breakpoint can be cleared by a breakpoint clear command, *c*, which can be preceded by labels and numbers, like the command *b*. There is a run command, *r*, in which the tracer executes until either a breakpoint, an exit call, or the end of the commands is encountered.

The tracer also keeps track of the subroutine level at which the program is running. This is shown in the window below the processor window and can also be seen through the indication numbers in the stack window. There are three commands that are based on these levels. The – command causes the tracer to run until the subroutine level is one less than the current level. What this command does is execute instructions until the current subroutine is finished. The converse is the + command, which runs the tracer until the next subroutine level is encountered. The = command runs until the same level is encountered, and can be used to execute a subroutine at the *CALL* command. If = is used, the details of the subroutine are not shown in the tracer window. There is a related command, *n*, which runs until the next line in the program is encountered. This command is especially useful when issued as a *LOOP* command; execution stops exactly when the bottom of the loop is executed.

C.7 GETTING STARTED

In this section, we will explain how to use the tools. First of all, it is necessary to locate the software for your platform. We have precompiled versions for Solaris, UNIX, for Linux and for Windows. The tools are located on the CD-ROM and on the Web at *www.prenhall.com/tanenbaum*. Once there, click on the *Companion Web Site* for this book and then click on the link in the left-hand menu. Unpack the selected zip file to a directory *assembler*. This directory and its subdirectories contain all the necessary material. On the CD-ROM, the main directories are *Bigendnx*, *LtlendNx*, and *MSWindos*, and in each there is a subdirectory *assembler* which contains the material. The three top-level directories are for Big-Endian UNIX (e.g. Sun workstations), Little-Endian UNIX (e.g., Linux on PCs), and Windows systems, respectively.

After unpacking or copying, the assembler directory should contain the following subdirectories and files: *READ_ME*, *bin*, *as_src*, *trce_src*, *examples*, and *exercise*. The precompiled sources can be found in the *bin* directory but, for convenience, there is also a copy of the binaries in the *examples* directory.

To get a quick preview of how the system works, go to the *examples* directory and type the command

```
t88 HlloWrld
```

This command corresponds to the first example in Sec. C.8.

The source code for the assembler is in the directory *as_src*. The source code files are in the language C, and the command *make* should recompile the sources.

For POSIX-compliant platforms, there is a *Makefile* in the source directory which does the job. For Windows, there is a batch file *make.bat*. It may be necessary to move the executable files after compilation to a program directory, or to change the PATH variable to make the assembler *as88* and the tracer *t88* visible from the directories containing the assembly source codes. Alternatively, instead of typing *t88*, the full path name can be used.

On Windows 2000 and XP systems, it is necessary to install the *ansi.sys* terminal driver by adding the line

```
device=%systemRoot%\System32\ansi.sys
```

to the configuration file, *config.nt*. The location of this file is as follows:

```
Windows 2000:  \winnt\system32\config.nt
Windows XP:    \windows\system32\config.nt
```

On UNIX and Linux systems, the driver is usually standard.

C.8 EXAMPLES

In Sec. C.2 through Sec. C.4, we discussed the 8088 processor, its memory, and its instructions. Then, in Sec. C.5, we studied the *as88* assembly language used in this tutorial. In Sec. C.6 we studied the tracer. Finally, in Sec. C.7, we described how to set up the toolkit. In theory, this information is sufficient to write and debug assembly programs with the tools provided. Nevertheless, it may be helpful for many readers to see some detailed examples of assembly programs and how they can be debugged with the tracer. That is the purpose of this section. All the example programs discussed in this section are available in the *examples* directory in the toolkit. The reader is encouraged to assemble and trace each one as it is discussed.

C.8.1 Hello World Example

Let us start with the example of Fig. C-12, *HlloWrld.s*. The program is listed in the left window. Since the assembler's comment symbol is the exclamation mark (!), it is used in the program window to separate the instructions from the line numbers that follow. The first three lines contain constant definitions, which connect the conventional names of two system calls and the output file to their corresponding internal representations.

The pseudoinstruction .SECT, on line 4, states that the following lines should be considered to be part of the TEXT section; that is, processor instructions. Similarly, line 17 indicates that what follows is to be considered data. Line 19 initializes a string of data consisting of 12 bytes, including one space and a line feed (\n) at the end.

Lines 5, 18 and 20 contain labels, which are indicated by a colon :. These labels represent numerical values, similar to constants. In this case, however, the assembler has to determine the numerical values. Since *start* is at the beginning of the TEXT section, its value will be 0, but the value of any subsequent labels in the TEXT section (not present in this example), would depend on how many bytes of code preceded them. Now consider line 6. This line ends with the difference of two labels, which is numerically a constant. Thus, line 6 is effectively the same as

```
MOV CX,12
```

except that it lets the assembler determine the string length, rather than making the programmer do it. The value indicated here is the amount of space in the data reserved for the string on line 19. The MOV on line 6 is the copy command, which requires the *de* − *hw* to be copied to CX.

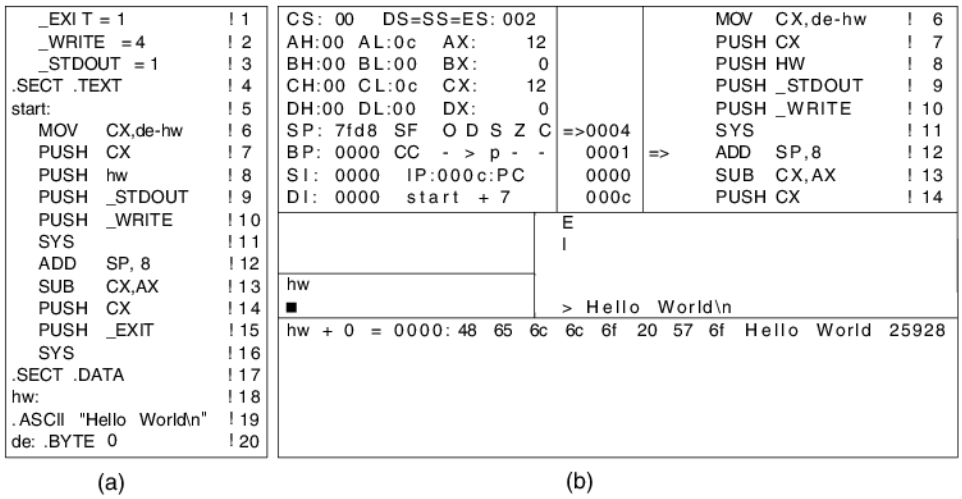


Figure C-12. (a) HlloWrld.s. (b) The corresponding tracer window.

Lines 7 through 11 show how system calls are made in the toolkit. These five lines are the assembly code translation of the C language function call

```
write(1, hw, 12);
```

where the first parameter is the file descriptor for standard output (1), the second is the address of the string to be printed (*hw*), and the third is the length of the string (12). Lines 7 through 9 push these parameters onto the stack in reverse order, which is the C calling sequence and the one used by the tracer. Line 10 pushes the system call number for write (4) onto the stack, and line 11 makes the actual call. While this calling sequence closely mimics how an actual assembly language program would work on a UNIX (or Linux) PC, for a different operating system, it would have to be modified slightly to use the calling conventions of that operating

system. The *as88* assembler and *t88* tracer use the UNIX calling conventions even when they are running on Windows, however.

The system call on line 11 does the actual printing. Line 12 performs a cleanup on the stack, resetting the stack pointer back to the value it had before the four 2-byte words were pushed onto the stack. If the write call is successful, the number of bytes written is returned in AX. Line 13 subtracts the system call result after line 11 from the original string length in CX to see whether the call was successful, that is, to see if all the bytes were written. Thus, the exit status of the program will be 0 on success and something else on failure. Lines 14 and 15 prepare for the exit system call on line 16 by pushing the exit status and function code for the EXIT call onto the stack.

Note that in the MOV and SUB instructions the first argument is the destination and the second is the source. This is the convention used by our assembler; other assemblers may reverse the order. There is no particular reason to choose one order over the other.

Now let us try to assemble and run *HlloWrld.s*. Instructions will be given for both UNIX and Windows platforms. For Linux, Solaris, MacOS X, and other UNIX variants, the procedure should be essentially the same as for UNIX. First, start up a command prompt (shell) window. On Windows, the click sequence is usually

```
Start > Programs > Accessories > Command prompt
```

Next, change to the *examples* directory using the *cd* (Change directory) command. The argument to this command depends on where the toolkit has been placed in the file system. Then verify that the assembler and tracer binaries are in this directory, using *ls* on UNIX and *dir* on Windows systems. They are called *as88* and *t88*, respectively. On Windows systems, they have the extension *.exe*, but that need not be typed in the commands. If the assembler and tracer are not there, find them and copy them there.

Now assemble the test program using

```
as88 HlloWrld.s
```

If the assembler is present in the *examples* directory but this command gives an error message, try typing

```
./as88 HlloWrld.s
```

on UNIX systems or

```
.\as88 HlloWrld.s
```

on Windows systems.

If the assembly process completes correctly, the following messages will be displayed:

```
Project HlloWrld listfile HlloWrld.$
```

```
Project HlloWrld num file HlloWrld.#
Project HlloWrld loadfile HlloWrld.88.
```

and the corresponding three files created. If there are no error messages, give the tracer command:

```
t88 HlloWrld
```

The tracer display will appear with the arrow in the upper right-hand window pointing to the

```
MOV CX,de-hw
```

instruction of line 6. Now hit the return (called Enter on PC keyboards) key. Notice that the instruction pointed to is now

```
PUSH CX
```

and the value of CX in the left-hand window is now 12. Hit return again and notice that the middle window on the top line now contains the value 000c, which is hexadecimal for 12. This window shows the stack, which now has one word containing 12. Now hit return three more times to see the PUSH instructions on lines 8, 9, and 10 being carried out. At this point, the stack will have four items and the program counter in the left-hand window will have the value 000b.

The next time return is hit, the system call is executed and the string "Hello World\n" is displayed in the lower right-hand window. Note that SP now has the value 0x7ff0. After the next return, SP is incremented by 8 and becomes 0x7ff8. After four more returns, the exit system call completes and the tracer exits.

To be certain that you understand how everything works, fetch the file *hlloWrld.s* into your favorite editor. It is better not to use a word processor. On UNIX systems, *ex*, *vi*, or *emacs* are good choices. On Windows systems, *notepad* is a simple editor, usually reachable from

```
Start > Programs > Accessories > Notepad
```

Do not use *Word* since the display will not look right and the output may be formatted incorrectly.

Modify the string on line 19 to display a different message, then save the file, assemble it, and run it with the tracer. You are now starting to do assembly language programming.

C.8.2 General Registers Example

The next example demonstrates in more detail how the registers are displayed and one of the pitfalls of multiplication on the 8088. In Fig. C-13, part of the program *genReg.s* is shown on the left. To its right are two tracer register windows, corresponding to different stages of the program's execution. Fig. C-13(b) shows the register state after line 7 has been executed. The instruction

```
MOV AX,258
```

on line 4 loads the value 258 in AX, which results in the value 1 being loaded into

AH and the value 2 being loaded into AL. Then line 5 adds AL to AH, making AH equal to 3. On line 6, the contents of the variable *times* (10) are copied into CX. On line 7, the address of the variable *muldat*, which is 2 because it is at the second byte of the DATA segment, is loaded into BX. This is the instant in time at which the dump of Fig. C-13(b) was made. Note that AH is 3, AL is 2, and AX is 770, which is to be expected, as $3 \times 256 + 2 = 770$.

```
start:                    ! 3
    MOV   AX,258          ! 4
    ADDB  AH,AL           ! 5
    MOV   CX,(times)      ! 6
    MOV   BX,muldat       ! 7
    MOV   AX,(BX)         ! 8
llp: MUL  2(BX)           ! 9
    LOOP  llp             ! 10
.SECT .DATA               ! 11
times: .WORD 10           ! 12
muldat :.WORD 625,2       ! 13
```

(a)

```
CS: 00   DS=SS=ES002
AH:03 AL:02  AX:   770
BH:00 BL:02  BX:     2
CH:00 CL:0a  CX:    10
DH:00 DL:00  DX:     0
SP: 7fe0 SF  O D S  Z C
BP: 0000 CC  - > p  - -
SI: 0000  IP:0009:PC
DI: 0000  start + 4
```

(b)

```
CS: 00   DS=SS=ES002
AH:38 AL:80  AX: 14464
BH:00 BL:02  BX:     2
CH:00 CL:04  CX:     4
DH:00 DL:01  DX:     1
SP: 7fe0 SF  O D S  Z C
BP: 0000 CC  v > p  - c
SI: 0000  IP:0011:PC
DI: 0000  start + 7
```

(c)

Figure C-13. (a) Part of a program. (b) The tracer register window after line 7 has been executed. (c) The registers.

The next instruction (line 8) copies the contents of *muldat* into AX. Thus, after the return key is hit, AX will be 625.

We are now ready to enter a loop that multiplies the contents of AX by the word addressed by 2BX (i.e., *muldat* + *2*), which has the value 2. The implied destination of the MUL instruction is the DX : AX long register combination. In the first iteration of the loop, the result fits in one word, so AX contains the result (1250), and DX remains 0. The contents of all the registers after 7 multiplications are shown in Fig. C-13.

Since AX started at 625, the result after those seven multiplications by 2 is 80,000. This result does not fit in AX, but the product is held in the 32-bit register formed by the concatenation of DX : AX, so DX is 1 and AX is 14,464. Numerically, this value is $1 \times 65{,}536 + 14{,}464$, which is, indeed, 80,000. Note that CX is 4 here, because the LOOP instruction decrements it every iteration. Because it started at 10, after seven executions of the MUL instruction (but only six iterations of the LOOP instruction) we have CX set to 4.

In the next multiplication, trouble crops up. Multiplication involves AX but not DX, so the MUL multiples AX (14464) by 2 to get 28,928. This results in AX being set to 28,928 and DX being set to 0, which is numerically incorrect.

C.8.3 Call Command and Pointer Registers

The next example, *vecprod.s* is a small program that computes the inner product of two vectors, *vec1* and *vec2*. It is listed in Fig. C-14.

The first part of the program prepares to call *vecmul* by saving SP in BP and then pushing the addresses of *vec2* and *vec1* onto the stack so that *vecmul* will have access to them. Then the length of the vector in bytes is loaded in CX on line 8. By shifting this result right one bit, on line 9, CX now contains the number of words in the vector, which is pushed onto the stack on line 10. The call to *vecmul* is made on line 11.

Once again, it is worth mentioning that the arguments of subroutines are, by convention, pushed onto the stack in reverse order to be compatible with the C calling convention. In this way, *vecmul* can also be called from C using

```
vecmul(count, vec1, vec2)
```

During the CALL instruction, the return address is pushed onto the stack. If the program is traced, then this address turns out to be 0x0011.

The first instruction in the subroutine is a PUSH of the base pointer, BP, on line 22. BP is saved because we will need this register to address the arguments and the local variables of the subroutine. Next, the stack pointer is copied to the BP register on line 23, so that the new value of the base pointer is pointing to the old value.

