Solutions Manual

Vincent P. Heuring

Computer Systems Design and Architecture

SECOND EDITION

Vincent P. Heuring · Harry F. Jordan

University of Colorado, Boulder

Preparation assisted by Mersedeh Tehranian

University of Colorado, Boulder

PEARSON Prentice Hall

Upper Saddle River, New Jersey 07458

Associate Editor: *Alice Dworkin*
Executive Managing Editor: *Vince O'Brien*
Managing Editor: *David A. George*
Production Editor: *Barbara Till*
Supplement Cover Manager: *Daniel Sandin*
Manufacturing Buyer: *Ilene Kahn*

Pearson Prentice Hall
Pearson Education, Inc.
Upper Saddle River, NJ 07458

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-147327-1

Pearson Education Ltd., *London*
Pearson Education Australia Pty. Ltd., *Sydney*
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd., *Hong Kong*
Pearson Education Canada, Inc., *Toronto*
Pearson Educación de Mexico, S.A. de C.V.
Pearson Education—Japan, *Tokyo*
Pearson Education Malaysia, Pte. Ltd.
Pearson Education, Inc., *Upper Saddle River, New Jersey*

Contents

Chapter 1
The General Purpose Machine

1.1 The PowerPC 601 processor addresses a maximum of 2^{32} bytes of memory. What is the maximum number of 64-bit words that can be stored in this memory? **(§1.1)**

Solution: A 64-bit word = 8 bytes =2^3 bytes. 2^{32} bytes of memory is equivalent to 2^{29} 64-bit words. In decimal this is about a half billion words.

1.2 A certain IBM 970 processor has a system clock frequency of 1.2 GHz, what is the clock period? **(§1.1)**

Solution: 1.2GHz is 1.2×10^9 clock cycles per second, so the period is 8.3×10^{-10}s, or 830 ps.

1.3 If the cost of RAM is $95 for a 4MB module, what will it cost to install 32 M words in an originally empty machine if the word size is 32 bits? **(§1.1)**

Solution: With a word size of 32 bits, one word = 4 bytes. So 32MW = 32M × 4B = 128MB. At $95 for 4MB, this gives 128MB/memory × $95/4MB = $3,040/memory.

1.4 How many 500MB tapes will be required to back up a 120GB hard drive? How long will the backup process require if one tape can be filled in 5 minutes? (No coffee breaks allowed.) **(§1.1)**

Solution: There are 120×2^{30} B/disk and 500×2^{20} B/tape. Thus one needs about $120 \times 2^{30}/500 \times 2^{20}$ tapes/disk, or 240 tapes. 240 tapes × 5minutes/tape = 1200 minutes = 20 hours.

1.5 a. A certain machine requires 1.5 μs to process each 64-byte data record in a database. How long will it take to process a database containing 100×10^8 records?

b. How many 700MB-capacity CD-ROMs will be required to store the database? **(§1.1)**

Solution: a. 1.5 μs/record × 100×10^8records/database = 15000 seconds/database = 250 minutes.

b. If a record takes 16 bytes, then there are 16×10^{10}bytes/database. At 700×10^6 B/CD-ROM, this means about 229 CD-ROMs/database. Unreasonable, even for small records!

1.6 a. What is the percentage relative error in using 2^{10} as an approximation for 10^3?

b. What is the percentage relative error in using 2^{30} as an approximation for 10^9?

c. What is the general formula for the relative error in using 2^{10k} as an approximation for

10^{3k}? **(§1.1)**

Solution: a. The relative error in approximating 1000 is $\frac{1024-1000}{1000} = 2.4\%$.

b. Relative error in using 2^{30} for 10^9 is $\frac{2^{30}-10^9}{10^9} = 7.37\%$.

c. $\frac{2^{10k}-10^{3k}}{10^{3k}}$ is an answer, but it is uninteresting. Strictly dullsville! The interesting thing that can be seen from parts a) and b) is that the relative error increases. How can we get a formula that helps us understand when the value of k is too large for the approximation to be any good. One way is as follows:
Relative error is

$\frac{2^{10k}}{10^{3k}} - 1$, and since $\ln(\frac{2^{10k}}{10^{3k}}) = k(10\ln 2 - 3\ln 10) = 0.0237k$, the relative error is $e^{0.0237k} - 1$. The first two terms of a power series expansion give an idea of the behavior: relative error $\approx 0.0237k + 0.0003k^2$. From this we see that if $k > 20$, the relative error exceeds 50%—not a good approximation.

1.7 If one printed character can be stored in 1 byte, approximately how many bytes will be required to store the text of this textbook? Do not include the graphics, and do not count the characters one by one. Show your work and state your assumptions. **(§1.1)**

Solution: In manuscript form there are about 96 characters (including spaces and punctuation) in a full line of text. About 3/4 of a page is full lines on the average, and there are 570 pages in the textbook. A full page could contain 50 lines.

