5

THE INSTRUCTION SET ARCHITECTURE LEVEL

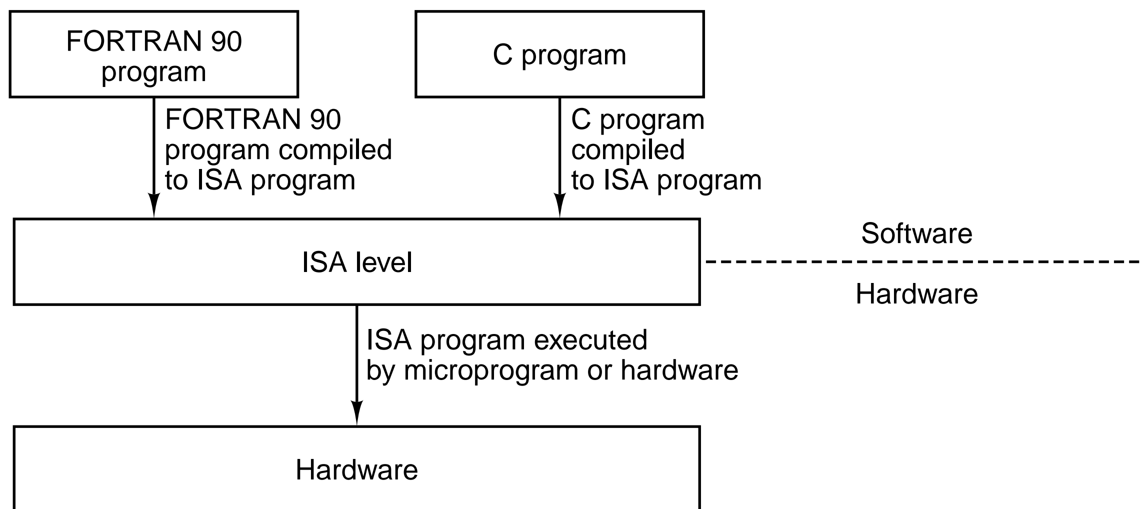

Figure 5-1. The ISA level is the interface between the compilers and the hardware.

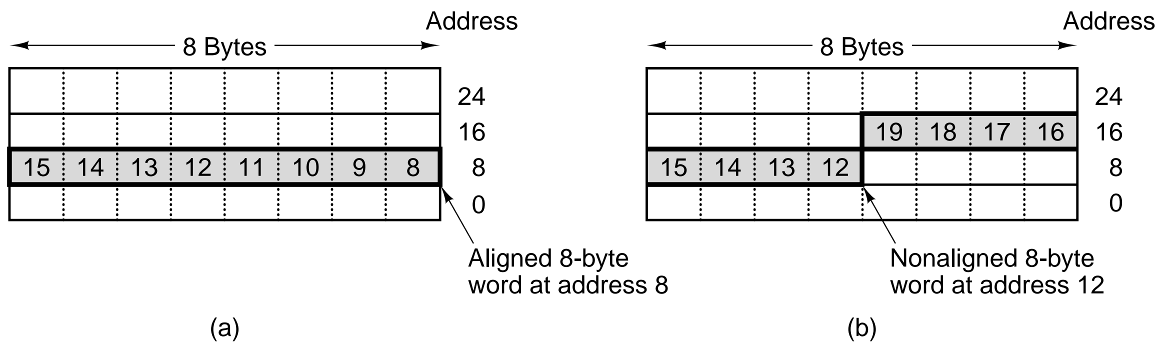

Figure 5-2. An 8-byte word in a little-endian memory. (a) Aligned. (b) Not aligned. Some machines require that words in memory be aligned.

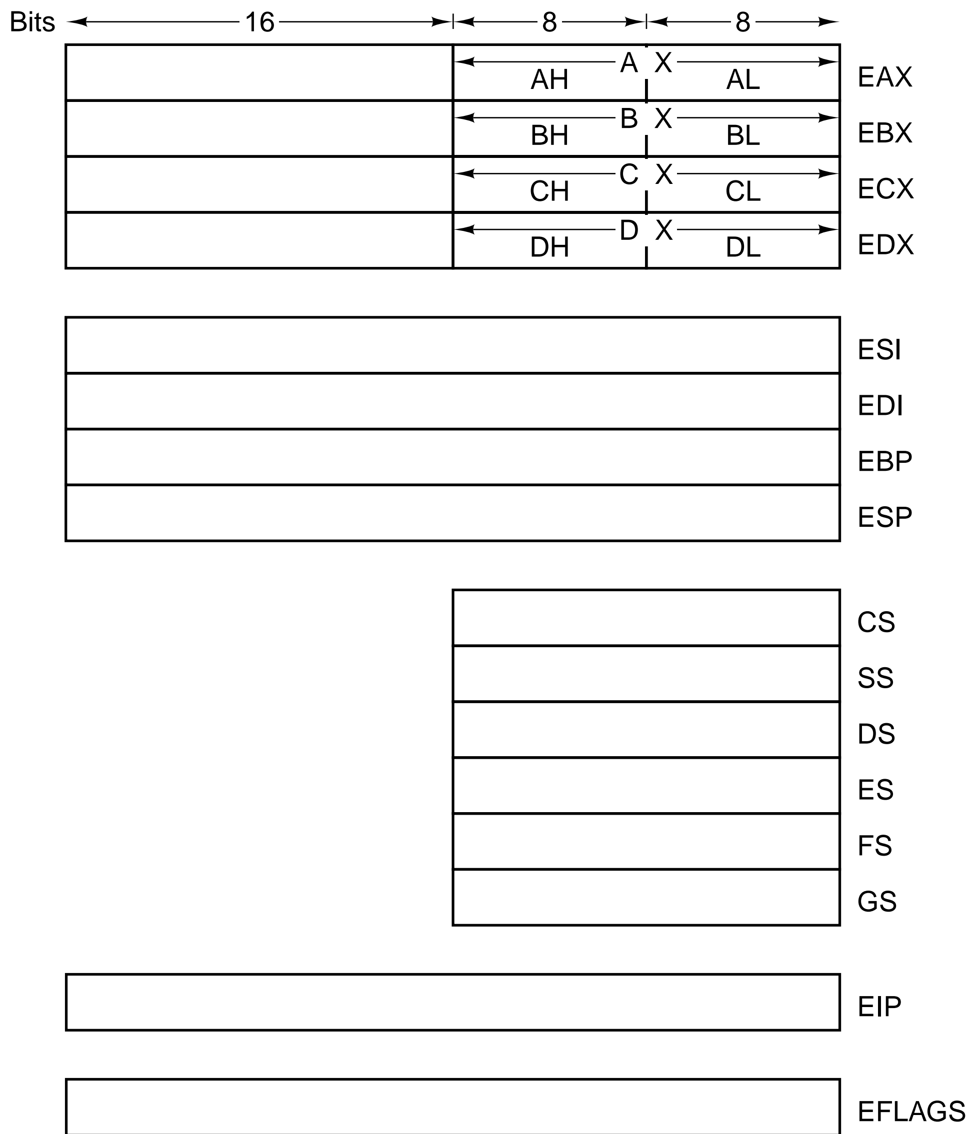

Figure 5-3. The Pentium II's primary registers.

Register	Alt. name	Function
R0	G0	Hardwired to 0. Stores into it are just ignored.
R1 – R7	G1 – G7	Holds global variables
R8 – R13	O0 – O5	Holds parameters to the procedure being called
R14	SP	Stack pointer
R15	O7	Scratch register
R16 – R23	L0 – L7	Holds local variables for the current procedure
R24 – R29	I0 – I5	Holds incoming parameters
R30	FP	Pointer to the base of the current stack frame
R31	I7	Holds return address for the current procedure

Figure 5-4. The UltraSPARC II's general registers.

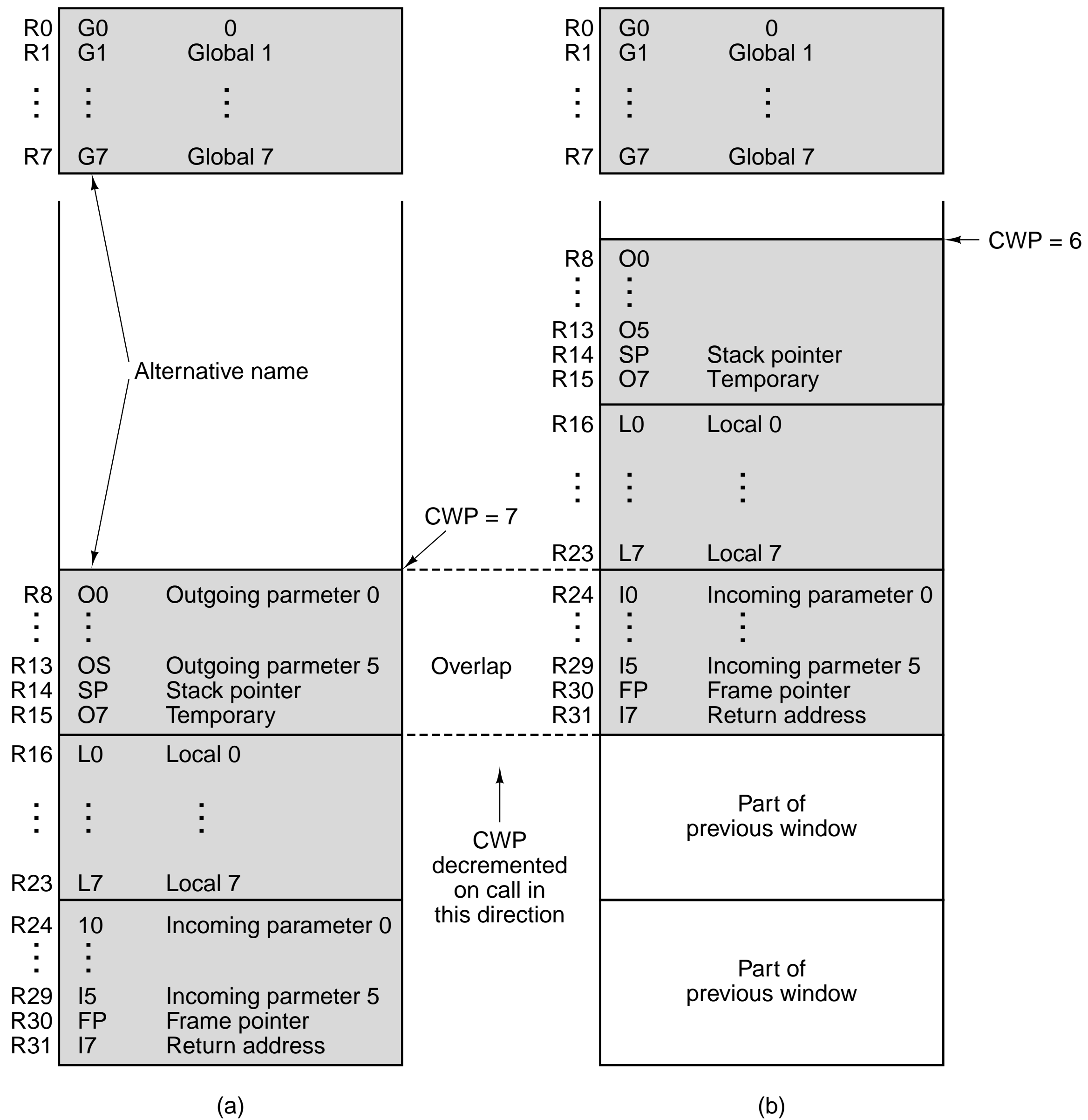

Figure 5-5. Operation of the UltraSPARC II register windows.

