Chapter 8
Processor Types and Instruction Sets

Douglas Comer. (2017). *Essentials of Computer Architecture* (2nd ed.). CRC Press.

M. Morris Mano, Charles R. Kime. (2015). *Logic and computer design fundamentals* (5th ed.). Pearson.

Contents

Contents

Contents

1. Mathematical Power, Convenience and Cost

- From a mathematical point of view, a wide variety of computational models provide equivalent computing power.
  - In theory, as long as a processor offers a few basic operations, the processor has sufficient power to compute any computable function.
    - In a mathematical sense, only three operations are needed to compute any computable function: add one, subtract one, and branch if a value is nonzero.
- Programmers understand that although only a minimum set of operations are necessary, a minimum is neither convenient nor practical.
  - That is, the set of operations is designed for convenience rather than for mere functionality.
    - For example, it is possible to compute a quotient by repeated subtraction. However, a program that uses repeated subtraction to compute a quotient runs slowly. Thus, most processors operations include hardware for each basic arithmetic operation: addition, subtraction, multiplication, and division.
- The set of operations a processor provides represents a tradeoff among the cost of the hardware, the convenience for a programmer, and engineering considerations such as power consumption.

2. Instruction Set Architecture

- When an architect designs a programmable processor, the architect must make two key decisions:
  - *Instruction set*: the set of operations the processor provides
    - We assume that on each iteration of its fetch-execute cycle, a processor *executes* one instruction.
  - *Instruction representation*: the format for each operation
    - Refers to the binary representation that the hardware uses for instructions.
- The definition of an instruction set and the corresponding representation define an *Instruction Set Architecture (ISA)*.

3. Opcodes, Operands, and Results

- Conceptually, each instruction contains three parts
  - *Opcode (operation code)*
    - The exact operation to be performed.
    - An opcode is a number. Each operation must be assigned a unique opcode. For example, integer addition might be assigned opcode 5, and integer subtraction might be assigned opcode 12.
  - *Operands*
    - Refers to a value that is needed to perform an operation. The definition of an instruction set specifies the exact number of operands for each instruction, and the possible values (e.g., the addition operation takes two signed integers).
  - *Results*
    - In some architectures, one or more of the operands specify where the processor should place results of an instruction (e.g., the result of an arithmetic operation); in others, the location of the result is determined automatically.

4. Typical Instruction Format

- Each instruction is represented as a binary string.

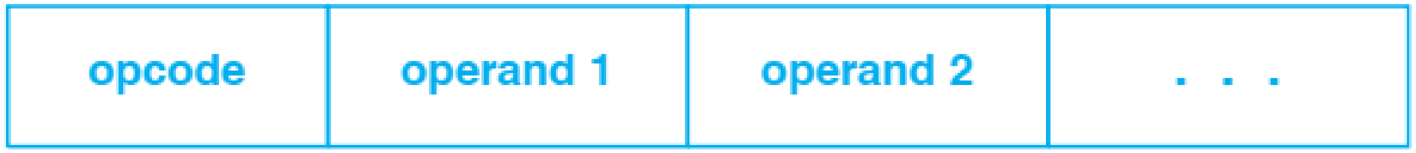


Figure 5.1 The general instruction format that many processors use. The opcode at the beginning of an instruction determines exactly which operands follow.

5. Variable-Length vs. Fixed-Length Instructions

- Should each instruction be the same size (i.e., occupy the same number of bytes) or should the length depend on the quantity and type of the operands?
- *Variable-length* to characterize an instruction set that includes multiple instruction sizes.
- *Fixed-length* to characterize an instruction set in which every instruction is the same size.
- By comparison, fixed-length instructions require less complex hardware.
    - Fixed-length instructions allow processor hardware to operate at higher speed because the hardware can compute the location of the next instruction easily. Thus, many processors force all instructions to be the same size, even if some instructions can be represented in fewer bits than others.
- When a fixed-length instruction set is employed, some instructions contain extra fields that the hardware ignores. The unused fields should be viewed as part of a hardware optimization, not as an indication of a poor design.

