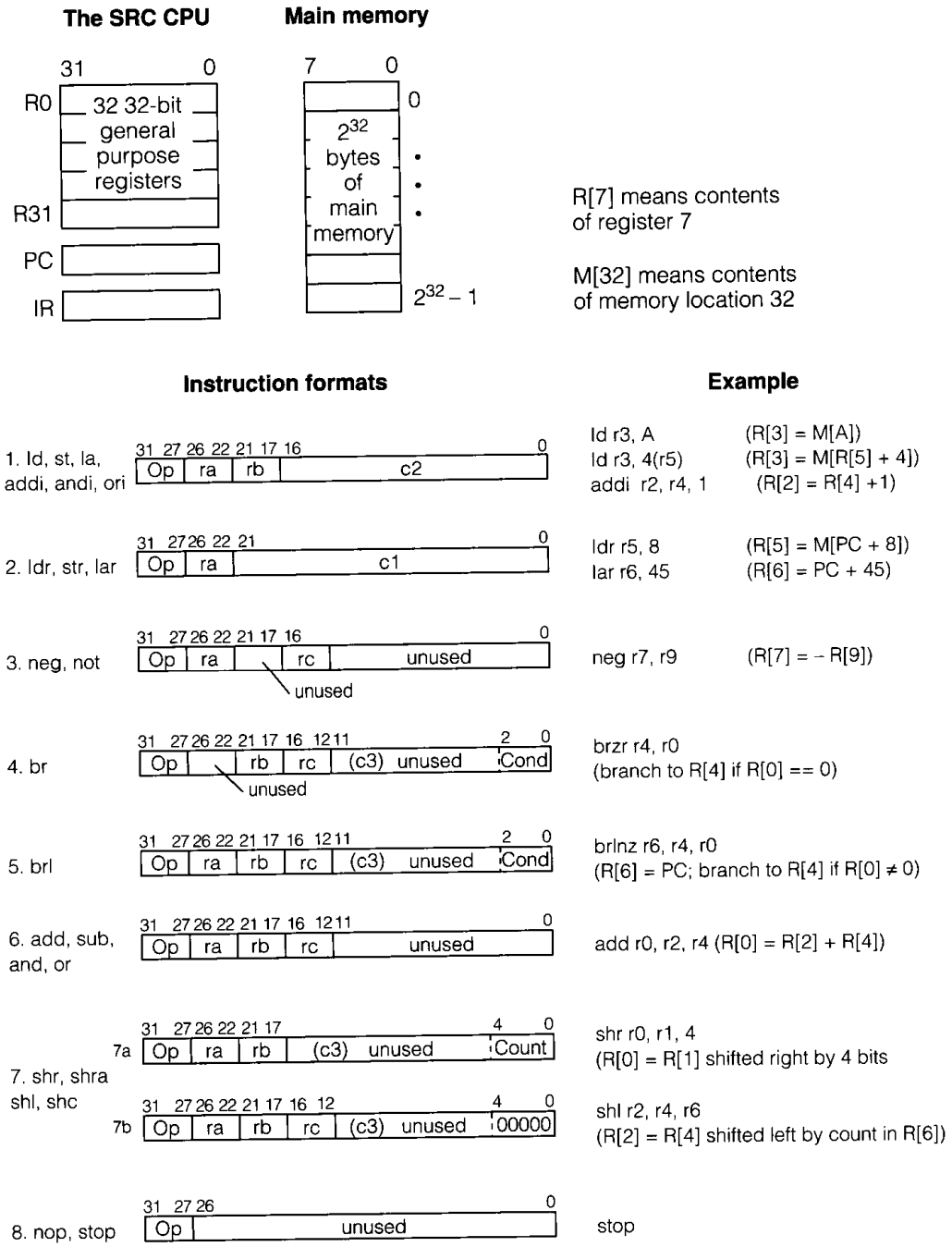
COE 405
Design & Modeling of Digital Systems