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	×	×	×		
Unsigned integer	×	×	×		
Binary coded decimal integer	×				
Floating point			×	×	

Figure 5-6. The Pentium II numeric data types. Supported types are marked with ×.

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	×	×	×	×	
Unsigned integer	×	×	×	×	
Binary coded decimal integer					
Floating point			×	×	×

Figure 5-7. The UltraSPARC II numeric data types.

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	×	×	×	×	
Unsigned integer					
Binary coded decimal integer					
Floating point			×	×	

Figure 5-8. The JVM numeric data types.

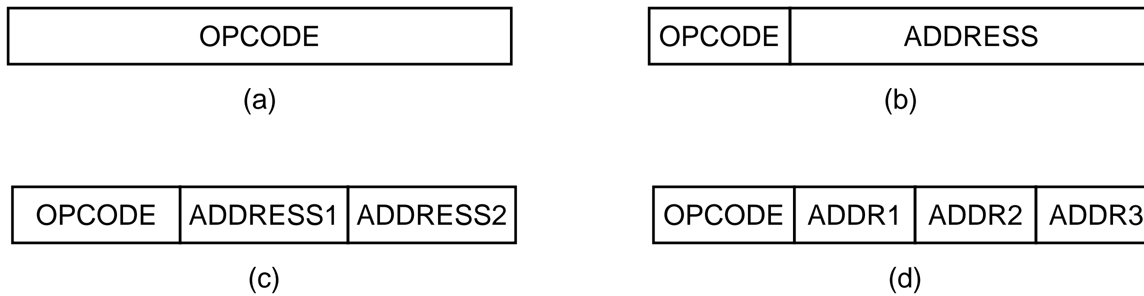

Figure 5-9. Four common instruction formats: (a) Zero-address instruction. (b) One-address instruction (c) Two-address instruction. (d) Three-address instruction.

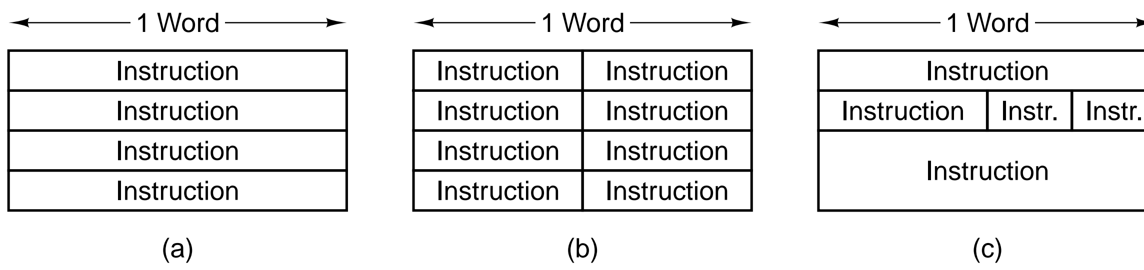

Figure 5-10. Some possible relationships between instruction and word length.

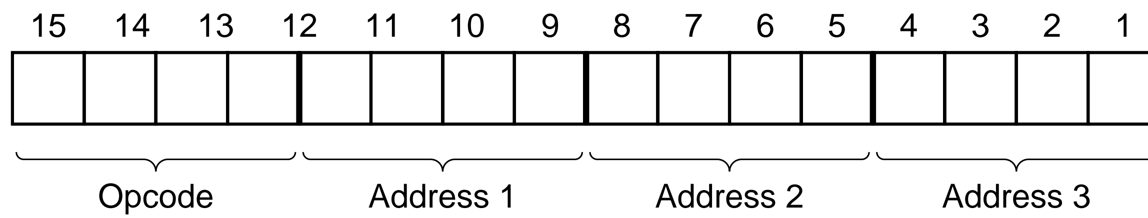

Figure 5-11. An instruction with a 4-bit opcode and three 4-bit address fields.

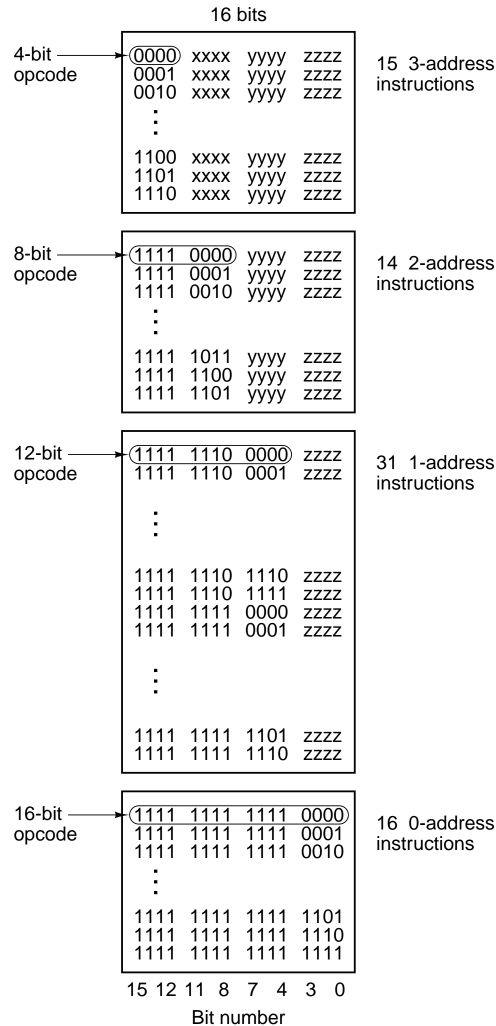

Figure 5-12. An expanding opcode allowing 15 three-address instructions, 14 two-address instructions, 31 one-address instructions, and 16 zero-address instructions. The fields marked *xxxx*, *yyyy*, and *zzzz* are 4-bit address fields.

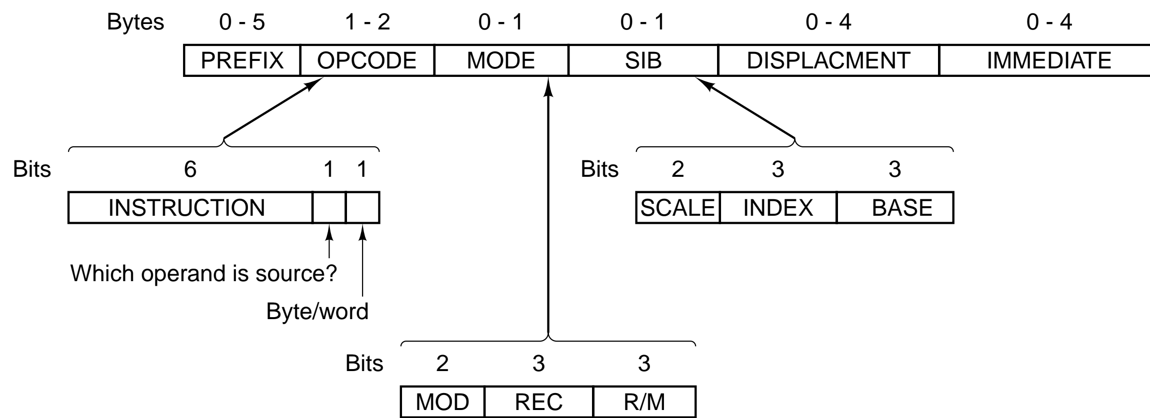

Figure 5-13. The Pentium II instruction formats.

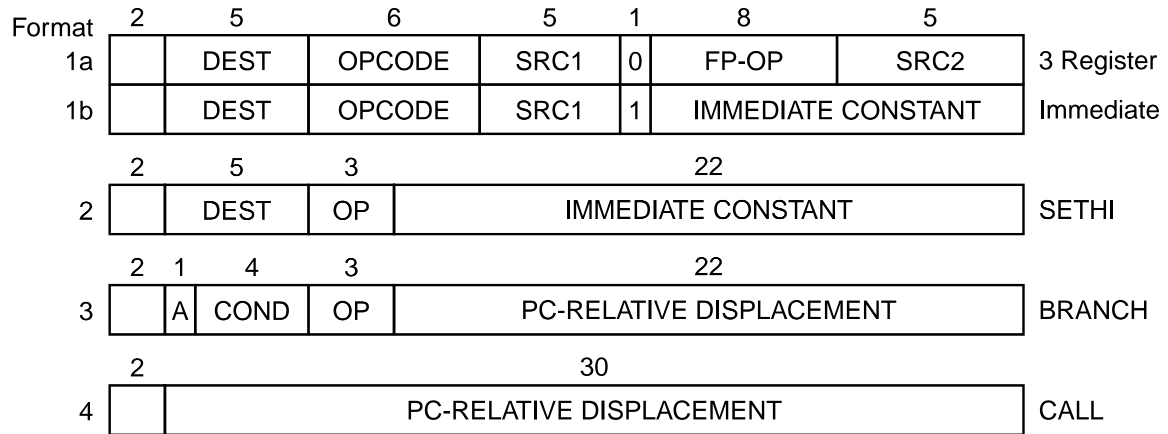

Figure 5-14. The original SPARC instruction formats.

