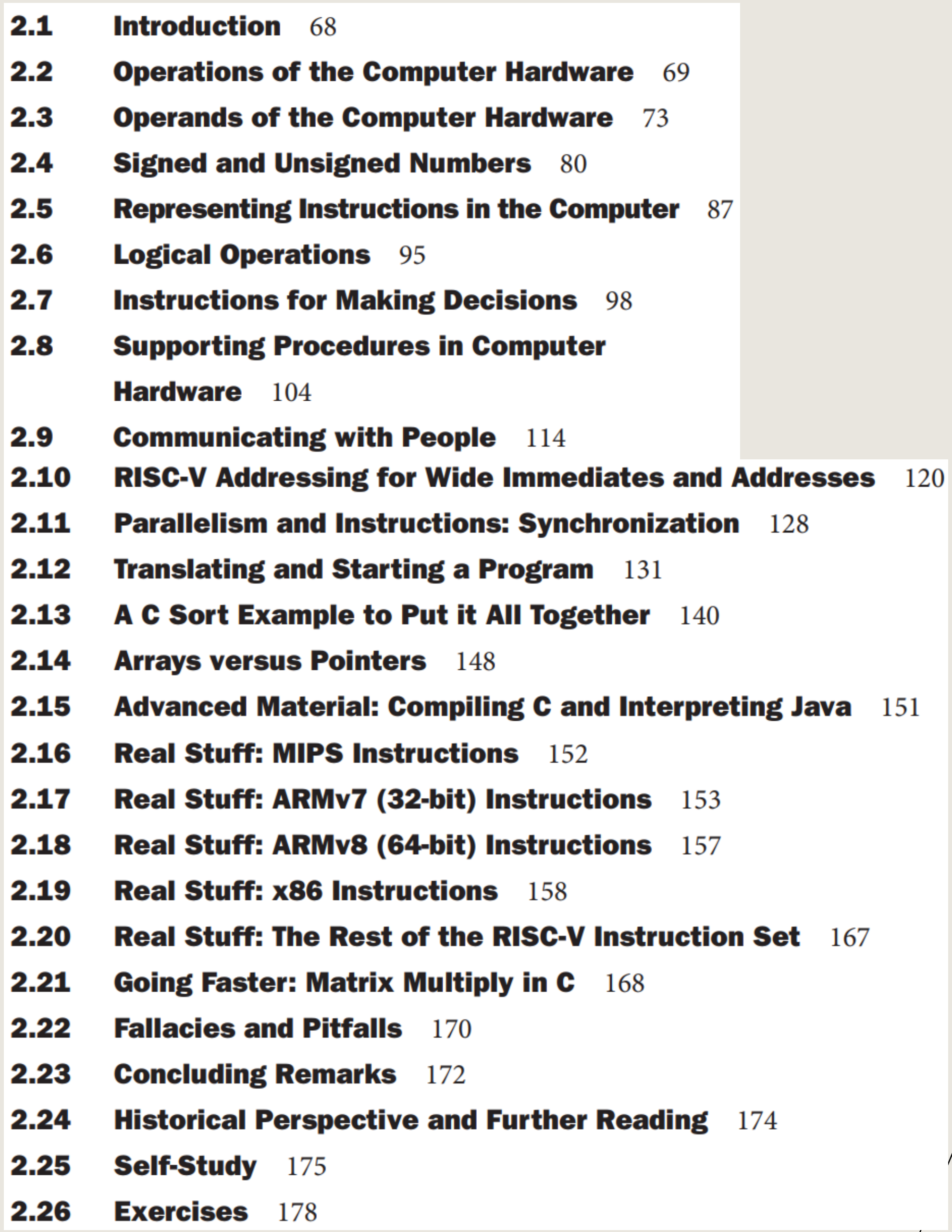
CHAPTER 2

Instructions: Language of the Computer

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University

國立成功大學
National Cheng Kung University
1931

Outline



× means not covered in this file

Instruction Set

- Instructions
  - To command a computer's hardware, one must speak its language
  - Instructions are the words of a computer's language
- An instruction set
  - The vocabulary of the language
  - Can be considered as the repertoire of instructions of a computer

- Different computers have different instruction sets
  - But with many aspects in common
  - There are a few basic operations that all computers must provide
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

The RISC-V Instruction Set

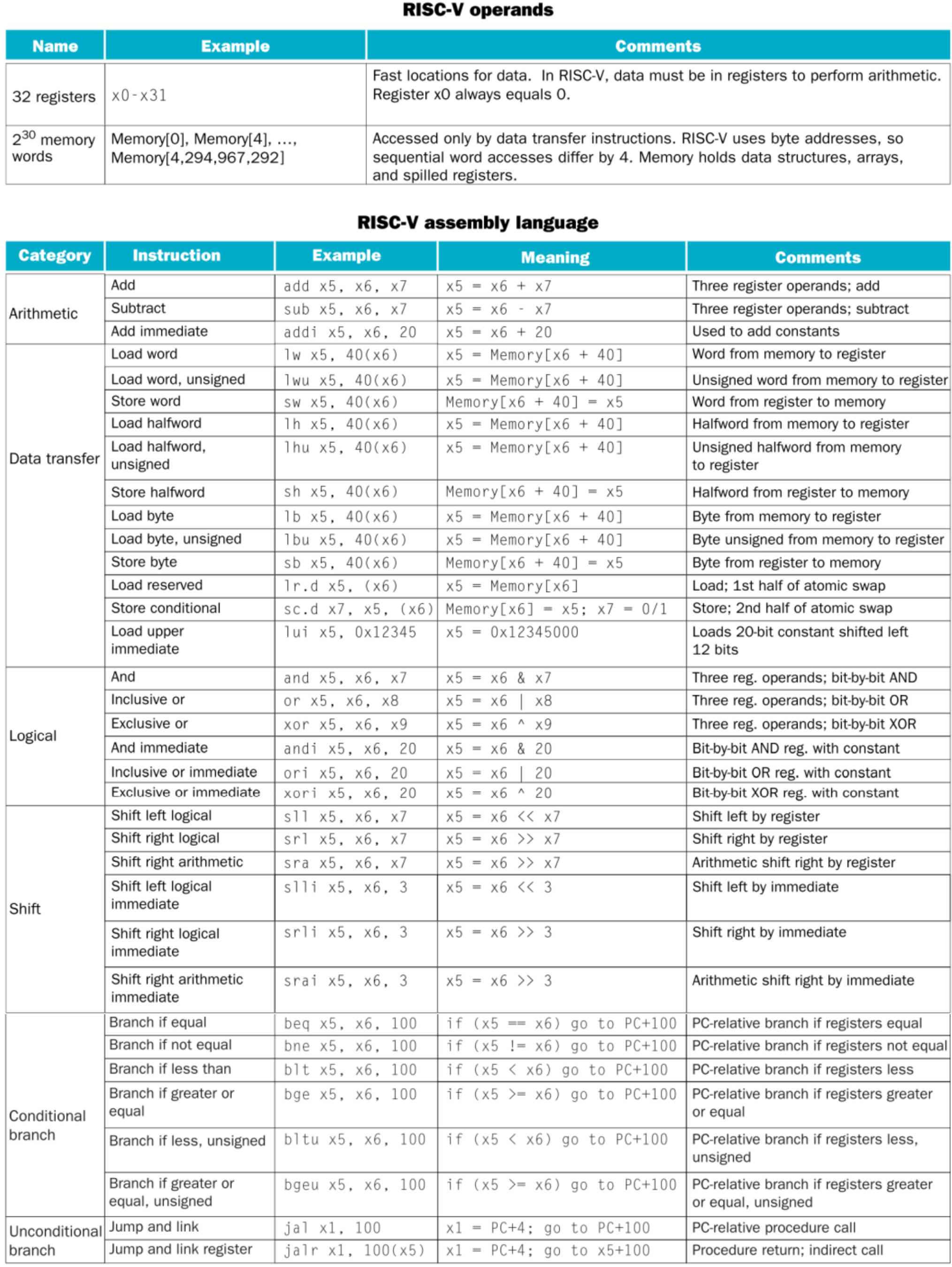
- RISC-V Instruction Set
  - ➢Is used as the example throughout this course
  - ➢Is developed at UC Berkeley as open ISA in 2010
  - ➢Now managed by the RISC-V Foundation

- Typical of many modern ISAs
  - ➢See RISC-V Reference Data card (also available at the front of the textbook) for RV64 Instructions, calling convention, opcode, etc.
- Similar ISAs have a large share of embedded core market
  - ➢Applications in consumer electronics, network/storage equipment, cameras, printers, …

RV32 Instructions

- 32 registers
  - ➢x0 ~ x31, x0 = 0
  - ➢32-bit wide registers
- Addressing 2^{30} words
  - ➢Byte addressable
  - ➢Sequential word accesses differ by 4 (2^2)
  - ➢E.g., Mem[0], Mem[4], Mem[8], … Mem[4,294,967,292]
- Instructions can be divided into six groups
  - ➢Arithmetic,
  - ➢data transfer,
  - ➢logical,
  - ➢shift,
  - ➢conditional branch, and
  - ➢unconditional branch

[Link to RISC-V ISA Manual]



RISC-V operands

Name	Example	Comments
32 registers	x0-x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4,294,967,292]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential word accesses differ by 4. Memory holds data structures, arrays, and spilled registers.

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	x5 = x6 + x7	Three register operands; add
	Subtract	sub x5, x6, x7	x5 = x6 - x7	Three register operands; subtract
	Add immediate	addi x5, x6, 20	x5 = x6 + 20	Used to add constants
Data transfer	Load word	lw x5, 40(x6)	x5 = Memory[x6 + 40]	Word from memory to register
	Load word, unsigned	lwu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned word from memory to register
	Store word	sw x5, 40(x6)	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	lh x5, 40(x6)	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	Memory[x6 + 40] = x5	Halfword from register to memory
	Load byte	lb x5, 40(x6)	x5 = Memory[x6 + 40]	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	x5 = Memory[x6 + 40]	Byte unsigned from memory to register
	Store byte	sb x5, 40(x6)	Memory[x6 + 40] = x5	Byte from register to memory
	Load reserved	lr.d x5, (x6)	x5 = Memory[x6]	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	Memory[x6] = x5; x7 = 0/1	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	x5 = 0x12345000	Loads 20-bit constant shifted left 12 bits
Logical	And	and x5, x6, x7	x5 = x6 & x7	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x8	x5 = x6 | x8	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	x5 = x6 ^ x9	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	x5 = x6 & 20	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	x5 = x6 | 20	Bit-by-bit OR reg. with constant
	Exclusive or immediate	xori x5, x6, 20	x5 = x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	sll x5, x6, x7	x5 = x6 << x7	Shift left by register
	Shift right logical	srl x5, x6, x7	x5 = x6 >> x7	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	x5 = x6 >> x7	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	x5 = x6 << 3	Shift left by immediate
	Shift right logical immediate	srli x5, x6, 3	x5 = x6 >> 3	Shift right by immediate
	Shift right arithmetic immediate	srai x5, x6, 3	x5 = x6 >> 3	Arithmetic shift right by immediate
Conditional branch	Branch if equal	beq x5, x6, 100	if (x5 == x6) go to PC+100	PC-relative branch if registers equal
	Branch if not equal	bne x5, x6, 100	if (x5 != x6) go to PC+100	PC-relative branch if registers not equal
	Branch if less than	blt x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	bge x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	bltu x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	bgeu x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	jal x1, 100	x1 = PC+4; go to PC+100	PC-relative procedure call
	Jump and link register	jalr x1, 100(x5)	x1 = PC+4; go to x5+100	Procedure return; indirect call

FIGURE 2.1 MIPS assembly language revealed in this chapter. This information is also found in Column 1 of the RISC-V Reference Data Card at the front of this book.