Now everything is ready for loading the arguments into registers and for reserving space for a local variable. In the next three lines, each of the arguments is fetched from the stack and put in a register. Recall that the stack is word oriented, so stack addresses should be even. The return address is next to the old base pointer so it is addressed by 2(BP). The *count* argument is next and addressed by 4(BP). It is loaded into CX on line 24. In lines 25 and 26, SI is loaded with *vec1* and DI is loaded with *vec2*. This subroutine needs one local variable with initial value 0 to save the intermediate result, so the value 0 is pushed on line 27.

The state of the processor just before the loop is entered on line 28 for the first time is shown in Fig. C-15. The narrow window in the middle of the top row (to the right of the registers) shows the stack. At the bottom of the stack is the address of *vec2* (0x0022), with *vec1* (0x0018) above it and the third argument, the number of items in each vector (0x0005) above that. Next comes the return address (0x0011). The number 1 to the left of this address indicates it is a return address one level from the main program. In the window below the registers, the same number 1 is shown, this time giving its symbolic address. Above the return address in the stack is the old value of BP (0x7fc0) and then the zero pushed on line 27. The arrow pointing to this value indicates where SP points. The window to the right of the stack shows a fragment of the program text, with the arrow indicating the next instruction to be executed.

Now let us examine the loop starting at line 28. The instruction LODS loads a memory word indirectly through the register SI from the data segment into AX. Because the direction flag is set, LODS is in auto-increment mode, so after the instruction SI will point to the next entry of *vec1*.

To see this effect graphically, start the tracer with the command

```
t88   vecprod
```

```
_EXIT = 1                              ! 1  define the value of _EXIT
_PRINTF = 127                          ! 2  define the value of _PRINTF
.SECT .TEXT                            ! 3  start the TEXT segment
inpstart:                              ! 4  define label inpstart
        MOV  BP,SP                     ! 5  save SP in BP
        PUSH vec2                      ! 6  push address of vec2
        PUSH vec1                      ! 7  push address of vec1
        MOV CX,vec2-vec1               ! 8  CX = number of bytes in vector
        SHR  CX,1                      ! 9  CX = number of words in vector
        PUSH CX                        ! 10  push word count
        CALL vecmul                    ! 11  call vecmul
        MOV  (inprod),AX               ! 12  move AX
        PUSH AX                        ! 13  push result to be printed
        PUSH pfmt                      ! 14  push address of format string
        PUSH _PRINTF                   ! 15  push function code for PRINTF
        SYS                            ! 16  call the PRINTF function
        ADD  SP,12                     ! 17  clean up the stack
        PUSH 0                         ! 18  push status code
        PUSH _EXIT                     ! 19  push function code for EXIT
        SYS                            ! 20  call the EXIT function

vecmul:                                ! 21  start of vecmul(count, vec1, vec2)
        PUSH BP                        ! 22  save BP on stack
        MOV  BP,SP                     ! 23  copy SP into BP to access arguments
        MOV  CX,4(BP)                  ! 24  put count in CX to control loop
        MOV  SI,6(BP)                  ! 25  SI = vec1
        MOV  DI,8(BP)                  ! 26  DI = vec2
        PUSH 0                         ! 27  push 0 onto stack

1:      LODS                           ! 28  move (SI) to AX
        MUL  (DI)                      ! 29  multiply AX by (DI)
        ADD  -2(BP),AX                 ! 30  add AX to accumulated value in memory
        ADD  DI,2                      ! 31  increment DI to point to next element
        LOOP 1b                        ! 32  if CX > 0, go back to label 1b
        POP  AX                        ! 33  pop top of stack to AX
        POP  BP                        ! 34  restore BP
        RET                            ! 35  return from subroutine

.SECT .DATA                            ! 36  start DATA segment
pfmt: .ASCIZ "Inner product is: %d\n"  ! 37  define string
.ALIGN 2                               ! 38  force address even
vec1:.WORD 3,4,7,11,3                  ! 39  vector 1
vec2:.WORD 2,6,3,1,0                   ! 40  vector 2
.SECT .BSS                             ! 41  start BSS segment
inprod: .SPACE 2                       ! 42  allocate space for inprod
```

Figure C-14. The program *vecprod.s*.

When the tracer window appears, type the command

```
/vecmul+7b
```

followed by a return to put a breakpoint at the line containing the LODS. From now on, we will not mention that all commands must be followed by the return key. Then give the command

```
g
```

to have the tracer execute commands until the breakpoint is encountered. It will stop at the line containing the LODS.

On line 29, the value of AX is multiplied to the source operand. The memory word for the MUL instruction is fetched from the data segment through the DI in register indirect mode. The implied destination of MUL is the DX : AX long register combination which is not mentioned in the instruction but which is implied by it.

On line 30, the result is added to the local variable at the stack address –2(BP). Because MUL does not autoincrement its operand, that must be done explicitly on line 31. Afterward, DI points to the next entry of *vec2*.

The LOOP instruction finishes this step. Register CX is decremented, and, if it is still positive, the program jumps back to the local label *1* on line 28. The use of the local label *1b* means the closest label *1* looking backward from the current location. After the loop, the subroutine pops the return value into AX (line 33), restores BP (line 34), and returns to the calling program (line 35).

Then the main program is resumed after the call with the MOV instruction on line 12. This instruction is the start of a five-instruction sequence whose goal is to print the result. The printf system call is modeled after the *printf* function in the standard C programming library. Three arguments are pushed onto the stack on lines 13-15. These arguments are the integer value to be printed, the address of the format string (*pfmt*), and the function code for printf (127). Note that the format string *pfmt* contains a *%d* to indicate that an integer variable can be found as argument to the printf call to complete the output.

Line 17 cleans up the stack. Since the program started on line 5 by saving the stack pointer in the base pointer, we could also use the instruction

```
MOV SP,BP
```

for a stack cleanup. The advantage of this solution is that the programmer does not need to keep the stack balanced in the process. For the main program this is not a big issue, but in subroutines this approach is an easy way to throw away garbage such as obsolete local variables.

The subroutine *vecmul* can be included in other programs. If the source file *vecprod.s* is put on the command line behind another assembler source file, the subroutine is available for multiplying two vectors of a fixed length. It is advisable to remove the constant definitions _EXIT and _PRINTF first, in order to avoid their being defined twice. If the header file *syscalnr.h* is included somewhere, then there is no need to define the system call constants anywhere else.

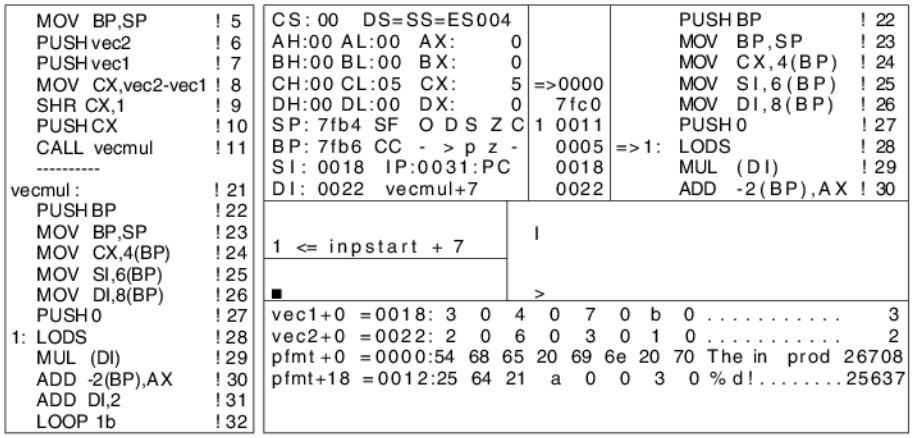


Figure C-15. Execution of *vecprod.s* when it reaches line 28 for the first time.

C.8.4 Debugging an Array Print Program

In the previous examples, the programs examined were simple but correct. Now we will show how the tracer can help debug incorrect programs. The next program is supposed to print the integer array, which is supplied after the label *vec1*. However, the initial version contains three errors. The assembler and tracer will be used to correct those errors, but first we will discuss the code.

Because every program needs system calls, and thus must define constants by which to identify the call numbers, we have put the constant definitions for those numbers in a separate header file *../syscalnr.h*, which is included on line 1 of the code. This file also defines the constants for the file descriptors

```
STDIN = 0
STDOUT = 1
STDERR = 2
```

which are opened at the start of the process, and header labels for the text and the data segments. It is sensible to include it at the head of all assembly source files, as these are much used definitions. If a source is distributed over more than one file, the assembler includes only the first copy of this header file, to avoid defining the constants more than once.

The program *arrayprt* is shown in Fig. C-16. Comments have been omitted here, as the instructions should be well known by now. This layout allows a two-column format. Line 4 puts the address of the empty stack in the base pointer register to allow the stack cleanup can be made on line 10 by copying the base pointer to the stack pointer, as described in the previous example. We also have seen the computation and pushing of the stack arguments before the call on lines 5 through 9 in the previous example. Lines 22 to 25 load the registers in the subroutine.

```
#include "../syscalnr.h"          ! 1

.SECT .TEXT                       ! 2
vecpstrt:                         ! 3
   MOV  BP,SP                     ! 4
   PUSH vec1                      ! 5
   MOV  CX,frmatstr-vec1          ! 6
   SHR  CX                        ! 7
   PUSH CX                        ! 8
   CALL vecprint                  ! 9
   MOV  SP,BP                     ! 10
   PUSH 0                         ! 11
   PUSH _EXIT                     ! 12
   SYS                            ! 13

.SECT .DATA                       ! 14
vec1:   .WORD 3,4,7,11,3          ! 15
frmatstr: .ASCIZ "%s"             ! 16

frmatkop:                         ! 17
.ASCIZ "The array contains "      ! 18
frmatint: .ASCIZ " %d"            ! 19

.SECT .TEXT                       ! 20
vecprint:                         ! 21
   PUSH BP                        ! 22
   MOV  BP,SP                     ! 23
   MOV  CX,4(BP)                  ! 24
   MOV  BX,6(BP)                  ! 25
   MOV  SI,0                      ! 26
   PUSH frmatkop                  ! 27
   PUSH frmatstr                  ! 28
   PUSH _PRINTF                   ! 29
   SYS                            ! 30
   MOV  -4(BP),frmatint           ! 31
1: MOV  DI,(BX)(SI)               ! 32
   MOV  -2(BP),DI                 ! 33
   SYS                            ! 34
   INC  SI                        ! 35
   LOOP 1b                        ! 36
   PUSH '\n'                      ! 37
   PUSH _PUTCHAR                  ! 38
   SYS                            ! 39
   MOV  SP,BP                     ! 40
   RET                            ! 41
```

Figure C-16. The program *arrayprt* before debugging.

Lines 27 to 30 show how a string can be printed, and 31 to 34 show the printf system call for an integer value. Note that the address of the string is pushed on line 27, while on line 33 the value of the integer is moved onto the stack. In both cases the address of the format string is the first argument of PRINTF. Lines 37 to 39 show how a single character can be printed using the putchar system call.

Now let us try assembling and running the program. When the command

```
as88 arrayprt.s
```

is typed, we get an operand error on line 28 of the file *arrayprt.$*. This file is generated by the assembler by combining the included files with the source file to get a composite file that is the actual assembler input. To see where line 28 really is, we have to examine line 28 of *arrayprt.$*. We cannot look at *arrayprt.s* to get the line number because the two files do not match on account of the header being included line by line in *arrayprt.$*. Line 28 in *arrayprt.$* corresponds to line 7 in *arrayprt.s* because the included header file, *syscalnr.h,* contains 21 lines.

One easy way to find line 28 of *arrayprt.$* on UNIX is to type the command

```
head –28 arrayprt.$
```

which displays the first 28 lines of the combined file. The line at the bottom of the listing is the one in error. In this way (or by using an editor and going to line 28)

we see that the error is on line 7, which contains the SHR instruction. Comparing this code with the instruction table in Fig. C-4 shows the problem: the shift count has been omitted. The corrected line 7 should read

```
SHR CX,1
```

It is very important to note that the error must be corrected in the original source file, *arrayprt.s*, and *not* in the combined source, *arrayprt.$*, as the latter is automatically regenerated every time the assembler is called.

The next attempt to assemble the source code file should succeed. Then the tracer can be started by the command:

```
t88 arrayprt
```

During the tracing process, we can see that the output is not consistent with the vector in the data segment. The vector contains: 3, 4, 7, 11, 3, but the values displayed start with: 3, 1024, ... Clearly, something is wrong.

To find the error, the tracer can be run again, step by step, examining the state of the machine just before the incorrect value is printed. The value to be printed is stored in memory on lines 32 and 33. Since the wrong value is being printed, this is a good place to see what is wrong. The second time through the loop, we see that SI is an odd number, when clearly it should be an even number, as it is indexing through words, not bytes. The problem is on line 35. It increments SI by 1; it should increment it by 2. To fix the bug, this line should be changed to

```
ADD SI,2
```

When this correction is made, the printed list of numbers is correct.

However, there is one more error waiting for us. When *vecprint* is finished and returns, the tracer complains about the stack pointer. The obvious thing to check for now is whether the value pushed onto the stack when *vecprint* is called is the value on top of the stack when the RET on line 41 is executed. It is not. The solution is to replace line 40 with two lines:

```
ADD SP,10
POP BP
```

The first instruction removes the 5 words pushed onto the stack during *vecprint*, thus exposing the value of BP saved on line 22. By popping this value to BP, we restore BP to its precall value and expose the correct return address. Now the program terminates correctly. Debugging assembly code is definitely more of an art than a science, but the tracer makes it much easier than running on the bare metal.

C.8.5 String Manipulation and String Instructions

The main purpose of this section is to show how to handle repeatable string instructions. In Fig. C-17, there are two simple string manipulation programs, *strngcpy.s* and *reverspr.s*, both present in the *examples* directory. The one in

```
.SECT .TEXT
stcstart:                          ! 1
   PUSH mesg1                      ! 2
   PUSH mesg2                      ! 3
   CALL strngcpy                   ! 4
   ADD  SP,4                       ! 5
   PUSH 0                          ! 6
   PUSH 1                          ! 7
   SYS                             ! 8
strngcpy:                          ! 9
   PUSH CX                         ! 10
   PUSH SI                         ! 11
   PUSH DI                         ! 12
   PUSH BP                         ! 13
   MOV  BP,SP                      ! 14
   MOV  AX,0                       ! 15
   MOV  DI,10(BP)                  ! 16
   MOV  CX,–1                      ! 17
   REPNZ SCASB                     ! 18
   NEG  CX                         ! 19
   DEC CX                          ! 20
   MOV  SI,10(BP)                  ! 21
   MOV  DI,12(BP)                  ! 22
   PUSH DI                         ! 23
   REP  MOVSB                      ! 24
   CALL stringpr                   ! 25
   MOV  SP,BP                      ! 26
   POP  BP                         ! 27
   POP  DI                         ! 28
   POP  SI                         ! 29
   POP  CX                         ! 30
   RET                             ! 31
.SECT .DATA                        ! 32
mesg1: .ASCIZ "Have a look\n"      ! 33
mesg2: .ASCIZ "qrst\n"             ! 34
.SECT .BSS
```

(a)

```
#include "../syscalnr.h"          ! 1

start: MOV DI,str                 ! 2
   PUSH AX                        ! 3
   MOV BP,SP                      ! 4
   PUSH _PUTCHAR                  ! 5
   MOVB AL,'\n'                   ! 6
   MOV CX,-1                      ! 7
   REPNZ SCASB                    ! 8
   NEG CX                         ! 9
   STD                            ! 10
   DEC CX                         ! 11
   SUB DI,2                       ! 12
MOV SI,DI                         ! 13
   1: LODSB                       ! 14
   MOV (BP),AX                    ! 15
   SYS                            ! 16
   LOOP  1b                       ! 17
   MOVB  (BP),'\n'                ! 18
   SYS                            ! 19
   PUSH 0                         ! 20
   PUSH _EXIT                     ! 21
SYS                               ! 22
.SECT .DATA                       ! 23
str: .ASCIZ "reverse\n"           ! 24
```

(b)

Figure C-17. (a) Copy a string (*strngcpy.s*). (b) Print a string backward (*reverspr.s*).