50 lines/page × 570 pages/book × 75% × 96 char/line ≈ 2,052,000 char ≈ 1.95 MB.

1.8 Consider computing the electric field in a box 1.5 cm on a side. The spatial resolution in each dimension is to be 50 μm. Assume that it takes 150 instructions for every point in the 3-D grid to do the calculation. How long does the computation take on a computer that can execute at a rate of 100 MFOPS (millions of floating point instructions per second)? **(§1.1)**

Solution: Each side of the box is 1.5×10^{-2} m long. The spatial resolution in each dimension is 50×10^{-6}m. Dividing the length of the box by the spatial resolution gives 300 points/side. Since the box is 3-dimensional, $300^3 = 27 \times 10^6$ points must be calculated. Each point needs 150 instructions, so 27×10^6 points/3-D grid × 150 instructions/point = 4.05×10^9 instructions/3-D grid. The processor can execute 100 MFOPS, so dividing 4.05×10^9 instructions/3-D grid by 100×10^6 instructions/s = 40.5 s.

1.9 Describe the tools used by the assembly language programmer. **(§1.3)**

Solution: The tools used by the assembly language programmer are editor, assembler, linker, loader, debugger, and development system. The *editor* is used to edit the source code (the assembly language). The *assembler* allows the programmer to generate machine language program from assembly language programs. It translates assembly language statements to their binary equivalents. The *linker* links separately assembled modules together into a single module suitable for loading and execution. The *loader* loads the executable binary code into the memory and changes some logical addresses to appropriate physical addresses. The *debugger* allows the programmer to observe the details of program execution. The *development system* is a collection of hardware and software that is used to support system development.

1.10 Describe the differences between assembly language and high-level languages as regards type checking. What is the source of these differences? **(§1.3)**

Solution: High-level languages usually have several primitive data types that are part of the language definition. In addition, most high-level languages provide some constructs to let the user define complex types by composition of the primitive types of the language. Correct type usage is usually enforced by the compiler during syntactic and semantic check phases. The rich data type structure built into higher level programming languages is missing from most assembly and machine languages. It's all "bits-n-bytes" at the machine level.

Assembly language programmers need to specify the addresses where the program and data should be located and to focus on the machine level implementation. High-level language programmers want the compiler to check the use of language constructs and to check some semantics of programs. High-level language programmers also need short representations of objects that are of interest to them, such as integers, reals, and characters.

1.11 What is the difference between a programmable calculator and a personal computer? **(§1.4)**

Solution: The programming capability of a programmable caculator is limited, and there's only one "language." It can only solve some particular kinds of numerical problems. The I/O capability is very poor.

A personal computer is a general purpose machine. Nearly all kinds of operating systems, programming languages, and application software can run on it. It can solve nearly all kinds of problems, such as mathematical computation, controlling, design, information processing, and artificial intelligence problems. It can drive all sorts of I/O devices—vastly more than a calculator. Different computers can also be connected via a network.

1.12 How would computers be operated if there were no stored programs? **(§1.4)**

Solution: If there were no stored programs, computers could only be operated by button pushing, switching, and rewiring. Furthermore, the operator would have to wait for the result of a step before he or she could enter the next step. It would be the operator's responsibility to determine the next step.

1.13 What is an ISA, and what are its components? **(§1.3)**

Solution: The ISA is the collection of instructions and resources. It includes the instruction set, the machine's memory, and all of the programmer-accessible registers in the CPU and elsewhere in the machine.

1.14 Using only the instructions in Table 1.3, compile by hand the following C statements into VAX11 assembly language. Assume all variables are integers. **(§1.3)**

a. `V = (W + X)*(Y + Z);`

b. `A = B*C*D + E;`

c. `z = x*y^2;`

d. `U = V; W = U + Y;`

Solution: T is a temporary memory location introduced to avoid overwriting operands.

a.
```
ADD W, X, T
ADD Y, Z, V
MPY T, V, V
```

b.
```
MPY B, C, T
MPY D, T, A
ADD E, A, A
```

c.
```
MPY y, y, z
MPY x, z, z
```

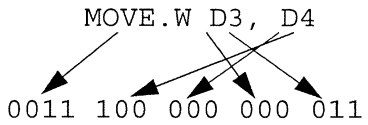
d.
```
MOV V, U
ADD U, Y, W
```

1.15 Using only the information in Table 1.2, encode the following MC68000 assembly language instructions into machine language. Express your results in binary. **(§1.3)**

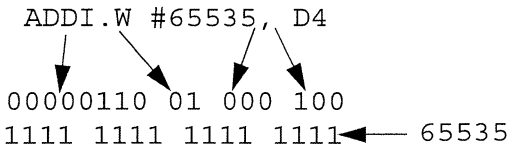
a. `MOVE.W D3, D4`

b. `ADDI.W #65535, D4`

Solution: a.



b.