Arithmetic Operations

- The instruction instructs a computer to add the two variables b and c and to put their sum in a

```
add a, b, c  // a gets b + c
```

- It takes four instructions to add four variables

```
add a, b, c // The sum of b and c is placed in a
add a, a, d // The sum of b, c, and d is now in a
add a, a, e //The sum of b, c, d, and e is now in a
```

comments

- Instructions, add and subtract, each has three operands
  - ➢Two sources and one destination
- All arithmetic operations have this form

- *Design Principle 1:* Simplicity favors regularity
  - ➢Regularity makes implementation simpler
  - ➢Simplicity enables higher performance at lower cost

Compiling a C Statement Example

- C code with the five variables, f, g, h, i, and j

```
f = (g + h) - (i + j);
```

- The compiled RISC-V code

```
add t0, g, h   // temp t0 = g + h
add t1, i,  j  // temp t1 = i + j
sub f,  t0, t1 // f = t0 - t1
```

- The single statement is braked into several instructions
  - ➢One operation is performed per RISC-V instruction
  - ➢Temporary variables (t0 and t1) are used to keep the intermediate results

Register Operands

- The operands of arithmetic instructions are kept in a limited number of special locations built directly in hardware called registers
- Registers are primitives used in hardware design
  - ➢ Registers are also visible to the programmer when the computer is completed, so you can think of registers as the bricks of computer construction

- A 32-bit RISC-V computer has 32 registers
  - ➢ Use for frequently accessed data
  - ➢ 32-bit data is called a "word"
  - ➢ 64-bit data is called a "doubleword"
    - ❖ For a 64-bit machine, there could be 32 x 64-bit general purpose registers x0 to x31

- *Design Principle 2:* Smaller is faster
  - ➢ c.f. main memory: millions of locations, but slower
- Why 32 registers?
  - ➢ 31 registers may not be faster than 32, but the designers have to balance the clock cycle and more registers, and number of bits in the instruction format
  - ➢ A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther

RISC-V Registers

- The common arrangement for x0~x31 registers

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer

- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Compiling a C Statement Example (Cont'd)

- C code with the five variables, f, g, h, i, and j

```
f = (g + h) - (i + j);
```

- The compiled RISC-V code

```
add t0, g, h   // temp t0 = g + h
add t1, i,  j  // temp t1 = i + j
sub f,  t0, t1 // f = t0 - t1
```

- In fact, a compiler has to associate program variables with registers, and the actual code should be…

```
add x5,  x20, x21
add x6,  x22, x23
sub x19, x5,  x6
```

Note:
1. f, g, h, i, j are in x19, x20, …, x23
2. x5 and x6 are for temporary data

Memory Operands

- Main memory used for composite data
  - Single data elements: integers, floats
  - Complex data structures: arrays, dynamic data
- Arithmetic operations occur only on registers
  - Data transfer instructions are required to
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Memory is considered as a large, single-dimensional array, where *address* is the index to this array
  - Each memory address identifies an 8-bit byte
- RISC-V does not require words to be aligned in memory
  - Unlike some other ISAs

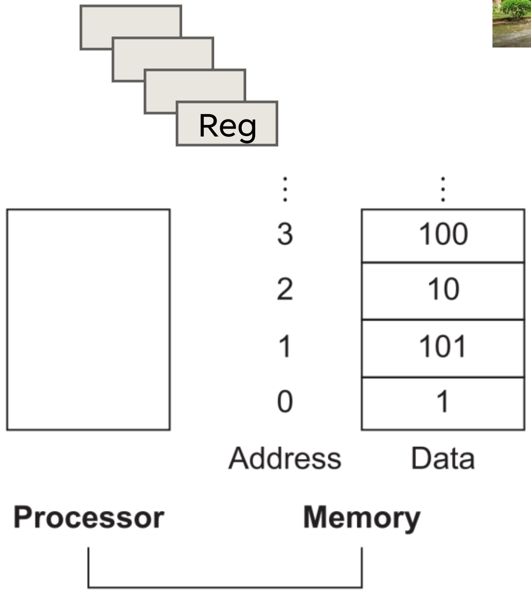
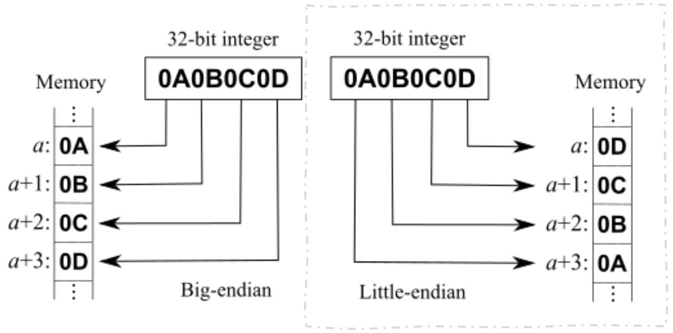


FIGURE 2.2 Memory addresses and contents of memory at those locations. If these elements were words, these addresses would be incorrect, since RISC-V actually uses byte addressing, with each word representing 4 bytes. Figure 2.3 shows the correct memory addressing for sequential word addresses.

- RISC-V is Little Endian
  - Least-significant byte at least address of a word
  - *c.f.* Big Endian: most-significant byte at least address



Courtesy of Img source 3/13/2024 11

Memory Operand Example

- A is an array of 100 words
  - ➢Compiler associates the variables g and h with the registers x20 and x21
  - ➢Base address of A is in x22 (base register)
- Given the C code below

```
g = h + A[8];
```

- its RISC-V assembly

```
lw   x9,   8(x22)          // Transfer A[8] to a register x9
add x20, x21,  x9          // g = h + A[8]; 8 is called the offset
```

Memory Address

- Compiler helps
  - ➢associate variables with registers, and
  - ➢allocate data structures, like arrays and structures, to memory locations
- RISC-V is byte addressable
  - ➢The address of a word matches the address of one of the 4 bytes within the word
  - ➢The addresses of sequential words differ by four (see Fig. 2.3)
  - ➢i.e., each array element is four-byte data
- Alignment restriction
  - ➢In many architectures (e.g., MIPS), words must start at addresses that are multiples of four (i.e., data be aligned in memory on natural boundaries) for better accessing efficiency

- With the byte addressing concept, the previous example assembly becomes:

```
lw      x9,  32(x22)
add     x20, x21,  x9
```

The offset to the 8th element is 32

- A sw instruction is needed if the result should be stored into a memory location (e.g., A[12])

```
sw      x20,  48(x22)
```

- What is the offset when we are handling the array in a 32-bit machine?
  - ➢ Each element is doubleword

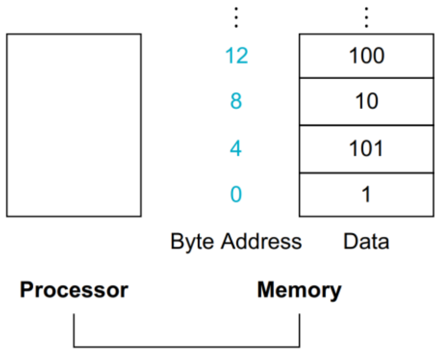


FIGURE 2.3 Actual RISC-V memory addresses and contents of memory for those words. The changed addresses are highlighted to contrast with Figure 2.2. Since RISC-V addresses each byte, word addresses are multiples of 4: there are 4 bytes in a word

Registers vs. memory

- Registers are faster to access than memory
  - ➢Assuming 32-bit data, registers are roughly 200 times faster (0.25 vs. 50 nanoseconds) and are 10,000 times more energy efficient (0.1 vs. 1000 picoJoules) than DRAM in 2020
- Operating on memory data requires loads and stores
  - ➢More instructions to be executed
- Many programs have more variables than computers have registers
  - ➢Compiler must use registers for variables as much as possible
  - ➢"Spilling registers" is putting less frequently used variables into memory
  - ➢Register optimization is important for execution performance

Immediate Operands

- Constants are often used in a program
  - ➢Increment an index (constant) to point to the next element of an array
  - ➢In fact, more than half of the RISC-V arithmetic instructions have a constant as an operand when running the SPEC CPU2006 benchmarks
- Using only the instructions we have seen, to use a constant, one has to *load* it from memory to register

```
lw    x9, AddrConstant4(x3) // x9 = 4 (a constant), Mem[x3+AddConstant4]=4
add x22, x22, x9            // x22 = x22 + x9 (where x9 == 4)
```

- Alternatively, *add immediate* (addi) instruction can be used to avoid the load instruction

```
addi x22, x22, 4            // x22 = x22 + 4
```

- The constant zero has a special role
  - ➢which is to simplify the instruction set by offering useful variations
  - ➢For example, you can negate the value in a register by using the sub instruction with zero for the first operand
  - ➢Hence, RISC-V dedicates register x0 to be hard-wired to the value zero

Unsigned Numbers (Bits)

- A computer represents numbers with binary digits (bits)
  - Base 2 numbers: 0 or 1 (on or off)
  - We are taught to think in base 10; 123 base 10 = 1111011 base 2
  - Decimal numbers for base 10 numbers, and binary numbers for base 2 numbers

$00000000\ 00000000\ 00000000\ 00000000_{two} = 0_{ten}$ Subscript to indicate the base
$00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}$
$00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}$
... ...
$11111111\ 11111111\ 11111111\ 11111101_{two} = 4{,}294{,}967{,}293_{ten}$
$11111111\ 11111111\ 11111111\ 11111110_{two} = 4{,}294{,}967{,}294_{ten}$
$11111111\ 11111111\ 11111111\ 11111111_{two} = 4{,}294{,}967{,}295_{ten}$

- We number the bits 0, 1, 2, 3, ... *from right to left*
  - In a 32-bit register, the bits are arranged as below

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

(32 bits wide)

  - Least Significant Bit (LSB) refers to the rightmost bit (bit 0)
  - Most Significant Bit (MSB) refers to the leftmost bit (bit 31)

Unsigned Numbers

- 32-bit binary numbers can be represented in terms of
  - ➢the bit value times a power of 2 (here x_i means the *i*-th bit of x)

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

  - ➢What we described so far are positive numbers, also known as unsigned numbers

- Convert a *n*-bit base 2 number into the base 10 number:

Range of decimal numbers: 0 to $+2^n - 1$

Example

$0000\ 0000 \ldots 0000\ 1011_2$

$= 0 + \ldots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

$= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

When using 64 bits (n=64): 0 to 18,446,774,073,709,551,615

Negative Numbers

- A representation that distinguishes the positive from the negative numbers
  - ➢The most obvious solution is to add a *separate* sign,
  - ➢which conveniently can be represented in a single bit; the name for this representation is sign and magnitude
  - ➢*Overflow* is said to have occurred
    - ❖if the number that is the proper result of an operation are larger than be represented in a register (thus, cannot be represented by the rightmost hardware bits)

- Sign and magnitude is soon abandoned because…
  1. It's not obvious where to put the sign bit. To the right? To the left? Early computers tried both
  2. Adders for sign and magnitude may need an extra step to set the sign because we can't know in advance what the proper sign of the sum will be
  3. A *separate* sign bit means that sign and magnitude has both a positive and a negative zero, which can lead to problems for inattentive programmers

Two's Complement

- The final solution is to make the hardware simple

→Leading 0s mean positive, and

→leading 1s mean negative

Sign bit

- ➢The positive half of the numbers ($0000 \cdots 0000_{two}$), from 0 to $2{,}147{,}483{,}647_{ten}$ ($2^{31}-1$), use the same representation as before
- ➢The following bit pattern ($1000 \cdots 0000_{two}$) represents the most negative number -2,147,483,648ten (-2^{31})
  - ❖i.e., a declining set of negative numbers: $-2{,}147{,}483{,}647_{ten}$ ($1000 \cdots 0001_{two}$) down to -1_{ten} ($1111 \cdots 1111_{two}$).

```
00000000 00000000 00000000 00000000two = 0ten
00000000 00000000 00000000 00000001two = 1ten
00000000 00000000 00000000 00000010two = 2ten
. . .                         . . .
01111111 11111111 11111111 11111101two = 2,147,483,645ten
01111111 11111111 11111111 11111110two = 2,147,483,646ten
01111111 11111111 11111111 11111111two = 2,147,483,647ten
10000000 00000000 00000000 00000000two = – 2,147,483,648ten
10000000 00000000 00000000 00000001two = – 2,147,483,647ten
10000000 00000000 00000000 00000010two = – 2,147,483,646ten
…                                          . . .
11111111 11111111 11111111 11111101two = – 3ten
11111111 11111111 11111111 11111110two = – 2ten
11111111 11111111 11111111 11111111two = – 1ten
```

More about Two's Complement

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers

- $-(-2^{n-1})$ can't be represented
  - It has no corresponding positive number for -2,147,483,648$_{ten}$ (- -2^{32-1})

- Non-negative numbers have the same <u>unsigned</u> and <u>2s-complement</u> representation

- Some specific numbers
  - 0: 0000 0000 ··· 0000
  - –1: 1111 1111 ··· 1111
  - Most-negative: 1000 0000 ··· 0000
  - Most-positive: 0111 1111 ··· 1111

Please go check

The elaboration (p. 87) in the textbook for the name of two's complement

It is right above the start position of Sec. 2.5

The description of one's complement (right below the above in p. 87)

Two's Complement Signed Numbers

- The value of a signed 32-bit number is evaluated by…
  - The sign bit is multiplied by -2^{31}, and
  - the rest of the bits are then multiplied by positive versions of their respective base values
  - The general form for n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

Overflow occurs when the leftmost retained bit of the binary bit pattern is not the same as the infinite number of digits to the left (the sign bit is incorrect) A 0 on the left of the bit pattern when the number is negative, or a 1 when the number is positive

- Convert a 32-bit two's complement number into the base 10 number
  - Range of decimal numbers: -2^{n-1} to $+2^{n-1} - 1$
  - Example

    $1111\ 1111\ \ldots\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648_{10} + 2{,}147{,}483{,}644_{10} = -4_{10}$
  - When using 64 bits: −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Negation Shortcut

- A useful shortcut to negate a two's complement number
  - ➢E.g., to find the native number from its positive counterpart
- Observation

  →The sum of a number and its *inverted* representation must be 111 ⋯ 111_{two} (-1_{ten})