Fig. C-17(a) is a subroutine for copying a string. It calls a subroutine, *stringpr*, which can also be found in a separate file *stringpr.s*. It is not listed in this appendix. In order to assemble programs containing subroutines in separate source files, just list all source files in the *as88* command, starting with the source file for

the main program, which determines the names of the executable and the auxiliary files. For example, for the program of Fig. C-17(a) use

```
as88 strngcpy.s stringpr.s
```

The program of Fig. C-17(b) outputs strings in reverse order. We will now look at them in turn.

To demonstrate that the line numbers are really just comments, in Fig. C-17(a) we have numbered the lines starting with the first label, omitting what comes before them. The main program, on lines 2 through 8, first calls *strngcpy* with two arguments, the source string, *mesg2*, and the destination string, *mesg1*, in order to copy the source to the destination.

Now let us look at *strngcpy*, starting on line 9. It expects that the addresses of the destination buffer and the string source have been pushed onto the stack just before the subroutine is called. On lines 10 to 13, the registers used are saved by pushing them onto the stack so that they can be restored later on lines 27 to 30. On line 14, we copy SP to BP in the usual way. Now BP can be used to load the arguments. Again, on line 26, we clean the stack by copying BP to SP.

The heart of the subroutine is the instruction REP MOVSB, on line 24. The instruction MOVSB moves the byte pointed to by SI to the memory address pointed to by DI. Both SI and DI are then incremented by 1. The REP creates a loop in which this instruction is repeated, decrementing CX by 1 for each byte moved. The loop is terminated when CX reaches 0.

Before we can run the REP MOVSB loop, however, we have to set up the registers, which is done in lines 15 through 22. The source index, SI, is copied from the argument on the stack on line 21; the destination index, DI, is set up on line 22. Obtaining the value of CX is more involved. Note that the end of a string is indicated by a zero byte. The MOVSB instruction does not affect the zero flag, but the instruction SCASB (scan byte string) does. It compares the value pointed to by DI with the value in AL, and it increments DI on the fly. Moreover, it is repeatable like MOVSB. So, on line 15 AX and hence AL is cleared, on line 16 the pointer for DI is fetched from the stack, and CX is initialized to −1 on line 17. On line 18, we have the REPNZ SCASB, which does the comparison in loop context, and sets the zero flag on equality. At each step of the loop, CX is decremented, and the loop stops when the zero flag is set, because the REPNZ checks both the zero flag and CX. The number of steps for the MOVSB loop is now computed as the difference of the current value of CX and the previous −1 on lines 19 and 20.

It is cumbersome that there are two repeatable instructions necessary, but this is the price for the design choice that move instructions never affect the condition codes. During the loops, the index registers have to be incremented, and to this end it is necessary that the direction flag is cleared.

Lines 23 and 25 print the copied string by means of a subroutine, *stringpr*, which is in the *examples* directory. It is straightforward and will not be discussed here.

In the reverse print program shown in Fig. C-17(b), the first line includes the usual system call numbers. On line 3, a dummy value is pushed onto the stack, and on line 4, the base pointer, BP, is made to point to the current top of stack. The program is going to print ASCII characters one by one, thus the numerical value _PUTCHAR is pushed onto the stack. Note that BP points to the character to be printed when a SYS call is made.

Line 2, 6 and 7 prepare the registers DI, AL and CX for the repeatable SCASB instruction. The count register and the destination index are loaded in a similar way as in the string copy routine, but the value of AL is the new line character, instead of the value 0. In this way, the SCASB instruction will compare the character values of the string *str* to \n instead of to 0, and set the zero flag whenever it is found.

The REP SCASB increments the DI register, so, after a hit, the destination index points at the zero character following the new line. On line 12, DI is decremented by two to have it point to the last letter of the word.

If the string is scanned in reverse order and printed character by character, we have obtained our goal, so on line 10 the direction flag is set to reverse the adjustment of the index registers in the string instructions. Now the LODSB on line 14 copies the character in AL, and on line 15 this character is put just next to the _PUTCHAR on the stack, so the SYS instruction prints it.

The instructions on lines 18 and 19 print an additional new line and the program closes with an _EXIT call in the usual way.

The current version of the program contains a bug. It can be found if the program is traced step by step.

The command */str* will put the string *str* in the tracer data field. Since the numerical value of the data address is also given, we can find out how the index registers run through the data with respect to the position of the string.

The bug, however, is encountered only after hitting the return many times. By using the tracer commands we can get to the problem faster. Start the tracer and give the command *13* to put us in the middle of the loop. If we now give the command *b* we set a breakpoint on this line 15. If we give two new lines, then we see that the final letter **e** is printed in the output field. The *r* command will keep the tracer running until either a breakpoint or the end of the process is encountered. In this way, we can run through the letters by giving the *r* command repeatedly until we are close to the problem. From this point, we can run the tracer at one step at a time until we see what happens at the critical instructions.

We can also put the breakpoint at a specific line, but then we must keep in mind, that the file *../syscalnr* is included, which causes the line numbers to be offset by 20. Consequently, the breakpoint on line 16 can be set by the command *36b*. This is not an elegant solution, so it is much better to use the global label *start* on line 2 before the instruction and give the command */start+14b*, which puts the breakpoint in the same place without having to keep track of the size of the included file.

C.8.6 Dispatch Tables

In several programming languages, there exist *case* or *switch* statements to select a jump from several alternatives according to some numerical value of a variable. Sometimes, such multiway branches are also needed in assembly language programs, too. Think, for instance, of a set of system call subroutines combined in a single SYS trap routine. The program *jumptbl.s*, shown in Fig. C-18 shows how such a multi-branch switch can be programmed in 8088 assembler.

The program starts by printing the string whose label is *strt*, inviting the user to type an octal digit (lines 4 through 7). Then a character is read from standard input (lines 8 and 9). If the value in AX is less than 5, the program interprets it as an end of file marker and jumps to the label *8* on line 22 to exit with a status code of 0.

If end of file has not been encountered, the incoming character, in AL, is inspected. Any character less than the digit 0 is considered to be white space and is ignored by the jump on line 13, which retrieves another character. Any character over digit 9 is considered to be incorrect input. On line 16, it is mapped onto the ASCII colon character, which is the successor of digit 9 in the ASCII character sequence.

Thus, on line 17 we have a value in AX between digit 0 and the colon. This value is copied into BX. On line 18, the AND instruction masks off all but the low-order four bits, which leaves the number between 0 and 10 (due to the fact that ASCII 0 is 0x30). Since we are going to index into a table of words, rather than bytes, the value in BX is multiplied by two using the left shift on line 19.

On line 20, we have a call instruction. The effective address is found by adding the value of BX to the numerical value of label *tbl*, and the contents of this composite address are loaded into the program counter, PC.

This program chooses one out of ten subroutines according to a character which is fetched from standard input. Each of those subroutines pushes the address of some message onto the stack and then jumps to a _PRINTF system subroutine call which is shared by all of them.

In order to understand what is happening, we need to be aware that the JMP and CALL instructions load some text segment address in PC. Such an address is just a binary number, and during the assembly process all addresses are replaced by their binary values. Those binary values can be used to initialize an array in the data segment, and this is done in line 50. Thus, the array starting at *tbl* contains the starting addresses of *rout*0, *rout*1, *rout*2, and so on, two bytes per address. The need for 2-byte addresses explains why we needed the 1-bit shift on line 19. A table of this type is often called a **dispatch table**.

How those routines work can be seen in the *erout* routine on lines 43 through 48. This routine handles the case of an out-of-range digit. First, the address of the message (in AX) is pushed onto the stack on line 43. Then the number of the _PRINTF system call is pushed onto the stack. After that, the system call is made, the stack is cleaned up, and the routine returns. The other nine routines, *rout0*

```
#include "../syscalnr.h"  ! 1
.SECT .TEXT               ! 2
jumpstrt:                 ! 3
    PUSH strt             ! 4
    MOV  BP,SP            ! 5
    PUSH _PRINTF          ! 6
    SYS                   ! 7
    PUSH _GETCHAR         ! 8
1:  SYS                   ! 9
    CMP  AX,5             ! 10
    JL   8f               ! 11
    CMPB AL,'0'           ! 12
    JL   1b               ! 13
    CMPB AL,'9'           ! 14
    JLE  2f               ! 15
    MOVB AL,'9'+1         ! 16
2:  MOV  BX,AX            ! 17
    AND  BX,0Xf           ! 18
    SAL  BX,1             ! 19
    CALL tbl(BX)          ! 20
    JMP  1b               ! 21
8:  PUSH 0                ! 22
    PUSH _EXIT            ! 23
    SYS                   ! 24

rout0: MOV  AX,mes0       ! 25
       JMP  9f            ! 26
rout1: MOV  AX,mes1       ! 27
       JMP  9f            ! 28
rout2: MOV  AX,mes2       ! 29
       JMP  9f            ! 30
rout3: MOV  AX,mes3       ! 31
       JMP  9f            ! 32
rout4: MOV  AX,mes4       ! 33
       JMP  9f            ! 34
rout5: MOV  AX,mes5       ! 35
       JMP  9f            ! 36
rout6: MOV  AX,mes6       ! 37
       JMP  9f            ! 38
rout7: MOV  AX,mes7       ! 39
       JMP  9f            ! 40
rout8: MOV  AX,mes8       ! 41
       JMP  9f            ! 42
erout: MOV  AX,emes       ! 43
9:     PUSH AX            ! 44
       PUSH _PRINTF       ! 45
       SYS                ! 46
       ADD  SP,4          ! 47
       RET                ! 48

.SECT .DATA                                                          ! 49
tbl: .WORD rout0,rout1,rout2,rout3,rout4,rout5,rout6,rout7,rout8,rout8,erout  ! 50
mes0: .ASCIZ "This is a zero.\n"                                     ! 51
mes1: .ASCIZ "How about a one.\n"                                    ! 52
mes2: .ASCIZ "You asked for a two.\n"                                ! 53
mes3: .ASCIZ "The digit was a three.\n"                              ! 54
mes4: .ASCIZ "You typed a four.\n"                                   ! 55
mes5: .ASCIZ "You preferred a five.\n"                               ! 56
mes6: .ASCIZ "A six was encountered.\n"                              ! 57
mes7: .ASCIZ "This is number seven.\n"                               ! 58
mes8: .ASCIZ "This digit is not accepted as an octal.\n"             ! 59
emes: .ASCIZ "This is not a digit. Try again.\n"                     ! 60
strt:  .ASCIZ "Type an octal digit with a return. Stop on end of file.\n"  ! 61
```

Figure C-18. A program demonstrating a multiway branch using a dispatch table.

through *rout8*, each load the addresses of their private messages in AX, and then jump to the second line of *erout* to output the message and finish the subroutine.

In order to get accustomed to the dispatch tables, the program should be traced with several different input characters. As an exercise, the program can be changed

in such a way that all characters generate a sensible action. For example, all characters other than the octal digits should give an error message.

C.8.7 Buffered and Random File Access

The program *InFilBuf.s*, shown in Fig. C-19, demonstrates random I/O on files. A file is assumed to consist of some number of lines, with different lines potentially having different lengths. The program first reads the file and builds a table in which entry *n* is the file position at which line *n* begins. Afterward, a line can be requested, its position looked up in the table, and the line read in by means of lseek and read system calls. The file name is given as the first input line on standard input. This program contains several fairly independent chunks of code, which can be modified for other purposes.

The first five lines simply define the system call numbers and the buffer size, and set the base pointer at the top of the stack, as usual. Lines 6 through 13 read the file name from standard input, and store it as a string at label *linein*. If the file name is not properly closed with a new line, then an error message is generated, and the process exits with a nonzero status. This is done in lines 38 through 45. Note that the address of the file name is pushed on line 39, and the address of an error message is pushed on line 40. If we examine the error message itself, (on line 113) then we have a *%s* string request in the _PRINTF format. The contents of the string *linein* are inserted here.

If the file name can be copied without problems, the file is opened on lines 14 to 20. If the open call fails, then the return value is negative and a jump is made to the label *9* on line 28 to print an error message. If the system call succeeds, then the return value is a file descriptor, which is stored in the variable *fildes*. This file descriptor is needed in the subsequent read and lseek calls.

Next, we read the file in blocks of 512 bytes, each of which is stored in the buffer *buf*. The buffer allocated is two bytes larger than the necessary 512 bytes, just to demonstrate how a symbolic constant and a integer can be mixed in an expression (on line 123). In the same way, on line 21 SI is loaded with the address of the second element of the *linh* array, which leaves a machine word containing 0 at the bottom of this array. The register BX will contain the file address of the first unread character of the file, and hence, it is initialized to 0 before the first time that the buffer is filled on line 22.

The filling of the buffer is handled by the *fillbuf* routine on lines 83 through 93. After pushing the arguments for the read, the system call is requested, which puts the number of characters actually read in AX. This number is copied into CX and the number of characters still in the buffer will be kept in CX thereafter. The file position of the first unread character in the file is kept in BX, so CX has to be added to BX in line 91. On line 92, the buffer bottom is put into DI in order to get ready to scan the buffer for the next new line character.

After returning from *fillbuf*, line 24 checks whether anything was actually read. If not, we jump out of the buffered read loop to the second part of the program in line 25.

Now we are ready to scan the buffer. The symbol \n (line feed) is loaded into AL on line 26, and in line 27 this value is scanned for by REP SCASB and compared to the symbols in the buffer. There are two ways to exit the loop: either when CX hits zero or when a scanned symbol is a new line character. If the zero flag is set, then the last symbol scanned was a \n and the file position of the current symbol (one after the new line), is to be stored in the *linh* array. The count is then incremented and the file position is computed from BX and the number of characters still available is in CX (lines 29 through 31). Lines 32 through 34 perform the actual store, but since STOS assumes that the destination is in DI instead of in SI, these registers are exchanged before and after the STOS. Lines 35 through 37 check whether more data is available in the buffer, and jump according to the value of CX.

When the end of the file is reached, we have a complete list of file positions of the heads of the lines. Because we started the *linh* array with a 0 word, we know that the first line started at address 0, and that the next line starts at the position given by *linh* + 2 etc. The size of line *n* can be found from the starting address of line *n* + 1 minus the start address of line *n*.