Course Project
Simple RISC Computer
Due on: Sunday Jan. 5, 2004

In this project, you are to model a simple RISC processor with 32 general purpose, 32-bit registers, plus a program counter (PC) and an instruction register (IR). Although the main memory is organized as an array of bytes, only 32-bit words can be fetched from or stored into main memory. A word at address A is defined as the 4 bytes at that address and the succeeding three addresses. The byte at the lowest address contains the most significant 8-bits, the byte at the next address contains the next most significant 8 bits, and so on. The programmer's model of the SRC machine is shown below. It has 23 instructions in 8 different formats:

- **Load and Store instructions**: There are four load instructions-- ld, ldr, la, and lar—and two store instructions– st and str.
- **Branch instructions**: There are two branch instructions, br and brl, that allow unconditional and conditional branches to an address contained in a specified register. The conditional branches test a register's contents and branch when the register contents are $=0, \neq 0, \geq 0, < 0$. The instruction brl stores the PC in a specified register.
- **Arithmetic instructions**: There are four arithmetic instructions: add, addi, sub, and neg. All except addi take two register operands and place the result in a register. The instruction addi adds an immediate constant contained in the c2 field to a register and places the result in a register.
- **Logical and shift instructions**: There are nine logical and shift instructions: and, andi, or, ori, not, shr, sha, shl, and shc. The shift instructions can shift by a count contained as a constant in the instruction or by a count in a register.
- **Miscellaneous instructions**: There are two zero-operand instructions—nop and stop.

All instructions are 32 bits long. Because the SRC is of the load-store class of machine, operands in memory can be accessed only through load and store instructions. All instructions have a 5-bit opcode field, allowing 32 different instructions. Here we use only 23 of these instructions. The ra, rb, and rc fields are 5-bit fields and specify one of the 32 general purpose registers. A more detailed description of the processor is attached in the Appendix.



(i) Write a behavioral model for the SRC machine modeling all its specified instruction set.

(ii) Write a behavioral model for the memory interfaced with the CPU. Assume that the CPU-Memory interface contains the following control signals: Read, Write, and MFC (Memory Function Complete). Both the CPU and Memory are synchronized by the same clock. When a Read or Write request is initiated by the CPU, it remains 1 until the memory indicates that it finished the requested operation by setting the MFC signal to 1. Assume that the MFC signal will remain 1 for 1 clock cycle. Also, assume that the MFC signal changes based on the falling edge of the clock while the CPU signals change on the rising edge of the clock.

(iii) Write a test bench to test your behavioral model of the SRC machine. The test bench should contain Memory, which contains an assembly program for the CPU to execute.

(iv) Partition your CPU into a data path unit and a control unit. Write a register transfer model for the designed CPU. The data path should be described structurally while the components in the data path, i.e. ALU, registers can be described in a behavioral or dataflow style. The control unit is to be modeled as a finite state machine, which can be modeled in a behavioral or dataflow style.

(v) In the modeling of the data path use the three-**bus design** given in the Appendix.

(vi) Use **std_logic and std_logic_vector** for all signals in the design.

(vii) Include all functions, procedures, and user-defined types and constants in a package to be used by the CPU.

(viii) For each entity that you model, test it and include simulation output indicating that it is working properly. The data-path should be tested before it is connected to the control unit. This way you can detect problems in the design and modeling early.

(ix) Assume that the CPU-Memory interface contains the following control signals: Read, Write, and MFC (Memory Function Complete). Both the CPU and Memory are synchronized by the same clock. When a Read or Write request is initiated by the CPU, it remains 1 until the memory indicates that it finished the requested operation by setting the MFC signal to 1. Assume that the MFC signal will remain 1 for 1 clock cycle. Also, assume that the MFC signal changes based on the falling edge of the clock while the CPU signals change on the rising edge of the clock.

(x) Test the register transfer model of the SRC machine using the same test bench used for the behavioral model.

*(xi) If you manage to synthesize your CPU, map it to FPGA and demonstrate its proper operation, you will get a **5% bonus** from your total mark in the course.*

This project can be conducted by a team of a maximum of three students. If one student or two do the project they will be given a bonus. If you are a team of three students, you can divide the work such that one student does the behavioral model of the CPU, one models the data path and one models the control unit. You have to understand all the work done in the project, the part you do and the parts done by the other team members. Each student will be tested for the three parts and each student in the team will be given a separate grade.