  $x + \bar{x} = 1111...111_2 = -1$

  - ➢That is, $x + \bar{x} = -1$, $\bar{x} + 1 = -x$

  $\bar{x} + 1 = -x$

  - ➢Simply invert every 0 to 1 and every 1 to 0, then add one to the result

Example of finding -2 from 2

$+2 = 0000\ 0000 \ldots 0010_{two}$

$-2 = 1111\ 1111 \ldots 1101_{two} + 1$

$= 1111\ 1111 \ldots 1110_{two}$

Exercise: You can find 2 from -2

Sign Extension Shortcut

- This shortcut converts a binary number represented in *n* bits to a number represented with more than *n* bits
  - ➢Preserve the numeric value
- Sign Extension
  - ➢Take the most significant bit from the smaller quantity (the sign bit) and
  - ➢replicate the sign bit to fill the new bits of the larger quantity
  - ➢The old nonsign bits are simply copied into the right portion of the new doubleword
- Examples: extends an 8-bit number to 16-bit number
  - ➢+2: 0000 0010 => 0000 0000 0000 0010
  - ➢–2: 1111 1110 => 1111 1111 1111 1110
- In RISC-V instruction set
  - ➢`lb`: sign-extend loaded byte
  - ➢`lbu`: zero-extend loaded byte; load byte unsigned

Representing Instructions

- The way computers see instructions

- Machine language
  - ➢is the <u>numeric version of instructions</u>
  - ➢binary representation used for communication within a computer system
- Machine code
  - ➢is a sequence of instructions encoded in <u>binary numbers,</u> translated from the assembly code
  - ➢To avoid reading/writing long, tiresome binary strings, hexadecimal (base 16) numbers are often adopted
  - ➢Four bits per hexadecimal (hex) digit
  - ➢Example: eca8 6420_{hex} = 1110 1100 1010 1000 0110 0100 0010 0000_{two}

- RISC-V instructions
  - ➢Encoded as 32-bit instruction *words*
  - ➢Regularity!
  - ➢An instruction encodes operation code (opcode), register numbers, etc.

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

FIGURE 2.4 The hexadecimal–binary conversion table. Just replace one hexadecimal digit by the corresponding four binary digits, and vice versa. If the length of the binary number is not a multiple of 4, go from right to left.

RISC-V
R-format Instructions

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Instruction fields (R-type for register)
  - ➢opcode: operation code, denoting the operation and format of an instruction
  - ➢rd: destination register number
  - ➢funct3: 3-bit function code (additional opcode)
  - ➢rs1: the first source register number
  - ➢rs2: the second source register number
  - ➢funct7: 7-bit function code (additional opcode)

R-format Inst. Example

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- add x9, x20, x21

add instruction in the R-format using *decimal* numbers

0	21	20	0	9	51

add instruction in the R-format using *binary* numbers

0000 000	10101	10100	000	01001	0110011

$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two}$ =
$015A\ 04B3_{hex}$

I-format Instructions

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

- An instruction will need longer fields than those shown above
  - E.g., a constant may be used to select elements from arrays or data structures, and
  - it often needs to be much larger than 31
  - The 5-bit field (e.g., rs2) is too small to be useful
  - For addi or lw inst.

- Immediate arithmetic (addi) and load instructions (lw)
  - rs1: source or base address register number
  - immediate: *constant* operand, or *offset* added to base address (for load inst.)
    - Two's-complement, sign extended numbers; integers from -2^{11} to $2^{11}-1$
    - A load word instruction refers to any word within a region of $\pm\ 2^{11}$ or 2,048 bytes ($\pm 2^9$ or 512 words) of the base address in the base register rd
- *Design Principle 3:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

S-format Instructions

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- We also need a format for the store word instruction
  - sw, which needs *two* source registers (for the base address and the store data) and an immediate for the address offset
  - It is a little bit different from R-type inst. (rd vs. imm), but with the same layout

- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: 12-bit immediate *offset* added to base address
    - *Split* so that rs1 and rs2 fields always in the same place

Summary of R-, I-, S-format Inst.

- The formats are distinguished by the values in the opcode field
  - ➢ each format is assigned a distinct set of opcode values in the first field (opcode)
  - ➢ so that the hardware knows how to treat the rest of the instruction

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
`add` (add)	R	0000000	reg	reg	000	reg	0110011
`sub` (sub)	R	0100000	reg	reg	000	reg	0110011

Instruction	Format	immediate	rs1	funct3	rd	opcode
`addi` (add immediate)	I	constant	reg	000	reg	0010011
`lw` (load word)	I	address	reg	010	reg	0000011

Instruction	Format	immed-iate	rs2	rs1	funct3	immed-iate	opcode
`sw` (store word)	S	address	reg	reg	010	address	0100011

FIGURE 2.5 RISC-V instruction encoding. In the table above, "reg" means a register number between 0 and 31 and "address" means a 12-bit address or constant. The funct3 and funct7 fields act as additional opcode fields

The "Stored Program" Concept

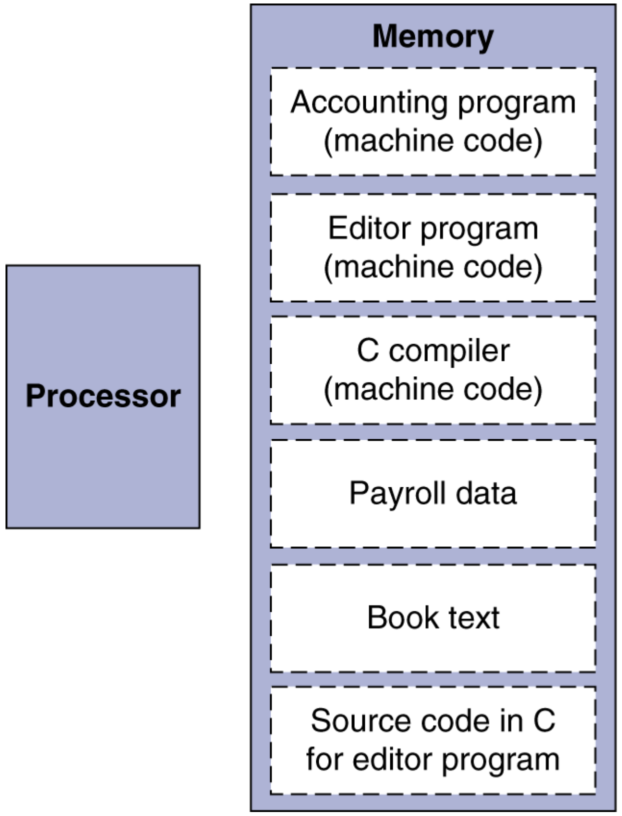

Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program
Processor

FIGURE 2.7 The stored-program concept. Stored programs allow a computer that performs accounting to become, in the blink of an eye, a computer that helps an author write a book. The switch happens simply by loading memory with programs and data and then telling the computer to begin executing at a given location in memory. Treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer systems. Specifically, the memory technology needed for data can also be used for programs, and programs like compilers, for instance, can translate code written in a notation far more convenient for humans into code that the computer can understand.

1. Instructions represented in binary, just like data
2. Instructions and data stored in memory

- Programs can operate on programs
  - ➢e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - ➢Standardized ISAs

Logical Operations

- It is useful to operate on fields of bits within a word or even on individual bits
  - ➢E.g., examining characters within a word, each of which is stored as 8 bits
- These operations were added to programming languages and instruction set architectures
  - ➢To simplify the packing and unpacking of bits into words
  - ➢Useful for extracting and inserting groups of bits in a word

Logical operations	C operators	Java operators	RISC-V instructions
Shift left	<<	<<	`sll, slli`
Shift right	>>	>>>	`srl, srli`
Shift right arithmetic	>>	>>	`sra, srai`
Bit-by-bit AND	&	&	`and, andi`
Bit-by-bit OR	|	|	`or, ori`
Bit-by-bit XOR	^	^	`xor, xori`
Bit-by-bit NOT	~	~	`xori`

FIGURE 2.8 C and Java logical operators and their corresponding RISC-V instructions. One way to implement NOT is to use XOR with one operand being all ones (FFFF FFFF FFFF FFFF$_{hex}$)

Shift Operations

funct6	immed	rs1	funct3	rd	opcode
6 bits	6 bits	5 bits	3 bits	5 bits	7 bits

- *Shifts* move all the bits in a word to the left or right, filling the emptied bits with 0s
- immed: how many positions to shift
- Shift *left* logical immediate
  - ➢Shift left and fill with 0 bits
  - ➢`slli` by *i* bits, multiplies by 2^i
  - ➢slli x11, x19, 4 // reg x11 = reg x19 << 4 bits

$00000000\ 00000000\ 00000000\ 00001001_{two} = 9_{ten}$

$\xleftarrow{4}$

$00000000\ 00000000\ 00000000\ 10010000_{two} = 144_{ten}$

- Shift *right* logical immediate
  - ➢Shift right and fill with 0 bits
  - ➢`srli` by *i* bits divides by 2^i (unsigned only)

A third type of shift, *shift right arithmetic (srai)*. This variant is similar to srli, except rather than filling the vacated bits on the left with zeros, it fills them with copies of the old ***sign*** bit.

AND Operations

- To isolate fields via bit-by-bit operation
  - ➢AND leaves a 1 in the result only if both bits of the operands are 1
  - ➢Useful to *mask* bits in a word
  - ➢Select some bits, clear others to 0

and x9,x10,x11

```
    x10   00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
and
    x11   00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
---------------------------------------------------------------------------------
    x9    00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000
```

OR and XOR Operations

- Useful to include bits in a word
  - A logical bit-by-bit operation with two operands that calculates a 1 if there is a 1 in *either* operand

or x9,x10,x11

```
x10   ... 00000000 00000000 00001101 11000000
or
x11   ... 00000000 00000000 00111100 00000000
-----------------------------------------------
x9    ... 00000000 00000000 00111101 11000000
```

- Differencing operation
  - A logical bit-by-bit operation with two operands that calculates the exclusive OR of the two operands
  - That is, it calculates a 1 only if the values are different in the two operands
  - It can be used to emulate NOT operation (to keep three-operand inst. Format)

xor x9,x10,x12 // NOT operation

NOT takes one operand and places a 1 in the result if one operand bit is a 0, and vice versa. Using our prior notation, it calculates $\overline{x}$.

```
x10   00000000 00000000 00000000 00000000 00000000 00000000 00001101  11000000
xor
x12   11111111 11111111  11111111 11111111 11111111 11111111 11111111 11111111
------------------------------------------------------------------------------
x9    11111111 11111111  11111111 11111111 11111111 11111111 11110010 00111111
```

Please read through…

Sec. 2.6

Shift also provides variants of all three shifts that take the shift amount from a register, rather than from an immediate: sll, srl, and sra

RISC-V also provides the instructions and immediate (andi), or immediate (ori), and exclusive or immediate (xori)

Conditional Operations

- Decision making is commonly represented in programming languages
  - ➢using the if statement, sometimes combined with go to statements and labels

- Two decision-making instructions (conditional branches*) in RISC-V assembly language
- beq (branch if equal)
  - ➢beq rs1, rs2, L1
  - ➢This instruction means go to the statement labeled L1 if the value in register rs1 equals the value in register rs2
- bne (branch if not equal)
  - ➢bne rs1, rs2, L1
  - ➢It means go to the statement labeled L1 if the value in register rs1 does not equal the value in register rs2

*An instruction that tests a value and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the test.

A higher-level view of branches

- Branch to a labeled instruction if a condition is true
  - ➢Otherwise, continue sequentially
- beq rs1, rs2, L1
  - ➢ if (rs1 == rs2) branch to instruction labeled L1
- bne rs1, rs2, L1
  - ➢ if (rs1 != rs2) branch to instruction labeled L1
- Decisions are important for if statements and for loops (iterating a computation)

Compiling If Statements

- C code

```
if (i==j)
        f = g+h;
else
        f = g-h;
```

NOTE: f, g, ⋯ in x19, x20, ⋯

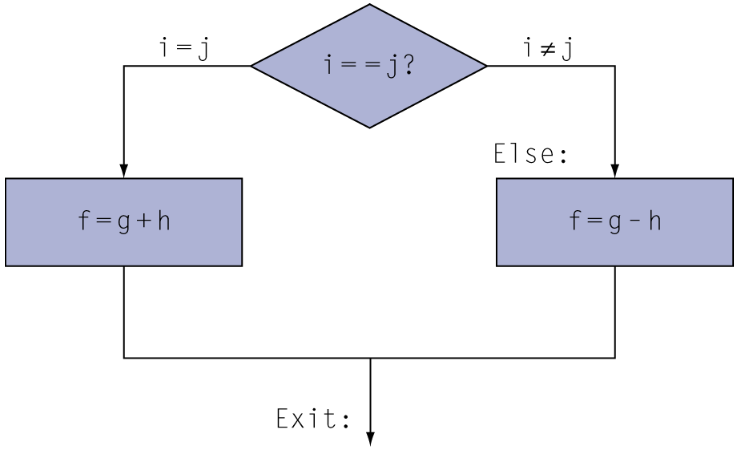


FIGURE 2.9 Illustration of the options in the if statement above. The left box corresponds to the then part of the if statement, and the right box corresponds to the else part.

