers, the high-order bit of the fraction must be 1. The exponent is interpreted as an unsigned binary number between 0 and 2047. If the exponent has value *e* and the fraction has value *f*, the absolute value of the number represented is

$$f * 2^{(e-1024)}.$$

The sign of the floating-point number is indicated by the value of *s* (0 = positive, 1 = negative). A value of zero is represented by setting all bits (including sign, exponent, and fraction) to 0.

Instruction Formats

- The larger memory available on SIC/XE means that an address will (in general) no longer fit into a 15-bit field; thus the instruction format used on the standard version of SIC is no longer suitable.
- There are two possible options either use some form of relative addressing, or extend the address field to 20 bits.
- Both of these options are included in SIC/XE (Formats 3 and 4 in the following description).
- In addition, SIC/XE provides some instructions that do not reference memory at all. Formats 1 and 2 in the following description are used for such instructions.
- The new set of instruction formats is as follows. The settings of the flag bits in Formats 3 and 4 are discussed under Addressing Modes.
- Bit *e is used to distinguish* between Formats 3 and 4 *(e = 0 means Format 3, e = 1 means Format* 4).
- **Appendix A indicates the format to be used with each machine instruction.**

Format 1 (1 byte):

8
op

Format 2 (2 bytes):

8	4	4
op	r1	r2

Format 3 (3 bytes):

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

Format 4 (4 bytes):

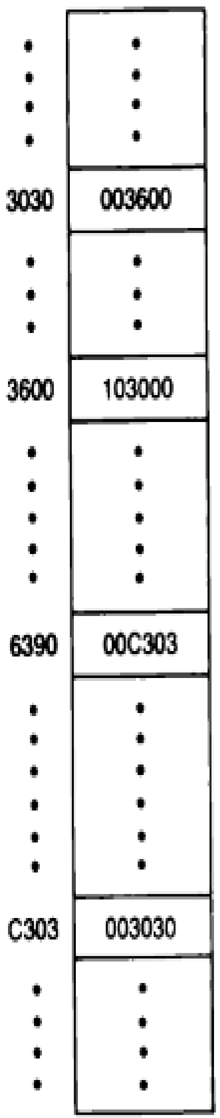
6	1	1	1	1	1	1	20
op	n	i	x	b	p	e	address

Addressing Modes

Mode	Indication	Target address calculation	
Base relative	b = 1, p = 0	TA = (B) + disp	($0 \le$ disp ≤ 4095)
Program-counter relative	b = 0, p = 1	TA = (PC) + disp	($-2048 \le$ disp ≤ 2047)

For *base relative addressing, the displacement field disp in a Format 3 instruction* is interpreted as a I2-bit unsigned integer.
For *program-counter relative addressing,* this field is interpreted as a I2-bit signed integer, with negative values represented in 2's complement notation.
If bits b and *p are both set to 0, the disp field from the Format 3 instruction* is taken to be the target address. For a Format 4 instruction, bits band pare normally set to 0, and the target address is taken from the address field of the instruction.
We will call this *direct addressing, to distinguish it from the relative* addressing modes described above.
Any of these addressing modes can also be combined with *indexed addressing-if* bit *x is set to 1, the term (X) is added in the target address calculation.*

- Bits *i and n in Formats 3 and 4 are used to specify how the target address is* used.
- If bit *i = 1 and n = 0, the target address itself is used as the operand* value; no memory reference is performed. This is called *immediate addressing.*
- If bit *i = 0 and n = 1, the word at the location given by the target address is* fetched; the *uaiue contained in this word is then taken as the address of the* operand value. This is called *ill direct addressing.*
- *If bits i and n are both 0 or* both 1, the target address is taken as the location of the operand; we will refer to this as *simple addressing.*
- *Indexing cannot be used with immediate or indirect* addressing modes.
- Many authors use the term eff*ective address to denote what we have called* the target address for an instruction.
- However, there is disagreement concerning the meaning of effective address when referring to an instruction that uses indirect addressing.
- To avoid confusion, we use the term *target address* throughout this book.