Clearly state your assumptions and have your design well documented. Write a professional report indicating all design stages, modeling and testing of each component and the final design. Include both a hard copy and a soft copy of your report and all VHDL files. The grading policy for the project is shown below:

Grading Criteria	**Mark**
Memory interface Modeling and Test	5
CPU Behavioral Description & Test Bench	25
Basic Components Modeling & Test	10
Data-path Design, Modeling & Test	15
Control Unit Design, Modeling & Test	20
Whole CPU Design, Modeling & Test	20
Report Organization	5
Total	100

direct addressing mode

The previous code examples have used the *direct addressing mode,* shown in Figure 2.8b. The address of the operand is specified as a constant contained in the instruction.

indirect addressing mode

In *indirect addressing,* shown in Figure 2.8c, a constant in the instruction specifies not the address of the value, but the address of the address of the value. An example of the use of indirect addressing is in implementing pointers, where the pointer, which is an address, is stored in memory at the pointer address. Thus, two memory references are required to access the value. The CPU must first fetch the pointer, which is stored in memory; then, having that address, the CPU accesses the value stored at that address.

register direct mode

In the *register direct mode,* shown in Figure 2.8d, the operand is contained in the specified register.

register indirect mode

When the address of the operand is in a register, the mode is referred to as the *register indirect mode,* shown in Figure 2.8e. This addressing mode is used to sequentially access the elements of an array stored in memory. The starting address of the array is stored in a register, an access made, and the register incremented to point to the next element.

indexed, or displacement, or based mode

To access arrays, or components of the C `struct`, or the Pascal `record` (which by definition are stored at a fixed offset from the start address of the structure), the *indexed mode,* sometimes called *displacement* or *based* addressing, is used, as shown in Figure 2.8f. The memory address is formed by adding a fixed constant, usually contained within the instruction, to the address value contained in a register. The term *indexed* is normally used when the constant value is the base of an array in memory, added to the "index" stored in a register. The term *displacement* is used when the base of a `struct` is held in a register and added to the constant offset, or displacement, of the field in the `struct`.

relative addressing mode

The *relative addressing mode,* shown in Figure 2.8g, is similar to indexed, but the base address is held in the PC rather than in another register. This allows the storage of memory operands at a fixed offset from the current instruction.

The previous discussion is only to provide a flavor for the complexity of addressing modes. A formal description of addressing modes is given in Section 2.5.

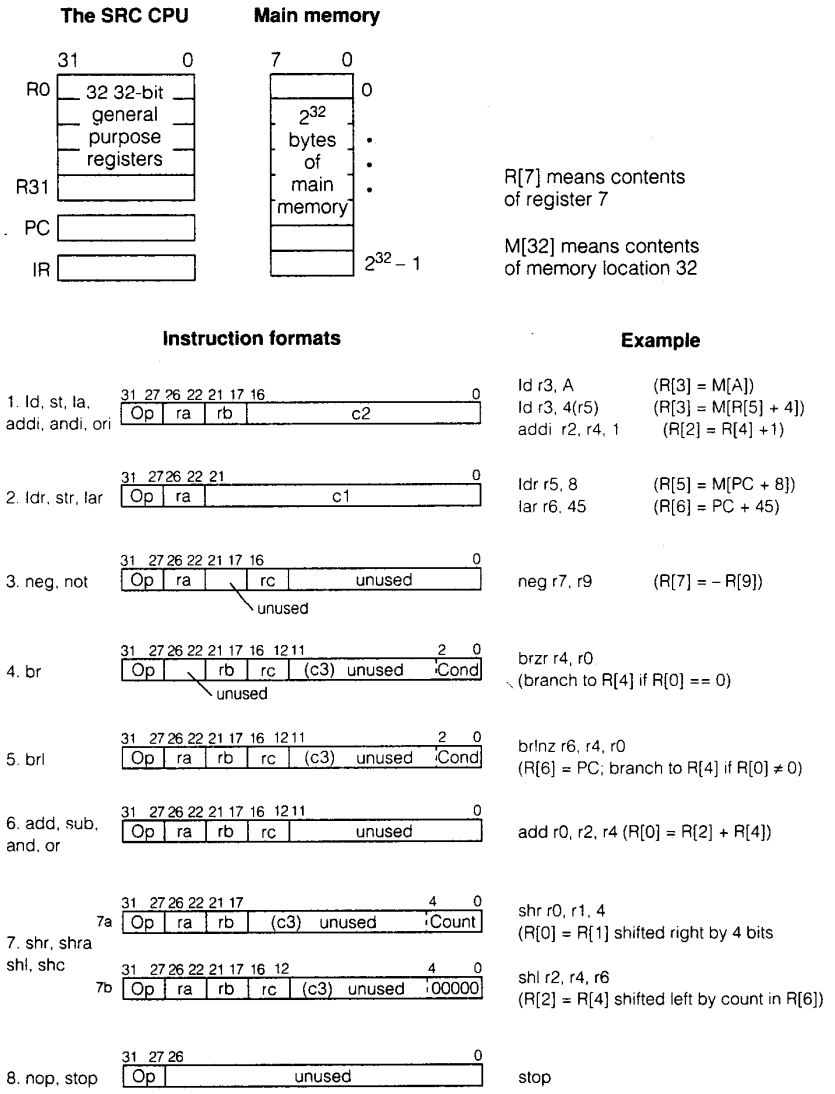
2.3 Informal Description of the Simple RISC Computer, SRC