1.16 Describe the advantages of data typing in high-level language programming. **(§1.3)**

Solution: Data typing helps prevent the programmer from making errors due to misuse of language constructs, and also provides guidance to the compiler about the meaning of the program. High-level language data types are designed for ease of representing objects

that are of interest to the programmer.

1.17 If assembly language is mostly free of data typing, how are data types expressed in assembly languages? **(§1.3)**

Solution: They are solely determined by the intent of the programmer, who performs "data typing" by the kind of instruction used.

1.18 Define the difference between a simulator and an emulator. Which would you prefer to work with, and why? **(§1.4)**

Solution: Simulators are software tools that mimic aspects of the system's behavior and allow a certain amount of performance estimation at an early part of the design process. Since simulators mimic hardware performance in software, they are usually slower in operation by orders of magnitude. Emulators can be thought of as hardware-assisted simulators that provide operation speed closer to the speed of the hardware being emulated.

Emulators are preferred because they provide results closer to the behavior of the real machine being emulated.

1.19 a. Define the term *bus*.

b. Why are buses used in computers?

c. Describe the similarities and differences between the computer bus and the electric power grid.

d. Describe the differences between the computer bus and the water system in your home. **(§1.4)**

Solution: a. A bus is a multiplexer interconnecting multiple devices and allowing multiple devices to use it for communication by time sharing. It provides both a data path and a signaling or control path.

b. It can save hardware and thereby reduce the cost. Using buses makes the design relatively simple, and it is not difficult to add more devices to the system.

c. They use a common medium. The electric power grid only delivers power. The computer bus transfers data and control signals. Once the connection is established, the power can flow to all terminals simultaneously. Different devices use the computer bus by time-sharing.

d. They use a common medium. The water system only delivers water. The computer bus transfers data and control signals. Once the connection is established, the water can flow to all terminals simultaneously. Different devices use the computer bus by time-sharing.

1.20 Provide a diagram similar to Figure 1.5 for the computer you work at most frequently. **(§1.5)**

Solution: The diagram should include CPU, memory, I/O, and their interconnection. The name of each bus should be specified. I/O devices may include keyboard, mouse, bit pad,

display, printer, disk drive, CD ROM, tape, network, other computers, and so on.

1.21 How does the computer architect communicate machine specifications to the logic designer? **(§1.5)**

Solution: Natural languages and some informal descriptions are inevitable. They provide a general view of the architecture and a more intuitive "feel" for the meaning of a construct. But they can be confusing, imprecise, or incomplete in their descriptive power. Formal description languages are often used to augment natural language in this communications process. These languages are called register transfer languages, as they are specifically designed to describe information transfer between registers, a feature that is central to the operation of the computer. Formal descriptions provide the means to be precise and exact.

1.22 What natural separation distinguishes computer logic design from classical logic design? **(§1.5)**

Solution: The computer designer does not design an entire digital computer using state machine design techniques. There is a natural separation or partitioning of concerns between the data path and the control path. Whereas the logic designer sees logic gates and flip-flops, the computer designer sees multiplexers, decoders, and register files.

1.23 Estimate the costs of the components in Figure 1.6 for the computer you work at most frequently. State where you got the component costs. **(§1.5)**

Solution: From *PC Magazine,* www.pcmag.com.

Component	**Price**
CPU: Intel Pentium 4 motherboard	$120
Cache: IBM 256 KB (94G3141) Cache Memory	$39
Main memory: 512MB 400MHz DDR PC3200 DIMM	$80
Disk memory: Western Digital 120GB (Ultra ATA/100, 7200 RPM)	$92
Tape memory: Exabyte 80/160 GB VXA-2 Tape Drive	$528

1.24 Describe as accurately as you can the implementation domain of the computer proposed by Charles Babbage. **(§1.5, 1.6)**

Solution: Gears, shafts, wheels, cranks, and dials.

1.25 Describe as accurately as you can the implementation domains of the first through the fourth generation of computers. **(§1.5, 1.6)**

Solution: The first generation: vacuum tubes, relays, and mercury delay lines. The second generation: discrete transistors. The third generation: small- and medium-scale integrated circuits, TTL chips. The fourth generation: VLSI on silicon, TTL chips, ECL chips, PLAs, and sea-of-gates gate arrays.

Chapter 2

Machines, Machine Languages, and Digital Logic

2.1 Refer to the four items on page 38 that an instruction must specify. What would need to be specified by the MC68000 instruction ADDI.W #5, D4 described in Table 1.2? **(§2.1)**

Solution: (1) opcode: add immediate (16- bit word. (2) operands: the immediate value 5 and the value in data register D4. (3) result: in data register D4. (4) next instruction: implicitly in the word following this instruction. (Notice this instruction consists of two words .)