(a)

(B) = 006000

(PC) = 003000

(X) = 000090

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

(b)

Figure 1.1 Examples of SIC/XE instructions and addressing modes.

Instruction Set

- SIC /XE provides all of the instructions that are available on the standard version.
- In addition, there are instructions to load and store the new registers (LDB, STB, etc.) and to perform floating-point arithmetic operations (ADDF, SUBF, MULF, DIVF).
- There are also instructions that take their operands from registers.
- Besides the RMO (register move) instruction, these include register-to-register arithmetic operations (ADDR, SUBR, MULR, DIVR).
- A special *supervisor call instruction (SVC) is provided. Executing this instruction* generates an interrupt that can be used for communication with the operating system.
- There are also several other new instructions.
- **Appendix A gives a complete list of all SIC/XE instructions, with their operation codes and a specification of the function performed by each.**

Input and Output

- The I/O instructions we discussed for SIC are also available on SIC/XE.
- In addition, there are 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, resulting in more efficient system operation.
- The instructions S10, TIO, and HIO are used to start, test, and halt the operation of I/O channels.

SIC Programming Examples (1.2)

```
        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)

```
        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
```

(b)

Figure 1.2 Sample data movement operations for (a) SIC and (b) SIC/XE.

Simple Arithmetic Operations for SIC and SIC/XE (1.3)

```
         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)

(1.3 Continued)

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

(b)

Figure 1.3 Sample arithmetic operations for (a) SIC and (b) SIC/XE.

Simple Loading and Indexing Operations for SIC and SIC/XE (1.4)

```
        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
```

(a)

(1.4 Continued)

```
         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
```

(b)

Figure 1.4 Sample looping and indexing operations for (a) SIC and (b) SIC/XE.

Simple Indexing and Looping Operations for SIC and SIC/XE (1.5)

```
          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)

(1.5 Continued)

```
         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
```

(b)

Figure 1.5 Sample indexing and looping operations for (a) SIC and (b) SIC/XE.

Sample Input and Output Operations for SIC (1.6)

```
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
```

Figure 1.6 Sample input and output operations for SIC.

Sample Subroutine Call and Record Input Operations for SIC and SIC/XE (1.7)

```
            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
.                                  ONE-WORD CONSTANTS
ZERO        WORD     0
K100        WORD     100
```

(a)

(1.7 Continued)