In this section we provide an informal description of SRC, and in the next section we provide a formal description. This example machine is sufficiently simple and lacking in the complications necessary in real machines that in Chapters 4 and 5 it will serve as an example of detailed machine hardware design.

2.3.1 Register and Memory Structure

Figure 2.9 shows the programmer's model of the SRC machine. It is a general register machine, with 32 general purpose, 32-bit registers, plus a program counter (PC) and an instruction register (IR). Although the main memory is or-

FIGURE 2.9 **Programmer's Model of the SRC**



ganized as an array of bytes, only 32-bit words can be fetched from or stored into main memory. Its memory operand access follows the load-store model described above. A word at address A is defined as the 4 bytes at that address and the succeeding three addresses. The byte at the lowest address contains the most significant 8 bits, the byte at the next address contains the next most significant 8 bits, and so on.

2.3.2 Instruction Formats

Figure 2.9 shows 23 instructions in 8 different formats:

- Load and store instructions: There are four load instructions—ld, ldr, la, and lar—and two store instructions—st and str.
- Branch instructions: There are two branch instructions, br and brl, that allow unconditional and conditional branches to an address contained in a specified register. The conditional branches test a register's contents and branch when the register contents are = 0, ≠ 0, ≥ 0, or < 0. The instruction brl stores the PC in a specified register.
- Arithmetic instructions: There are four arithmetic instructions: add, addi, sub, and neg. All except addi take two register operands and place the result in a register. The instruction addi adds an immediate constant contained in the c2 field to a register and places the result in a register.
- Logical and shift instructions: There are nine logical and shift instructions: and, andi, or, ori, not, shr, sha, shl, and shc. The shift instructions can shift by a count contained as a constant in the instruction or by a count in a register.
- Miscellaneous instructions: There are two zero-operand instructions: nop and stop.

All instructions are 32 bits long. Because the SRC is of the load-store class of machine, operands in memory can be accessed only through load and store instructions. All instructions have a 5-bit opcode field, allowing 32 different instructions. Here we define only 23 of these instructions. The ra, rb, and rc fields are 5-bit fields that specify one of the 32 general purpose registers. Constants c1, c2, c3, Cond, and Count are used in various ways that we will describe as we discuss the individual instructions. We first discuss the memory addressing modes and then each of the 23 instructions. As we describe each instruction, we include a comment field beginning with a semicolon that describes the operation of the instruction in pseudo C code. The notation M[x] means the value stored at word x in memory.

Notice that there are many unused "holes" in the instructions that waste memory space. This willingness to trade off less efficient use of memory for having all instructions be exactly one word long is a feature of most modern RISC machines. This issue is discussed in more detail in Chapter 3.

2.3.3 Accessing Memory: The Load and Store Instructions

The load and store instructions are the only SRC instructions to access operands in memory.