2.2 Repeat Exercise 2.1 for the JCXZ instruction in Table 2.3. **(§2.1)**

Solution: (1) opcode: Jump if register CX is equal to zero. (2) operand: in register CX. (3) result: none. (4) next instruction: location of the Addr if the condition is true, otherwise implicitly the instruction after the current instruction (JCXZ).

2.3 According to the discussion in Section 2.2.1, how many instructions would the MC6800 have to execute to move a 128-bit floating point-number from one memory location to another? **(§2.1)**

Solution: If we assume that data can only be moved one byte at a time from memory to an accumulator or from an accumulator to memory, then 16 pairs of (load, store) instructions are needed to move a 128-bit word. If single instructions are available to load and store a 16-bit register, say IX, the number of instructions can be reduced from 32 to 16.

2.4 Write the code to implement the expression A = (B - C)*D on 3-, 2-, 1-, and 0-address machines. Do not rearrange the expression. In accordance with programming language practice, computing the expression should not change the values of its operands. **(§2.2)**

Solution:

3-address	**2-address**	**1-address**	**0-address**
`SUB A, B, C`	`LOAD A, B`	`LDA B`	`PUSH B`
`MPY A, A, D`	`SUB A, C`	`SUB C`	`PUSH C`
	`MPY A, D`	`MPY D`	`SUB`
		`STA A`	`PUSH D`
			`MPY`
			`POP A`

2.5 Compute the total memory traffic in bytes for both instruction fetch and instruction execution for the code that implements the expression evaluation the four machines in Exercise 2.4 above. Assume opcodes occupy one byte, addresses occupy two bytes, and data values also occupy two bytes. **(§2.2)**

Solution: **Solution:**

Machine	Instruction Fetch	Instruction Execution	Total memory traffic
3-address	7+7=14	6+6=12	14+12=26
2-address	5+5+5=15	4+6+6=16	15+16=31
1-address	3+3+3+3=12	2+2+2+2=8	12+8=20
0-address	3+3+1+3+1+3=14	2+2+2+2=8	14+8=22

2.6 Do problem 2.4 above, but for the expression A= B*C + D*E. (Feel free to use a temporary variable, called, say, T, if you feel you need one.) Assuming that addresses are 16 bits, data values are 16 bits, and opcodes are 8 bits, compute the size of your program, in bytes, and the amount of memory traffic the program would generate, in bytes, when it executes. When you compute the amount of memory traffic generated by the program, compute separately the amount of traffic due to instruction fetch and instruction execution. **(§2.2)**

Solution: T is a memory location used as a temporary.

3-address	**2-address**	**1-address**	**0-address**
`MPY A, B, C`	`LOAD A, B`	`LDA D`	`PUSH D`
`MPY T, D, E`	`MPY A, C`	`MPY E`	`PUSH E`
`ADD A, A, T`	`LOAD T, D`	`STA T`	`MPY`
	`MPY T, E`	`LDA B`	`PUSH C`
	`ADD A, T`	`MPY C`	`PUSH B`
		`ADD T`	`MPY`
		`STA A`	`ADD`
			`POP A`

Amount of traffics:

Machine	Instruction Fetch	Instruction Execution	Total memory traffic
3-address	7+7+7=21	6+6+6=18	21+18=39
2-address	5+5+5+5+5=25	4+6+4+6+6=26	25+26=51
1-address	$3 \times 7 = 21$	$2 \times 7 = 14$	21+14=35
0-address	$(3 \times 5) + (1 \times 3) = 18$	$2 \times 5 = 10$	18+10=28

The size of the program for each machine is as follows:
3-address: The program contains 3 instructions and each instruction takes $(2 \times 3) + 1 = 7$ bytes, therefore the size of the program in memory would be

$3 \times 7 = 21$ bytes.

2-address: The program contains 5 instructions and each instruction takes $(2 \times 2) + 1 = 5$ bytes, therefore the size of the program in memory would be $5 \times 5 = 25$ bytes.

1-address: The program contains 7 instructions and each instruction takes $(2 \times 1) + 1 = 3$ bytes, therefore the size of the program in memory would be $7 \times 3 = 21$ bytes.

0-address: The program contains 8 instructions, 5 of the instructions take $(2 \times 1) + 1 = 3$ bytes and 3 of them take only 1 byte, therefore the size of the program in memory would be $(5 \times 3) + (3 \times 1) = 18$ bytes.