- Compiled RISC-V code

```
      bne x22, x23, Else // go to Else if i ≠ j
      add x19, x20, x21  // f = g + h (skipped if i ≠ j)
      beq x0,   x0,   Exit // unconditional; // if 0 == 0, go to Exit

Else:
      sub x19, x20, x21  // f = g − h (skipped if i = j)

Exit:
         …
```

Assembler calculates addresses

Compiling a *while* loop in C

- C code

```
while (save[i] == k)
        i += 1;
```

Note: i in x22, k in x24, address of save in x25

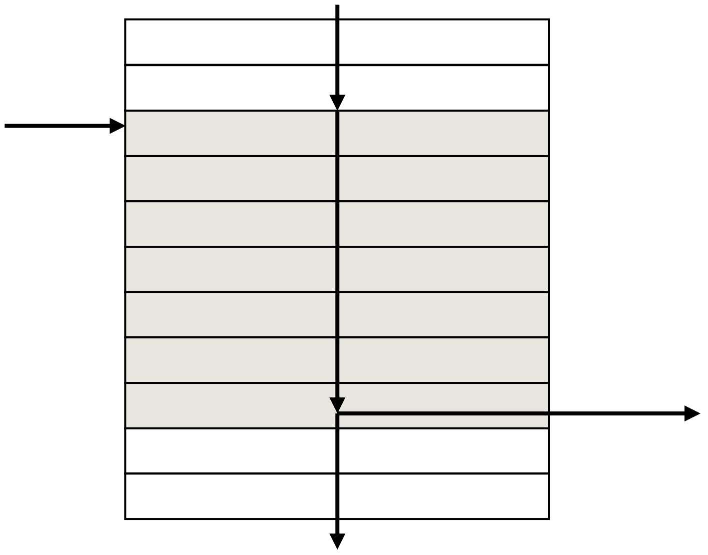
- Compiled RISC-V code

```
Loop:
     slli   x10, x22, 2       // Temp reg x10 = i * 4
     add  x10, x10, x25     // x10 = address of save[i]
     lw     x9,   0(x10)       // Temp reg x9 = save[i]
     bne  x9,   x24, Exit     // go to Exit if save[i] ≠ k
     addi x22, x22, 1          // i = i + 1
     beq  x0,   x0,   Loop // go to Loop
Exit:
     ...
```

Basic Blocks

- A *basic block* is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)

- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks



Please read through…

Bounds Check and Other Types of Branches (p. 101 ~ 102)

To know many other relationships between two numbers.

For example, a for loop may want to test to see if the index variable is less than 0. The full set of comparisons is less than (), greater than or equal ($\geq$), equal ($=$), and not equal ($\neq$)

Case/Switch Statement (p. 103)

Procedures

- A procedure (or function)
  - ➢Is a stored subroutine that performs a specific task based on the parameters with which it is provided
  - ➢Is one tool programmers use to structure programs, both to make them easier to understand and to allow code to be reused
  - ➢Is one way to implement abstraction in software

- Six steps required for invoking a procedural call
  1. Place parameters in registers x10 to x17
  2. Transfer control to procedure (with branch inst.)
  3. Acquire storage for procedure (spill registers)
  4. Perform procedure's operations
  5. Place result in register for caller (restore registers as well)
  6. Return to place of call (return address in x1 and jump back)

Procedure Calls

- RISC-V software follows the following procedure calling convention in allocating its 32 registers
  - x10–x17: eight parameter registers in which to pass parameters or return values
  - x1: one return address register to return to the point of origin

- RISC assembly language has the two instructions for procedures
  - jal (jump-and-link): It branches to an address and simultaneously saves the address of the following instruction to the destination register rd (the "link" portion)
  - jalr (jump-and-link register): it helps return from a procedure; branches to the address stored in the return register x1, which is just what we want

Procedure Calls (Cont’d)

- Procedure call w/ jump and link

  `jal x1, ProcedureLabel`

  - Address of following instruction put in x1
  - Jumps to target address
  - Caller is the program that instigates a procedure and provides the necessary parameter values
  - Program counter (PC) is a register to hold the address of the current instruction being executed
    - PC can be seen as instruction address register
  - The jal instruction actually saves PC + 4 in its designation register (usually x1) to link to the byte address of the following instruction to set up the procedure return

- Procedure return w/ jump and link register

  `jalr x0, 0(x1)`

  - Like jal, but jumps to 0 plus the address in x1
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements
  - Callee is a procedure (ProcedureLabel) that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller

A Procedure Call Example (Six Steps)

Caller

```
void non_leaf () {
  int a = 0;

  a = leaf_example(4, 3, 2, 1);

}
```

Callee

```
int leaf_example (
          int g, int h,
          int i, int j) {
      int f;
      f = (g + h) - (i + j);
      return f;
}
```

1. Place parameters in registers x10 to x17
2. Transfer control to procedure (with jal inst.) return addr. Is in x1

3. Acquire storage for procedure (spill registers)

4. Perform procedure's operations

5. Place result in register for caller and adjust stack pointer
6. Return to place of call (return address in x1 and jump back, jalr)

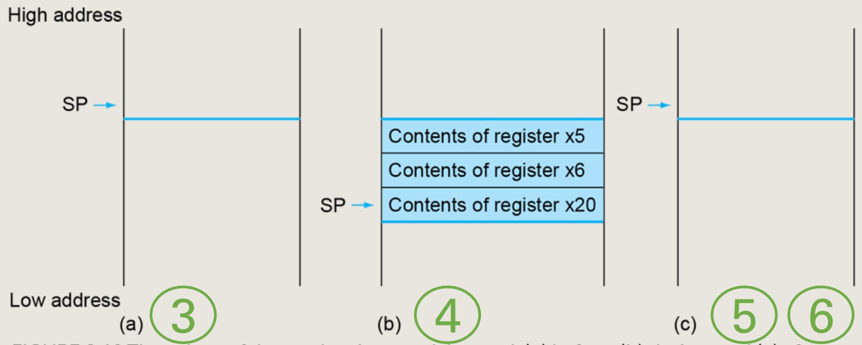


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Compiling a C Procedure

- C code

```
int leaf_example (
      int g, int h,
      int i, int j) {
   int f;
   f = (g + h) - (i + j);
   return f;
}
```

  - ➢Arguments g, ..., j in x10, ..., x13
  - ➢f in x20
  - ➢Temporary variables in x5, x6
  - ➢Need to save x5, x6, x20 on stack

- RISC-V code

```
leaf_example:
  addi sp,sp,-12      // adjust stack to make room for 3 items: x5, x6, x20
  sw   x5,8(sp)       // save register x5, x6, x20 for use afterwards
  sw   x6,4(sp)
  sw   x20,0(sp)
  add  x5,x10,x11     // x5 = g + h
  add  x6,x12,x13     // x6 = i + j
  sub  x20,x5,x6      // f = x5 – x6
  addi x10,x20,0      // copy f (x10) to return register
  lw   x20,0(sp)      // Restore x5, x6, x20 from stack
  lw   x6,4(sp)
  lw   x5,8(sp)
  addi sp, sp,12      // adjust stack to delete 3 items: x5, x6, x20
  jalr x0,0(x1)       // Return to caller
```

Stack

- It is a data structure organized as a last-in-first-out queue for spilling registers (to preserve the data contents)
  - ➢Placing data onto the stack is called a push, and removing data from the stack is called a pop
- Stack Pointer (SP) is a value denoting the most recently allocated address in a stack
  - ➢that shows where registers should be spilled or where old register values can be found
  - ➢In RISC-V, it is register sp, or x2

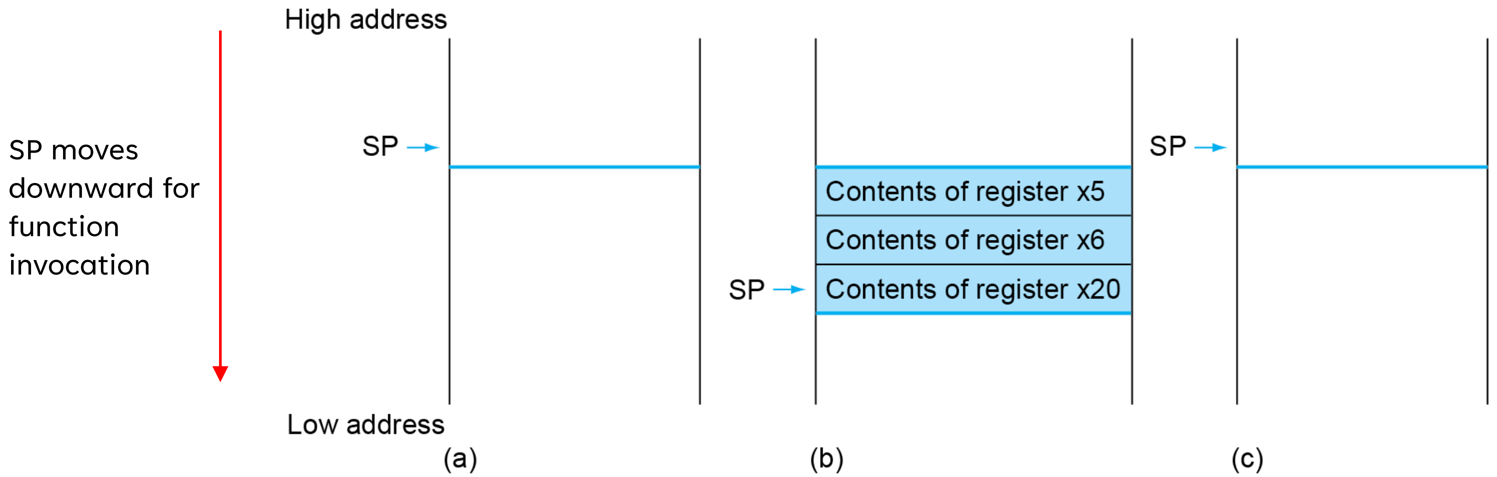


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Temporary and Preserved Registers

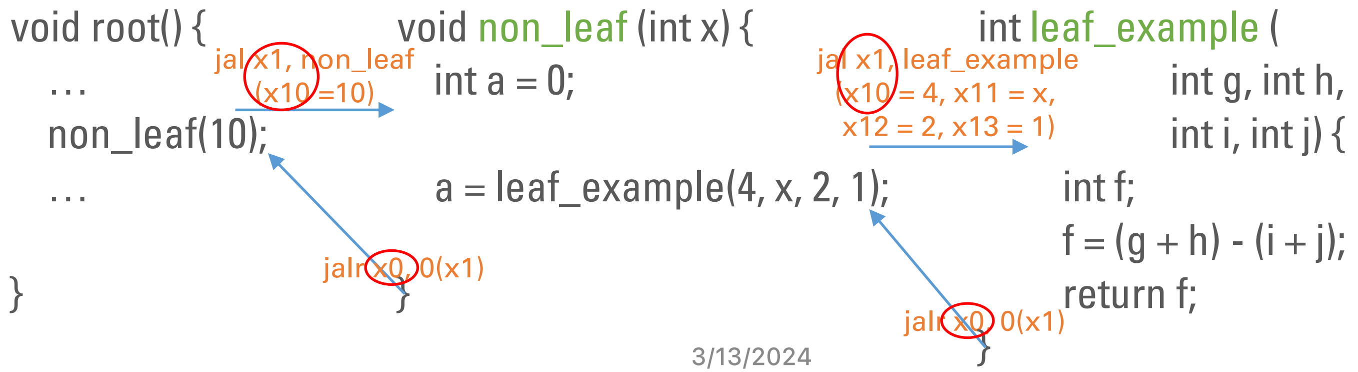
- RISC-V SW separates following 19 registers into two groups
  - ➢to avoid saving/restoring a register whose value isn't used
  - ➢which might happen with a temporary register

1. x5－x7 and x28－x31: temporary registers that are not preserved by the callee (called procedure) on a procedure call
   - ❖The caller does not expect registers x5 and x6 to be preserved across a procedure call (these registers are maintained by caller)
   - ❖The example in p. 49 does not follow this arrangement
2. x8－x9 and x18－x27: saved registers that *must* be preserved on a procedure call
   - ❖if used, the callee saves and restores them
   - ❖These registers are maintained by callee

Nested Procedures: Leaf and Non-leaf Procedures

- Leaf procedures are procedures that do not call others
  - Only be callee (e.g., leaf_example)
- Non-leaf procedures are procedures that do call others
  - Be both caller and callee (e.g., non_leaf)

- Without proper protection of registers, there will be conflicts on their contents
  - E.g., since non_leaf hasn’t finished its task yet, there is a conflict over the use of registers (e.g., x0, x1, x10)
- For a nested call, caller needs to save on the stack
  - Its return address
  - Any arguments and temporaries needed after the call
  - Restore from the stack after the call



```
void root() {
  ...
  non_leaf(10);
  ...

}
```

```
void non_leaf (int x) {
  int a = 0;

  a = leaf_example(4, x, 2, 1);


}
```

```
int leaf_example (
      int g, int h,
      int i, int j) {
  int f;
  f = (g + h) - (i + j);
  return f;
}
```

