Figure A.1.1 The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

Figure A.1.2 MIPS machine language code for a routine to compute and print the sum of the squares of integers between 0 and 100.

Figure A.1.3 The same routine written in assembly language. However, the code for the routine does not label registers or memory locations nor include comments.

Figure A.1.4 The same routine written in assembly language with labels, but no comments. The commands that start with periods are assembler directives (see pages A-47–49). .text indicates that succeeding lines contain instructions. .data indicates that they contain data. .align n indicates that the items on the succeeding lines should be aligned on a 2n byte boundary. Hence, .align 2 means the next item should be on a word boundary. .globl main declares that main is a global symbol that should be visible to code stored in other files. Finally, .asciiz stores a null-terminated string in memory

Figure A.1.5 The routine written in the C programming language.

Figure A.1.6 Assembly language either is written by a programmer or is the output of a compiler.

Figure A.2.1 Object file. A UNIX assembler produces an object file with six distinct sections.

Figure A.3.1 The linker searches a collection of object files and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files.

Figure A.5.1 Layout of memory.

Figure A.6.1 MIPS registers and usage convention.

Figure A.6.2 Layout of a stack frame. The frame pointer ($fp) points to the first word in the currently executing procedure’s stack frame. The stack pointer ($sp) points to the last word of the frame. The first four arguments are passed in registers, so the fifth argument is the first one stored on the stack.

Figure A.6.3 Stack frames during the call of fact(7).

Figure A.7.1 The Status register.

Figure A.7.2 The Cause register.

Figure A.8.1 The terminal is controlled by four device registers, each of which appears as a memory location at the given address. Only a few bits of these registers are actually used. The others always read as 0s and are ignored on writes.

Figure A.9.1 System services.

Figure A.10.1 MIPS R2000 CPU and FPU.

Figure A.10.2 MIPS opcode map. The values of each field are shown to its left. The first column shows the values in base 10, and the second shows base 16 for the op field (bits 31 to 26) in the third column. This op field completely specifies the MIPS operation except for six op values: 0, 1, 16, 17, 18, and 19. These operations are determined by other fields, identified by pointers. The last field (funct) uses “f” to mean “s” if rs = 16 and op = 17 or “d” if rs = 17 and op = 17. The second field (rs) uses “z” to mean “0”, “1”, “2”, or “3” if op = 16, 17, 18, or 19, respectively. If rs = 16, the operation is specified elsewhere: if z = 0, the operations are specified in the fourth field (bits 4 to 0); if z = 1, then the operations are in the last field with f = s. If rs = 17 and z = 1, then the operations are in the last field with f = d.