2.7 Write SRC code to implement the expression in Exercise 2.4. Assume SRC has a multiply instruction. **(§2.3)**

Solution:

Assume that operands and results are stored in memory addresses that can be accessed with direct addressing. Also assume that SRC has a multiply instruction, `mpy`, that uses format 6.

```
ld  r0, B
ld  r1, C
sub r0, r0, r1
ld  r1, D
mpy r0, r0, r1
st  r0, A
```

2.8 Compute the total memory traffic in bytes for both instruction fetch and instruction execution for the code in Exercise 2.7. **(§2.3)**

Solution:

Instructions	Instruction Fetch	Instruction Execution	Total memory traffic
`ld r0, B`	4B	4B	4+4=8B
`ld r1, C`	4B	4B	4+4=8B
`sub r0, r0, r1`	4B	0	4B
`ld r1, D`	4B	4B	4+4=8B
`mpy r0, r0, r1`	4B	0	4B
`st r0, A`	4B	4B	4+4=8B
Total	**24B**	**16B**	**40B**

2.9 Repeat Exercise 2.6 for a general register machine. Assume 8-bit opcodes, 5-bit register numbers, 16 bits data words and 24-bit addresses. **(§2.2)**

Solution:

Assume that operands and results are stored in memory addresses that can be accessed with direct addressing.

```
load  R0, B
load  R1, C
mul R0, R0, R1
load  R1, D
load  R2, E
mul R1, R1, R2
add R0, R0, R1
store R0, A
```

The amount of traffics for this general register machine is as the follows:

Instructions	Instruction Fetch	Instruction Execution	Total memory traffic
`load R0, B`	8+5+24=27b=4B	16bits=2B	27+16=43b=6B
`load R1, C`	8+5+24=27b=4B	16bits=2B	27+16=43b=6B
`mul R0, R0, R1`	8+5+5+5=23b=3B	0	23b=3B
`load R1, D`	8+5+24=27b=4B	16bits=2B	27+16=43b=6B
`load R2, E`	8+5+24=27b=4B	16bits=2B	27+16=43b=6B
`mul R1, R1, R2`	8+5+5+5=23b=3B	0	23b=3B
`add R0, R0, R1`	8+5+5+5=23b=3B	0	23b=3B
`store R0, A`	8+5+24=27b=4B	16bits=2B	27+16=43b=6B
Total	**204b or 29B**	**80b =10B**	**284b or 39B**

Size of the program:

The program contains 8 instructions, 5 of the instructions take 8+5+24=27bits and 3 of them take 8+5+5+5=23bits, therefore the size of the program in memory would be $(27 \times 5) + (23 \times 3) =$ 204 bits or 29 bytes.

2.10 Suppose the instruction word in a general register machine has space for an opcode and either three register numbers or one register number and an address. What different instruction formats might be used for an ADD instruction, and how would they work? **(§2.2)**

Solution: Format 1: `ADD Rdst, Rsrc1, Rsrc2`

Fetch the contents of register `Rsrc1` and `Rsrc2`, add them, and then store the result into register `Rdst`.

Format 2: `ADD Reg, Mem-addr`

Fetch the contents from register `Reg` and memory address `Mem-addr`, add them, and then store the result to register `Reg`.

2.11 There are reasons for machine designers to want all instructions to be the same length. Why is this not a good idea in a stack machine? **(§2.2)**

Solution: In a stack machine, an arithmetic instruction only needs an opcode field, while a PUSH/POP instruction needs both opcode and a much longer address field. If all instructions were forced to be the same length, a considerable amount of memory space would be wasted in arithmetic instructions.

2.12 In the last two instructions of Table 2.3, which of the five items on page 44 are explicitly specified and which are implicit? **(§2.2)**

Solution:

	`SOB R4 LOOP`	`JCXZ Addr`
Operation to be performed	Explicit: `SOB`	Explicit: `JCXZ`
Location of first operand	Explicit: `R4`	Explicit: `CX`
Location of second operand	Implicit: -1	none
Place to store the result	Explicit: `R4`	none
Location of next instruction	Explicit: Location of LOOP if result is not equal 0 Implicit: The instruction after SOB, or "PC" if result = 0	Explicit: Location of Addr if CX = 0 Implicit: The instruction after JCXZ, or "PC" if CX is not equal 0

2.13 Tell which addressing modes are used by each of the instructions in Table 2.2. **(§2.2)**

Solution:

Instruction	Source address mode	Destination address mode
`MULF A,B,C`	Direct addressing	Direct addressing
`nabs r3, r1`	Register direct addressing	Register direct addressing
`ori $2,$1,255`	Register direct addressing	Register direct addressing
`DEC R2`	Register direct addressing	Register direct addressing
`SHL AX, 4`	Register direct addressing	Register direct addressing

2.14 Suppose that SRC instruction formats are considered different only when field boundaries in the instruction word change and not when some fields or parts of fields are unused. How many different formats should appear in Figure 2.10 in this case? **(§2.3)**

Solution: Formats 3, 4, 5, 6, and 7 in Figure 2.9 could be considered as one format. Format 1 uses a 17-bit constant, so it is another format. Format 2 is also distinct because it uses a 22-bit constant. Format 8 can be combined with any format that has operand field, giving 3 different formats.