A Non-leaf Example

- C code

```
int fact (int n) {
   if (n < 1)
      return f;
   else
      return n * fact(n - 1);
}
```

- ➢ Argument n in x10
- ➢ Result in x10

- RISC-V code

```
fact:
    addi  sp,sp,-8          Adjust stack for 2 items (return addr. and n)
    sw    x1,  4(sp)        Save the return address
    sw    x10, 0(sp)        Save the argument n
    addi  x5,  x10,-1       x5 = n - 1
    bge   x5,  x0,L1        if n >= 1, go to L1
    addi  x10,x0,1          else, set return value to 1
    addi  sp,  sp,8         Pop stack, don't bother restoring values
    jalr  x0,  0(x1)        Return to caller
L1: addi  x10,x10,-1        n = n – 1, n>=1 arguments gets (n-1)
    jal   x1, fact          Call fact(n-1)
    addi  x6, x10,0         Move result of fact(n - 1) to x6; return from jal
    lw    x10,0(sp)         Restore caller's argument n
    lw    x1, 4(sp)         Restore caller's return address
    addi  sp, sp,8          Pop stack; adjust SP to pop 2 items
    mul   x10,x10,x6        Return n * fact(n-1)
    jalr  x0,  0(x1)        Return to the caller
```

Preserved	Not preserved
Saved registers: x8-x9, x18-x27	Temporary registers: x5-x7, x28-x31
Stack pointer register: x2(sp)	Argument/result registers: x10-x17
Frame pointer: x8(fp)	
Return address: x1(ra)	
Stack above the stack pointer	Stack below the stack pointer

FIGURE 2.11 What is and what is not preserved across a procedure call. If the software relies on the global pointer register, discussed in the following subsections, it is also preserved

Note: The above is a general rule. Compiler will preserve the used register(s) when necessary (e.g., x10 in the above example).

Local Data on the Stack

- The stack is also used to store variables
  - ➢that are local to the procedure (and maintained by callee)
  - ➢but do not fit in registers
  - ➢E.g., local arrays, structures, C automatic variables
- The stack grows downwards at runtime
  - ➢SP value might change during the procedure
  - ➢so references to a local variable in memory might have different offsets depending on where they are in the procedure
  - ➢making the procedure harder to understand

- Procedure frame (activation record)
  - ➢A stack segment contains a procedure's saved registers and local variables
- A frame pointer fp (x8) points to the first word of the frame of a procedure
  - ➢ Solution: fp offers a stable base register for procedure frame for local variable accesses

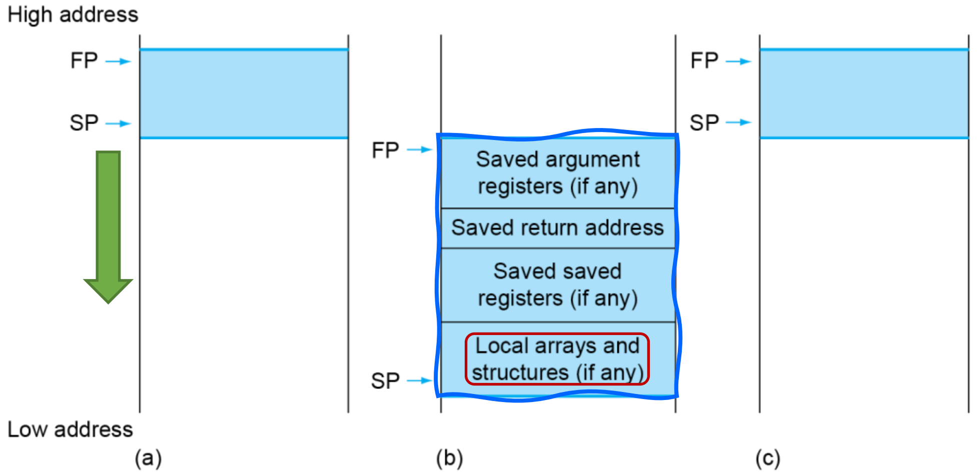

High address
FP
SP
FP
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
SP
FP
SP
Low address
(a)
(b)
(c)

FIGURE 2.12 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call. The frame pointer (fp or x8) points to the first word of the frame, often a saved argument register, and the stack pointer (sp) points to the top of the stack. The stack is adjusted to make room for all the saved registers and any memory-resident local variables. Since the stack pointer may change during program execution, it's easier for programmers to reference variables via the stable frame pointer, although it could be done just with the stack pointer and a little address arithmetic. If there are no local variables on the stack within a procedure, the compiler will save time by not setting and restoring the frame pointer. When a frame pointer is used, it is initialized using the address in sp on a call, and sp is restored using fp. This information is also found in Column 4 of the RISC-V Reference Data Card at the front of this book

Memory Layout

- In addition to local variables, global data in a program is also kept in the memory
  - ➢E.g., static variables and dynamic data structures in C programs
- The memory layout shown in figure shows
  - ➢the RISC-V convention for allocation of memory when running the Linux operating system

- Text segment
  - ➢ Starts at 0000 0000 0040 0000_{hex}
  - ➢ The RISC-V program (machine) code
- Static data segment
  - ➢ Starts at 0000 0000 1000 0000_{hex}
  - ➢ Constants and other static variables
  - ➢ Pointed by gp (global pointer)
- Dynamic data (heap; grows upwards)
  - ➢ Structures grow/shrink during their lifetimes, e.g., linked lists
  - ➢ The C functions, malloc and free, manipulates the heap data
- Stack (grows downwards)
  - ➢ Automatic storage

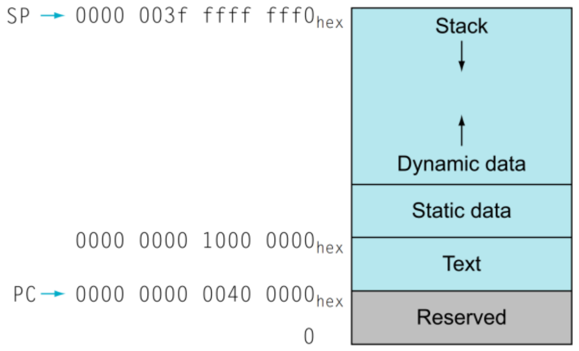


FIGURE 2.13 The RISC-V memory allocation for program and data. These addresses are only a software convention, and not part of the RISC-V architecture. The stack pointer is initialized to 0000 003f ffff $fff0_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at 0000 0000 0040 0000hex. The static data starts immediately after the end of the text segment; in this example, we assume that address is 0000 0000 1000 0000_{hex}. Dynamic data, allocated by malloc in C and by new in Java, is next. It grows up toward the stack in an area called the heap. This information is also found in Column 4 of the RISC-V Reference Data Card at the front of this book

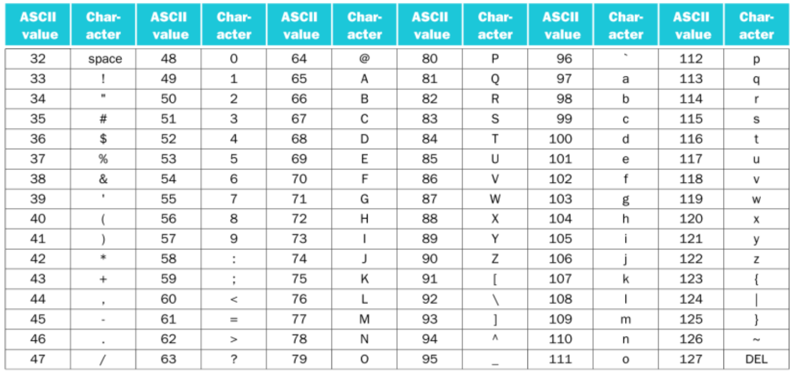
Please read through…

Global pointer (p. 110)

Representing ASCII Characters

- Byte-encoded character sets
  - American Standard Code for Information Interchange (ASCII)
  - ASCII: 128 characters (95 graphic, 33 control)
  - Latin-1: 256 characters (ASCII, +96 more graphic characters)

- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings



ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

FIGURE 2.15 ASCII representation of characters. Note that upper- and lowercase letters differ by exactly 32; this observation can lead to shortcuts in checking or changing upper- and lowercase. Values not shown include formatting characters. For example, 8 represents a backspace, 9 represents a tab character, and 13 represents a carriage return. Another useful value is 0 for null, the value the programming language C uses to mark the end of a string.

Load/Store Data in Different Widths

- LOAD/STORE byte/halfword/word
  - Load byte/halfword/word: Sign extend to bits in rd
    - lb rd, offset(rs1)
    - lh rd, offset(rs1)
    - lw rd, offset(rs1)
  - Load byte/halfword/word unsigned: Zero extend to bits in rd
    - lbu rd, offset(rs1)
    - lhu rd, offset(rs1)
    - lwu rd, offset(rs1)
  - Store byte/halfword/word: Store rightmost 8/16/32 bits
    - sb rs2, offset(rs1)
    - sh rs2, offset(rs1)
    - sw rs2, offset(rs1)

lbu for load byte unsigned, sb for store byte

lbu x12, 0(x10) // Read byte from source, placing it in the rightmost 8 bits in x12

sb x12, 0(x11) // Takes a byte from the rightmost 8 bits of x12 and writes the byte to x11[0]

A String Copy Example

- C code

```
void strcpy (char x[], char y[]) {
  size_t i;
  i = 0;
  while ((x[i]=y[i])!='\0')
     i += 1;
}
```

  - Null-terminated string
  - Assume that base addresses for arrays x and y are found in x10 and x11

- RISC-V code

```
strcpy:
        addi sp,sp,-4     // adjust stack for 1 word
        sw   x19,0(sp)     // push x19
        add  x19,x0,x0    // i=0+0
L1:     add  x5,x19,x10 // x5 = addr of y[i]
        lbu  x6,0(x5)      // x6 = y[i]
        add  x7,x19,x10 // x7 = addr of x[i]
        sb   x6,0(x7)      // x[i] = y[i]
        beq  x6,x0,L2     // if y[i] == 0 then exit
        addi x19,x19, 1  // i = i + 1 (advance to next byte)
        jal  x0,L1         // next iteration of loop
L2:     lw   x19,0(sp)     // restore saved x19
        addi sp,sp,4      // pop 1 doubleword from stack
        jalr x0,0(x1)      // and return
```

Since the procedure strcpy above is a leaf procedure, the compiler could allocate i to a temporary register and avoid saving and restoring x19

Please read through …

Characters and Strings in Java (p. 117~120)

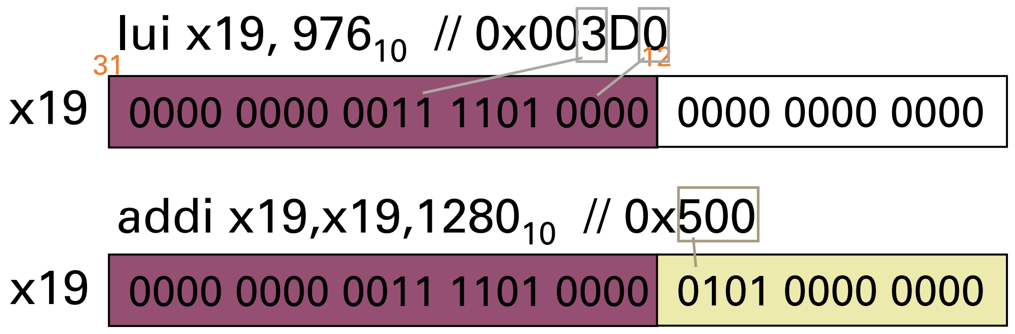
Immediate Operands (32-bit Const.)