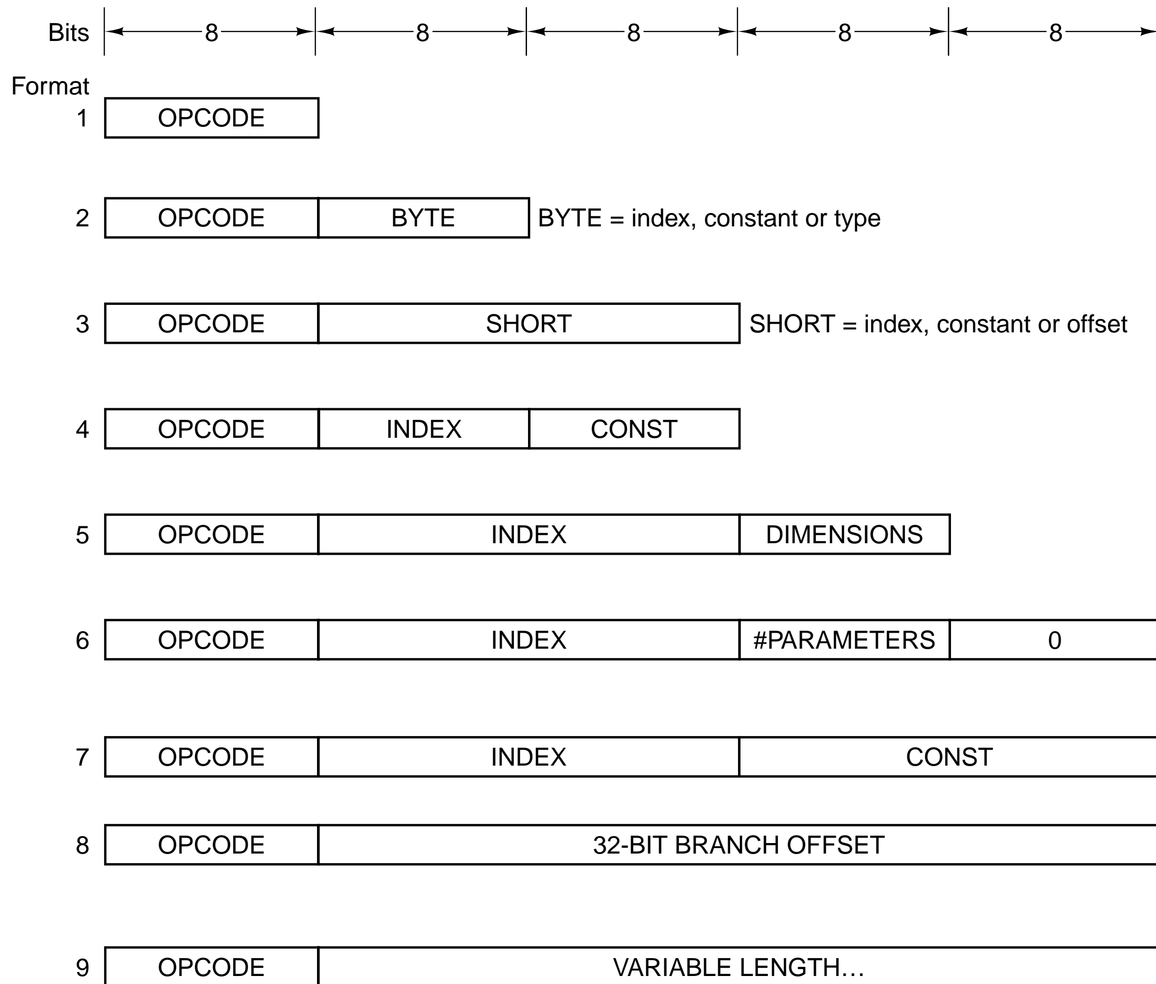

Figure 5-15. The JVM instruction formats.

MOV	R1	4

Figure 5-16. An immediate instruction for loading 4 into register 1.

```
        MOV R1,#0      ; accumulate the sum in R1, initially 0
        MOV R2,#A      ; R2 = address of the array A
        MOV R3,#A+1024; R3 = address if the first word beyond A
LOOP:                  ADD R1,(R2); register indirect through R2 to get operand
        ADD R2,#4      ; increment R2 by one word (4 bytes)
        CMP R2,R3      ; are we done yet?
        BLT LOOP       ; if R2 < R3, we are not done, so continue
```

Figure 5-17. A generic assembly program for computing the sum of the elements of an array.

```
        MOV R1,#0       ; accumulate the OR in R1, initially 0
        MOV R2,#0       ; R2 = index, i, of current product: A[i] AND B[i]
        MOV R3,#4096; R3 = first index value not to use
LOOP:                   MOV R4,A(R2); R4 = A[i]
        AND R4,B(R2); R4 = A[i] AND B[i]
        OR R1,R4        ; OR all the Boolean products into R1
        ADD R2,#4       ; i = i + 4 (step in units of 1 word = 4 bytes)
        CMP R2,R3       ; are we done yet?
        BLT LOOP        ; if R2 < R3, we are not done, so continue
```

Figure 5-18. A generic assembly program for computing the OR of A_i AND B_i for two 1024-element arrays.

MOV	R4	R2	124300

Figure 5-19. A possible representation of MOV R4,A(R2).

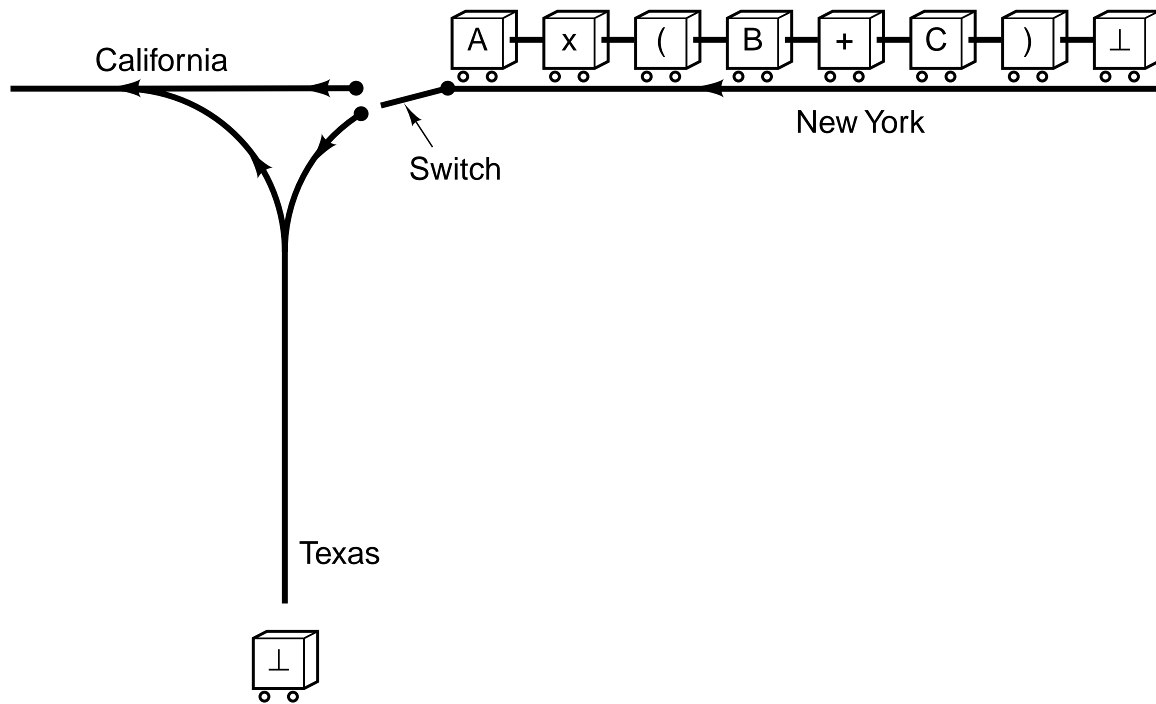

Figure 5-20. Each railroad car represents one symbol in the formula to be converted from infix to reverse Polish notation.

Car at the switch

Most recently arrived car on the Texas line

	⊥	+	–	x	/	(	)
⊥	4	1	1	1	1	1	5
+	2	2	2	1	1	1	2
–	2	2	2	1	1	1	2
x	2	2	2	2	2	1	2
/	2	2	2	2	2	1	2
(	5	1	1	1	1	1	3

Figure 5-21. Decision table used by the infix-to-reverse Polish notation algorithm

Infix	Reverse Polish notation
A + B × C	A B C × +
A × B + C	A B × C +
A × B + C × D	A B × C D × +
(A + B) / (C – D)	A B + C D – /
A × B / C	A B × C /
((A + B) × C + D)/(E + F + G)	A B + C × D + E F + G + /

Figure 5-22. Some examples of infix expressions and their reverse Polish notation equivalents.

Step	Remaining string	Instruction	Stack
1	8 2 5 × + 1 3 2 × + 4 – /	BIPUSH 8	8
2	2 5 × + 1 3 2 × + 4 – /	BIPUSH 2	8, 2
3	5 × + 1 3 2 × + 4 – /	BIPUSH 5	8, 2, 5
4	× + 1 3 2 × + 4 – /	IMUL	8, 10
5	+ 1 3 2 × + 4 – /	IADD	18
6	1 3 2 × + 4 – /	BIPUSH 1	18, 1
7	3 2 × + 4 – /	BIPUSH 3	18, 1, 3
8	2 × + 4 – /	BIPUSH 2	18, 1, 3, 2
9	× + 4 – /	IMUL	18, 1, 6
10	+ 4 – /	IADD	18, 7
11	4 – /	BIPUSH 4	18, 7, 4
12	– /	ISUB	18, 3
13	/	IDIV	6

Figure 5-23. Use of a stack to evaluate a reverse Polish notation formula.

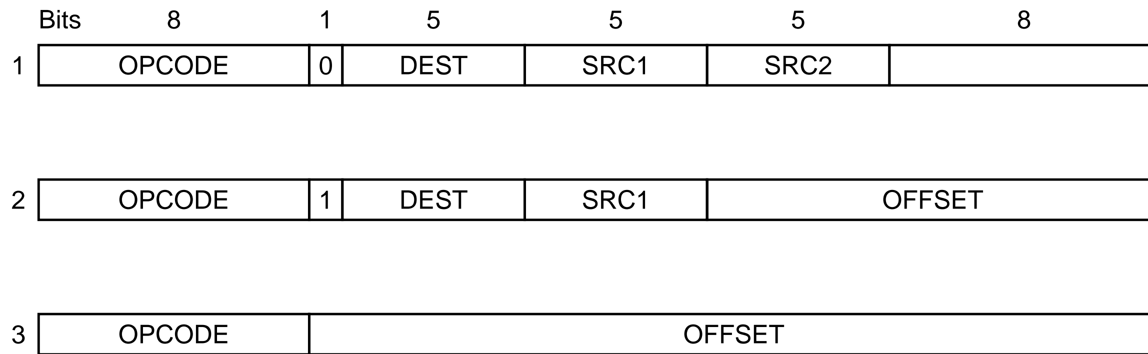

Figure 5-24. A simple design for the instruction formats of a three-address machine.