6. General-Purpose Registers

- A register has a fixed size (e.g., 32 or 64 bits) and supports two basic operations: *fetch* and *store*.
- *General-purpose* registers that are used as a temporary storage mechanism.
- A processor usually has a small number of general-purpose registers, and each register is usually the size of an integer.
    - For example, on a processor that provides thirty-two bit arithmetic, each general-purpose register holds thirty-two bits.
    - A general-purpose register can hold an operand needed for an arithmetic instruction or the result of such an instruction.
- General-purpose registers have the same semantics as memory: a fetch operation returns the value specified in the previous store operation. Similarly, a store operation replaces the contents of the register with a new value.

7. Floating Point Registers and Register Identification

- Processors that support floating point arithmetic often use a separate set of registers to hold floating point values.
- Confusion can arise because both general-purpose registers and floating point registers are usually numbered starting at zero – the instruction determines which registers are used.
- For example, if registers 3 and 6 are specified as operands for an integer instruction, the processor will extract the operands from the general-purpose registers.
    - However, if registers 3 and 6 are specified as operands for a floating point instruction, the floating point registers will be used.

8. Programming with Registers

- Many processors require operands to be placed in general-purpose registers before an instruction is executed.
- Some processors also place the results of an instruction in a general-purpose register.
- For example, to add two integers in variables X and Y and place the result in variable Z, a programmer must create a series of instructions that move values to the corresponding registers. If general-purpose registers 3, 6, and 7 are available, the program might contain four instructions that perform the following steps:
    - Load a copy of variable X from memory into register 3
    - Load a copy of variable Y from memory into register 6
    - Add the value in register 3 to the value in register 6, and place the result in register 7
    - Store a copy of the value in register 7 to variable Z in memory

9. Register Banks

- Some architectures divide registers into multiple banks, and require the operands for an instruction to come from separate banks.
  - For example, on a processor that uses two register banks, an integer add instruction may require the two operands to be from separate banks.
- Register banks allow the hardware to operate faster because each bank has a separate physical access mechanism and the mechanisms operate simultaneously.
  - Thus, when the processor executes an instruction that accesses two operands in registers, both operands can be obtained at the same time.

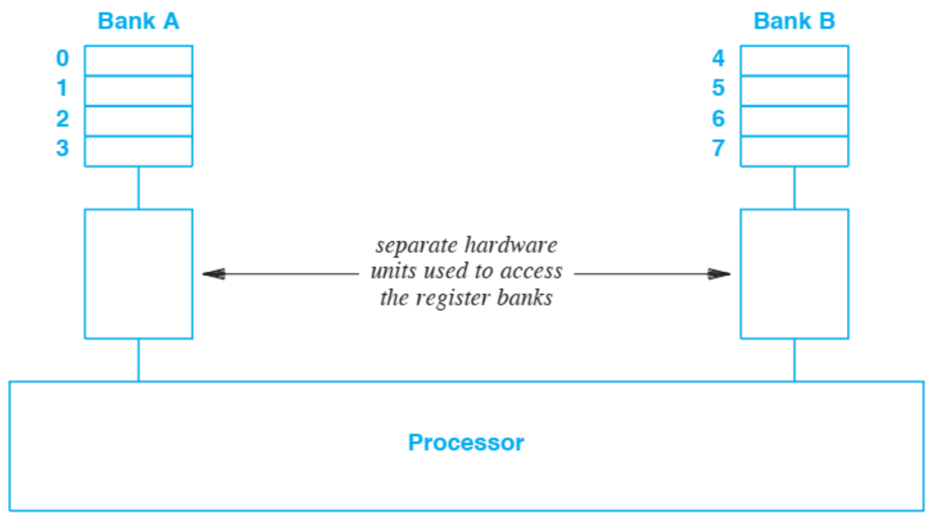


Figure 5.2 Illustration of eight registers divided into two banks. Hardware allows the processor to access both banks at the same time.

9. Register Banks

- *Register conflict*
  - Assume we want to implement the statements on a processor that has two register banks A and B.