- Most constants are small
  - 12-bit immediate is sufficient
  - But, sometimes there are bigger immediates (constants)
  - In this case, lui instruction can be used to set up a 32-bit immediate into a register for further use

- lui (load upper immediate)
  - lui rd, constant
    To copy a 20-bit constant to bits [31:12] of rd, and clear bits [11:0] of rd to 0
  - The example below loads a 32-bit immediate 0x003D 0500hex into x19 by using a lui and an addi instruction
  - For 64-bit, it can extend bit 31 to bits [63:32]

Load bits 12 to 31 with 0x003D0

Add 0x500 in the lowest 12 bits of x19



Branch Addressing

- Branch instructions specify the contents
  - ➢Opcode, two registers, offset (to target address)
- Most branch targets are near branch
  - ➢Forward or backward relative to PC
  - → PC-relative addressing (in byte)

    Target address = PC + offset

    = PC + immediate * 2

Since RISC-V instructions are **4 bytes long**, the RISC-V branch instructions could have been designed to stretch their reach by having the PC-relative address refer to the number of words between the branch and the target instruction, rather than the number of bytes. However, the RISC-V architects wanted to support the possibility of instructions that are only 2 bytes long, so the branch instructions represent the number of ***halfwords*** between the branch and the branch target. Thus, the 20-bit address field in the jal instruction can encode a distance of $\pm 2^{19}$ halfwords, or ±1MiB from the current PC

- SB format
  - ➢A 7-bit opcode, a 3-bit function code, two 5-bit register operands (rs1 and rs2), and a 12-bit address immediate
  - ➢bne x10, x11, 2000_{10}
  - ➢If x10 != x11, go to location 2000_{10} = 0111 1101 0000

0011111	01011	01010	001	01000	1100111
imm[12:6]	rs2	rs1	funct3	imm[5:1]	opcode

Unit is halfwords
Accessing to *even* address only
The discrepancy between *offset* and *immediate* may be handled by ***assembler***

Jump Addressing

- jal (unconditional jump and link)
  - jal rd, immed
  - uses 20-bit immediate for larger range

- UJ format
  - A 7-bit opcode, a 5-bit destination register operand (rd), and a 20-bit address immediate
  - The link address, which is the address of the instruction following the jal, is written to rd
  - jal x0, 2000_{10}
  - *Go to location 2000_{10} = 0111 1101 0000

00000000001111101000	00000	1101111
imm[20:1]	rd	opcode

For long jumps (e.g., to 32-bit absolute address), RISC-V allows very long jumps to any 32- bit address with a two-instruction sequence:
lui (load address[31:12] to temp register), and
Jalr (add the lower address[11:0] and jump to target)

*A simplified format; actual format is described in Section 4.4

Addressing Mode Summary

- Addressing mode
  - One of several addressing regimes delimited by their varied use of operands and/or addresses

1. Immediate addressing, where the operand is a constant within the instruction itself
2. Register addressing, where the operand is a register
3. Base or displacement addressing, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction PC = Reg + offset (in mem.)
4. PC-relative addressing, where the branch address is the sum of the PC and a constant in the instruction

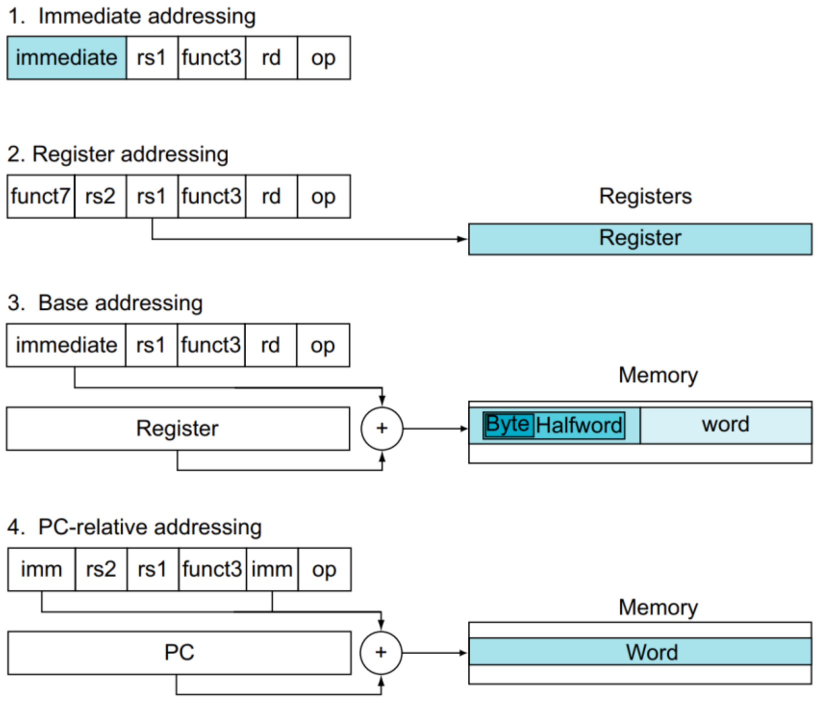
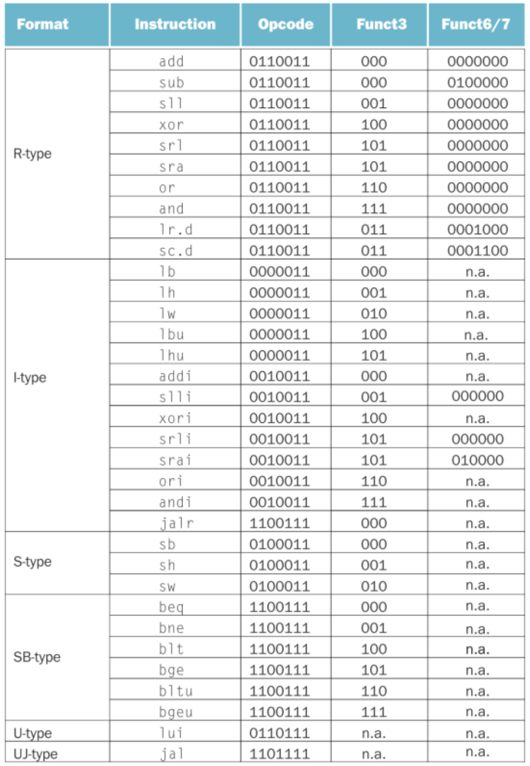


FIGURE 2.17 Illustration of four RISC-V addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, words. For mode 1, the operand is part of the instruction itself. Mode 4 addresses instructions in memory, with mode 4 adding a long address to the PC. Note that a single operation can use more than one addressing mode. Add instruction, for example, uses both immediate (addi) and register (add) addressing.
Branch instruction often uses PC-relative addressing.

Inst. Encoding

- To find out the exact operation of a given inst.
  - ➢ Check the fields: opcode, func3, and (optional) funct7



Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

FIGURE 2.18 RISC-V instruction encoding. All instructions have an opcode field, and all formats except U-type use the funct3 field. R-type instructions use the funct7 field, and immediate shifts (slli, srli, srai) use the funct6 field.

Name	**Field**						**Comments**
(Field Size)	**7 bits**	**5 bits**	**5 bits**	**3 bits**	**5 bits**	**7 bits**	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

FIGURE 2.19 Four RISC-V instruction formats. Figure 4.14.6 reveals the missing RISC-V formats for conditional branch (SB) and unconditional jumps (UJ), whose formats match the lengths of the fields in the S and U types, but the bits are swirled around. The rationale for SB and UJ makes more sense once you have an understanding of hardware given in Chapter 4, as SB and UJ simplify the hardware but give the assembler a little more to do

Please read through …

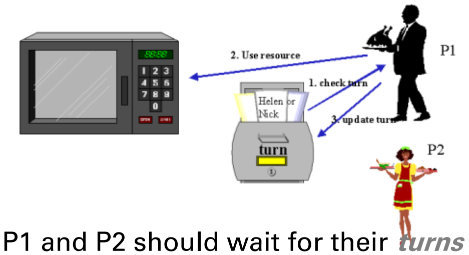
Section 2.10 (p.121~128)

for Addressing in Branches, RISC-V Addressing Mode Summary, Decoding Machine Language

Parallel Execution and Synchronization

- Parallel execution is hard when tasks need to cooperate
  - ➢E.g., some tasks are writing new values that others must read
- Synchronization is used to know when a task is finished writing so that it is safe for another to read
  - ➢A data race would occur if they don't synchronize
  - ➢where the results of the program can change depending on how events happen to occur
- Example
  - ➢Two processors sharing an area of memory
  - ➢P1 writes, then P2 reads
  - ➢A *data race* occurs if P1 and P2 don't synchronize
    - ❖Result depends on order of accesses

Synchronization



P1 and P2 should wait for their *turns* for the microwave (IMG source)

- Synchronization mechanisms are typically built with user-level routines (e.g., lock and unlock)
  - ➢that rely on hardware-supplied synchronization instructions
  - ➢Lock/unlock can be used straightforwardly to create regions where only a single processor can operate, called a mutual exclusion,
  - ➢as well as to implement more complex synchronization mechanisms

- Hardware-supplied primitives for read/write memory atomically
  - ➢Nothing else can interpose itself between the read or the write of the memory location
- Two possible implementation of the HW primitives
  - ➢A single instruction → e.g., atomic swap of register ↔ memory
  - ➢An atomic pair of instructions → the second instruction returns a value showing whether the pair of instructions was executed as if the pair was atomic

Synchronization in RISC-V

- In RISC-V, this special pair of instructions include a special load (lr.w) and a special store (sc.w)
  - ➢ if the contents of the memory location specified by the load-reserved are changed before the store-conditional to the same address occurs, then
  - ➢ the store-conditional fails and does not write the value to memory
  - ➢ Check Sec. 2.11, p. 128~131, in the textbook

- lr.w (load-reserved word)
- lr.w rd,(rs1)
  - ➢ Load from address in rs1 to rd
  - ➢ Place reservation on memory address

- sc.w (store-conditional word)
- sc.w rd,(rs1),rs2

  1. Store from rs2 to address in rs1

  2-1. Succeeds if location not changed since the lr.w
  - ❖ Write 0 in rd

  2-2. Fails if location is changed
  - ❖ Write non-zero value in rd

Synchronization Primitive Examples

- Example 1: Impl. atomic swap (to test/set lock variable) Swap the contents between [x20] and x23

```
again:   lr.w  x10,(x20)
         sc.w x11,(x20),x23        // x11 = status ( 1 means failure)
         bne  x11,x0,again         // branch if store failed
         addi x23,x10,0            // X23 = loaded value
```

- Example 2: Impl. lock acquire

```
         addi x12,x0,1             // copy locked value
again:   lr.w   x10,(x20)          // read lock
         bne  x10,x0,again         // check if it is 0 yet
         sc.w  x11,(x20),x12       // attempt to store
         bne  x11,x0,again         // branch if fails
Unlock:
         sw    x0,0(x20)           // free lock
```

C Program Execution Flow

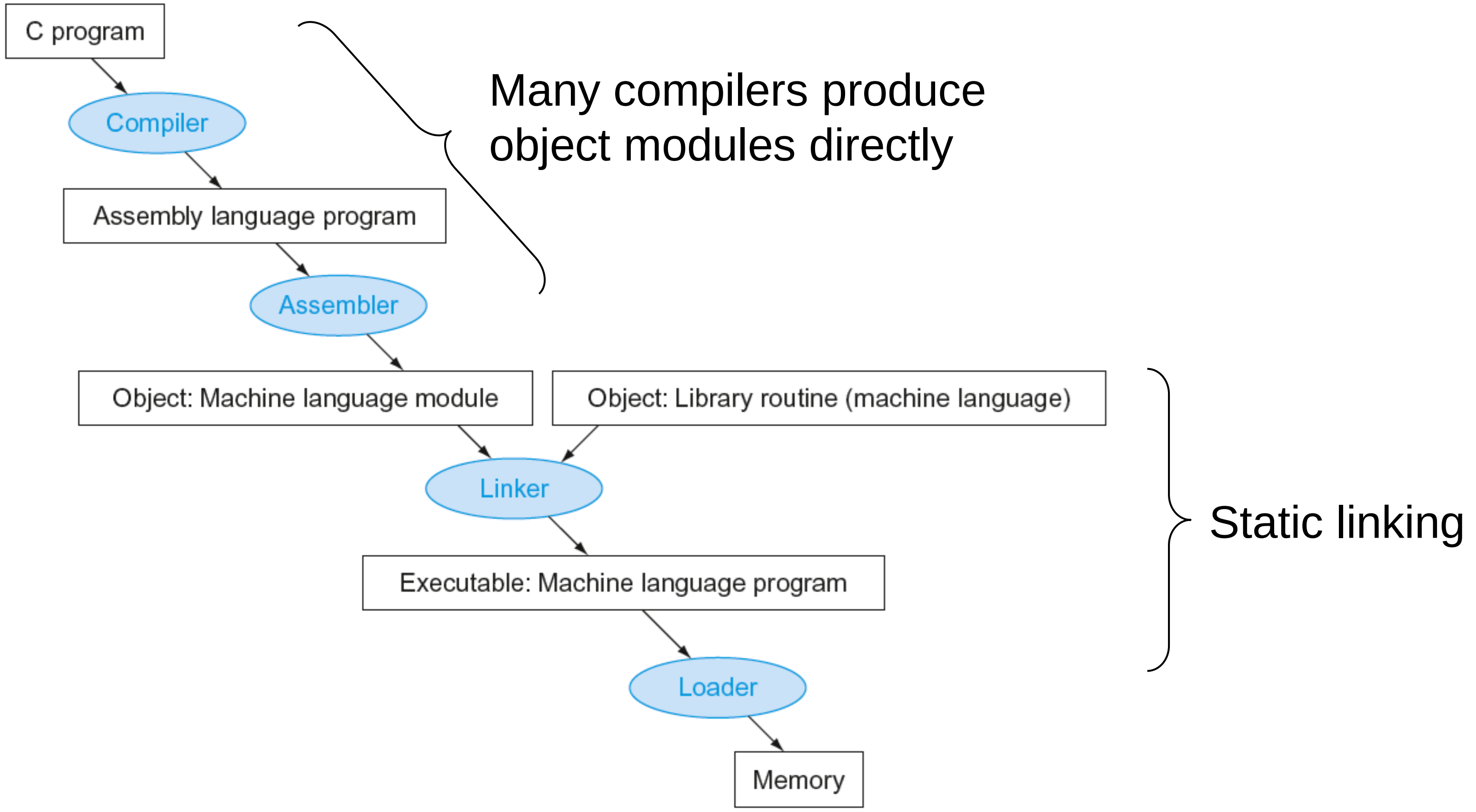

C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

FIGURE 2.20 A translation hierarchy for C. A high-level language program is first compiled into an assembly language program and then assembled into an object module in machine language. The linker combines multiple modules with library routines to resolve all references. The loader then places the machine code into the proper memory locations for execution by the processor. To speed up the translation process, some steps are skipped or combined. Some compilers produce object modules directly, and some systems use linking loaders that perform the last two steps. To identify the type of file, UNIX follows a suffix convention for files: C source files are named x.c, assembly files are x.s, object files are named x.o, statically linked library routines are x.a, dynamically linked library routes are x.so, and executable files by default are called a.out. MS-DOS uses the suffixes .C, .ASM, .OBJ, .LIB, .DLL, and .EXE to the same effect