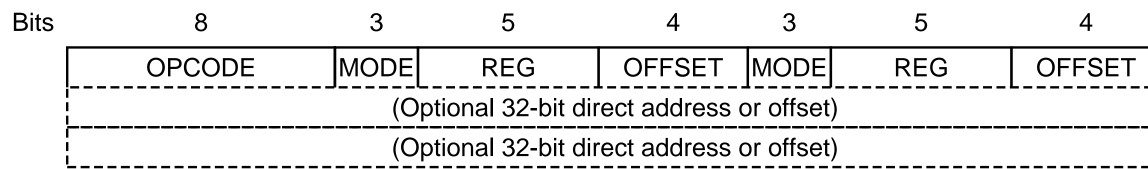

Figure 5-25. A simple design for the instruction formats of a two-address machine.

	MOD			
R/M	**00**	**01**	**10**	**11**
000	M[EAX]	M[EAX + OFFSET8]	M[EAX + OFFSET32]	EAX or AL
001	M[ECX]	M[ECX + OFFSET8]	M[ECX + OFFSET32]	ECX or CL
010	M[EDX]	M[EDX + OFFSET8]	M[EDX + OFFSET32]	EDX or DL
011	M[EBX]	M[EBX + OFFSET8]	M[EBX + OFFSET32]	EBX or BL
100	SIB	SIB with OFFSET8	SIB with OFFSET32	ESP or AH
101	Direct	M[EBP + OFFSET8]	M[EBP + OFFSET32]	EBP or CH
110	M[ESI]	M[ESI + OFFSET8]	M[ESI + OFFSET32]	ESI or DH
111	M[EDI]	M[EDI + OFFSET8]	M[EDI + OFFSET32]	EDI or BH

Figure 5-26. The Pentium II 32-bit addressing modes. M[x] is the memory word at x.

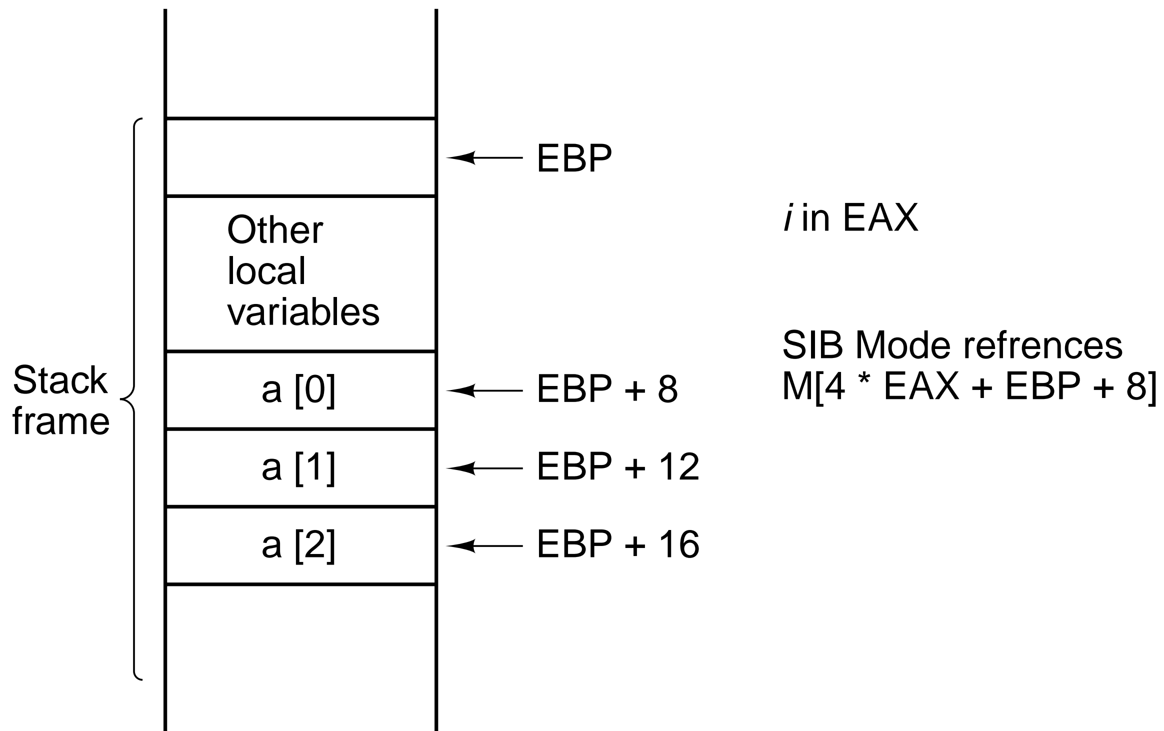

Figure 5-27. Access to *a*[*i*].

Addressing mode	Pentium II	UltraSPARC II	JVM
Immediate	×	×	×
Direct	×		
Register	×	×	
Register indirect	×		
Indexed	×	×	×
Based-indexed		×	
Stack			×

Figure 5-28. A comparison of addressing modes.

```
    i = 1;

L1: first-statement;
    .
    .
    .
    last-statement;
    i = i + 1;
    if (i < n) goto L1;
```

(a)

```
    i = 1;
L1: if (i > n) goto L2;
    first-statement;
    .
    .
    .
    last-statement
    i = i + 1;
    goto L1;
L2:
```

(b)

Figure 5-29. (a) Test-at-the-end loop. (b) Test-at-the-beginning loop.

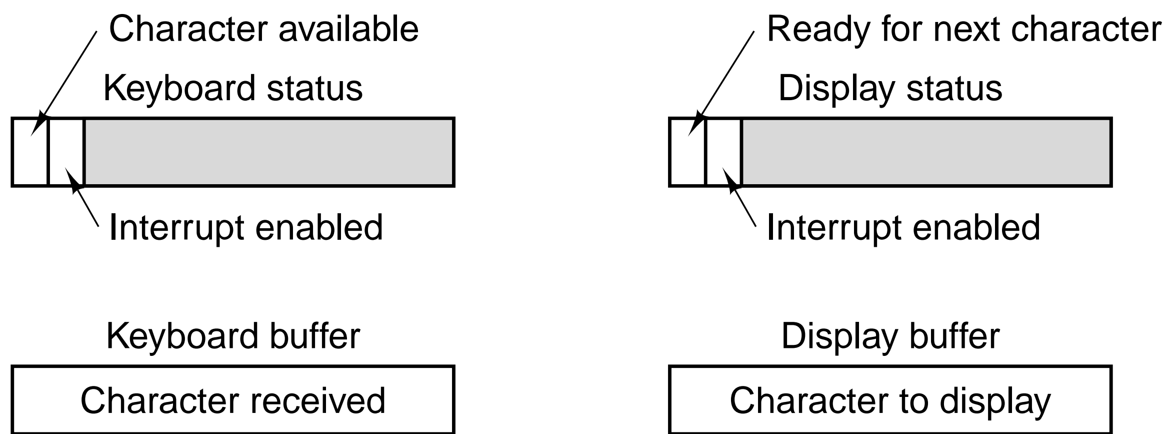

Figure 5-30. Device registers for a simple terminal.

```
public static void output_buffer(int buf[ ], int count) {
    // Output a block of data to the device
    int status, i, ready;

    for (i = 0; i < count; i++) {
        do {
            status = in(display_status_reg);// get status
            ready = (status << 7) & 0x01;// isolate ready bit
        } while (ready == 1);
        out(display_buffer_reg, buf[i]);
    }
}
```

Figure 5-31. An example of programmed I/O.

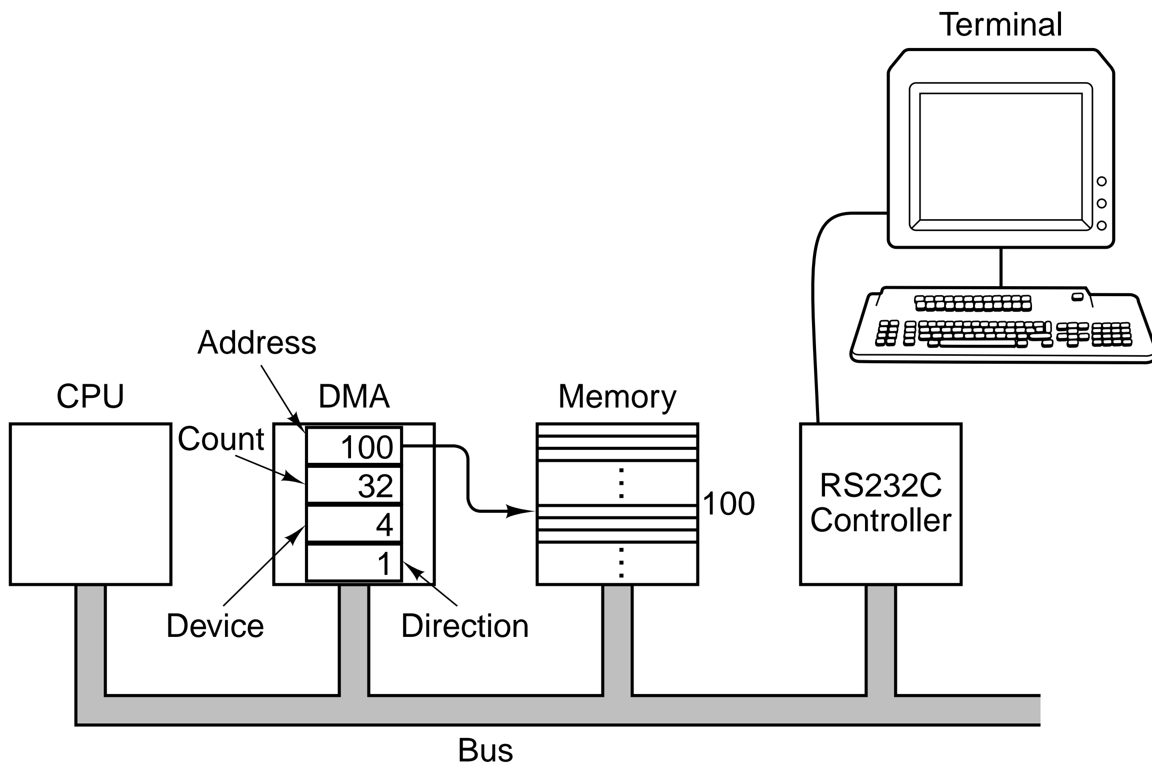

Figure 5-32. A system with a DMA controller.