Load and Store Instructions

```
ld ra, c2       ;Direct addressing: R[ra] = M[c2]
ld ra, c2(rb)   ;Indexed addressing(rb ≠ 0); R[ra] = M[c2 + R[rb]]
st ra, c2       ;Direct addressing: M[c2] = R[ra]
st ra, c2(rb)   ;Indexed addressing(rb ≠ 0); M[c2 + R[rb]] = R[ra]
la ra, c2       ;Load displacement address: R[ra] = c2
la ra, c2(rb)   ;Load displacement address: R[ra] = c2 + R[rb]
```

direct addressing with rb = 0

These instructions use format 1 from Figure 2.9. The register to be loaded or stored is specified in the 5-bit field `ra`, and the address is specified as the 17-bit value in the `c2` field. The `rb` field serves double duty. If rb = 0 (that is, if the value of the 5-bit field is zero, specifying r0), this serves as a signal to the machine control unit that the memory address is just the value of `c2` as a sign-extended 2's complement number. If any of the other 31 registers are specified—that is, if rb ≠ 0—then the memory address is formed by adding R[rb] + c2, resulting in the based, or displacement, addressing mode. Be aware that the addition of `c2` to R[rb] takes place when the instruction is executing, that is, at run time. Notice that when `c2` is set to 0, the addressing mode becomes register indirect.

Thus the `ld` instruction loads into register R[ra] the operand stored at address `c2` (direct addressing) if rb = 0, or at address c2 + R[rb] if rb ≠ 0 (indexed or displacement addressing); the st instruction does the reverse, storing the operand in R[ra] at address `c2` when rb = 0, or at address c2 + R[rb] when rb ≠ 0.

The la (load address) instruction calculates the operand address as above, but then rather than fetching the operand, it stores the calculated value in R[ra]. Operationally, it loads the value of `c2` or c2 + R[rb] itself into a register. This allows complex address calculations to be performed explicitly. In this way, addressing modes not available in the instruction set can be simulated by a series of explicit arithmetic steps.

direct address restrictions

Several points should be made about these instructions. First, because `c2` is a 17-bit value, only operands stored in the first or last 2^{16} bytes of memory can be accessed using the direct addressing mode, or, in the case of the `la` instruction, only positive or negative constants with magnitudes smaller than 2^{16} can be loaded. To access operands stored elsewhere in memory, the displacement or register indirect addressing modes must be used, with the value in R[rb] serving as the base and the value of `c2` serving as an offset. (Recall that the register indirect addressing mode can be achieved by setting `c2` equal to 0.) Note also that the address addition is 2's complement, so the 17-bit displacement must be sign-extended to 32 bits before the address addition. See Chapter 6 for a more thorough discussion of 2's complement arithmetic.

TABLE 2.4 **Example SRC Load and Store Instructions**

Instruction	op	ra	rb	c1	Meaning	Addressing Mode
`ld r1, 32`	1	1	0	32	R[1] ← M[32]	Direct
`ld r22, 24(r4)`	1	22	4	24	R[22] ← M[24 + R[4]]	Displacement
`st r4, 0(r9)`	3	4	9	0	M[R[9]] ← R[4]	Register indirect
`la r7, 32`	5	7	0	32	R[7] ← 32	Immediate
`ldr r12, -48`	2	12	-	-48	R[12] ← M[PC -48]	Relative
`lar r3, 0`	6	3	-	0	R[3] ← PC	Register (!)

Relative addressing computes the operand address as an address *relative to the PC.*

Load and Store Relative

```
ldr ra, c1   ;Load register relative: R[ra] = M[PC + c1]
str ra, c1   ;Store register relative: M[PC + c1] = R[ra]
lar ra, c1   ;Load relative address: R[ra] = PC + c1
```

relocatable instructions

The effective address is formed by the run-time addition, c1+PC. These relative addressing modes make the instructions *relocatable.* Because the address of the data is specified as a value that is a constant offset from the PC, and hence from the current instruction, the entire module of program and data can be moved, or relocated, anywhere in the machine memory without changing the values of the displacements. This is in contrast to the direct addressing mode, which specifies addresses as absolute memory locations. Because the displacement constant c1 has 22 bits, addresses within $\pm 2^{21}$ of the current instruction can be specified.