The aim of the rest of the program is to read the number of a line, to read that line into the buffer, and to output it by means of a write call. All the necessary information can be found in the *linh* array, whose *n*th entry contains the position of the start of line *n* in the file. If the line number requested is either 0 or out of range, the program exits by jumping to label 7.

This part of the program starts with a call to the *getnum* subroutine on line 46. This routine reads a line from standard input and stores it in the *linein* buffer, (on lines 95 through 103). Next, we prepare for the SSCANF call. Considering the reverse order of the arguments, we first push the address of *curlin*, which can hold an integer value, then the address of the integer format string *numfmt*, and finally the address of the buffer *linein* containing the number in decimal notation. The system subroutine SSCANF puts the binary value in *curlin* if possible. On failure, it returns a 0 in AX. This return value is tested on line 48; on failure, the program generates an error message through label 8.

If the *getnum* subroutine returns a valid integer in *curlin*, then we first copy it in BX. Next we test the value against the range in lines 49 through 53, generating an EXIT if the line number is out of range.

Then we must find the end of the selected line in the file and the number of bytes to be read, so we multiply BX by 2 with a left shift SHL. The file position of the intended line is copied to AX on line 55. The file position of the next line is in CX and will be used to compute the number of bytes in the current line.

To do a random read from a file, an lseek call is needed to set the file offset to the byte to be read next. The lseek is performed with respect to the start of the file,

```
#include "../syscalnr.h" ! 1
bufsiz = 512             ! 2
.SECT .TEXT              ! 3
infbufst:                ! 4
    MOV  BP,SP           ! 5
    MOV  DI,linein       ! 6
    PUSH _GETCHAR        ! 7
1:  SYS                  ! 8
    CMPB AL,'\n'         ! 9
    JL   9f              ! 10
    JE   1f              ! 11
    STOSB                ! 12
    JMP  1b              ! 13
1:  PUSH 0               ! 14
    PUSH linein          ! 15
    PUSH _OPEN           ! 16
    SYS                  ! 17
    CMP  AX,0            ! 18
    JL   9f              ! 19
    MOV  (fildes),AX     ! 20
    MOV  SI,linh+2       ! 21
    MOV  BX,0            ! 22
1:  CALL fillbuf         ! 23
    CMP  CX,0            ! 24
    JLE  3f              ! 25
2:  MOVB AL,'\n'         ! 26
    REPNE SCASB          ! 27
    JNE  1b              ! 28
    INC  (count)         ! 29
    MOV  AX,BX           ! 30
    SUB  AX,CX           ! 31
    XCHG SI,DI           ! 32
    STOS                 ! 33
    XCHG SI,DI           ! 34
    CMP  CX,0            ! 35
    JNE  2b              ! 36
    JMP  1b              ! 37
9:  MOV  SP,BP           ! 38
    PUSH linein          ! 39
    PUSH errmess         ! 40
    PUSH _PRINTF         ! 41
    SYS                  ! 42
    PUSH _EXIT           ! 43
    PUSH _EXIT           ! 44
    SYS                  ! 45
3:  CALL getnum          ! 46
    CMP  AX,0            ! 47
    JLE  8f              ! 48
    MOV  BX,(curlin)     ! 49
    CMP  BX,0            ! 50
    JLE  7f              ! 51
    CMP  BX,(count)      ! 52
    JG   7f              ! 53
    SHL  BX,1            ! 54
    MOV  AX,linh-2(BX)   ! 55
    MOV  CX,linh(BX)     ! 56
    PUSH 0               ! 57
    PUSH 0               ! 58
    PUSH AX              ! 59
    PUSH (fildes)        ! 60
    PUSH _LSEEK          ! 61
    SYS                  ! 62
    SUB  CX,AX           ! 63
    PUSH CX              ! 64
    PUSH buf             ! 65
    PUSH (fildes)        ! 66
    PUSH _READ           ! 67
    SYS                  ! 68
    ADD  SP,4            ! 69
    PUSH 1               ! 70
    PUSH _WRITE          ! 71
    SYS                  ! 72
    ADD  SP,14           ! 73
    JMP  3b              ! 74
8:  PUSH scanerr         ! 75
    PUSH _PRINTF         ! 76
    SYS                  ! 77
    ADD  SP,4            ! 78
    JMP  3b              ! 79
7:  PUSH 0               ! 80
    PUSH _EXIT           ! 81
    SYS                  ! 82
fillbuf:                 ! 83
    PUSH bufsiz          ! 84
    PUSH buf             ! 85
    PUSH (fildes)        ! 86
    PUSH _READ           ! 87
    SYS                  ! 88
    ADD  SP,8            ! 89
    MOV  CX,AX           ! 90
    ADD  BX,CX           ! 91
    MOV  DI,buf          ! 92
    RET                  ! 93

getnum:                  ! 94
    MOV  DI,linein       ! 95
    PUSH _GETCHAR        ! 96
1:  SYS                  ! 97
    CMPB AL,'\n'         ! 98
    JL   9b              ! 99
    JE   1f              !100
    STOSB                !101
    JMP  1b              !102
1:  MOVB (DI),'\0'       !103
    PUSH curlin          !104
    PUSH numfmt          !105
    PUSH linein          !106
    PUSH _SSCANF         !107
    SYS                  !108
    ADD  SP,10           !109
    RET                  !110

.SECT .DATA              !111
errmess:                 !112
.ASCIZ "Open %s failed\n" !113
numfmt:  .ASCIZ "%d"     !114
scanerr:                 !115
.ASCIZ "Type a number.\n" !116
.ALIGN 2                 !117
.SECT .BSS               !118
linein:  .SPACE 80       !119
fildes:  .SPACE 2        !120
linh:    .SPACE 8192     !121
curlin:  .SPACE 4        !122
buf:     .SPACE bufsiz+2 !123
count:   .SPACE 2        !124
```

Figure C-19. A program with buffered read and random file access.

so first an argument of 0 is pushed to indicate this on line 57. The next argument is the file offset. By definition, this argument is a long (i.e., 32-bit) integer, so we first push a 0 word and then the value of AX on lines 58 and 59 to form a 32-bit integer. Then the file descriptor and code for LSEEK are pushed and the call is made on line 62. The return value of LSEEK is the current position in the file and can be found in the DX : AX register combination. If the number fits into a machine word (which it will for files shorter than 65536 bytes), then AX contains the address, so subtracting this register from CX (line 63), yields the number of bytes to be read in order to bring the line into the buffer.

The rest of the program is easy. The line is read from the file on lines 64 through 68 and then it is written to standard output via file descriptor 1 on lines 70 through 72. Note that the count and the buffer value are still on the stack after the partial stack cleanup on line 69. Finally, on line 73, we reset the stack pointer completely and we are ready for the next step, so we jump back to label 3, and restart with another call to *getnum*.

Acknowledgements

The assembler used in this appendix is part of the "Amsterdam Compiler Kit." The full kit is available online at *www.cs.vu.nl/ack*. We thank the people who were involved in the original design: Johan Stevenson, Hans Schaminee, and Hans de Vries. We are especially indebted to Ceriel Jacobs, who maintains this software package, and who has helped adapt it several times to meet the teaching requirements of our classes, and also to Elth Ogston for reading the manuscript and testing the examples and exercises.

We also want to thank Robbert van Renesse and Jan-Mark Wams, who designed tracers for the PDP-11 and the Motorola 68000, respectively. Many of their ideas are used in the design of this tracer. Moreover, we wish to thank the large group of teaching assistants and system operators who have assisted us during many assembly language programming courses over a period of many years.

PROBLEMS

1. After the instruction MOV *AX*, 702 is executed, what are the decimal values for the contents of AH and AL?
2. The CS register has the value 4. What is the range of absolute memory addresses for the code segment?
3. What is the highest memory address the 8088 can access?

4. Suppose that CS = 40, DS = 8000, and IP = 20.

a. What is the absolute address of the next instruction?
b. If MOV *AX*, (2) is executed, which memory word is loaded into AX?

5. A subroutine with three integer arguments is called following the calling sequence described in the text, that is, the caller pushes the arguments onto the stack in reverse order, then executes a CALL instruction. The callee then saves the old BP and sets the new BP to point to the saved old one. Then the stack pointer is decremented to allocate space for local variables. With these conventions, give the instruction needed to move the first argument into AX.

6. In Fig. C-1 the expression *de* − *hw* is used as an operand. This value is the difference of two labels. Might there be circumstances in which *de* + *hw* could be used as a valid operand? Discuss your answer.

7. Give the assembly code for computing the expression:

```
x = a + b + 2
```

8. A C function is called by

```
foobar(x, y);
```

Give the assembly code for making this call.

9. Write an assembly language program to accept input expressions consisting of an integer, an operator, and another integer and output the value of the expression. Allow the +, −, ×, and / operators..Pc

INDEX

This page intentionally left blank

INDEX

A

Absolute path, 494
Accelerated graphics port bus, 217
Access control list, 501
Access token, 501
Accumulator, 19, 695
Achieving high performance, 650, 650–652
ACK (*see* Amsterdam Compiler Kit)
Acknowledgment packet, 226
ACL (*see* Access Control List)
Acorn Archimedes computer, 45
Active matrix display, 117
Actual parameter, macro, 526
Adder, 164–166
  carry select, 165–166
  full, 165–166
  half, 164–165
  rippler carry, 165
Additive inverse, 390
Address, 74, 694
Address decoding, 233
Address modes, 371
Address space, 439
Address-space identifier, 461
Addressing, 362, 371–386
  8088, 701–704
  ARM, 384
  ATmega168, 384–385
  based-indexed, 376
  branch instructions, 380–381
  Core i7, 382–384
  direct, 372
  displacement, 703
  immediate, 372
  indexed, 374
  register, 372
  register indirect, 373–374
  stack, 376–379
Addressing and memory, 8088, 699–704
Addressing direct, 372
Addressing modes, 384–385
  ARM, 384
  discussion, 385–386
ADSL (*see* Asymmetric DSL)
Advanced microcontroller bus architecture, 574
Advanced programmable interrupt controller, 205
Aggregate bandwidth, 647
AGP bus (*see* Accelerated Graphics Port bus)
Aiken, Howard, 16

Algorithm, 8
Allocation/renaming unit, 327
Alpha, 14, 26
ALU (*see* Arithmetic Logic Unit)
AMBA (*see* Advanced Microcontroller Bus Architecture)
Amdahl's law, 649
American Standard Code for Information Interchange, 138
Amplitude modulation, 128
Amsterdam compiler kit, 716
Analytical engine, 15
APIC (*see* Advanced Programmable Interrupt Controller)
Apple II, 24
Apple Lisa, 14
Apple Macintosh, 24
Apple Newton, 14, 45
Application layer, 654
Application programming interface, 487
Application-level shared memory, 640–646
Application-specific integrated circuit, 578
Architecture, computer, 8, 60
Arduino, 34
Argument, 711
Arithmetic circuit, 163–166
Arithmetic logic unit, 6, 56–57, 166–167
ARM,
  addressing, 384
  bi-endianness, 354
  data types, 361
  instruction formats, 368–370
  instructions, 400–402
  introduction, 45–47
  ISA level, 354–356
  microarchitecture of OMAP4430, 329–334
  virtual memory, 460–462
ARM (*see also* OMAP4430)
ARM addressing modes, 385
ARM data types, 361
ARM instruction formats, 368–370
ARM instructions, 400–402
ARM v7, 346–347
ARM virtual memory, 460–462
As88, 716–720
ASCII (*see* American Standard Code for Information Interchange)
ASIC (*see* Application Specific Integrated Circuit)
Assembler, 7, 518, 692, 715–721
  8088, 715
Assembler directive, 522
Assembler pass one, 530–534
Assembler pass two, 534–536
Assembly language, 518–529, 691–745
  example, 693–694
Assembly-language level, 517–550
Assembly-language macro, 524–529
Assembly-language program, 716
Assembly-language statement, 520–521
Assembly process, 529–536
Asserted signal, 178
Associative memory, 454, 535
Asymmetric DSL, 130
Asynchronous bus, 194–196
AT attachment disk, 92
ATA packet interface, 92
ATA-3 disk, 92
Atanasoff, John, 16
ATAPI-4 disk, 92
ATAPI-5 disk, 92
ATmega168, 212–214
  ISA level, 356–358
  addressing modes, 384–385
  data types, 361–362
  instruction formats, 370–371
  instructions, 402
  microarchitecture, 334–336
Atmel ATmega168 (*see* ATmega168)
Attraction memory, 615
Auto decrement addressing, 704
Auto increment addressing, 704
Auxiliary carry flag, 708
AVR, 47–49

B

Babbage, Charles, 14–15
Baby feeding algorithm, 445
Backward compatibility, 344
Ball grid array, 211
Bank, 206
Banking, Core i7, 328
Bardeen, John, 19
Base, 148
Base pointer, 698
Base register, 695
Based-indexed addressing, 376

Basic block, 320
Basic input/output system, 91
Batch system, 10–11
Baud, 129
Bayer filter, 135
BCD (*see* Binary Coded Decimal)
Benchmark, 293
Best-fit algorithm, 454
Bi-endian processor, 354
Big endian memory, 76–78
Binary, 671
Binary arithmetic, 678
Binary coded decimal, 74, 702
Binary number,
  addition of, 678
  conversion between radices, 673–675
  negative, 675–677
Binary program, 692
Binary search, 536
Binding time, 542–545
BIOS (*see* Basic Input Output System)
Bipolar transistor, 150
Bisection bandwidth, 618
Bit, 74, 672
Bit map, 360
Bit slice, 166
Block cache, 484
Block started by symbol, 717
Blocking network, 606
Blu-ray disc, 108
BlueGene, 622–626
BlueGene/L, 622
BlueGene/P, 622–626
BlueGene/P vs. Red Storm, 629–631
Boole, George,
Boolean algebra, 147–157
Boolean function, 152–153
Branch history shift register, 314
Branch instructions, addressing, 380–381
Branch prediction, 310–315, 315
  dynamic, 312–315
Branch predictor, Core i7, 326
Branch target buffer, 327
Brattain, Walter, 19
Breakpoint, 724
Broadband, 129
BSS section, 717
BTB (*see* Branch Target Buffer)
Bubblejet printer, 126
Bucket, 536
Buffered message passing, 637
Bundle, Itanium 2, 425
Burroughs B5000, 14, 21
Bus, 20, 55, 108–112, 187–201, 214–232
  ACP, 217
  asynchronous, 194–196
  Core i7, 206–209
  EISA, 111
  ISA, 110
  multiplexed, 191
  PCI, 111–112
  PCIe, 111–112
  synchronous, 191–194
  system, 188
Bus arbiter, 110
Bus arbitration, 196–198
Bus clocking, 191–194
Bus cycle, 191
Bus driver, 189
Bus grant, 196
Bus master, 189
Bus operation, 198–201
Bus protocol, 188
Bus receiver, 189
Bus skew, 112, 191
Bus slave, 189
Bus transceiver, 189
Bus width, 190–191
Busy waiting, 395
Byron, Lord, 15
Byte, 75, 347
Byte instruction, 701
Byte ordering, 76–78
Byte register, 701