Moves

MOV DST,SRC	Move SRC to DST
PUSH SRC	Push SRC onto the stack
POP DST	Pop a word from the stack to DST
XCHG DS1,DS2	Exchange DS1 and DS2
LEA DST,SRC	Load effective addr of SRC into DST
CMOV DST,SRC	Conditional move

Arithmetic

ADD DST,SRC	Add SRC to DST
SUB DST,SRC	Subtract DST from SRC
MUL SRC	Multiply EAX by SRC (unsigned)
IMUL SRC	Multiply EAX by SRC (signed)
DIV SRC	Divide EDX:EAX by SRC (unsigned)
IDIV SRC	Divide EDX:EAX by SRC (signed)
ADC DST,SRC	Add SRC to DST, then add carry bit
SBB DST,SRC	Subtract DST & carry from SRC
INC DST	Add 1 to DST
DEC DST	Subtract 1 from DST
NEG DST	Negate DST (subtract it from 0)

Binary coded decimal

DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

Boolean

AND DST,SRC	Boolean AND SRC into DST
OR DST,SRC	Boolean OR SRC into DST
XOR DST,SRC	Boolean Exclusive OR SRC to DST
NOT DST	Replace DST with 1's complement

Shift/rotate

SAL/SAR DST,#	Shift DST left/right # bits
SHL/SHR DST,#	Logical shift DST left/right # bits
ROL/ROR DST,#	Rotate DST left/right # bits
RCL/RCR DST,#	Rotate DST through carry # bits

Test/compare

TST SRC1,SRC2	Boolean AND operands, set flags
CMP SRC1,SRC2	Set flags based on SRC1 - SRC2

Transfer of control

JMP ADDR	Jump to ADDR
Jxx ADDR	Conditional jumps based on flags
CALL ADDR	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT ADDR	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

Strings

LODS	Load string
STOS	Store string
MOVS	Move string
CMPS	Compare two strings
SCAS	Scan Strings

Condition codes

STC	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS reg
STI	Set interrupt bit in EFLAGS register
CLI	Clear interrupt bit in EFLAGS reg
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
SAHF	Store AH in EFLAGS register

Miscellaneous

SWAP DST	Change endianness of DST
CWQ	Extend EAX to EDX:EAX for division
CWDE	Extend 16-bit number in AX to EAX
ENTER SIZE,LV	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN AL,PORT	Input a byte from PORT to AL
OUT PORT,AL	Output a byte from AL to PORT
WAIT	Wait for an interrupt

SRC = source # = shift/rotate count
DST = destination LV = # locals

Figure 5-33. A selection of the Pentium II integer instructions.

Loads

LDSB ADDR,DST	Load signed byte (8 bits)
LDUB ADDR,DST	Load unsigned byte (8 bits)
LDSH ADDR,DST	Load signed halfword (16 bits)
LDUH ADDR,DST	Load unsigned halfword (16)
LDSW ADDR,DST	Load signed word (32 bits)
LDUW ADDR,DST	Load unsigned word (32 bits)
LDX ADDR,DST	Load extended (64-bits)

Stores

STB SRC,ADDR	Store byte (8 bits)
STH SRC,ADDR	Store halfword (16 bits)
STW SRC,ADDR	Store word (32 bits)
STX SRC,ADDR	Store extended (64 btis)

Arithmetic

ADD R1,S2,DST	Add
ADDCC "	Add and set icc
ADDC "	Add with carry
ADDCCC "	Add with carry and set icc
SUB R1,S2,DST	Subtract
SUBCC "	Subtract and set icc
SUBC "	Subtract with carry
SUBCCC "	Subtract with carry and set icc
MULX R1,S2,DST	Multiply
SDIVX R1,S2,DST	Signed divide
UDIVX R1,S2,DST	Unsigned divide
TADCC R1,S2,DST	Tagged add

Shifts/rotates

SLL R1,S2,DST	Shift left logical (64 bits)
SLLX R1,S2,DST	Shift left logical extended (64)
SRL R1,S2,DST	Shift right logical (32 bits)
SRLX R1,S2,DST	Shift right logical extended (64)
SRA R1,S2,DST	Shift right arithmetic (32 bits)
SRAX R1,S2,DST	Shift right arithmetic ext. (64)

Boolean

AND R1,S2,DST	Boolean AND
ANDCC "	Boolean AND and set icc
ANDN "	Boolean NAND
ANDNCC "	Boolean NAND and set icc
OR R1,S2,DST	Boolean OR
ORCC "	Boolean OR and set icc
ORN "	Boolean NOR
ORNCC "	Boolean NOR and set icc
XOR R1,S2,DST	Boolean XOR
XORCC "	Boolean XOR and set icc
XNOR "	Boolean EXCLUSIVE NOR
XNORCC "	Boolean EXCL. NOR and set icc

Transfer of control

BPcc ADDR	Branch with prediction
BPr SRC,ADDR	Branch on register
CALL ADDR	Call procedure
RETURN ADDR	Return from procedure
JMPL ADDR,DST	Jump and Link
SAVE R1,S2,DST	Advance register windows
RESTORE "	Restore register windows
Tcc CC,TRAP#	Trap on condition
PREFETCH FCN	Prefetch data from memory
LDSTUB ADDR,R	Atomic load/store
MEMBAR MASK	Memory barrier

Miscellaneous

SETHI CON,DST	Set bits 10 to 31
MOVcc CC,S2,DST	Move on condition
MOVr R1,S2,DST	Move on register
NOP	No operation
POPC S1,DST	Population count
RDCCR V,DST	Read condition code register
WRCCR R1,S2,V	Write condition code register
RDPC V,DST	Read program counter

SRC = source register
DST = destination register
R1 = source register
S2 = source: register or immediate
ADDR = memory address

TRAP# = trap number
FCN = function code
MASK = operation type
CON = constant
V = register designator

CC = condition code set
R =destination register

cc = condition
r = LZ,LEZ,Z,NZ,GZ,GEZ

Figure 5-34. The primary UltraSPARC II integer instructions.

Instruction	How to do it
MOV SRC,DST	OR SRC with G0 and store the result DST
CMP SRC1,SRC2	SUBCC SRC2 from SRC1 and store the result in G0
TST SRC	ORCC SRC1 with G0 and store the result in G0
NOT DST	XNOR DST with G0
NEG DST	SUB DST from G0 and store in DST
INC DST	ADD 1 to DST (immediate operand)
DEC DST	SUB 1 from DST (immediate operand)
CLR DST	OR G0 with G0 and store in DST
NOP	SETHI G0 to 0
RET	JMPL %I7+8,%G0

Figure 5-35. Some simulated UltraSPARC II instructions.

Loads

typeLOAD IND8	Push local variable onto stack
typeALOAD	Push array element on stack
BALOAD	Push byte from an array on stack
SALOAD	Push short from an array on stack
CALOAD	Push char from an array on stack
AALOAD	Push pointer from an array on "

Stores

typeSTORE IND8	Pop value and store in local var
typeASTORE	Pop value and store in array
BASTORE	Pop byte and store in array
SASTORE	Pop short and store in array
CASTORE	Pop char and store in array
AASTORE	Pop pointer and store in array

Pushes

BIPUSH CON8	Push a small constant on stack
SIPUSH CON16	Push 16-bit constant on stack
LDC IND8	Push constant from const pool
typeCONST_#	Push immediate constant
ACONST_NULL	Push a null pointer on stack

Arithmetic

typeADD	Add
typeSUB	Subtract
typeMUL	Multiple
typeDIV	Divide
typeREM	Remainder
typeNEG	Negate

Boolean/shift

ilAND	Boolean AND
ilOR	Boolean OR
ilXOR	Boolean EXCLUSIVE OR
ilSHL	Shift left
ilSHR	Shift right
ilUSHR	Unsigned shift right

Conversion

x2y	Convert x to y
i2c	Convert integer to char
i2b	Convert integer to byte

Stack management

DUPxx	Six instructions for duping
POP	Pop an int from stk and discard
POP2	Pop two ints from stk and discard
SWAP	Swap top two ints on stack

Comparison

IF_ICMPrel OFFSET16	Conditional branch
IF_ACMPEQ OFFSET16	Branch if two ptrs equal
IF_ACMPNE OFFSET16	Branch if ptrs unequal
IFrel OFFSET16	Test 1 value and branch
IFNULL OFFSET16	Branch if ptr is null
IFNONNULL OFFSET16	Branch if ptr is nonnull
LCMP	Compare two longs
FCMPL	Compare 2 floats for <
FCMPG	Compare 2 floats for >
DCMPL	Compare doubles for <
DCMPG	Compare doubles for >

Transfer of control

INVOKEVIRTUAL IND16	Method invocation
INVOKESTATIC IND16	Method invocation
INVOKEINTERFACE ...	Method invocation
INVOKESPECIAL IND16	Method invocation
JSR OFFSET16	Invoke finally clause
typeRETURN	Return value
ARETURN	Return pointer
RETURN	Return void
RET IND8	Return from finally
GOTO OFFSET16	Unconditional branch

Arrays

ANEWARRAY IND16	Create array of ptrs
NEWARRAY ATYPE	Create array of atype
MULTINEWARRAY IN16,D	Create multidim array
ARRAYLENGTH	Get array length

Miscellaneous

IINC IND8,CON8	Increment local variable
WIDE	Wide prefix
NOP	No operation
GETFIELD IND16	Read field from object
PUTFIELD IND16	Write field to object
GETSTATIC IND16	Get static field from class
NEW IND16	Create a new object
INSTANCEOF OFFSET16	Determine type of obj
CHECKCAST IND16	Check object type
ATHROW	Throw exception
LOOKUPSWITCH ...	Sparse multiway branch
TABLESWITCH ...	Dense multiway branch
MONITORENTER	Enter a monitor
MONITOREXIT	Leave a monitor

IND8/16 = index of local variable
CON8/16, D, ATYPE = constant

type, x, y = I, L, F, D
OFFSET16 for branch

Figure 5-36. The JVM instruction set.