Table 2.4 provides examples of the assembly language and resulting machine encoding of several load and store instructions. You should study it until you understand each entry. The operation codes for each instruction are given in the op column.

Example 2.2: Binary Encoding of an SRC Instruction As an example of SRC instruction encoding, let us encode the second instruction in Table 2.4, which is ld r22, 24(r4). Working from the msb, the encoding will be

```
op=1  ra=22  rb=4        c1=24
00001 10110 00100 0000000000011000 = 0D880018H
```

You should verify this encoding and try several examples for yourself.

2.3.4 Arithmetic and Logic Instructions

This class of instructions uses the ALU of the SRC machine to do arithmetic, logical, and shift operations. We first cover the "1-operand" instructions `not` and `neg`.

1-Operand ALU Instructions

```
neg ra, rc      ;Negate: R[ra] = -R[rc]
not ra, rc      ;Not: R[ra] = R[rc]
```

These format 3 instructions take one register operand and provide one register result. The instruction `neg` (op = 15) takes the 2's complement of the contents of register R[rc] and stores it in register R[ra]. The `not` (op = 24) instruction takes the logical (1's) complement of the contents of register R[rc] and stores it in register R[ra]. All other fields in the instruction are unused.

The instructions `add` (op = 12), `sub` (op = 14), `and` (op = 20), and `or` (op = 22) are 2-operand, 1-result instructions. All must be in the general purpose registers. They are specified using format 6. Notice that the least significant 12 bits are unused, because the first 4 fields are sufficient to describe the entire operation.

2-Operand ALU Instructions

```
add  ra,  rb,  rc   ;2's complement addition: R[ra] = R[rb] + R[rc]
sub  ra,  rb,  rc   ;2's complement subtraction: R[ra] = R[rb] - R[rc]
and  ra,  rb,  rc   ;Logical AND: R[ra] = R[rb]∧R[rc]
or   ra,  rb,  rc   ;Logical OR: R[ra] = R[rb]∨R[rc]
```

There are three ALU instructions that use the immediate addressing mode: `addi` (op = 13), `andi` (op = 21), and `ori` (op = 23). The constant is contained in the 17-bit `c2` field and is sign-extended to a 32-bit value before the arithmetic operation is performed. All of these instructions use format 1.

Immediate Addressing ALU Instructions

```
addi ra,  rb,  c2   ;Immediate 2's compl. addition: R[ra] = R[rb] + c2
andi ra,  rb,  c2   ;Immediate logical and: R[ra] = R[rb]∧c2
ori  ra,  rb,  c2   ;Immediate logical or: R[ra] = R[rb]∨c2
```

The shift instructions shift the operand in R[rb] right, left, or "circularly," from 1 to 32 bits, and place the result in R[ra]; the amount of the shift is governed by an encoded 5-bit unsigned integer, so shifts from 0 to 31 bits are possible. The integer representing the shift count is stored either as an immediate value in the 5 least significant bits in the instruction (format 7a), or, if that value is 0, then the shift count is taken from the least significant 5 bits of register R[rc] (format 7b).

There are two forms of right shift, `shr` and `shra` (op = 26 and 27, respectively). The first form shifts zeros in from the left as the value is shifted right, and the second form, the so-called arithmetic shift, continually shifts copies of the msb into the word on the left as the contents are shifted right. This arithmetic form of the shift preserves the arithmetic sign of 2's complement numbers during the shift operation.

The left shift, `shl` (op = 28), shifts zeros in on the right as the value in the register is shifted left. The circular shift, `shc` (op = 29), shifts the value left by *count* bits, but the value shifted out of the register on the left is placed back into the register on the right. The assembly language forms are shown below.