C

Cache, 304–310
  direct-mapped, 306–308
  level 2, 305
  MESI protocol, 601–603
  set-associative, 308–310
  snooping, 599–606
  split, 84, 305
  unified, 84
  update strategy, 600
  write back, 310

Cache (*continued*)
write-deferred, 310
write-through, 310, 599
write-allocate, 310
Cache coherence, 599
Cache coherence protocol, 599
Cache coherent NUMA, 607–608
Cache consistency, 599
Cache hit, 307
Cache invalidate strategy, 600
Cache line, 84, 306, 599
Cache memory, 41, 82–85
Cache miss, 307
Cache only memory access, 592, 614–616
Call gate, 460
Camera, digital, 135–137
Capacitor, 114
Carrier, 128
Carry select adder, 165–166
Catamount, 629
Cathode-ray tube, 115
CC-NUMA (*see* Cache Coherent NUMA)
CCD (*see* Charge-Coupled Device)
CD-R (*see* CD Recordable)
CD-Recordable, 103–105
CD-Rewritable, 105
CD-ROM (*see* Compact Disc-Read Only Memory)
CD-ROM track, 105
CD-ROM XA, 105
CDC (*see* Control Data Corporation)
CDC 6600, 14, 21
Celeron, 42
Cell, memory, 74
Central processing unit, 19, 55–73
Character code, 137–142
Charge-coupled device, 135
Checkerboarding, 453
Checksum generation, 581
Checksum verification, 580
Child process, 503
Chip, 158
Chipset, 205
Circuit,
arithmetic, 163–166
combinational, 159–163
Circuit equivalence, 153–158
CISC (*see* Complex Instruction Set Computer)
Clock, 168–169
Clock cycle time, 168
Clocked D latch, 171–172
Clocked SR latch, 170–171
Clone, 24
Closure, 670
Cloud computing, 38
Cluster, 37, 502, 593
Cluster computing, 631–636
Cluster of workstations, 593
CMYK printer, 125
Coarse-grained multithreading, 563
COBOL, 39, 359
Code generation, 716
Code page, 138
Code point, 139
Code segment, 694
Codesign, 27
Codeword, 78
Collective layer, 653
Collector, 148
Color gamut, 125
Color palette, 118
Color printing, 124–127
COLOSSUS, 14, 16–17
COMA (*see* Cache Only Memory Access computer)
COMA multiprocessor, 614
Combinational circuit, 159–163
Committed page, 490
Commodity off-the-shelf cluster, 37
Communicator, 637
Compact disc-read only memory, 101–103
Comparator, 162–163
Comparison instructions, 390–392
Comparison of instruction sets, 402
Compiler, 7, 518
Completeness property, 153
Complex instruction set computer, 62
Compute unified device architecture, 583
Computer,
data parallel, 70–72
disposable, 31–33
game, 35–36
Computer architecture, 8
milestones, 13–28
Computer center, 23
Computer zoo, 28–39
Condition code, 350
Condition code register, 699
Condition variable, 506
Conditional branch instructions, 390–392
Conditional execution, 427
Conditional jump, 710

Consistency,
cache, 599
processor, 595–596
release, 597–598
sequential, 594–595
strict, 594
weak, 596–597, 597–598
Consistency model, 594
Constant pool, 260
Consumer, 474
Control Data Corporation, 21
Control signal, 249
Control store, 61, 253
Controller, 109
Conversion between radices, 673–675
Coprocessor, 574–586
Copy on write, 489
Core, 25, 568
Core 2 duo, 42
Core dump, 10
Core i7,
addressing, 382–384
addressing modes, 382–384
banking, 328
branch predictor, 326
data types, 360
hyperthreading, 564–568
introduction, 35–47
instruction formats, 367–368
instructions, 397–400
ISA level, 351–353
memory model, 347–348
microarchitecture, 323–329
multiprocessor, 569
photo of the die, 43
pinout, 204–208
pipelining, 206–208
reorder buffer, 327
retirement unit, 328
scheduler, 327
virtual memory, 455–460
virtualization, 464
CoreConnect, 572
Coroutine, 410–413
Cortex A9, 329–334
COTS (*see* Commodity Off The Shelf)
Counter register, 697
COW (*see* Collection of Workstations)
CP/M, 24
CPU (*see* Central Processing Unit)
CPU chip, 185–187
CPU organization, 56
Cray, Seymour, 21
CRAY-1, 14
CRC (*see* Cyclic Redundancy Check)
CRC (*see* Cyclic Redundancy Code)
Critical section, 509
Crossbar switch, 604
Crosspoint, 604
CRT (*see* Cathode Ray Tube)
Cryptography,
public-key, 585
symmetric-key, 585
Cryptoprocessor, 585–586
Cube network, 620
CUDA (*see* Compute Unified Device Architecture)
Cycle stealing, 110, 397
Cyclic redundancy check, 226
Cylinder, disk, 89

D

D latch, 171–172
Daisy chaining, 196
Data cache, OMAP4430, 331
Data center, 37
Data movement instructions, 386–387
Data parallel computer, 70–72
Data path, 6, 56, 244–250
Mic-1, 254
Mic-2, 292
Mic-3, 296
Mic-4, 301
Data path cycle, 58
Data path timing, 247–249
Data register, 697
Data section, 717
Data segment, 709
Data type, nonnumeric, 359–360
numeric, 358–359
Data types, 358–362
ARM, 360
ATmega168, 361–362
Core i7,360
DDR (*see* Double Data Rate RAM)
De Morgan's law 156

DEC (*see* Digital Equipment Corporation)
DEC Alpha, 14, 26
DEC PDP-1, 14, 20
DEC PDP-11, 14
DEC PDP-8, 14, 20
DEC VAX, 14, 61
Decimal number, 718
Decoder, 161–162
Decoding unit, 300
Degree, node, 618
Delay slot, 311
Demand paging, 443–446, 445
Demultiplexer, 161
Denormalized number, 687
Dependence, 297, 318
Design principles, 63–65
Destination determination, 580
Destination index, 698
Destination operand, 701
Device level, 5
Device register bus, 573
Diameter, network 618
Dibit encoding, 129
Difference engine, 15
Digital camera, 135–137
Digital Equipment Corporation, 14, 19–20
Digital logic circuits, 158–169
Digital logic level, 5
  buses, 185–201, 214–232
  circuits, 158–169
  CPU chips, 201–214
  gates, 147–157
  I/O interfacing, 232–235
  memory, 169–185
  PCI bus, 215–223
  PCI express bus, 223–228
Digital subscriber line, 129–132, 130
Digital subscriber line access multiplexer, 132
Digital versatile disk, 106–108
Digital video disk, 106–108
Dimensionality, 619
DIMM (*see* Dual Inline Memory Module)
DIP (*see* Dual Inline Package)
Direct addressing, 372, 702
Direct memory access, 109, 396
Direct-mapped cache, 306
Direction flag, 704
Directory, 471
Directory management instructions, 471
Directory-based multiprocessor, 608
Disambiguation, 329
Disk,
  ATAPI, 92
  CD-ROM, 101–103
  DVD, 106–108
  IDE, 91–92
  magnetic, 87–97
  optical, 99–108
  RAID, 94–97
  SCSI, 92–94
  SSD, 97–99
  Winchester, 111
Disk controller, 90
Diskette, 89
Disposable computer, 31–33
Distributed memory system, 587
Distributed shared memory, 589–591, 640–642
DLL (*see* Dynamic Link Library)
DMA (*see* Direct Memory Access)
Dot, 716
Dots per inch, 126
Double data rate RAM, 181
Double indirect block, 497
Double integer, 697
Double torus, 620
Double-precision number, 358
DPI (*see* Dots Per Inch)
DRAM (*see* Dynamic RAM)
DSL (*see* Digital Subscriber Line)
DSLAM (*see* Digital Subscriber Line Access Multiplexer)
DSM (*see* Distributed Shared Memory)
Dual, 155
Dual inline memory module, 85
Dual inline package, 158
DVD (*see* Digital Video Disk)
Dyadic operations, 387–388
Dye sublimation printer, 127
Dye-based ink, 126
Dynamic branch prediction, 312–315
Dynamic link library, 547
Dynamic linking, 545–549
  MULTICS, 545–547
  UNIX, 549
  Windows, 547–549
Dynamic RAM, 181
Dynamic voltage scaling, 209

E

Eckert, J. Presper, 17
ECL (*see* Emitter-Coupled Logic)
Edge-triggered flip-flop, 172
EDO (*see* Extended Data Output)
EDSAC, 14
EDVAC, 17
EEPROM (*see* Electrically Erasable PROM)
Effective address, 704
Egress processing, 580
EHCI (*see* Enhanced Host Controller Interface)
EIDE (*see* Extended IDE)
EISA bus (*see* Extended ISA bus)
Electrically erasable PROM, 182–184
Electronic discrete variable automatic computer, 17
Electronic numerical integrator and computer, 17
Emitter, 148
Emitter-coupled logic, 150
Emulation, 22
Encoding, 8b/10b, 226
Endian memory,
  big, 76–78
  little, 76–78
Enhanced host controller interface, 231
ENIAC, 14, 17
ENIGMA, 16
Entry point, 542
EPIC (*see* Explicitly Parallel Instruction Computing)
EPROM (*see* Erasable PROM)
EPT (*see* Extended Page Table)
Erasable PROM, 182
Error-correcting code, 78–82
Escape code, 367
Estridge, Philip, 24
Ethernet, 575
Event, 509
Evolution of multilevel machines, 8–13
Example computer families, 39
Example programs, 8088, 726–745
Excess notation, 676
Executable binary file, 716
Executable binary program, 518, 537
Executive, NTOS, 487
Expanding opcode, 365–367
Explicit linking, 548
Explicitly parallel instruction computing, 423
Exponent, 682
Extended data output, 181
Extended IDE disk, 91
Extended ISA bus, 111
Extended page table, 464
External fragmentation, 453
External reference, 539
External symbol, 542
Extra segment, 709

F

Fabric layer, 653
False sharing, 641
Fanout, 618
Far call, 710
Far jump, 8088, 709
Fast page mode memory, 181
FAT (*see* File Allocation Table)
Fat tree, 620
Fermi GPU, 70–71, 582–585
Fetch-decode-execute cycle, 58, 244
Field extraction, 580
Field-programmable gate array, 25, 183–185, 578
FIFO algorithm (*see* First-In First-Out algorithm)
Fifth-generation computers, 26–28
Fifth-generation project, Japanese, 26
File, 465–467
File allocation table, 498
File descriptor, 492, 713
File index, 468
Filter, 494
Fine-grained multithreading, 562
Finite state machine, 290, 314
Finite-precision numbers, 669–671
Firewall, 576
First-fit algorithm, 454
First pass, assembler, 716
First-generation computers, 16–19
First-in, first-out algorithm, 447
Flags register, 350, 699
Flash memory, 183
Flat panel display, 115–117
Flip-flop, 172–174
Floating-point number, 681–688
Floppy disk, 89
Flow control, 227
Flow of control, 403–417, 404–417
  branches, 405–406
  coroutines, 410–413

Flow of control (*continued*)
interrupts, 414–417
procedures, 406–410
sequential, 405–406
traps, 413
Flynn's taxonomy, 591–593
Formal parameter, macro, 526
Forrester, Jay, 19
FORTRAN, 10, 392
FORTRAN Monitor System, 10
Forward reference problem, 529
Fourth-generation computers, 23–26
FPGA (*see* Field Programmable Gate Array)
FPGA (*see* Field-Programmable Gate Array)
FPM (*see* Fast Page Mode)
Fraction, 682
Fragmentation, 448–449
external, 453
internal, 448
Fragmentation and reassembly, 581
Frame, 101
local variable, 258–260
Frame pointer, 353
Free list, 469
Free page, 490
Frequency modulation, 128
Frequency shift keying, 128
FSM (*see* Finite State Machine)
Full adder, 165–166
Full handshake, 195
Full interconnect, 620
Full resource sharing, 566
Full-duplex line, 129
Function, 392

G

Game computer, 35–36
Game controller, 120–122
Gamut, color, 125
Gate, 5, 148–150
Gate delay, 159
GDT (*see* Global Descriptor Table)
General registers, 695–697
General-purpose CPU, 584
Ghosting, 114
Global descriptor table, 455
Global label, 717
Goldstine, Herman, 18
Google cluster, 632–636
GPGPU (*see* General Purpose GPU)
GPU (*see* Graphics Processing Unit)
Graphical user interface, 24, 485
Graphics processing unit, 70–72, 582–585
Graphics processor, 582–585
Green book, 103
Grid, 620
Grid computing, 652–655
GridPad, 14
Guest operating system, 464
GUI (*see* Graphical User Interface)

H

H register 245–246
Half adder, 164–165
Half-duplex line, 129
Halftone screen frequency, 124
Halftoning, 124
Hamming, Richard, 30
Hamming distance, 78
Handle, 488
Hard disk, 87–97
Hardware, 8
equivalence with software, 8
Hardware abstraction layer, 486
Hardware DSM, 607
Hardware virtualization, 463–464
Harvard architecture, 84
Harvard Mark I, 16
Hash coding, 536
Hashing, 536
Hawkins, Jeff, 27
Hazard, 297
Headend, 132
Header, PCI packet, 225
Header management, 581
Headless workstation, 631
Hello world example, 8088, 726
Heterogeneous multiprocessor, 570–574
Hexadecimal, 671
Hexadecimal number, 718
High Sierra, 103
High-level language, 7

History,
1642–1945, 13–16
1945–1955, 16–19
1955–1965, 19–21
1965–1980, 21–23
ARM, 45–47
AVR, 47–49
Intel, 39–45
Hit ratio, 84
Hoagland, Al, 30
Hoff, Ted, 40
Hoisting, 321
Homogeneous multiprocessors, 568–569
Host library, 549
HTTP (*see* HyperText Transfer Protocol)
Hypercube, 621
Hypertext transfer protocol, 576
Hyperthreading, Core i7, 564–568
Hypervisor, 463

I

I-node, 496
I/O (*see* Input/Output)
I/O interface, 232–236
IA-32 architecture, 351
IA-32 flaws, 421–423
IA-64, 420–429, 423–429
bundle, 425
EPIC model, 423
instruction scheduling, 424–426
speculative loads, 429
IAS, 14
IAS machine, 18
IBM 1401, 14, 20, 21–22
IBM 360, 14, 22–23
IBM 701, 19
IBM 704, 19
IBM 709, 10
IBM 7094, 14, 20–21, 21–22
IBM 801, 62
IBM CoreConnect, 572
IBM Corporation, 19–21
IBM PC, 14, 24–25
IBM POWER4, 14, 26
IBM PS/2, 41
IBM RS6000, 14
IBM Simon, 14
IC (*see* Integrated Circuit)
IDE disk, 91–92
IEEE Floating-point standard 754, 684–688
IFU (*see* Instruction Fetch Unit)
IJVM, 258–282
compiling Java to, 265–267
IJVM (*see* Integer JVM)
IJVM implementation, 271–282
IJVM instruction set, 262–265
IJVM memory model, 260
ILC (*see* Instruction Location Counter)
ILLIAC, 17, 70
Immediate addressing, 372, 704
Immediate file, 503
Immediate operand, 372
Implicit linking, 548
Implied addressing, 704
Import library, 548
Imprecise interrupt, 318
Improving network performance, 582
Improving performance, 650–652
Index register, 698
Indexed addressing, 374–376
Indexed color, 118
Indirect block, 497
Industry standard architecture bus, 110
Infix, 376
Informative section, 346
Ingress processing, 580
Initiator, PCI bus, 218
Ink,
dye-based, 126
pigment-based, 126
solid, 126
Inkjet printer, 125–126
Input/output, 108–142
Input/output device,
color printer, 124–125
digital camera, 135–137
flat panel display, 115–118
game controller, 120–122
inkjet printer, 125–126
keyboard, 113
laser printer, 122–124
mice, 118–120
modem, 127–129
specialty printers, 126–127
telecommunication equipment, 127–132
touch screen, 113–115