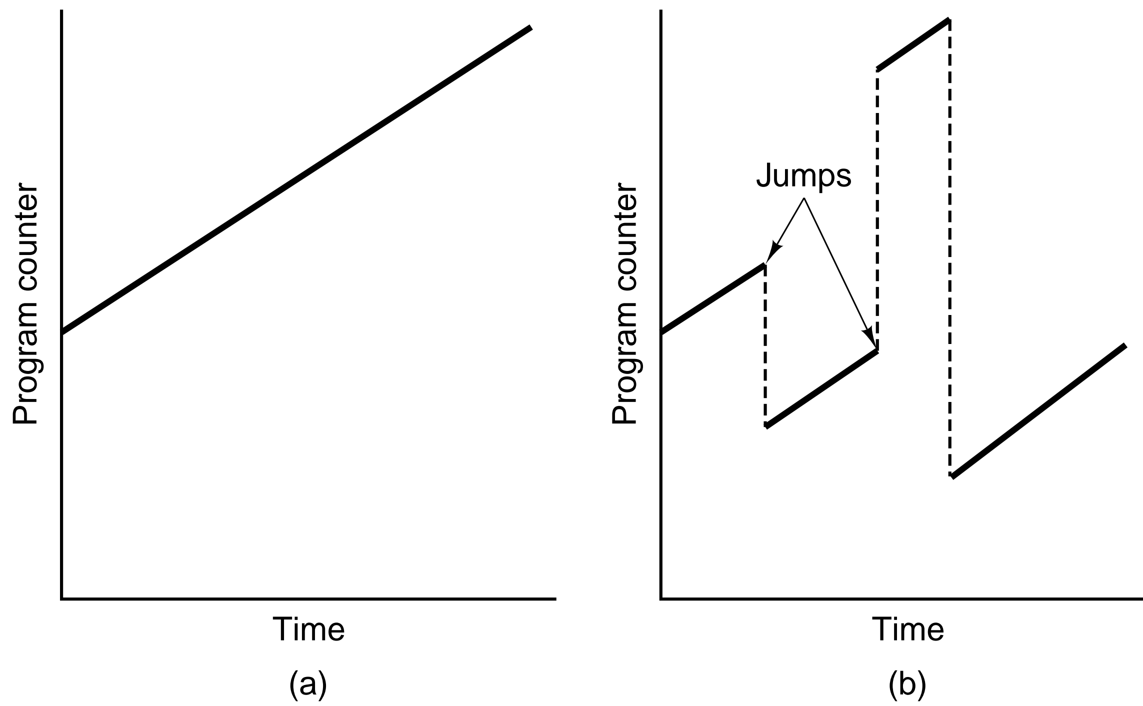

Figure 5-37. Program counter as a function of time (smoothed). (a) Without branches. (b) With branches.

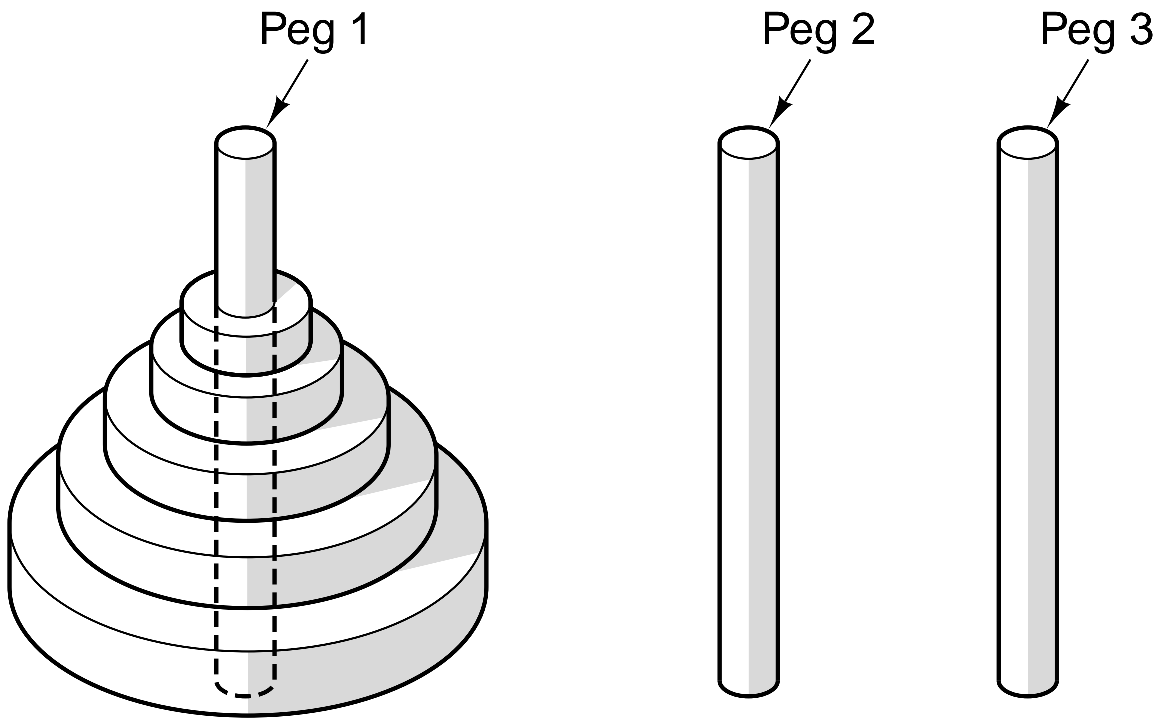

Figure 5-38. Initial configuration for the Towers of Hanoi problem for five disks.

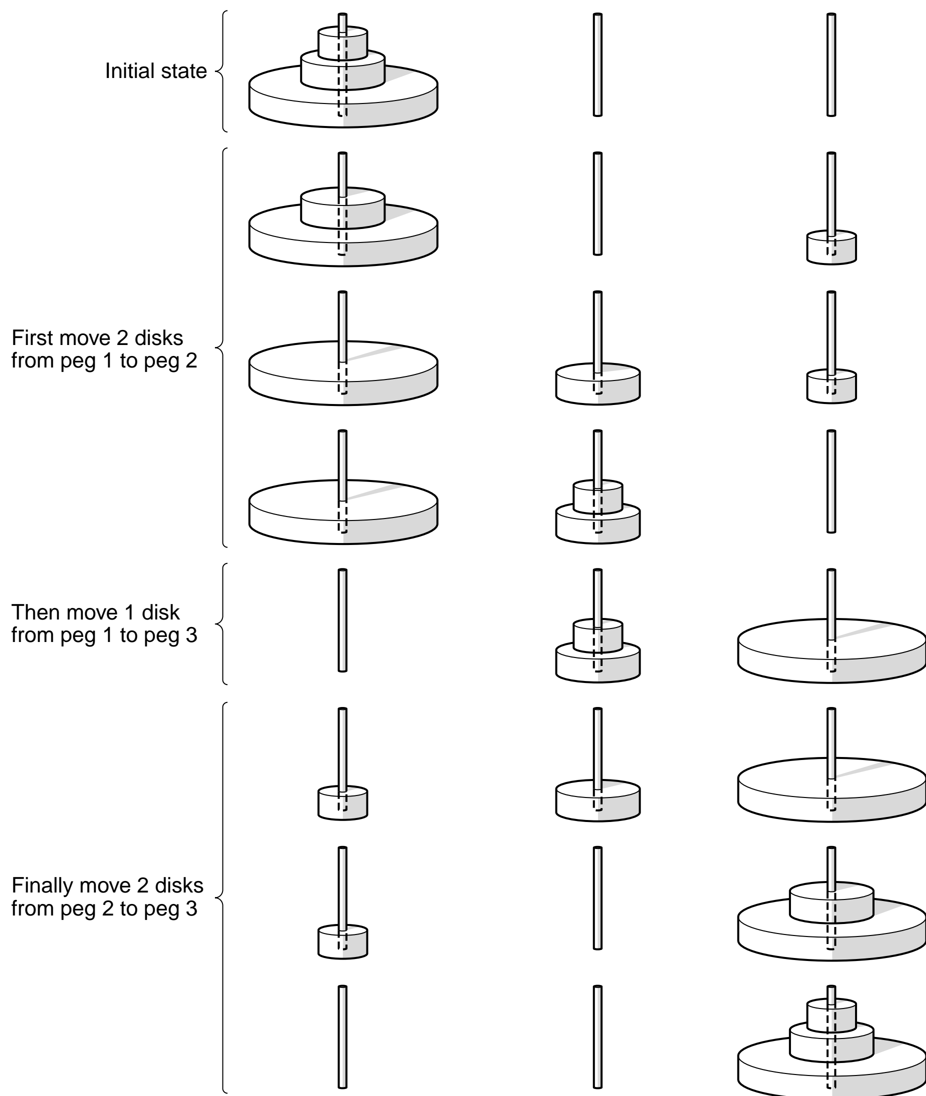

Figure 5-39. The steps required to solve the Towers of Hanoi for three disks.

```
public void towers(int n, int i, int j) {
   int k;

   if (n == 1)
      System.out.println("Move a disk from " + i + " to " + j);
   else {
      k = 6 – i – j;
      towers(n – 1, i, k);
      towers(1, i, j);
      towers(n – 1, k, j);
  }
}
```

Figure 5-40. A procedure for solving the Towers of Hanoi.

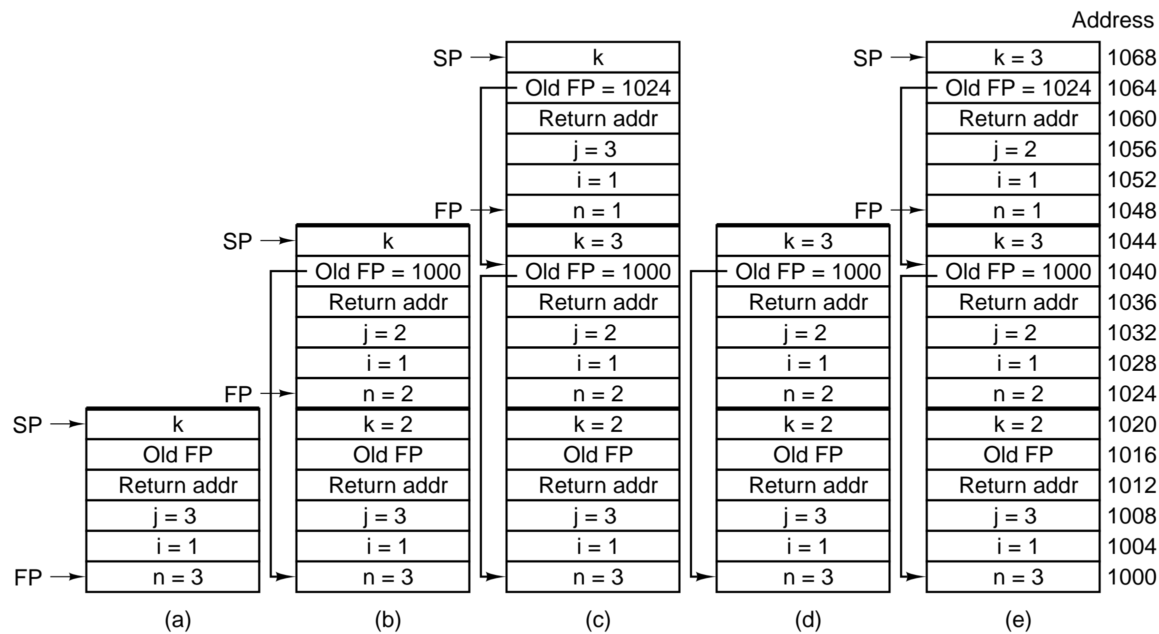

Figure 5-41. The stack at several points during the execution of Fig. 5-40.