Shift Instructions

```
shr ra, rb, rc      ;Shift R[rb] right into R[ra] by count in R[rc]
shr ra, rb, count   ;Shift R[rb] right into R[ra] by count in c3
shra ra, rb, rc     ;AShift R[rb]right into R[ra] by count in R[rc]
shra ra, rb, count  ;AShift R[rb] right into R[ra] by count in c3
shl ra, rb, rc      ;Shift R[rb] left into R[ra] by count in R[rc]
shl ra, rb, count   ;Shift R[rb] left into R[ra] by count in c3
shc ra, rb, rc      ;Shift R[rb] circ. into R[ra] by count in R[rc]
shc ra, rb, count   ;Shift R[rb] circ. into R[ra] by count in c3
```

All of these instructions are encoded using format 7 from Figure 2.9. If the count field $\neq 0$ (format 7a), then the shift count is taken to be the 5 least significant bits (lsbs), of the c3 field, called "count" in the figure. If the count field = 0 (format 7b), then the shift count is taken from the register encoded in bits 12–16 of the instruction, called `rc` in format 7 of Figure 2.9.

2.3.5 Branch Instructions

The branch instructions `br` (op = 8) and `brl` (op = 9) are encoded using formats 4 and 5. Format 4, `br`, is used to specify a branch instruction that replaces the PC with the target address of the branch. Format 5, `brl`, is used for the branch and link instruction, which copies the PC into a so-called linkage register prior to the branch. This *link* register allows return from subroutine calls and is used to implement high-level language procedures and functions. Notice that the PC is copied into the linkage register regardless of whether the branch is taken. These two instructions allow branching under five different branch conditions. We mentioned in the previous discussion of branch instructions that many machines maintain a set of condition codes in a status register within the CPU that can be tested as part of a conditional branch. SRC does not use this approach. Rather, it allows any of the 32 general purpose registers to hold a value to be tested for conditional branching. The branch condition to be tested is specified by the least significant 3 bits of field c3, c3⟨2..0⟩, as shown in Table 2.5. A two-letter code, `nv`, `zr`, `nz`, `pl`, or `mi`, appended to the mnemonic is converted by the assembler to the branch condition code in c3. The meaning of the ra, rb, and rc fields in the branch instructions is shown on the next page.

subroutine return link

TABLE 2.5 **Branch Conditions and Encoding**

Assembly Language	c3⟨2..0⟩	Branch Condition
`brnv, brlnv`	0	Never
`br, brl`	1	Always (unconditional)
`brzr, brlzr`	2	If R[rc] = 0
`brnz, brlnz`	3	If R[rc] ≠ 0
`brpl, brlpl`	4	If R[rc]⟨31⟩ = 0 (R[rc] ≥ 0)
`brmi, brlmi`	5	If R[rc]⟨31⟩ = 1 (R[rc] negative)

Branch Instructions

```
br  rb, rc, c3        ;Branch to R[rb] if R[rc] meets condition in c3
brl ra, rb, rc, c3    ;R[ra] ← PC; branch as above
```

Table 2.6 shows examples of the forms and encoding of all the branch instructions. The assembler is responsible for converting the branch mnemonic and appended two-letter code of column 1 of the table to operation codes 8 and 9 of `br` and `brl` instructions, respectively, as well as the correct c3⟨2..0⟩ field value selecting the branch condition. The fields that the assembler will assemble into the 32-bit instruction are given in the table: op, ra, rb, rc, and c3⟨2..0⟩. The form in the first column will fix the op and c3⟨2..0⟩ fields, and the remaining operands will fix the ra, rb, and rc fields. Note that in some cases one or more of these fields are unused, indicated by a long dash in the corresponding table entry.

2.3.6 Miscellaneous Instructions

no operation: nop

In addition to the instructions mentioned above, there are two instructions: `nop` (op = 0), whose purpose is to do nothing, and `stop` (op = 31), whose purpose is to halt the machine. The `nop` instruction is used as a placeholder or as a time waster and is very important in pipelined implementations (see Chapter 5). The `stop` instruction is used to halt the machine at a specified point in program execution. It is useful in debugging, as it can be inserted at problematical points in the program, and if the `stop` instruction is reached, the person doing the debugging can examine the machine state at his or her leisure.