2.15 Encode the program on page 63 in hexadecimal. **(§2.3)**

Solution:

Address	Label	Instruction	Hexadecimal
1388H		`lar r0, Over`	3000000CH
138CH		`ld r1, X`	084003E8H
1390H		`brpl r1, r0`	40001004H
1394H		`neg r1, r1`	78401000H
1398H	`Over`	`addi r1, r1, cost`	4842007DH
139CH		`st r1, X`	1840000CH

2.16 Write SRC code to implement the following C statements, assuming all variables are 32-bit integers:

`a.if (a < 0) a = -a; else a = 0;`

`b.for ( i = 0; i < 10; i++ )`
`ndigit[i] = i+1; assuming a declaration of ndigit[10]` **(§2.3)**

Solution: a.

```
        ld   r0, a      ;Get value of a
        la   r1, 0      ;Get constant 0
        lar  r2, sign   ;Set branch target
        brmi r2, r0     ;Skip next if a<0
        st   r1, a      ;Store 0 into a
sign:   neg  r0, r0     ;Converts a to -a
        st   r0, a      ;Store -a into a
```

b.

```
        la   r4, 0          ;Constant 0
        la   r3, ndigit     ;R[3] points to ndigit[i]
        lar  r2, loop       ;Branch target
        addi r1, r4, 0      ;Make R[1], =i, 0
        addi r5, r1, 1      ;Make R[5], =i+1
loop:   st   r5, 0(r3)      ;ndigit[i] = i+1
        addi r3, r3, 4      ;Advance array pointer
        addi r1, r1, 1      ;i++
        addi r0, r1, -10    ;R[0]<0 iff i<10
        addi r5, r5, 1      ;i+1
        brmi r2, r0         ;Repeat if i<10
```

2.17 Testing a difference against zero is not the same as comparing two numbers in finite precision arithmetic. Propose an encoding for an SRC branch instruction that specifies two registers to be compared, rather than one register to be compared against zero.

a.What potential problems might there be with implementing the modified instruction?

b.How would condition codes improve the situation?

c.Can you suggest a restructuring of the SRC branch that would help without using condition codes? **(§2.3)**

Solution: a. Two numbers are usually compared by a subtraction followed by testing the result. The problem is that the 32-bit difference does not contain enough information. In case of overflow, the 32-bit 2's complement difference cannot correctly show which of the two compared numbers is greater.

b. Condition codes are flags in the processor state that are set as a side effect of some arithmetic instruction. The usual condition code flags are N (negative), Z (zero), V (overflow), and C (carry out). Testing these flags gives enough information to tell the correct result of the comparison.

c. The register tested in a branch instruction could hold condition codes rather than the 32-bit difference. A comparison instruction could be added to the instruction set that compares two numbers and stores the condition codes in the destination register. The new branch instructions could still use format 4 and 5 in Figure 2.9. The comparison instruction could use format 6.

2.18 Procedure-calling sequences are standard groups of instructions that transfer control from the main program to a procedure, supplying it with input arguments if necessary. Return sequences finish the procedure by setting up any output arguments and transferring control back to the point following the call. Write a call and return sequence for an SRC procedure that computes the absolute value of an integer passed and returned in r0. Assume r31 is the linkage register. **(§2.3)**

Solution:

```
Main:
        ...
        ld   r0, Integer     ;load the  input argument
        la   r1, 0           ;load constant 0
        addi r30, r30, -4    ;assume stack pointer is r30
        st   r0, (r30)       ;push the integer
        addi r30, r30, -4    ;leave the space for the output
        la   r29, Proc       ;load the address of procedure
        brl  r31, r29        ;call with return address in r31

        ld   r0, (r30)       ;on return,
        st   r0, Output      ;get the output result
        addi r30, r30, 8     ;restore the stack pointer
        ...
```

```
Proc:   addi r30, r30, -4     ;push return address
        st   r31, (r30);
        la   r28, Abs         ;load the address of Abs
        la   r27, Ret         ;load the address of Ret
        ld   r0, 8(r30)       ;load integer
        brmi r28, r0          ;branch to Abs if integer < 0
Ret:    st   r0, 4(r30)       ;store result from r0
        ld   r29, (r30)       ;pop the return address
        addi r1, r1, 4
        br   r29              ;return to calling program

Abs:    neg  r0, r0           ;get the absolute value
        br   r27              ;return to Ret
```

2.19 Examine the RTN descriptions for `la` and `addi`.

a.How do the instructions differ?

b.Give the pros and cons of eliminating one or the other. **(§2.4)**

Solution: First expand la to compare with addi.

la → R[ra] ← ((rb = 0) → c2⟨16..0⟩{sign extend}:
(rb≠0) → R[rb] + c2⟨16..0⟩{sign extend, 2's complement}):

addi → R[ra] ← R[rb] + c2⟨16..0⟩{sign extend, 2's complement}:

a. Both instructions add an immediate constant to a register, but la treats R[0] as if it contained zero when used as an operand, while addi treats it like any other register.

b. Eliminating either one has the advantage of saving an opcode. Eliminating la makes it impossible to load a small constant into a register unless some register is known to contain zero. Eliminating addi retains the ability to load an immediate constant but makes it impossible to use R[0] as the first operand of an immediate add.