Input/output instructions, 394–397
Instruction,
  ARM 400–402
  ATmega168, 402
  comparison, 390–392
  conditional branch, 390–392
  Core i7, 397–400
  data movement, 386–387
  dyadic, 387–388
  input/output, 394–397
  ISA level, 351
  loop control, 393–394
  monadic, 388–390
  predicated, 370
  procedure call, 392–393
Instruction execution, 58–62
Instruction fetch unit, 288–291
Instruction format, 362–371
  ARM, 368–370
  ATmega168, 370–371
  Core i7, 367–368
  design criteria, 362–365
Instruction group, Itanium 2, 424
Instruction issue unit, OMAP4430, 330
Instruction location counter, 530
Instruction pointer, 694, 698
Instruction register, 56
Instruction scheduling, Itanium 2, 424
Instruction set, 8088, 705–715
Instruction set architecture, 691
Instruction set architecture level, 343–430
Instruction types, 386–403
Instruction-level parallelism, 65–69, 555–561
Instructions,
  ARM, 400–402
  ATmega168, 403
  Core i7, 397–400
Integer JVM, 258–267
  compiling Java to, 265–267
  implementation, 267–282
  instruction set, 262–265
  memory model, 260–261
  stack, 258–260
Integrated circuit, 158–159
Integrated drive electronics, 91
Intel 386, 14
Intel 4004, 40
Intel 80286, 41
Intel 80386, 41
Intel 80486, 41
Intel 8080, 14
Intel 8086, 41
Intel 8088, 694–704
  addressing, 699–704
  example programs, 726–745
  far jump, 709
  near jump, 709
  segments, 700
Intel 8088 instruction set, 705–715
Intel Core i7 (*see* Core i7)
Intel Xeon, 43
Interconnection network, 617–621
  bisection bandwidth, 618
  topology, 618
Interfacing, 232–236
Interleaved memory, 606
Internal fragmentation, 448
Internet over cable, 132–135
Internet protocol, 576
Internet service provider, 576
Interpretation, 2
Interpreter, 2, 58–59, 693
Interrupt, 109, 414–417
  imprecise, 318
  precise, 318
  transparent, 415
Interrupt handler, 109
Interrupt vector, 200, 414
Intersector gap, 87
Introduction, 715
Invalidate strategy, 600
Inversion bubble, 149
Inverter, 149
Inverting buffer, 177
Invisible computer, 26–28
IP header, 576
IP protocol, 576
IR (*see* Instruction Register)
Iron Oxide Valley, 20
ISA bus (*see* Industry Standard Architecture bus)
ISA level, 6
  addressing, 371–386
  ARM, 354–356
  ATmega168, 356–358
  Core i7, 351–353
  data types, 358–362
  flow of control, 404–417
  instruction formats, 362–371
  instruction types, 386–404
  overview, 345–358

ISA level properties, 345–347
ISP (*see* Internet Service Provider)
Itanium 2, 420–429

J

Java virtual machine, 260
Jobs, Steve, 24
Johnniac, 17
Joint Photographic Experts Group, 137
Joint test action group, 206
JPEG (*see* Joint Photographic Experts Group)
JTAG (*see* Joint Test Action Group)

K

Kernel mode, 347
Key, record, 468
Keyboard, 113
Kilby, Jack, 21
Kildall, Gary, 24
Kinect controller, 122

L

Label, 692
  assembly language, 717
LAN (*see* Local Area Network)
Land, 100
Land grid array, 158
Lane, PCI Express, 112, 226
Language, 1
Large memory model, 720
Laser printer, 122–134
Latch, 169–172
  D, 171–172
Latency, 67
Latin-1, 138
Layer, 3
LBA (*see* Logical Block Addressing)
LCD (*see* Liquid Crystal Display)
LDT (*see* Local Descriptor Table)
Least recently used algorithm, 309, 446
LED (*see* Light Emitting Diode)
Left value, 701
Leibniz, Gottfried Wilhelm von, 13
Level, 3
  assembly-language, 517–550
  device, 5
  digital logic, 147–237
  ISA, 6, 343–430
  microarchitecture, 6, 243–338
  operating-system machine, 7, 437–510
Level 2 cache, 305
Level-triggered latch, 172
LGA (*see* Land Grid Array)
Light emitting diode, 119
Linda, 642–644
Linear address, 457
Lines per inch, 124
Link, file, 494
Link layer, PCI Express, 226
Linkage editor, 536
Linkage segment, 545
Linker, 536, 716
  tasks performed, 538–541
Linking, 536–549
  binding time, 542–545
  dynamic, 545–549
  MULTICS, 545–547
  UNIX, 549
  Windows, 547–548
Linking loader, 536
Liquid crystal display, 115
Lisa, 14
Literal, 532
Little endian, 702
Little endian memory, 76–78
Load/store architecture, 356
Loading, 536–549
Local descriptor table, 455
Local label, 717
Local loop, 130
Local variable frame, 258–260
Local-area network, 575
Locality principle, 83, 446
Location, 74
Location counter, 716
Logical block addressing, 91
Logical record, 466
Long integer, 697

Lookup table, 183–185
Loop control instructions, 393–394
Loosely coupled system, 73, 554
Lovelace, Ada, 15
LPI (*see* Lines Per Inch)
LRU (*see* Least Recently Used algorithm)
LRU algorithm, (*see* Least Recently Used algorithm)
LRU algorithm (*see* Least Recently Used algorithm)
LUT (*see* LookUp Table)

M

M.I.T.
  TX-0, 19
  TX-2, 19
Machine language, 1, 691
Macintosh, Apple, 24
Macro, assembly language, 524–529
Macro call, 525
Macro definition, 524
Macro expansion, 525
Macro parameter, 526–527
Macro processor, implementation, 528–529
Macroarchitecture, 258
Magnetic disk, 87–97
Mainframe, 38–39
MAL (*see* Micro Assembly Language)
MANIAC, 17
Mantissa, 682
MAR (*see* Memory Address Register)
Mark I, 14
Mask, 387
MASM (*see* Microsoft ASseMbler)
Massively parallel processor, 593, 621–631
  BlueGene, 622–626
  Red Storm, 626–631
Master, bus, 189
Master file table, 502
Masuoka, Fujio, 98
Mauchley, John, 16
MDR (*see* Memory Data Register)
Memory, 73–85, 169–185
  associative, 454, 535
  attraction, 615
  big endian, 76–78
  cache, 41, 82–85
  cache only access, 614–616
  compact disc-read only, 101–103
  fast page mode, 181
  flash, 183
  interleaved, 606
  little endian, 76–78
  nonvolatile, 182–183
  random access, 73–85, 180–183
  read only, 182
  secondary, 86–108
  virtual, 438–462
Memory address, 74–76
Memory address register, 249–250
Memory and addressing, 8088, 699–704
Memory chip, 178–180
Memory data register, 249–250
Memory hierarchy, 86–87
Memory management unit, 442
Memory map, 440
Memory model, 347–349, 720
  Core i7, 347–348
Memory operation, 249–250
Memory organization, 174–178
  8088, 699–701
Memory packaging, 85
Memory semantics, 593–598
  cache, 599
  processor, 595–596
  release, 597–598
  sequential, 594–595
  strict, 594
  weak, 596–597
Memory-mapped I/O, 234
Mesh, 620
MESI protocol, 601–603
Message queue, 504
Message-passing interface, 637–639
Message-passing multicomputer, 616–652
Metal oxide semiconductor, 150
Method, 392
Method area, 260
Metric units, 49–50
MFT (*see* Master File Table)
Mic-1, 253, 253–258, 267–282
Mic-2, 291–293
Mic-3, 293–300
Mic-4, 300–303
Mickey, 120
Micro assembly language, 268–271
Micro-op cache, 325
Micro-operation, 300

Microarchitecture,
ARM, 329–324
ATmega168, 334–336
Core i7, 323–329
Mic-1, 267–282
Mic-2, 291–293
Mic-3, 293–300
Mic-4, 300–303
OMAP4430, 329–334
three-bus, 287
Microarchitecture level, 6, 243–338
design, 283–303
examples, 323–336
Microarchitecture of the Core i7 CPU, 323
Microcode, 11–13
Microcontroller, 33–35
Microdrive, 137
Microinstruction, 62, 251–258
Microinstruction register, 255
Microprogram, 6, 695
Microprogram counter, 255
Microprogramming, 9, 12–13
Microsoft assembler, 520
Microstep, 297
Milestones in computer architecture, 13–28
MIMD (*see* Multiple Instruction stream Multiple Data stream computer)
Minislot, 134
MIPS, 14
MIPS computer, 62
MIR (*see* MicroInstruction Register)
Miss ratio, 84
MMU (*see* Memory Management Unit)
MMX (*see* MultiMedia eXtensions)
Mnemonic, 692, 715
Modem, 127–129
Modulation, 128
Modulator demodulator, 128
Monadic operations, 388–390
Monitor, 115–117
Moore, Gordon, 28
Moore's law, 28
MOS (*see* Metal Oxide Semiconductor)
Motherboard, 108
Motif, 485
Motion Picture Experts Group, 571
Motorola 68000, 61
Mouse, 118–120
MPC (*see* MicroProgram Counter)
MPEG-2, 571
MPI (*see* Message Passing Interface)
MPP (*see* Massively Parallel Processor)
MS-DOS, 25
Multicomputer, 73, 587–591, 616–652, 621–631
BlueGene, 622–626
Google cluster, 632–636
MPP, 593
Red Storm, 626–631
Multicomputer performance, 650–652
Multicomputer software, 636–639, 636–646
MULTICS (*see* MULTiplexed Information and Computing Service)
Multilevel machine, 5–13
Multilevel machines, evolution, 8–13
Multimedia extensions, 41
Multiple instruction stream multiple data stream computer, 591–592
Multiplexed bus, 191
Multiplexed information and computing service, 454–455, 545–547
Multiplexer, 159–161
Multiprocessor, 72–73, 586–616
COMA, 614–616
Core i7, 569
heterogeneous, 570–574
NUMA, 606–610
symmetric, 587, 598–606
vs. multicomputer, 586–593
Multiprogramming, 22
Multisession CD-ROM, 105
Multistage switching network, 604–606
Multitouch screen, 114
Mutex, 505
Mutual capacitance, 114
Myhrvold, Nathan, 29

N

N-way set associative cache, 308–310
NaN (*see* Not a Number)
Nathan's law, 29
NC-NUMA (*see* No Cache NUMA)
Near call, 710
Near jump, 8088, 709
Negated signal, 178
Negative binary numbers, 675–677
Negative logic, 157

NEON, 331
Network,
  Ethernet, 575
  local-area, 575
  ring, 569
  store-and-forward, 575
  wide-area, 575
Network interface device, 131
Network of workstations, 593
Network processor, 574–582, 577–579, 578
Networking, introduction, 575–577
Newton, 14, 45
Nibble, 399
NID (*see* Network Interface Device)
No cache NUMA, 606
No remote memory access computer, 593
Nonblocking message passing, 637
Nonblocking network, 604–606
Noninverting buffer, 177
Nonnumeric data type, 359–360
Nonuniform memory access, 606–610
Nonuniform memory access computer, 592
Nonvolatile memory, 182–183
NORMA (*see* NO Remote Memory Access computer)
Normalized floating-point number, 684
Normative section, in standard, 346
Not a number, 688
Notation, Mic-1, 267–271
NOW (*see* Network Of Workstations)
Noyce, Robert, 21
NT File System, 498
NTFS (*see* NT File System)
NTOS executive, 487
NUMA (*see* NonUniform Memory Access computer)
NUMA multiprocessor, 606–610
Numeric data type, 358–359
Nvidia Fermi GPU, 582–585
Nvidia Tegra, 46

O

Object file, 716
Object module, 541–542
Object program, 518
OCP-IP (*see* Open Core Protocol-International Partnership)
Octal, 671
Octal number, 718
Octet, 75, 347
OGSA (*see* Open Grid Services Architecture)
OHCI (*see* Open Host Controller Interface)
OLED (*see* Organic Light Emitting Diode)
Olsen, Kenneth, 19
OMAP4430,
  addressing, 384
  bi-endianness, 354
  data cache, 331
  data types, 361
  instruction formats, 368–370
  instruction issue unit, 330
  instructions, 400–402
  internal organization, 209
  introduction, 45–47
  ISA level, 354–356
  microarchitecture, 329–334
  pinout, 211
  pipeline, 331–334
  store buffer, 331
  virtual memory, 460–462
Omega network, 604–606
Omnibus, PDP-8, 20
On-chip multithreading, 562–568
On-chip paralellism, 554–574
One's complement number, 675
Opcode, 244
Open collector, 189
Open core protocol-international partnership, 574
Open grid services architecture, 654
Open host controller interface, 231
Operand stack, 259–260
Operating system, 437
  CP/M, 24
  history, 9–11
  OS/2, 25
  timesharing, 11
  UNIX, 482–485, 488–489, 492–498, 503–506
  Windows, 490–492
  Windows 7, 485–488, 498–503, 506–509
Operating-system machine level, 7, 437–510
Operating-system macro, 11
Operation code, 244
Optical discs, 99–108
Orange book, 105
Orca, 644–646

Organic light emitting diode, 117
OS/2, 25
Osborne-1, 14
Out-of-order execution, 315–320
Overlay, 439