Compiler and Assembler

- Compiler translates a high-level language program (e.g., C) into machine instructions

- Assembler further converts the instructions into machine code (binary)
    - Handle pseudoinstructions that are not implemented by the target hardware but found in machine instructions
    - li x9, 123 // load immediate value 123 into register x9
    - RISC-V assembler converts it into the following code (li is acceptable by assembler, but is not implemented in RISC-V machine language)
    - addi x9, x0, 123 // register x9 gets register x0 + 123

Assembler

- In fact, the assembler turns the assembly language program into an object file
  - ➢which is a combination of code, data, and information needed to place instructions properly in memory
  - ➢It must determine the addresses corresponding to all labels
  - ➢A symbol table helps assemblers keep track of labels used in branches and data transfer instructions

- An object file contains the six parts:
  - ➢Header: described contents of object module
  - ➢Text segment: translated instructions
  - ➢Static data segment: data allocated for the life of the program
  - ➢Relocation info: for contents that depend on absolute location of loaded program
  - ➢Symbol table: global definitions and external references
  - ➢Debug info: for associating with source code

Linker

- It takes all the independently assembled machine language programs and "stitches" them together
- It produces an *executable* file running on a computer
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external references
  - ➢ NOTE: It could leave location dependencies to be fixed by a relocating loader later

Merged file

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
A	$0000\ 0000\ 0040\ 0000_{hex}$	`lw x10, 0(x3)`
	$0000\ 0000\ 0040\ 0004_{hex}$	`jal x1, 252`$_{ten}$
	...	...
B	$0000\ 0000\ 0040\ 0100_{hex}$	`sw x11, 32(x3)`
	$0000\ 0000\ 0040\ 0104_{hex}$	`jal x1, -260`$_{ten}$
	...	...
Data segment	Address	
A	$0000\ 0000\ 1000\ 0000_{hex}$	(X)
	...	...
B	$0000\ 0000\ 1000\ 0020_{hex}$	(Y)
	...	...

List of obj files

Object file header			
	Name	Procedure A A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`lw x10, 0(x3)`	
	4	`jal x1, 0`	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`lw`	X
	4	`jal`	B
Symbol table	Label	Address	
	X	–	
	B	–	
	Name	Procedure B B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sw x11, 0(x3)`	
	4	`jal x1, 0`	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`sw`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	–	
	A	–	

- From Figure 2.14 on page 113, we know that the text segment starts at address $0000\ 0000\ 0040\ 0000_{hex}$ and the data segment at $0000\ 0000\ 1000\ 0000_{hex}$
- The text of procedure A is placed at the first address and its data at the second
- The object file header for procedure A says that its text is 100_{hex} bytes and its data is 20_{hex} bytes, so the starting address for procedure B text is $40\ 0100_{hex}$, and its data starts at $1000\ 0020_{hex}$

- Procedure A needs to find the address for the variable labeled X to put in the load instruction and to find the address of procedure B to place in the jal instruction
- Procedure B needs the address of the variable labeled Y for the store instruction and the address of procedure A for its jal instruction

Loader

- Loader is a system software
  - ➢that is called by the operating system
  - ➢to place an object program in main memory so that it is ready to execute

- In UNIX systems, the loader follows these steps:
  1. Reads the executable file header to determine size of the text and data segments
  2. Creates an address space large enough for the text and data
  3. Copies the instructions and data from the executable file into memory (also initialize the data)
     - Or set page table entries so they can be faulted in
  4. Copies the parameters (if any) to the main program onto the stack
  5. Initializes the processor registers (including sp, fp, gp) and sets the stack pointer to the first free location
  6. Branches to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program
     - Copies arguments to x10, … and calls main
     - When main returns, the start-up routine terminates the program with an exit syscall

Static vs. Dynamic Linking

- A linker combines independently assembled machine language programs (object files) and resolves all undefined labels into an executable file

Two ways to link to external object files:

- Static link: Library routines that are linked to a program during linking
  - ➢External references are resolved during the linking stage
  - ➢Bad for upgrade and code size, since the library routines become part of the executable code
- Dynamic link: Library routines that are linked to a program during execution
  - ➢Only load and link library the *procedure* when it is called
  - ➢Require procedure code to be *relocatable*
  - ➢Avoid image bloat caused by static linking of all (transitively) referenced libraries (i.e., smaller size of the built executable)
  - ➢Automatically pick up new library versions
  - ➢Check Figure 2.21 for more information

Execution Scheme of Java Programs

- The above contents emphasizes on the fast execution time for a program targeted to a specific instruction set architecture
- Java was invented with a different set of goals
  - Portability: One goal was to run safely on any computer, even if it might slow execution time; and, it is done by introducing Java bytecode instruction set
  - An interpreter, Java Virtual Machine, can execute Java bytecodes
  - Just-In-Time compilers is introduced to preserve portability and improve execution speed

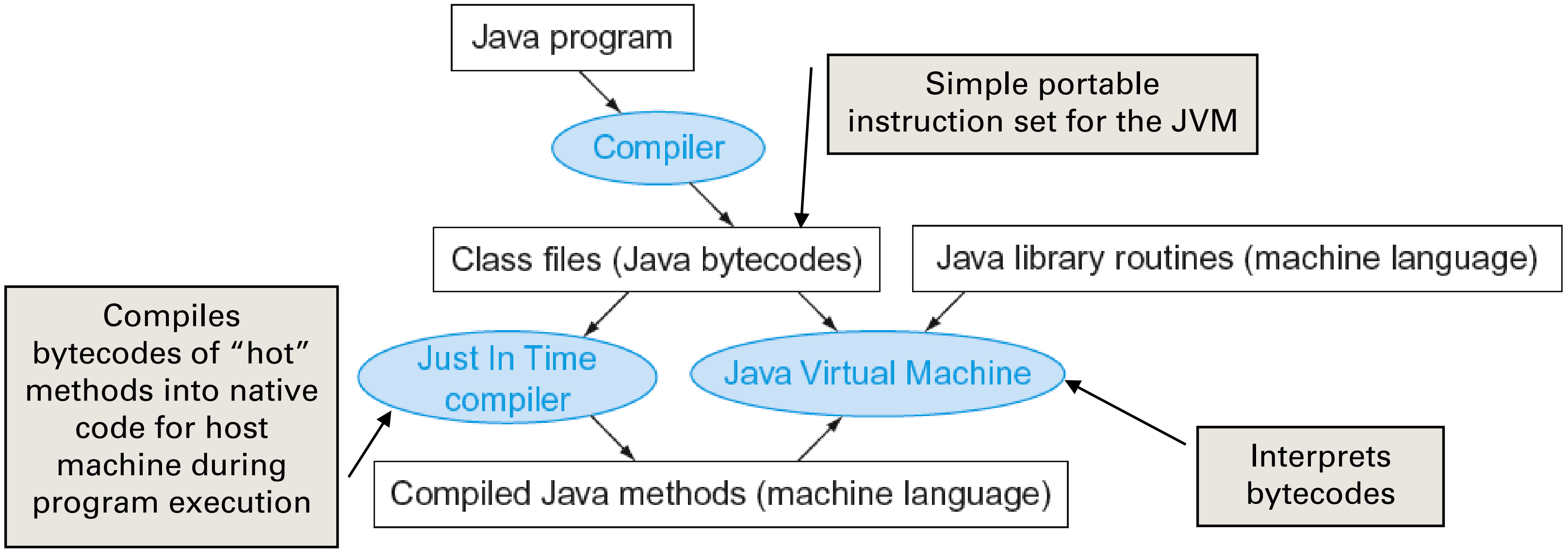
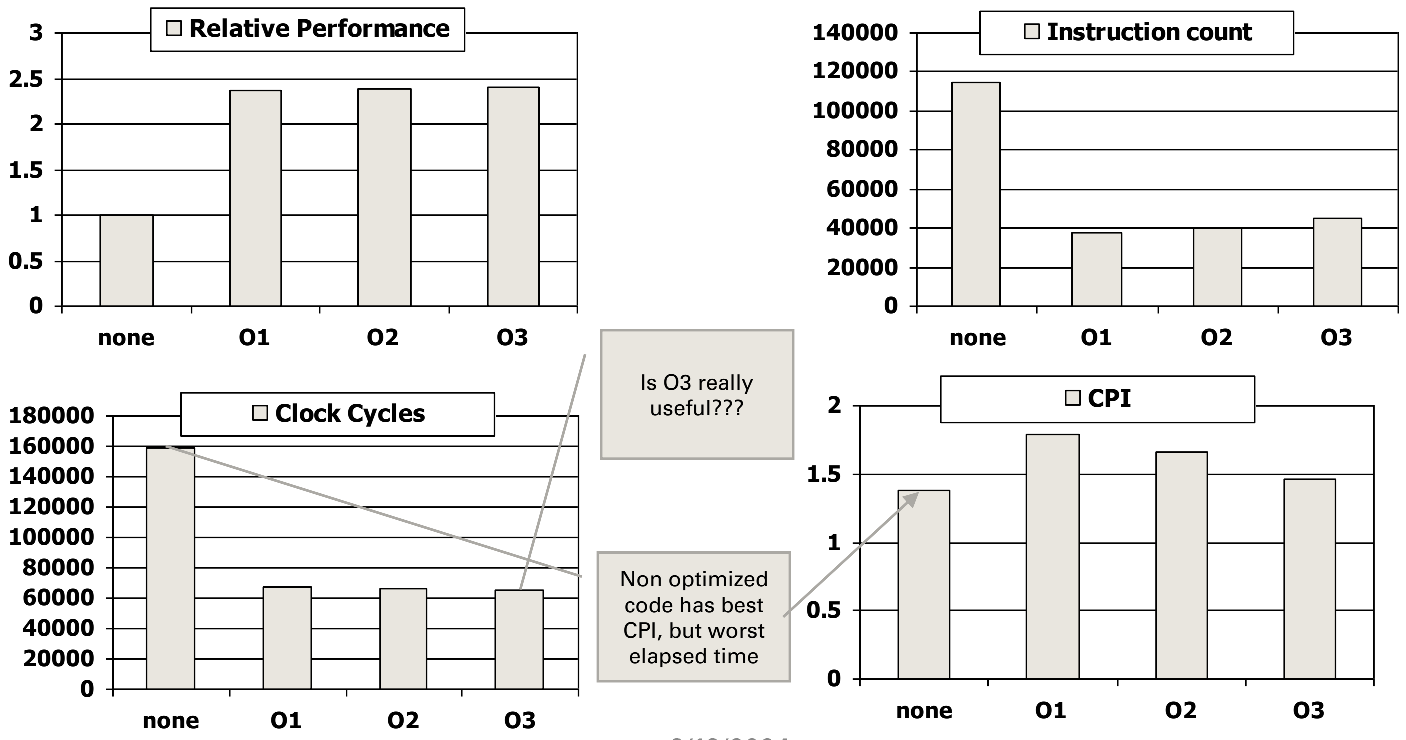

Java program
Compiler
Simple portable instruction set for the JVM
Class files (Java bytecodes)
Java library routines (machine language)
Compiles bytecodes of "hot" methods into native code for host machine during program execution
Just In Time compiler
Java Virtual Machine
Compiled Java methods (machine language)
Interprets bytecodes

FIGURE 2.22 A translation hierarchy for Java. A Java program is first compiled into a binary version of Java bytecodes, with all addresses defined by the compiler. The Java program is now ready to run on the interpreter, called the Java Virtual Machine (JVM). The JVM links to desired methods in the Java library while the program is running. To achieve greater performance, the JVM can invoke the JIT compiler, which selectively compiles methods into the native machine language of the machine on which it is running.