2.20 Modify the SRC RTN to include a SingleStep button. SingleStep functions in the following way: when Run is true, SingleStep has no effect. When Run is false, that is, when the machine is halted, pressing SingleStep causes the machine to execute a single instruction and then return to the halted state. **(§2.4)**

Solution: instruction_interpretation := (

¬Run ∧ Strt → Run ← 1:

Run → (IR ← M[PC]: PC ← PC + 4; instruction_execution):

¬Run ∧ ¬Strt ∧ SingleStep → (SingleStep ← 0: IR ← M[PC]:
PC ← PC + 4; instruction_execution):

2.21 Modify the SRC to include the swap (op = 7) instruction that exchanges the contents of two registers, ra and rb, by writing RTN for the new instruction. **(§2.4)**

Solution: swap (:= op = 7) → (R[rb] ← R[ra]: R[ra] ← R[rb]):

2.22 a. Modify the SRC RTN to include a conditional jump instruction, `jpr` (op = 25). It should use format 2 in Figure 2.10. The jpr instruction uses relative addressing, rel, instead of a branch target register. The jump should be taken only if ra = 0.

b.Change the meaning of `jpr` so that the jump is taken only if the register specified by the ra field has a nonzero value. **(§2.4)**

Solution: a. jpr (:= op = 25) → ((ra = 0) → PC ← rel:)

b. jpr (:= op = 25) → ((R[ra] ≠ 0) → PC ← rel):

2.23 Describe in words the difference between the two addressing modes described in RTN as follows:

a.M[M[X + R[a]]]

b.M[M[X] + R[a]]. **(§2.5)**

Solution: They are both indexed indirect addressing modes. In mode a, the index register is added to the address of the indirect pointer, so it is called preindexing. In mode b, the indirect pointer is fetched from memory address X and the index is then added to the pointer, so it is called postindexing.

2.24 Write SRC instructions to load a value into a register using each of the addressing modes of Table 2.8. Use multiple SRC instructions for a mode only when necessary. **(§2.5)**

Solution:

Addressing mode	**Assembler syntax**	**SRC instructions**
Register	Ra	`addi rt, ra, 0`
Register indirect	(Ra)	`ld rt, 0(ra)`
Immediate	#x	`la rt, x(r0)`
Direct, absolute	x	`ld rt, x`
Indirect	(x)	`ld rn, x` `ld rt, 0(rn)`
Indexed, based, or displacement	x(Ra)	`ld rt, x(ra)`
Relative	x(PC)	`ldr rt, x`
Autoincrement	(Ra)+	`ld rt, 0(ra)` `addi ra, ra, 4`
Autodecrement	–(Ra)	`addi ra, ra, -4` `ld rt, 0(ra)`

2.25 Assume that in a certain byte-addressed machine all instructions are 32 bits long. Assume

the following state of affairs for the machine:

Address	Value
PC	100
r0	200
r1	300
100	200
104	300
108	400
200	500
300	600
500	700

Fill in the following table, assuming that each statement executes from the initial state defined above. The lea, load effective address, instruction is similar to the LEA instruction shown in Table 2.1 **(§2.5)**

Solution: .

Instruction	Addressing Mode	Value of r0 after Execution
`load r0, #200`	Immediate	200
`load r0, 200`	Direct	500
`load r0, (200)`	Indirect	700
`load r0, r1`	Register	300
`load r0, [r1]`	Reg. Ind.	600
`load r0, -100[r1]`	Based	500
`lea r0 -100[r1]`	Based	200
`load r0, 200[PC]`	Relative	600

2.26 Consider the C `int` array variable V[2]. Assume that C ints are 32 bits in size and that the base address of V is in r3.

Write a single SRC instruction similar to those in Exercise 2.25 that will store V[2] in r4. **(§2.5)**

Solution: The SRC instruction to store V[2] is : load r4, 8[r3]

2.27 Using the hardware in Figure 2.24, write the RTN description and the control sequence that implements the following:

a.R[0] ← R[1] + R[2] + 2

b.R[3] ← R[4] + R[5] + R[6]. **(§2.6)**

Draw timing diagrams similar to Figure 2.25 for the control sequences developed above. **(§2.6)**

Solution:

a.