```
         JSUB    READ              CALL READ SUBROUTINE
         .
         .
         .
.                                  SUBROUTINE TO READ 100-BYTE RECORD
READ     LDX     #0                INITIALIZE INDEX REGISTER TO 0
         LDT     #100              INITIALIZE REGISTER T TO 100
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
         TIXR    T                 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
```

(b)

Figure 1.7 Sample subroutine call and record input operations for (a) SIC and (b) SIC/XE.

1.4: RISC and CISC machine architecture

CISC Machine: Introduce briefly!!!

VAX (Virtual Address Extension) Architecture

- The VAX family of computers was introduced by Digital Equipment Corporation (DEC) in 1978.
- The VAX architecture was designed for compatibility with the earlier PDP-II (Programmable Data Procesor)machines.
- A compatibility mode was provided at the hardware level so that many PDP-II programs could run unchanged on the VAX.
- It was even possible for PDP-ll programs and VAX programs to share the same machine in a multi-user environment.
- This section summarizes some of the main characteristics of the VAX architecture.

Main characteristics of the VAX architecture

Memory

- The VAX memory consists of 8-bit bytes. All addresses used are byte addresses.
- Two consecutive bytes form a *word; four bytes form a longwoord; eight* bytes form a *quadword; sixteen bytes form an octawoord.*
- *Some operations are* more efficient when operands are aligned in a particular way- for example, a longword operand that begins at a byte address that is a multiple of 4.
- All VAX programs operate in a *virtual address space of* 2^{32} *bytes. This virtual* memory allows programs to operate as though they had access to an extremely large memory, regardless of the amount of memory actually present on the system.
- Routines in the operating system take care of the details of memory management.
- One half of the VAX virtual address space is called *system space, which contains the operating system, and is shared* by all programs.
- The other half of the address space is called *process space, and* is defined separately for each program.
- A part of the process space contains stacks that are available to the program.
- Special registers and machine instructions aid in the use of these stacks.

Registers

- There are 16 general-purpose registers on the VAX, denoted by RO through R15. Some of these registers, however, have special names and uses.
- All general registers are 32 bits in length. Register R15 is the *program counter, also* called Pc.
- R14 is the *stack pointer SP, which 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.
- Registers R6 through R11 have no special functions, and are available for general use by the program.
- Registers RO through R5 are likewise available for general use; however, these registers are also used by some machine instructions.
- There is a *processor status longtoord* (PSL), which contains state variables and flags associated with a process. The PSL includes, among many other items of information, a condition code and a flag that specifies whether PDP-11 compatibility mode is being used by a process.
- There are also a number of control registers that are used to support various operating system functions.

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.
- There are four different floating-point data formats on the VAX, ranging in length from 4 to 16 bytes. Two of these are compatible with those found on the PDP-II, and are standard on all VAX processors. The other two are available as options, and provide for an extended range of values by allowing more bits in the exponent field. In each case, the principles are the same as those we discussed for SIC/XE: a floating-point value is represented as a fraction that is to be multiplied by a specified power of 2.
- VAX processors provide a *packed decimal data format. In this format, each* byte represents two decimal digits, with each digit encoded using 4 bits of the byte. The sign is encoded in the last 4 bits.
- There is also a *numeric format that* is used to represent numeric values with one digit per byte. In this format, then sign may appear either in the last byte, or as a separate byte preceding the first digit. These two variations are called *trailing numeric and leading separate numeric.*
- VAX also supports queues and variable-length bit strings.
- There are single machine instructions that insert and remove entries in queues, and perform a variety of operations on bit strings. The existence of such powerful machine instructions and complex primitive data types is one of the more unusual features of the VAX architecture.

Instruction Formats

- VAX machine instructions use a variable-length instruction format.
- Each instruction consists of an operation code (1 or 2 bytes) followed by 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 necessary to locate the operand.

Addressing Modes

- VAX provides a large number of addressing modes.
- With few exceptions, any of these addressing modes may be used with any instruction.
- The operand itself may be in a register *(register mode), or its address may be specified by a* register *(register deferred mode).*
- *If the operand address is in a register, the register* contents may be automatically incremented or decremented by the operand length *in autoincrement and autodecrement modes).*
- *There are several* base relative addressing modes, with displacement fields of different lengths; when used with register PC, these become program-counter relative modes.
- All of these addressing modes may also include an index register, and many of them are available in a form that specifies indirect addressing (called *deferred* modes on VAX).
- In addition, there are immediate operands and several special-purpose addressing modes.

Instruction Set

- One of the goals of the VAX designers was to produce an instruction set that is symmetric with respect to data type.
- Many instruction mnemonics are formed by combining the following elements:
  - a prefix that specifies the type of operation,
  - a suffix that specifies the data type of the operands,
  - a modifier (on some instructions) that gives the number of operands involved.

Input and Output

- Input and output on the VAX are accomplished by I/O device controllers.
- Each controller has a set of control/status and data registers, which are as- signed locations in the physical address space.