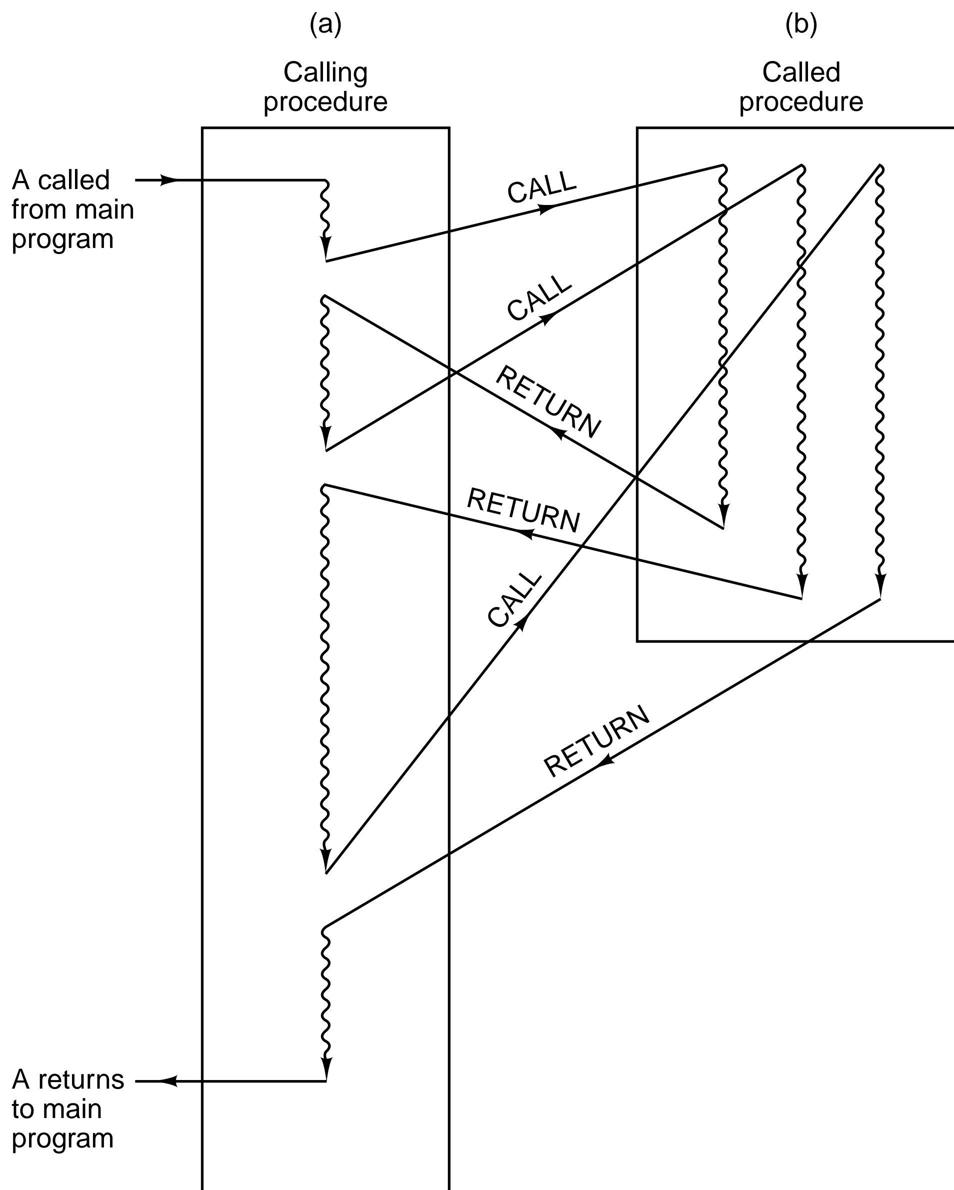

Figure 5-42. When a procedure is called, execution of the procedure always begins at the first statement of the procedure.

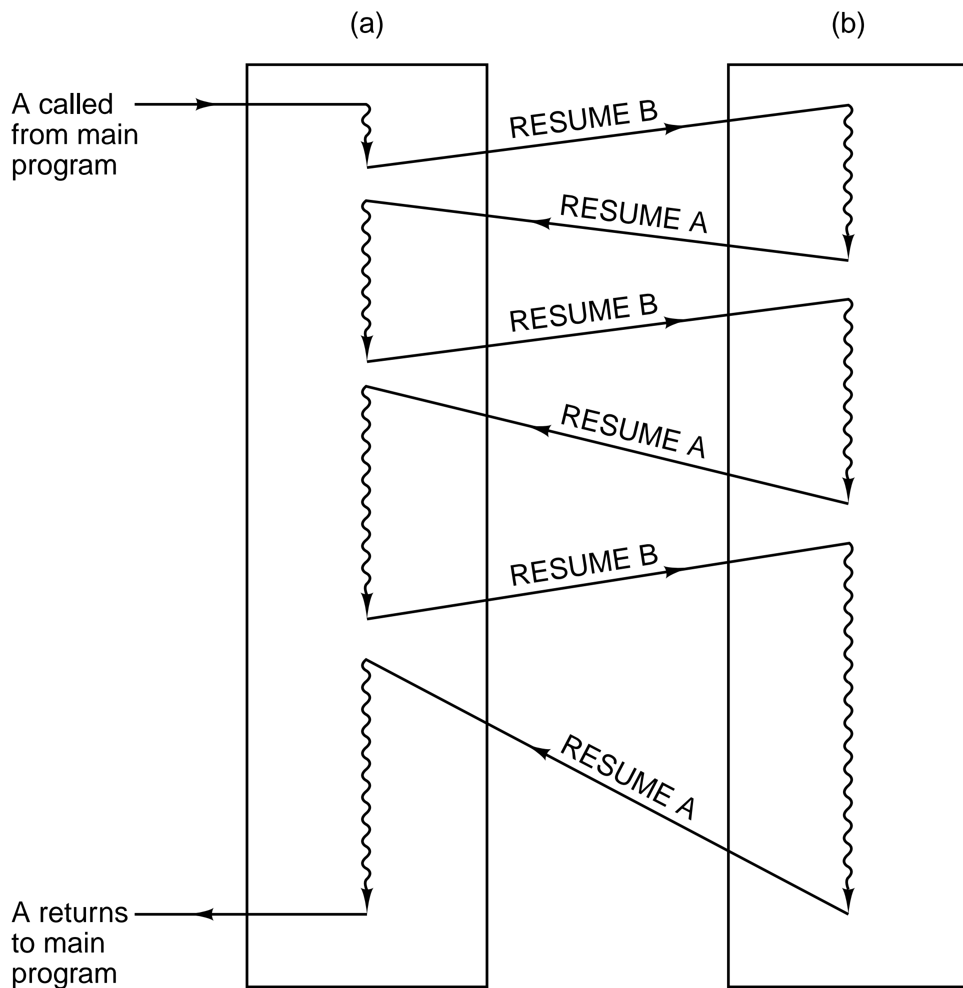

Figure 5-43. When a coroutine is resumed, execution begins at the statement where it left off the previous time, not at the beginning.