Example 2.3: SRC Assembly Code for a C Conditional Statement In presenting SRC code, we will assume some assembly language conventions that are summarized in Appendix C. Let us encode the C conditional statement,

```
#define Cost 125
if (X<0) X = -X;
```

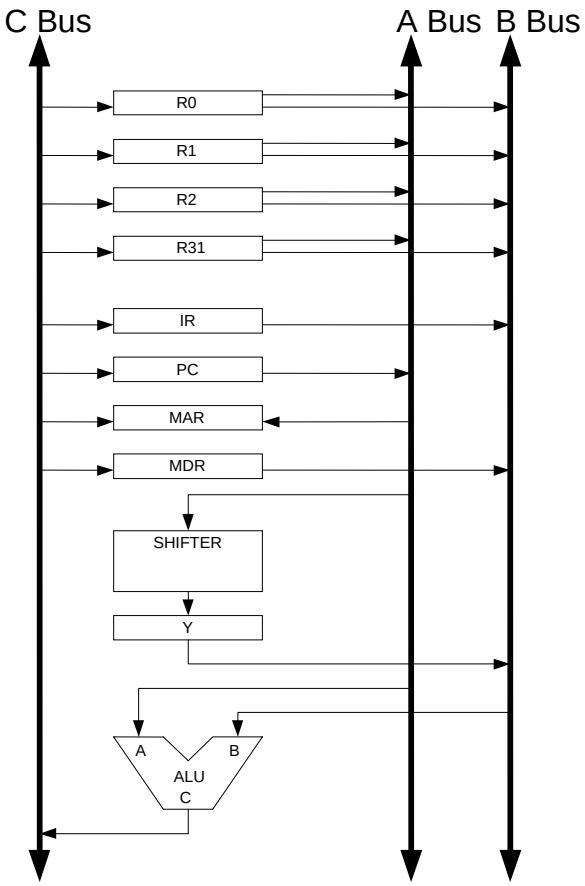
originally discussed on page 39, using SRC assembly language:

TABLE 2.6 **Forms and Formats of the br and brl Instructions**

Assembly Language	Example Instruction	Meaning	op	ra	rb	rc	c3 ⟨2..0⟩	Condition
`brlnv`	`brlnv r6`	R[6] ← PC	9	6	—	—	000	never
`br`	`br r4`	PC ← R[4]	8	—	4	—	001	always
`brl`	`brl r6,r4`	R[6] ← PC; PC ← R[4]	9	6	4	—	001	always
`brzr`	`brzr r5,r1`	if (R[1]=0) PC ← R[5]	8	—	5	1	010	zero
`brlzr`	`brlzr r7,r5,r1`	R[7] ← PC; if (R[1]=0) PC← R[5]	9	7	5	1	010	zero
`brnz`	`brnz r1, r0`	if (R[0]≠0) PC ← R[1]	8	—	1	0	011	nonzero
`brlnz`	`brlnz r2,r1,r0`	R[2] ← PC; if (R[0]≠0) PC← R[1]	9	2	1	0	011	nonzero
`brpl`	`brpl r3, r2`	if (R[2]≥0) PC← R[3]	8	—	3	2	100	plus
`brlpl`	`brlpl r4,r3,r2`	R[4] ← PC; if (R[2]≥0) PC← R[3]	9	4	3	2	100	plus
`brmi`	`brmi r0, r1`	if (R[1]<0) PC← R[0]	8	—	0	1	101	minus
`brlmi`	`brlmi r3,r0,r1`	R[3] ← PC; if (r1<0) PC← R[0]	9	3	0	1	101	minus

```
Cost: .equ   125        ;Define symbolic constant
      .org   1000       ;Next word will be loaded at address 1000₁₀
X:    .dw    1          ;Reserve 1 word for variable X
      .org   5000       ;Program will be loaded at location 5000₁₀
      lar    r0, Over   ;Load address of jump location if expression is false
      ld     r1, X      ;Get value of X into r1
      brpl   r0, r1     ;Branch to Over if r1 ≥ 0
      neg    r1, r1     ;Negate value
Over: ...
```

The three "pseudo ops," `.equ`, `.org`, and `.dw`, are instructions to the assembler and program loader, and do not result in any executable code. The pseudo op `.equ` allows the programmer to specify constants symbolically, in almost exact analogy to the `#define` operation of C; `.org` specifies the locations of data and program in memory; and `.dw` reserves space for program variables.



Data Path Design