P

Packet, 575, 618
  PCI, 225
Packet classification, 580
Packet processing, 580–581
Packet processing engine, 579
Packet switching, 575
Page, 440
Page directory, 457
Page fault, 443
Page frame, 442
Page scanner, 607
Page table, 440
Page-replacement policy, 446–448
Paging, 439–452
  implementation, 441–443
Paging algorithm, 446–448
  FIFO, 447
  LRU, 446
Palm PDA, 27
Parallel computer,
  coprocessor, 574–586
  grid, 652–655
  multicomputer, 616–652
  multiprocessor, 586–61
  on-chip parallelism, 554–574
Parallel computers,
  performance, 646–652
  taxonomy, 591–593
Parallel input/output, 232
Parallel processing, 471–480
Parallel virtual machine, 637
Parallelism, instruction-level, 65–69, 555–561
  on-chip, 554–574
Parameter, macro, 526–537
Parent process, 503
Parity bit, 79
Parity flag, 708
Partial address decoding, 235
Partitioned resource sharing, 566
Pascal, Blaise, 13
Pass one, assembler, 530–534
Pass two, assembler, 534–536
Passive matrix display, 117
Path, 494
Path length, 283–291
  reducing, 285–291
Path selection, 580
Payload, PCI packet, 225
PC (*see* Program counter)
PCI Bus (*see* Peripheral Component Interconnect Bus)
PCI Express, 223–228
  architecture, 224–225
PCI Express bus, 111–112
PCI Express. protocol stack, 225–228
PCIe (*see* PCI Express)
PCIe bus (*see* PCI Express bus)
PDA (*see* Personal Digital Assistant)
PDP-1, 14, 20
PDP-11, 14
PDP-8, 14, 20
Pentium, 25, 41
Pentium 4, 44
Perfect shuffle, 605
Performance,
  hardware metrics, 647–648
  improving, 650–652
  improving network, 582
  software metrics, 648–650
Performance metrics, 647–650
  hardware, 647–648
  software, 648–650
Performance of parallel computers, 646–652
Peripheral bus, 573
Peripheral component interconnect bus,
    111–112, 215–218
  arbitration, 219–220
  operation, 218–219
  signals, 220–222
  transactions, 222–223
Perpendicular recording, 88
Personal computer, 23–25, 36
Personal digital assistant, 27
Pervasive computing, 28
PGA (*see* Pin Grid Array)
Phase modulation, 128
Physical address space, 440
Physical layer, PCI Express, 226
Pigment-based ink, 126
Pin grid array, 158

Pinout, 185
PIO (*see* Parallel Input/Output)
Pipe, 504
Pipeline, 65–67, 293
  Mic-3, 298–300
  Mic-4, 300–303
  OMAP, 331–334
  Pentium, 68
  seven-stage, 300–303
Pipeline stall, 298
Pipelining, Core i7 memory, 206–208
Pit, 100
Pixel, 117
Plain old telephone service, 130
PlayStation 3, 35
Pointer, 373
Pointer register, 698–699
Poison bit, 322
Polish notation, 376–379
Portable Operating System-IX, 483
Position independent code, 545
Positive logic, 157
POSIX (*see* Portable Operating System-IX)
Postfix, 376
POTS (*see* Plain Old Telephone Service)
Power gating, 209
PPE (*see* Packet Processing Engine)
Preamble, 87
Precise interrupt, 318
Predicated instruction, 370
Predication, 426–428
Prefetch buffer, 65
Prefetching, 651
Prefix, 400
Prefix byte, 278, 367
Present/absent bit, 442
Print engine, 123
Printer, 122–127, 124–127
  bubblejet, 126
  CMYK, 125
  dye-sublimination, 127
  inkjet. 125–126
  laser, 122–124
  solid-ink, 126
  specialty, 126–127
  wax, 127
Procedure, 392, 406–410
Procedure call instructions, 392–393
Procedure epilog, 409
Procedure prolog, 409
Process creation, 473
Process management, 503, 506–509
  UNIX, 503–506
  Windows 7, 506–509
Process synchronization, 478–480
Processor, 55–73
Processor bandwidth, 67
Processor bus, 572
Processor consistency, 595–596
Processor cycle, 695
Processor-level parallelism, 69–73
Producer, 474
Program, 1
Program counter, 56, 694
Program status word, 350, 459
Programmable interconnect, 183
Programmable processing engine, 579
Programmable ROM, 182
Programmed I/O, 394
PROM (*see* Programmable ROM)
Protocol, 576
  Bus, 188
  IP, 576
  MESI, 601–603
  PCI Express, 225–228
Protocol processing engine, 579
Protocol stack, PCI Express, 225–228
Pseudoinstruction, 522–524, 692, 718
PSW (*see* Program Status Word)
Pthreads, 505
Public-key cryptography, 585
PVM (*see* Parallel Virtual Machine)

Q

Queue management, 581
Queueing unit, 300

R

Race condition, 473–478
Radio frequency identification, 31–33
Radix, 671
Radix number systems, 671–673

RAID (*see* Redundant Array of Inexpensive Disks)
RAM (*see* Random Access Memory)
Random access memory, 73–85, 180–183
  DDR, 181
  dynamic RAM, 181
  SDRAM, 181
Ranging, 134
RAW dependence, 297
Read only memory, 182
Read/write pointer, 714
Real mode, 352
Recursion, 392
Recursive procedure, 406
Red book, 100
Red Storm, 626–631
Red Storm vs. BlueGene/P, 629–631
Reduced instruction set computer, 62
  design principles, 63–65
  vs. CISC
Redundant array of inexpensive disks, 94–97
Reed-Solomon code, 87
Refresh, memory, 181
Register, 5, 174, 694
  PSW, 350
Register addressing, 372
Register displacement, 703
Register indirect addressing, 373–374, 702
Register renaming, 315–320
Register with index, 703
Register with index and displacement, 704
Registers, 349–351
  flags, 350
Relative error, 683
Relative path, 494
Release consistency, 597–598
Relocation constant, 541
Relocation problem, 539
Reorder buffer, Core i7, 327
Replicated worker model, 644
Reserved page, 490
Resource layer, 653
Retirement unit, Core i7, 328
Reverse Polish notation, 376–379
RFID (*see* Radio Frequency IDentification)
Right-justified alignment, 387
Ring network, 569, 620
RISC (*see* Reduced Instruction Set Computer)
RISC design principles, 63–65
RISC vs. CISC, 62–63
ROB (*see* ReOrder Buffer)
ROM (*see* Read Only Memory)
Root directory, 494
Root hub, 229
Rotational latency, 89
Rounding, 683
Route lookup, 581
Router, 575

S

Sandy Bridge, 323–329
Saturated arithmetic, 559
Scalable, 650
Scalable design, 589
Scale, index, base, byte 368, 383
Scheduler, Core i7, 327
Scheduling, multicomputer, 639–640
Scoreboard, 316
SCSI (*see* Small Computer System Interface)
SCSI disk, 92
SDRAM (*see* Synchronous DRAM)
Seastar, 627
Second pass, assembler, 716
Second-generation computers, 19–21
Secondary memory, 86–108
Sector, 87
Security descriptor, 501
Security ID, 501
Seek, disk, 89
Segment, 450, 699
  8088, 700
Segment override, 715
Segment register group, 699
Segmentation, 449–455
  implementation, 452–455
Self-modifying, 374
Semantics, memory, 593–598
Semaphore, 478–480
Sequencer, 253
Sequential consistency, 594–595
Sequential flow of control, 405–406
Serial ATA, 92
Server, 36
Session, CD-ROM, 105
Set-associative cache, 308–310
Shard, 632
Shared library, 549

Shared-memory, application-level, 640–646
Shell, 485
Shifter, 163–164
Shockley, William, 19
SIB (*see* Scale, Index, Base byte)
SID (*see* Security ID)
Sign extension, 250
Signed magnitude number, 675
Significand, 686
SIMD (*see* Single Instruction stream Multiple Data steam)
SIMM (*see* Single Inline Memory Modules)
Simon smartphone, 27
Simple COMA, 615
Simplex line, 129
Simultaneous multithreading, 564
Single inline memory module, 85
Single instruction stream multiple data stream, 70, 583
Single large expensive disk, 94
Single-chip multiprocessors, 568–574
Skew, bus, 112
Slave, 189
  bus, 189
SLED (*see* Single Large Expensive Disk)
SM (*see* Streaming Multiprocessor)
Small computer system interface, 92–94
Small memory model, 720
Small outline DIMM, 85
Smartphone, 27
SMP (*see* Symmetric MultiProcessor)
Snoop, 202
Snooping cache, 599, 599–606
Snoopy cache, 599
SO-DIMM (*see* Small Outline DIMM)
SoC, 208
Socket, 483
Software, 8
Software layer, 227
Software metric, 648
Solid-ink printer, 126
Solid-state disk, 97–99
Source index, 698
Source language, 517
Source operand, 701
SPARC, 14
Spatial locality, 305
Specialty printer, 126–127
Speculative execution, 320–323, 321
Speculative load, 429
Speed vs cost, 283–285
Split cache, 84, 305
Splitter, 131
SR latch, 169–170
SRAM (*see* Static RAM)
SSD (*see* Solid-State Disk)
SSE (*see* Streaming SIMD Extensions)
Stack, 258–260
  operand, 259
Stack addressing, 376–379
Stack frame, 698
Stack pointer, 698
Stage, pipeline, 65
Stale data, 599
Stall, 311
Stalling, 298
Standard error, 494
Standard input, 494
Standard output, 494
Star, 620
State, 244
  finite state machine, 290
Statement, assembly language, 520–521
Static branch prediction, 315
Static RAM, 181
Stibbitz, George, 16
Storage, 73
Store, 73
Store buffer, OMAP4430, 331
Store-and-forward network, 575
Store-to-load forwarding, 329
Stream, 484
Streaming multiprocessor, 582
Streaming SIMD extensions, 42
Strict consistency, 594
Striping, 95
Strobe, 171
Structured computer organization, 2–13
Subroutine, 392, 710
Sun Fire E25K, 610–614
Sun SPARC, 14
Supercomputer, 21
Supercomputers, 39
Superscalar architecture, 67–69, 68–69
Superuser, 496
Supervisor call, 11
Switching algebra, 150
Switching network, multistage, 604–606
Symbol table, 530, 535–536, 716
Symbolic name, 692
Symmetric key cryptography, 585

Symmetric multiprocessor, 587, 598–606
Synchronous bus, 191–194
Synchronous DRAM, 181
Synchronous memory interface, 208
Synchronous message passing, 636
System bus, 188
System call, 11, 437, 712–715
System-on-a-chip, 208
Systems programmer, 7

T

Target, 218
Target language, 517
Target library, 549
Task bag, 644
TAT-12/13 cable, 30
Taxonomy of parallel computers, 591–593
TCP (*see* Transmission Control Protocol)
TCP header, 576
Tegra, Nvidia, 46
Telco, 129
Telecommunications equipment, 127–135
Telephone company, 129
Template, 643
Temporal locality, 306
Terminal, 113–118
Texas Instruments OMAP4430 (*see* OMAP4430)
Text section, 717
TFT display, 117
Thermal printer, 127
Thermal throttling, 206
Thin film transistor, 117
Third-generation computers, 21–23
Thrashing, 448
Thread, 505
Three-bus architecture, 287
Threshold sharing, 567
TI OMAP4430 (*see* OMAP4430)
Tightly coupled system, 73, 554
Timesharing system, 11
Tiny memory model, 720
TLB (*see* Translation Lookaside Buffer)
TLB miss, 461
TN (*see* Twisted Nematic display)
Token, 572
Topology, 618
Touch screen, 113–115
Towers of Hanoi, 417–420
Towers of Hanoi for ARM, 418–420
Towers of Hanoi for Core i7, 418–419
Tracer, 693, 721–725
Transaction layer, PCI Express, 227
Transistor, invention, 19
Transistor-transistor logic, 150
Transition, 290
Translation, 2
Translation lookaside buffer, 331, 461
Translator, 517
Transmission control protocol, 576
Transparent interrupt, 415
Trap, 413, 706
Trap handler, 413
Tree, 620
Tri-state device, 177
TriMedia processor, 557–561
Triple indirect block, 498
True dependence, 297
Truth table, 150
TTL (*see* Transistor-Transistor Logic)
Tuple, 642
Tuple space, 642
Twin, 642
Twisted nematic display, 116
Two's complement number, 675
Two-pass assembler, 529–536
Two-pass translator, 529
TX-0, 19
TX-2, 19

U

U pipeline, 68
UART (*see* Universal Asynchronous Receiver Transmitter)
Ubiquitous computing, 28
UCS (*see* Universal Character Set)
UCS transformation format, 141
UHCI (*see* Universal Host Controller Interface)
UMA (*see* Uniform Memory Access computer)
UMA (*see* Uniform Memory Access computer)
Unicode, 138–141
Unified cache, 84
Uniform memory access computer, 592

Universal asynchronous receiver transmitter, 232
Universal character set, 141
Universal host controller interface, 231
Universal serial bus, 228–232
  USB 2.0, 231
  USB 3.0, 231
Universal synchronous asynchronous receiver transmitter, 232
UNIX, 482–485
  virtual memory, 488–489
UNIX I/O, 492–498
UNIX process management, 503–506
Update strategy, cache, 600
USART (*see* Universal Synchronous Asynchronous Receiver Transmitter)
USB 2.0, 231
USB 3.0, 231
User mode, 347
UTF-8, 141–142

V

V pipeline, 68
Vacuum tube computers, 16–19
Vampire tap, 575
VAX, 14, 61
VCI (*see* Virtual Component Interconnect)
Vector processor, 71
Vector register, 71
Very large scale integration, 23
Very long instruction word, 555
VFP, 331
Video RAM, 117–118
Virtual 8086 mode, 352
Virtual address space, 440
Virtual circuit, 227
Virtual component interconnect, 574
Virtual cut through network, 625
Virtual machine, 3, 463–464
Virtual memory, 438–462
  ARM, 460–462
  compared to caching, 462
  Core i7, 455–460
  UNIX, 488–489
  Windows 7, 490–492
Virtual organization, 652
Virtual register, 258
Virtual topology, 638
Virtual-machine control structure, 464
Virtualization, 464
  hardware, 463–464
Virtuous circle, 29
VLIW (*see* Very Long Instruction Word)
VLSI (*see* Very Large Scale Integration)
VMCS (*see* Virtual-Machine Control Structure)
Volume table of contents, 105
Von Neumann, 18–19
Von Neumann machine, 18–19
VTOC (*see* Volume Table Of Contents)

W

Wait state, 192
WAN (*see* Wide Area Network)
WAR dependence, 318
Watson, Thomas, 19
WAW dependence, 318
Wax printer, 127
Weak consistency, 596–597
Wear leveling, 99
Weiser, Mark, 28
Weizac, 17
Whirlwind I, 14
Wide-area network, 575
Wiimote controller, 120–122
Wilkes, Maurice, 60
Win32 API, 487
Winchester disk, 88
Windows 7, 485–488
Windows 7 I/O, 498–503
Windows 7 process management, 506–509
Windows 7 virtual memory, 490–492
Windows drivers, 487
Windows new technology, 486
Windows NT (*see* Windows New Technology)
Wired-OR, 189
Word, 75
Word instruction,, 701
Word register, 701
Working directory, 494
Working-set model, 443–446
Wozniak, Steve, 24
Write allocation, 310
Write back cache, 310

Write deferred cache, 310
Write through cache, 310, 599
Write-allocate policy, 600
Write-back protocol, 601
Write-once protocol, 601