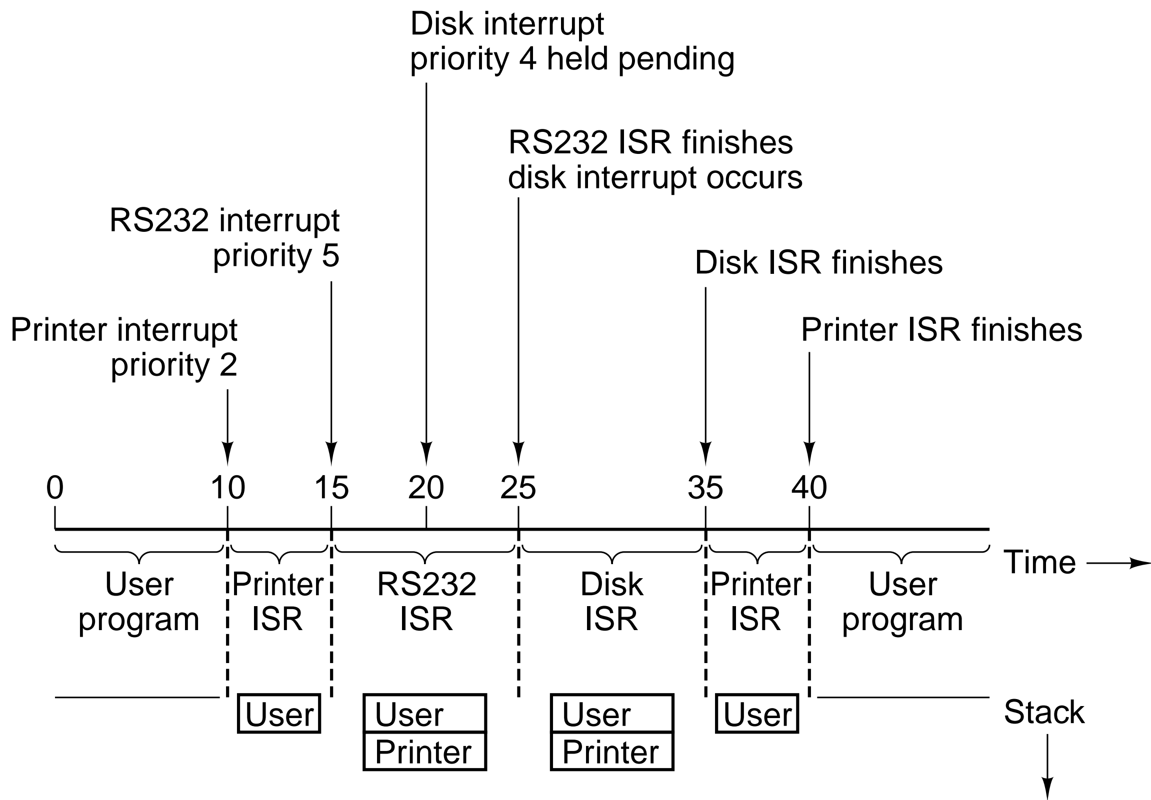

Figure 5-44. Time sequence of multiple interrupt example.

```
        .586                          ; compile for Pentium (as opposed to 8088 etc.)
.MODEL FLAT
PUBLIC _towers                        ; export 'towers'
EXTERN _printf:NEAR                   ; import printf
.CODE
 _towers:                             PUSH EBP; save EBP (frame pointer)
        MOV EBP, ESP                  ; set new frame pointer above ESP
        CMP [EBP+8], 1                ; if (n == 1)
        JNE L1                        ; branch if n is not 1
        MOV EAX, [EBP+16]             ; printf(" ...", i, j);
        PUSH EAX                      ; note that parameters i, j and the format
        MOV EAX, [EBP+12]             ; string are pushed onto the stack
        PUSH EAX                      ; in reverse order. This is the C calling convention
        PUSH OFFSET FLAT:format; offset flat means the address of format
        CALL _printf                  ; call printf
        ADD ESP, 12                   ; remove params from the stack
        JMP Done                      ; we are finished
  L1:   MOV EAX, 6                    ; start k = 6 – i – j
        SUB EAX, [EBP+12]             ; EAX = 6 – i
        SUB EAX, [EBP+16]             ; EAX = 6 – i – j
        MOV [EBP+20], EAX             ; k = EAX
        PUSH EAX                      ; start towers(n – 1, i, k)
        MOV EAX, [EBP+12]             ; EAX = i
        PUSH EAX                      ; push i
        MOV EAX, [EBP+8]              ; EAX = n
        DEC EAX                       ; EAX = n – 1
        PUSH EAX                      ; push n – 1
        CALL _towers                  ; call towers(n – 1, i, 6 – i – j)
        ADD ESP, 12                   ; remove params from the stack
        MOV EAX, [EBP+16]             ; start towers(1, i, j)
        PUSH EAX                      ; push j
        MOV EAX, [EBP+12]             ; EAX = i
        PUSH EAX                      ; push i
        PUSH 1                        ; push 1
        CALL _towers                  ; call towers(1, i, j)
        ADD ESP, 12                   ; remove params from the stack
        MOV EAX, [EBP+12]             ; start towers(n – 1, 6 – i – j, i)
        PUSH EAX                      ; push i
        MOV EAX, [EBP+20]             ; push 20
        PUSH EAX                      ; push k
        MOV EAX, [EBP+8]              ; EAX = n
        DEC EAX                       ; EAX = n–1
        PUSH EAX                      ; push n – 1
        CALL _towers                  ; call towers(n – 1, 6 – i – j, i)
        ADD ESP, 12                   ; adjust stack pointer
 Done:  LEAVE                         ; prepare to exit
        RET 0                         ; return to the caller
.DATA
format  DB "Move disk from %d to %d\n"; format string
END
```

Figure 5-45. The Towers of Hanoi for the Pentium II.

```
#define N %i0                    /* N is input parameter 0 */
#define I %i1                    /* I is input parameter 1 */
#define J %i2                    /* J is input parameter 2 */
#define K %l0                    /* K is local variable 0 */
#define Param0 %o0               /* Param0 is output parameter 0 */
#define Param1 %o1               /* Param1 is output parameter 1 */
#define Param2 %o2               /* Param2 is output parameter 2 */
#define Scratch %l1              /* as an aside, cpp uses the C comment convention */
        .proc 04
        .global towers

towers:                          save %sp, –112, %sp
        cmp N, 1                 ! if (n == 1)
        bne Else                 ! if (n != 1) goto Else

        sethi %hi(format), Param0 ! printf("Move a disk from %d to %d\n", i, j)
        or Param0, %lo(format), Param0! Param0 = address of format string
        mov I, Param1            ! Param1 = i
        call printf              ! call printf BEFORE parameter 2 (j) is set up
        mov J, Param2            ! use the delay slot after call to set up parameter 2
        b Done                   ! we are done now
        nop                      ! fill delay slot


Else:   mov 6, K                 ! start k = 6 –i – j
        sub K, J, K              ! k = 6 – j
        sub K, I, K              ! k = 6 – i – j

        add N, –1, Scratch       ! start towers(n – 1, i, k)
        mov Scratch, Param0      ! Scratch = N – 1
        mov I, Param1            ! parameter 1 = i
        call towers              ! call towers BEFORE parameter 2 (k) is set up
        mov K, Param2            ! use the delay slot after call to set up parameter 2

        mov 1, Param0            ! start towers(1, i, j)
        mov I, Param1            ! parameter 1 = i
        call towers              ! call towers BEFORE parameter 2 (j) is set up
        mov J, Param2            ! parameter 2 = j

        mov Scratch, Param0      ! start towers(n – 1, k, j)
        mov K, Param1            ! parameter 1 = k
        call towers              ! call towers BEFORE parameter 2 (j) is set up
        mov J, Param2            ! parameter 2 = j

Done:   ret                      ! return
        restore                  ! use the delay slot after ret to restore windows

format:                          .asciz "Move a disk from %d to %d\n"
```

Figure 5-46. The Towers of Hanoi for the UltraSPARC II.

```
        ILOAD_0                 // local 0 = n; push n
        ICONST_1                // push 1
        IF_ICMPNE L1            // if (n != 1) goto L1

        GETSTATIC #13           // n == 1; this code handles the println statement
        NEW #7                  // allocate buffer for the string to be built
        DUP                     // duplicate the pointer to the buffer
        LDC #2                  // push pointer to string "move a disk from "
        INVOKESPECIAL #10       // copy the string to the buffer
        ILOAD_1                 // push i
        INVOKEVIRTUAL #11       // convert i to string and append to the new buffer
        LDC #1                  // push pointer to string " to "
        INVOKEVIRTUAL #12       // append this string to the buffer
        ILOAD_2                 // push j
        INVOKEVIRTUAL #11       // convert j to string and append to buffer
        INVOKEVIRTUAL #15       // string conversion
        INVOKEVIRTUAL #14       // call println
        RETURN                  // return from towers

L1:     BIPUSH 6                // Else part: compute k = 6 – i – j
        ILOAD_1                 // local 1 = i; push i
        ISUB                    // top-of-stack = 6 – i
        ILOAD_2                 // local 2 = j; push j
        ISUB                    // top-of-stack = 6 – i – j
        ISTORE_3                // local 3 = k = 6 – i – j; stack is now empty

        ILOAD_0                 // start working on towers(n – 1, i, k); push n
        ICONST_1                // push 1
        ISUB                    // top-of-stack = n – 1
        ILOAD_1                 // push i
        ILOAD_3                 // push k
        INVOKESTATIC #16        // call towers(n – 1, 1, k)

        ICONST_1                // start working on towers(1, i, j); push 1
        ILOAD_1                 // push i
        ILOAD_2                 // push j
        INVOKESTATIC #16        // call towers(1, i, j)

        ILOAD_0                 // start working on towers(n – 1, k, j); push n
        ICONST_1                // push 1
        ISUB                    // top-of-stack = n – 1
        ILOAD_3                 // push k
        ILOAD_2                 // push j
        INVOKESTATIC #16        // call towers(n – 1, k, j)
        RETURN                  // return from towers
```

Figure 5-47. The Towers of Hanoi for JVM.

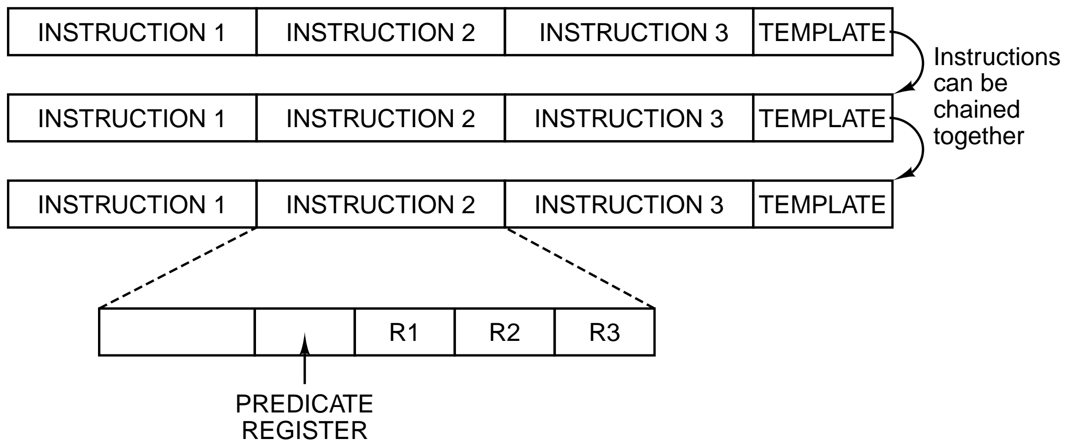

Figure 5-48. IA-64 is based on bundles of three instructions.

```
if (R1 == 0)          CMP R1,0          CMOVZ R2,R3,R1
    R2 = R3;          BNE L1
                      MOV R2,R3
                  L1:
```

(a) (b) (c)

Figure 5-49. (a) An if statement. (b) Generic assembly code for (a). (c) A conditional instruction.

```
if (R1 == 0) {
    R2 = R3;
    R4 = R5;
} else {
    R6 = R7;
    R8 = R9;
}
```

(a)

```
        CMP R1,0
        BNE L1
        MOV R2,R3
        MOV R4.R5
        BR L2
L1:     MOV R6,R7
        MOV R8,R9
L2:
```

(b)

```
CMOVZ R2,R3,R1
CMOVZ R4,R5,R1
CMOVN R6,R7,R1
CMOVN R8,R9,R1
```

(c)

Figure 5-50. (a) An if statement. (b) Generic assembly code for (a). (c) Conditional execution.

```
if (R1 == R2)
    R3 = R4 + R5;
else
    R6 = R4 – R5
```

(a)

```
     CMP R1,R2
     BNE L1
     MOV R3,R4

     BR L2
L1:  MOV R6,R4
     SUB R6,R5
L2:
```

(b)

```
CMPEQ R1,R2,P4
<P4> ADD R3,R4,R5
<P5> SUB R6,R4,R5
     ADD R3,R5
```

(c)

Figure 5-51. (a) An if statement. (b) Generic assembly code for (a). (c) Predicated execution.