Understanding Program Performance

- The performance impact of compiler optimization on the C sort program performance (ref. Figure 2.26)
  - Metrics: compile time (relative performance), clock cycles (elapsed time), instruction count, and CPI
  - Compiled with GCC for Pentium 4 under Linux
    - The programs sorted 100,000 32-bit words with the array initialized to random values. These programs were run on a Pentium 4 with a clock rate of 3.06GHz and a 533MHz system bus with 2GB of PC2100 DDR SDRAM. It used Linux version 2.4.20



Performance C vs. Java

- Figures compares the performance impact of *sorts* in three different aspects
  - ➢programming languages,
  - ➢compilation versus interpretation, and
  - ➢algorithms
  - ➢Sorts: Bubblesort and Quicksort

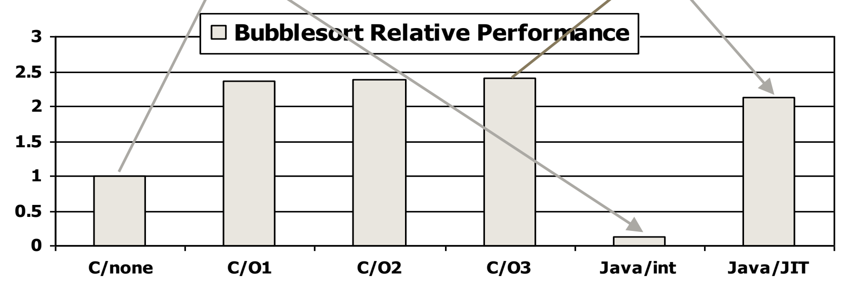
- The ratios are not as close for Quicksort, especially for Java/JIT

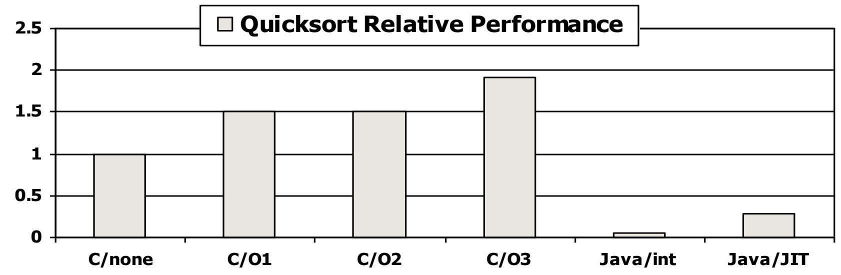
- Quicksort beats Bubble Sort

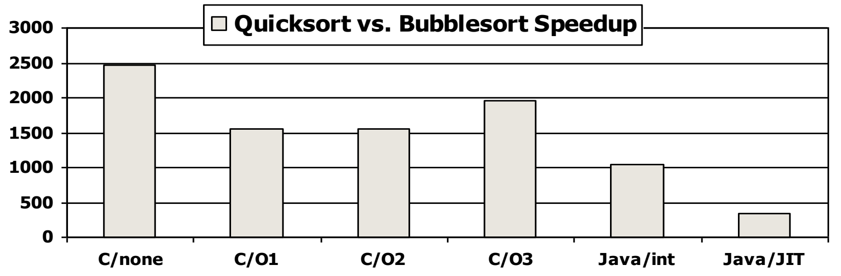
The unoptimized C program is 8.3 times faster than the interpreted Java code for Bubble Sort

Using the JIT compiler makes Java 2.1 times faster than the unoptimized C and within a factor of 1.13 (2.41/2.13) of the highest optimized C code

(ref. Figure 2.27)







Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

Arrays vs. Pointers

- This section shows
  - the relationships between <u>high-level language statements</u> and <u>machine instructions</u>
  - the impact of modern compiler optimizations the array and pointer implementation of an array data initialization
    - i.e., clear1 implemented w/ array while clear2 implemented with pointer

- The difference of the manipulation of array and pointer
- Array indexing involves
  - multiplying index by element size
  - adding to array base address
- Pointers correspond directly to memory addresses
  - Avoid indexing complexity

Comparing Two Versions of clear?

- Two compiler optimizations can do the same jobs (of the pointer code)
  - ➢ Induction variable elimination (eliminating array address calculation within loops)
  - ➢ Better to make program clearer and safer

Multiply "strength reduced" to shift

- Seeking array[i] requires slli and add within the loop
- Incrementing pointer is simpler

clear1	clear2

`clear1(int array[], int size) {`
`  int i;`
`  for (i = 0; i < size; i += 1)`
`    array[i] = 0;`
`}`
	`clear2(int *array, int size) {`
	`  int *p;`
	`  for (p = &array[0]; p < &array[size];`
	`       p = p + 1)`
	`    *p = 0;`
	`}`

`  li   x5,0          // i = 0`
`loop1:`
`  slli x6,x5,2       // x6 = i * 4`
`  add  x7,x10,x6     // x7 = address`
`                     // of array[i]`
`  sd   x0,0(x7)      // array[i] = 0`
`  addi x5,x5,1       // i = i + 1`
`  blt  x5,x11,loop1  // if (i<size)`
`                     // go to loop1`
	`  addi x5,x10,0    // p = address`
	`                   // of array[0]`
	`  slli x6,x11,2    // x6 = size * 4`
	`  add x7,x10,x6    // x7 = address`
	`                   // of array[size]`
	`loop2:`
	`  sd x0,0(x5)      // Memory[p] = 0`
	`  addi x5,x5,4     // p = p + 4`
	`  bltu x5,x7,loop2 // if (p<&array[size])`
	`                   // go to loop2`

RISC-V Instruction Set

- Targeting a wide variety of computers, the RISC-V ISA is divided into:
  - a base architecture with
  - several extensions (five standard extensions are listed)
  - There are more extensions available listed in [this page]
- Each extension is named with a letter of the alphabet
  - The base architecture is named I for integer (Figure 2.42)
- Figure 2.41 lists the remaining five instructions in the base RISC-V ISA

Mnemonic	Description	Insn. Count
I	Base architecture	51
M	Integer multiply/divide	13
A	Atomic operations	22
F	Single-precision floating point	30
D	Double-precision floating point	32
C	Compressed instructions	36

FIGURE 2.42 The RISC-V instruction set architecture is divided into the base ISA, named I, and five standard extensions, M, A, F, D, and C. RISC-V International is developing many other optional instruction extensions. Unlike most architectures, the RISC-V software stack only assumes the base architecture (I), with other extensions optional that are only issued by the compiler if the processor includes those options

Instruction	Name	Format	Description
Add upper immediate to PC	`auipc`	U	Add 20-bit upper immediate to PC; write sum to register
Set if less than	`slt`	R	Compare registers; write Boolean result to register
Set if less than, unsigned	`sltu`	R	Compare registers; write Boolean result to register
Set if less than, immediate	`slti`	I	Compare registers; write Boolean result to register
Set if less than immediate, unsigned	`sltiu`	I	Compare registers; write Boolean result to register

FIGURE 2.41 The remaining five instructions in the base RISC-V ISA

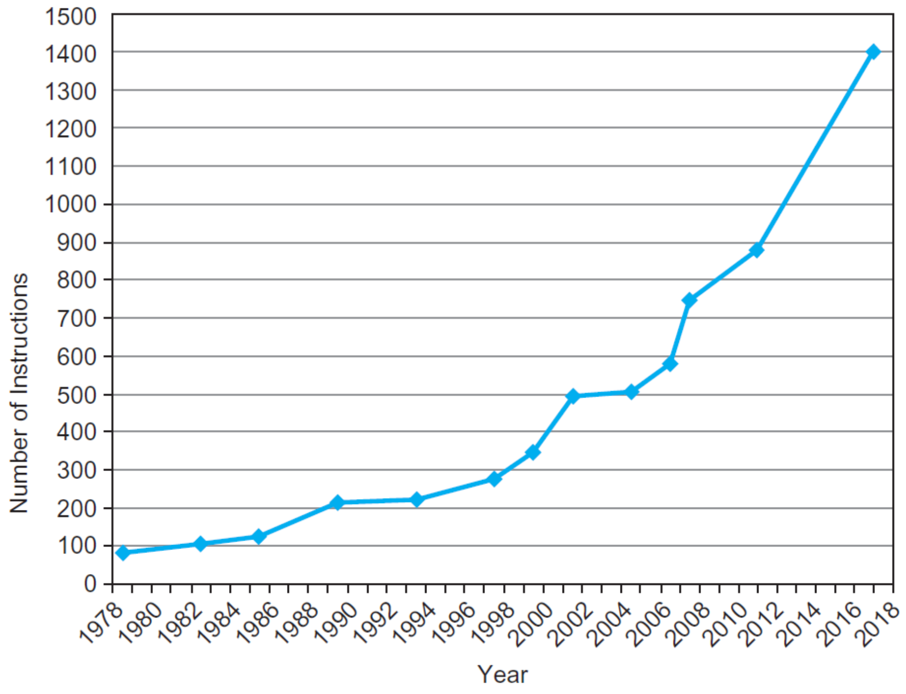
Fallacies

- Powerful instruction ⇒ higher performance
  - ➢Fewer instructions required
  - ➢But complex instructions are hard to implement
    - ❖May slow down all instructions, including simple ones
  - ➢Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - ➢But modern compilers are better at dealing with modern processors
  - ➢More lines of code ⇒ more errors and less productivity

Fallacies (Cont'd)

- Backward compatibility ⇒ instruction set doesn't change
  - ➢But they do accrete more instructions

x86 instruction set



Pitfalls

- Sequential words are not at sequential addresses
  - ➢Increment by 4 (for 32-bit architecture), not by 1 !!!
- Keeping a pointer to an automatic variable after procedure returns
  - ➢E.g., passing pointer back via an argument
  - ➢Pointer becomes invalid when stack popped

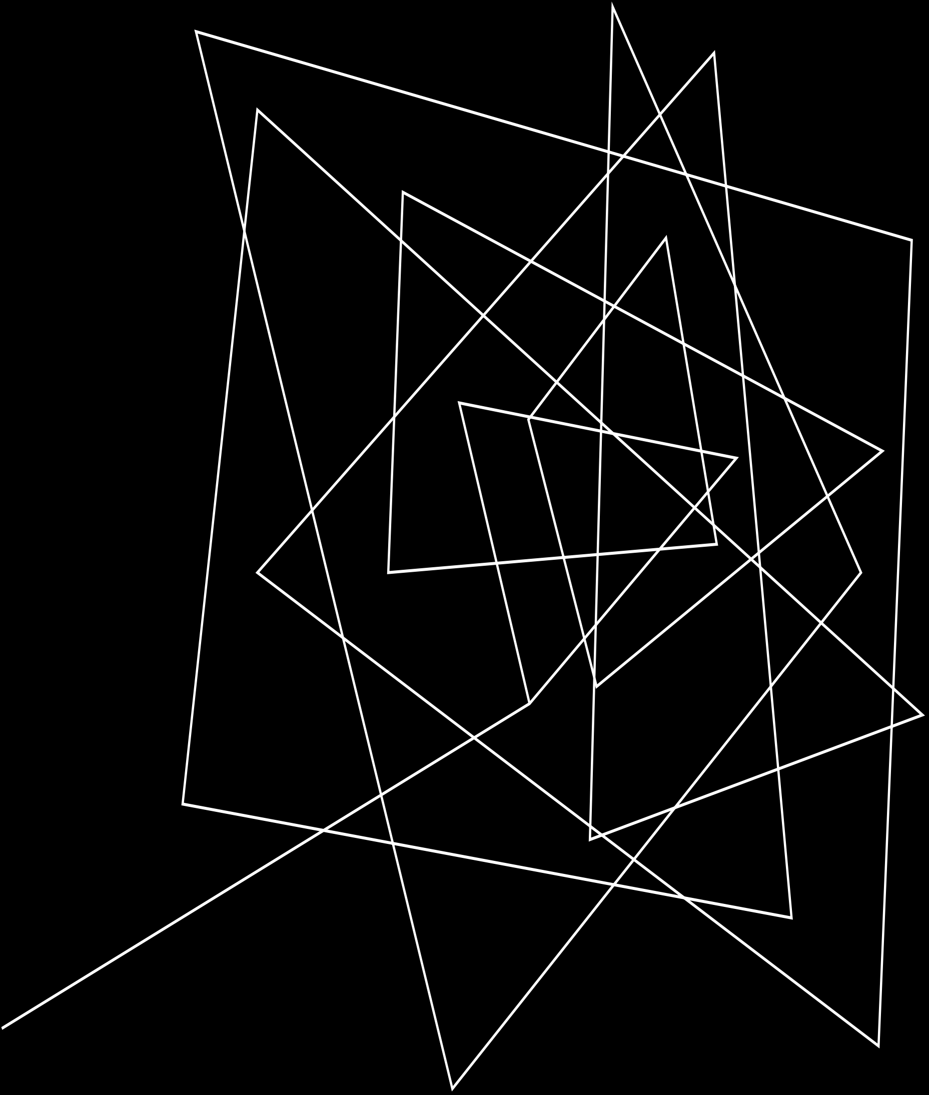
Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Good design demands good compromises

- Make the common case fast

- Layers of software/hardware
  - ➢Compiler, assembler, hardware

- RISC-V: typical of RISC ISAs
  - ➢c.f. x86



Questions?