X

X Windows, 485
x86, 25, 39–45, 347
x86 architecture, 39
XC2064, 14
Xeon, 43

Y

Y2K problem, 39, 359
Yellow book, 101

Z

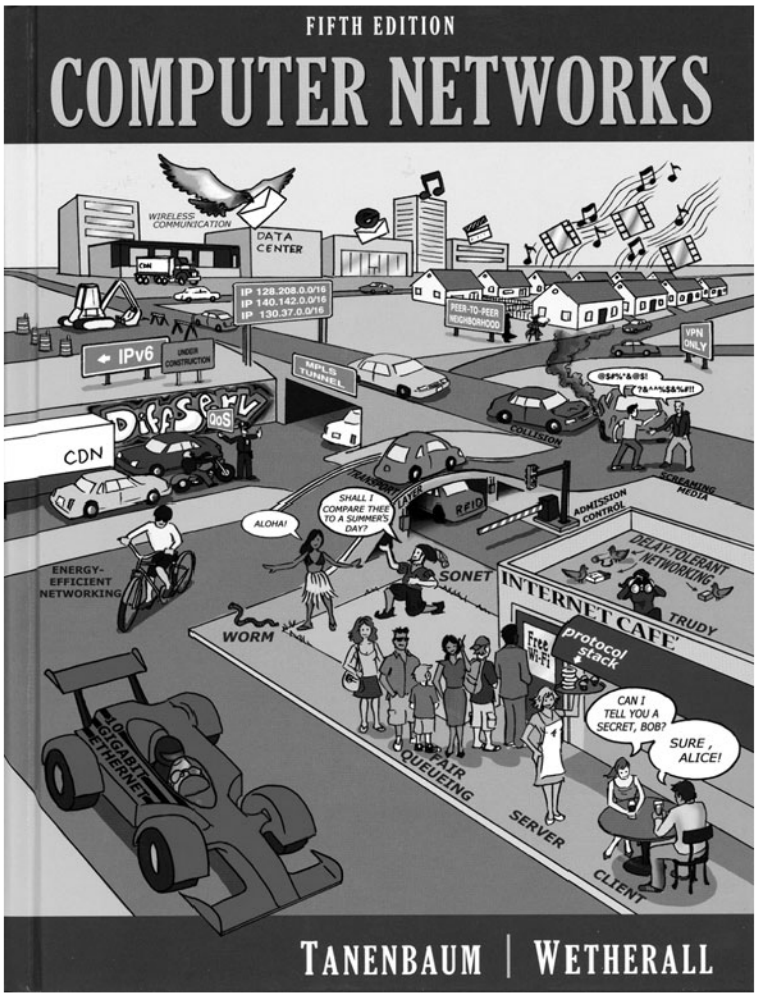
Z1, 14
Zeroth-generation computers, 13–16
Zilog Z8000, 61
Zuse, Konrad, 15–16
Zuse Z1, 14

This page intentionally left blank

Also by Andrew S. Tanenbaum and David J. Wetherall

Computer Networks, 5th ed.

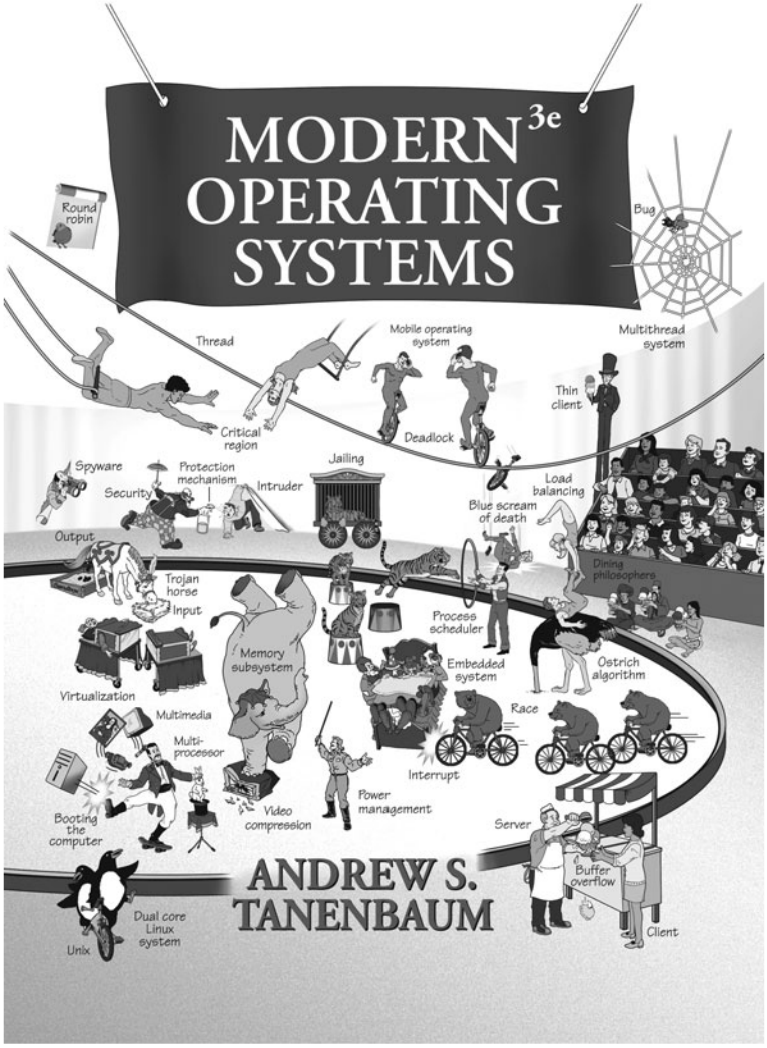
This widely read classic, with a fifth edition co-authored with David Wetherall, provides the ideal introduction to today's and tomorrow's networks. It explains in detail how modern networks are structured. Starting with the physical layer and working up to the application layer, the book covers a vast number of important topics, including wireless communication, fiber optics, data link protocols, Ethernet, routing algorithms, network performance, security, DNS, electronic mail, the World Wide Web, and multimedia. The book has especially thorough coverage of TCP/IP and the Internet.



Also by Andrew S. Tanenbaum

Modern Operating Systems, 3rd ed.

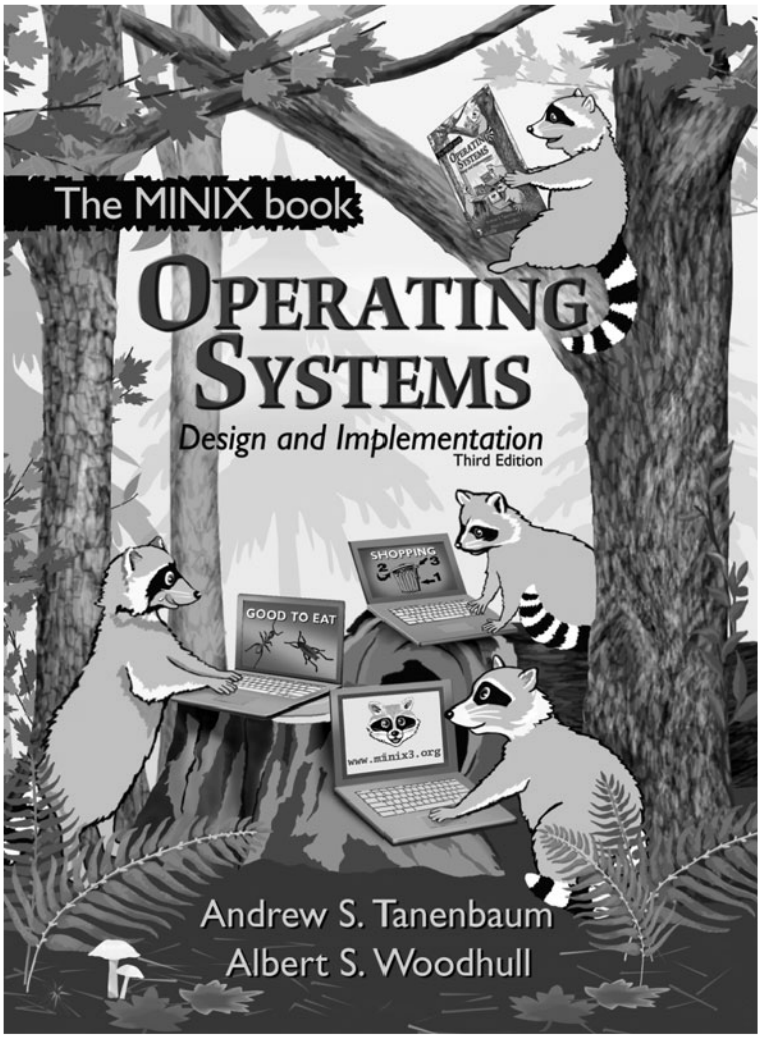
This worldwide best-seller incorporates the latest developments in operating systems. The book starts with chapters on the principles, including processes, memory management, file systems, I/O, and so on. Then it goes into three chapter-long case studies: Linux, Windows, and Symbian. Tanenbaum's experience as the designer of three operating systems (Amoeba, Globe, and MINIX) gives him a background few other authors can match, so the final chapter distills his long experience into advice for operating system designers.



Also by Andrew S. Tanenbaum and Albert S. Woodhull

Operating Systems: Design and Implementation, 3rd ed.

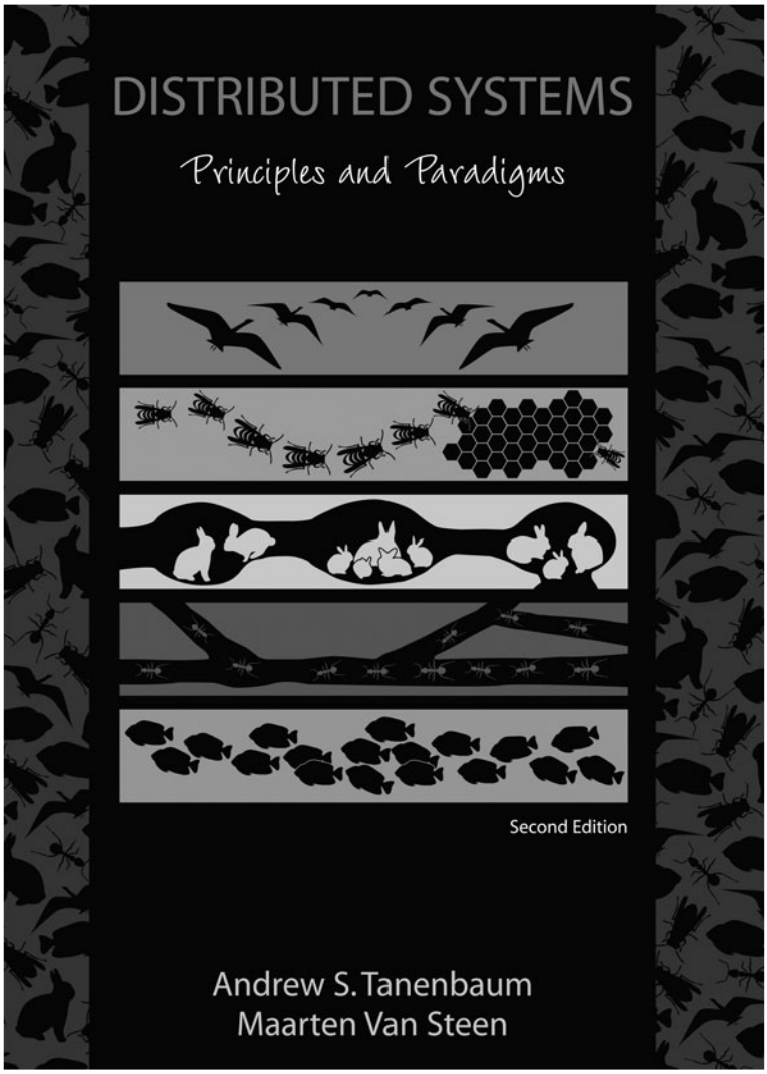
All other textbooks on operating systems are long on theory and short on practice. This one is different. In addition to the usual material on processes, memory management, file systems, I/O, and so on, it contains a CD-ROM with the source code (in C) of a small, but complete, POSIX-conformant operating system called MINIX 3 (see *www.minix3.org*). All the principles are illustrated by showing how they apply to MINIX 3. The reader can also compile, test, and experiment with MINIX 3, leading to in-depth knowledge of how an operating system really works.



Also by Andrew S. Tanenbaum and Maarten van Steen

Distributed Systems: Principles and Paradigms, 2nd ed.

Distributed systems are becoming ever-more important in the world and this book explains their principles and illustrates them with numerous examples. Among the topics covered are architectures, processes, communication, naming, synchronization, consistency, fault tolerance, and security. Examples are taken from distributed object-based, file, Web-based, and coordination-based systems.



CREDITS

ABOUT THE AUTHORS

Andrew S. Tanenbaum has an S.B. degree from M.I.T. and a Ph.D. from the University of California at Berkeley. He is currently a Professor of Computer Science at the Vrije Universiteit where he has taught operating systems, networks, and related topics for over 30 years. His current research is on highly reliable operating systems although he has worked on compilers, distributed systems, security, and other topics over the years. These research projects have led to over 150 refereed papers in journals and conferences.

Prof. Tanenbaum has also (co)authored five books which have now appeared in 19 editions. The books have been translated into 21 languages, ranging from Basque to Thai and are used at universities all over the world. In all, there are 159 versions (language/edition combinations). See at *www.cs.vu.nl/~ast/book_covers for images of them.*

Prof. Tanenbaum has also produced a considerable volume of software, including the Amsterdam Compiler Kit (a retargetable portable compiler), Amoeba (an early distributed system used on LANs), and Globe (a wide-area distributed system).

He is also the author of MINIX, a small UNIX clone initially intended for use in student programming labs. It was the direct inspiration for Linux and the platform on which Linux was initially developed. The current version of MINIX, called MINIX 3, is now focused on being an extremely reliable and secure operating system. Prof. Tanenbaum will consider his work done when no computer is equipped with a reset button and no living person has ever experienced a system crash. MINIX 3 is an on-going open-source project to which you are invited to contribute. Go to *www.minix3.org* to download a free copy.

Tanenbaum is a Fellow of the ACM, a Fellow of the the IEEE, and a member of the Royal Netherlands Academy of Arts and Sciences. He has also won numerous scientific prizes, including:

- 2010 TAA McGuffey Award for Computer Science and Engineering books
- 2007 IEEE James H. Mulligan, Jr. Education Medal
- 2002 TAA Texty Award for Computer Science and Engineering books
- 1997 SIGCSE Award for Outstanding Contributions to Computer Science Education
- 1994 ACM Karl V. Karlstrom Outstanding Educator Award

He also holds two honorary doctorates.

His home page on the World Wide Web can be found at *http://www.cs.vu.nl/~ast/.*

Todd Austin is a Professor of Electrical Engineering and Computer Science at the University of Michigan in Ann Arbor. His research interests include computer architecture, reliable and secure system design, hardware and software verification, and performance analysis tools and techniques. Prior to joining academia, Prof. Austin was a Senior Computer Architect in Intel's Microcomputer Research Labs, a product-oriented research laboratory in Hillsboro, Oregon. Prof. Austin is the first to take credit (but the last to accept blame) for creating the SimpleScalar Tool Set, a popular collection of computer architecture performance analysis tools. In 2002, Prof. Austin was a Sloan Research Fellow, and in 2007 he received the ACM Maurice Wilkes Award for "innovative contributions in Computer Architecture including the SimpleScalar Toolkit and the DIVA and Razor architectures." Austin received his Ph.D. in Computer Science from the University of Wisconsin in 1996.