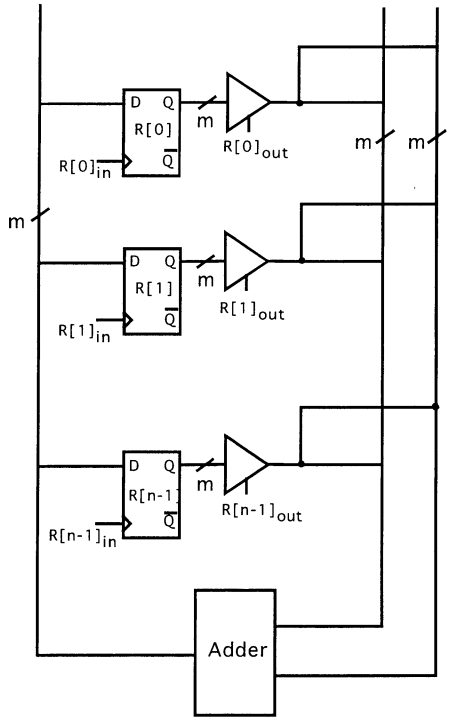
W ← R[2] + 1;	$R[2]_{out}$, W_{in};
W ← W + 1	W_{out}, W_{in}
Y ← W;	W_{out}, Y_{in};
Z ← R[1] + Y;	$R[1]_{out}$, Z_{in};
R[0] ← Z;	Z_{out}, $R[0]_{in}$;

b.

Y ← R[6];	$R[6]_{out}$, Y_{in};
Z ← R[5] + Y;	$R[5]_{out}$, Z_{in};
Y ← Z;	Z_{out}, Y_{in};
Z ← R[4] + Y;	$R[4]_{out}$, Z_{in};
R[3] ← Z;	Z_{out}, $R[3]_{in}$;

2.28 Design a circuit similar to Figure 2.24, but without the incrementer, so that any register can be added to any register and the result stored in any register *in one clock cycle*.**(§2.5)**

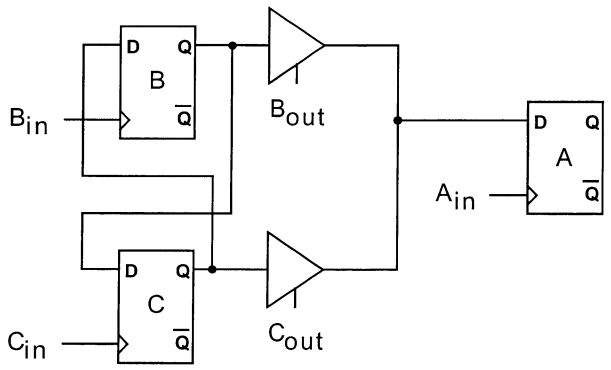
Solution:



2.29 Design data path logic that will allow any two of the following register transfers that do not have the same destination to be done in one step. **(§2.5)**

A ← B:
A ← C:
C ← B:
B ← C:

Solution:



2.30 It is a widespread practice to develop simulators for computers that are under development. The simulators run on other available machines, and they allow designers and programmers to evaluate machine hardware and software in advance of the availability of the machine. A simulator may simulate only the abstract behavior of the machine, without any pretense of performing the operations the way an actual machine would perform them, or it may simulate the structure and function of the machine down to the gate level and below. If a compiler were available for RTN, the compiled code could be thought of as being a simulator for SRC at the most abstract level.

Begin the process of writing a behavior-level simulator for SRC by declaring the memory, formats, and effective address parts of the machine in ANSI C. Assume that only the first 4,096 32-bit words are implemented, and assume that integers on your machine are 32-bits long. **(§2.4)**

Solution:

```
/* Simulator for SRC */
int PC;                         /* Program counter */
int R[32];                      /* General registers */
short int Run, Strt;            /* Flags */
union {int M[4096];
            char Mem[16384];
            } memory;           /* Main memory */
union { struct { op:5; ra:5; c1:22;
                        } oneR;
            struct { op:5; ra:5; rb:5; c2:17;
                        } twoR;
            struct { op:5; ra:5; rb:5; rc:5; c3:12;
                        } threeR;
            } IR;                       /* Inst. register */
int sge22(c1)                   /* Sign extend 22 bits */
 int ce;
 {ce = c1; if (ce >= 2097152) ce = 4194304 - ce;
 return ce;}
```

```
int sge17(c2)                /* Sign extend 17 bits */
 int ce;
 {ce = c2; if (ce >= 65536) ce = 131072 - ce;
 return ce;}
int sge12(c3)                /* Sign extend 12 bits */
 int ce;
{ce = c3; if (ce >= 2048) ce = 4096 - ce; return ce;}
int disp(IR)                 /* Displacement address */
 int d;
 { if (IR.twoR.rb == 0) d = sge17(IR.twoR.c2);
   else d = R[IR.twoR.rb] + sge17(IR.twoR.c2);
   return d; }
int rel(IR)                  /* Relative address */
 int r;
 { r = PC + sge22(IR.oneR.c1); return r; }
```