- The portion of the address space into which the device controller registers are mapped is 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 they were physical memory locations.
- Likewise, software routines may read these registers to obtain status information.
- The association of an address in I/O space with a physical register in a device controller is handled by the memory management routines.

Home Assignment 1

- **Pentium Pro Architecture**

RISC MACHINES

- In general, a RISC system is characterized by a standard, fixed instruction length (usually equal to one machine word), 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 relatively large number of general-purpose registers.
- The number of machine instructions, instruction formats, and addressing modes is relatively small.

UltraSPARC Architecture

- The UltraSPARC processor, announced by Sun Microsystems in 1995, is the latest member of the SPARC family.
- Other members of this family include a variety of SPARC and SuperSPARC processors.
- The original SPARC architecture was developed in the mid-1980s, and has been implemented by a number of manufacturers.
- The name SPARC stands for scalable processor architecture.
- This architecture is intended to be suitable for a wide range of implementations, from microcomputers to supercomputers.

Memory

- Memory consists of 8-bit bytes; all addresses used are byte addresses.
- Two consecutive bytes form a *half word; four bytes form a word; eight bytes form a doubletoord. Halfwords are stored in memory beginning at byte addresses that* are multiples of 2. Similarly, words begin at addresses that are multiples of 4, and doublewords at addresses that are multiples of 8.
- UltraSPARC programs can be written using a virtual address space of 2^{64} bytes. This address space is divided into *pages; multiple page sizes are supported.*
- Some of the pages used by a program may be in physical memory, while others may be stored on disk.
- When an instruction is executed, the hardware and the operating system make sure that the needed page is loaded into physical memory. The virtual address specified by the instruction is automatically translated into a physical address by the UltraSPARC Memory Management Unit (MMU).

Registers

- The SPARC architecture includes a large *register file that usually contains more* than 100 general-purpose registers. (The exact number varies from one implementation to another.)
- However, any procedure can access only 32 registers, designated rO through r31.
- The first eight of these registers (rO through r7) are global-that is, they can be accessed by all procedures on the system.
- The other 24 registers available to a procedure can be visualized as a *window through which part of the register file can be seen. These windows overlap,* so some registers in the register file are shared between procedures.
- The SPARC hardware manages the windows into the register file.
- If a set of concurrently running procedures needs more windows than are physically available, a "window overflow" interrupt occurs.
- The operating system must then save the contents of some registers in the file (and restore them later) to provide the additional windows that are needed.

Data Formats

- The UltraSPARC architecture provides for the storage of integers, floating point values, and characters.
- Integers are stored as 8-, 16-, 32-, or 64-bit binary numbers. Both signed and unsigned integers are supported; 2's complement is used for negative values.
- In the original SPARC architecture, the most significant part of a numeric value is stored at the lowest-numbered address. (This is commonly called *big-endian byte ordering, because the "big end" of the value* comes first in memory.)
- UltraSPARC supports both big-endian and little-endian byte orderings.
- There are three different floating-point data formats. The single-precision format is 32 bits long. It stores 23 significant bits of the floating-point value, and allows for an 8-bit exponent (power of 2). (The remaining bit is used to store the sign of the floating-point value.)
- The double-precision format is 64 bits long. It stores 52 significant bits, and allows for a 11-bit exponent.
- The quad-precision format stores 63 significant bits, and allows for a 15-bit exponent.
- Characters are stored one per byte, using their 8-bit ASCII codes.

Instruction Formats

- There are three basic instruction formats in the SPARC architecture.
- All of these formats are 32 bits long; the first 2 bits of the instruction word identify which format is being used.
- Format 1 is used for the Call instruction. Format 2 is used for branch instructions (and one special instruction that enters a value into a register). The remaining instructions use Format 3, which provides for register loads and stores, and three-operand arithmetic operations.
- The fixed instruction length in the SPARC architecture is typical of RISC systems, and is intended to speed the process of instruction fetching and decoding.

Addressing Modes

As in most architectures, an operand value may be specified as part of the instruction itself *(immediate mode), or it may be in a register (register direct* mode).

Operands in memory are addressed using one of the following three modes:

Mode	Target address 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)

Instruction Set

- The basic SPARC architecture has fewer than 100 machine instructions, reflecting its RISC philosophy.
- The only instructions that access memory are loads and stores.
- All other instructions are register-to-register operations. Instruction execution on a SPARC system is *pipelined-while one instruction* is being executed, the next one is being fetched from memory and decoded.
- In most cases, this technique speeds instruction execution. However, an ordinary branch instruction might cause the process to "stall."
- The instruction following the branch (which had already been fetched and decoded) would have to be discarded without being executed.
- To make the pipeline work more efficiently, SPARC branch instructions (including subroutine calls) are *delayed branches. This means that the instruction* immediately following the branch instruction is actually executed *before the* branch is taken.

Input and Output

- In the SPARC architecture, 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 address, or 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.
- Thus input and output can be performed with the regular instruction set of the computer, and no special I/O instructions are needed.

Home Assignment 2

- **PowerPC Architecture**