    $R \leftarrow X + Y$ Suppose X is in bank A, Y is in bank B.

    $S \leftarrow Z - X$ Z must be in the opposite bank than X. Z is in bank B.

    $T \leftarrow Y + Z$ **Y and Z must be in different banks. !!!**

    What happens when a register conflict arises?

    - The programmer must either reassign registers or insert an instruction to copy values. For example, we could insert an extra instruction that copies the value of Z into a register in bank A before the final addition is performed.

10. Complex And Reduced Instruction Sets

- Computer architects divide instruction sets into two broad categories that are used to classify processors:
  - Complex Instruction Set Computer (CISC)
  - Reduced Instruction Set Computer (RISC)
- A *CISC processor* usually includes many instructions (typically hundreds), and each instruction can perform an arbitrarily complex computation.
  - Intel’s x86 instruction set is classified as CISC because a processor provides hundreds of instructions, including complex instructions that require a long time to execute (e.g., one instruction manipulates graphics in memory and others compute the sine and cosine functions).

10. Complex And Reduced Instruction Sets

- A *RISC processor* strives for a minimum set that is sufficient for all computation (e.g., thirty-two instructions).
  - Instead of allowing a single instruction to compute an arbitrary function, each instruction performs a basic computation.
  - To achieve the highest possible speed, RISC designs constrain instructions to be a fixed size. A RISC processor is designed to execute an instruction in one clock cycle.
    - Arm Limited and MIPS Corporation have each created RISC architectures with limited instructions that can be executed in one clock cycle. The ARM designs are especially popular in smart phones and other low-power devices.

10. Complex And Reduced Instruction Sets

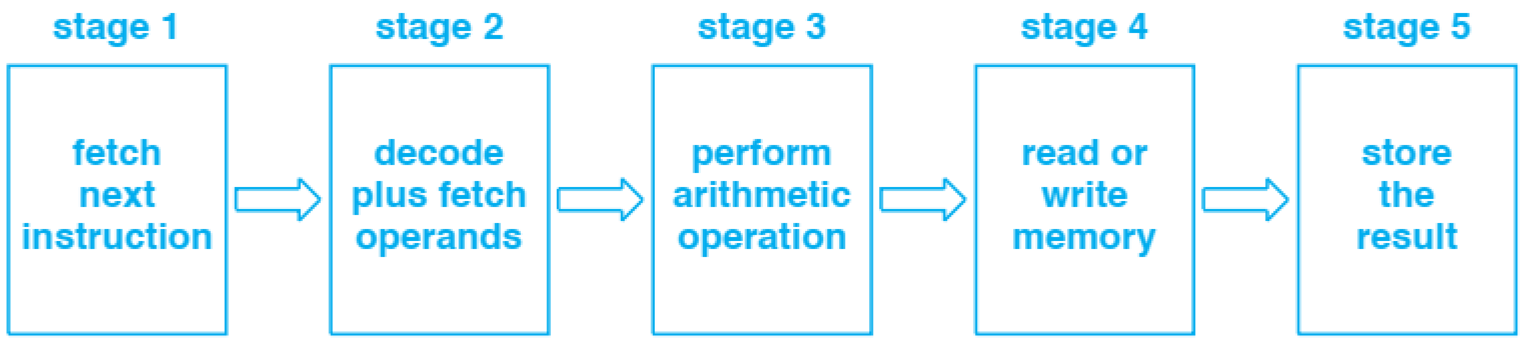
- A processor is classified as CISC if the instruction set contains instructions that perform complex computations that can require long times; a processor is classified as RISC if it contains a small number of instructions that can each execute in one clock cycle.

11. RISC Design and The Execution Pipeline

- A RISC processor is designed so the processor can *complete* one instruction on each clock cycle.
- Processors divide the fetch-execute cycle into several steps:
  - Fetch the next instruction
  - Decode the instruction and fetch operands from registers
  - Perform the arithmetic operation specified by the opcode
  - Perform memory read or write, if needed
  - Store the result back to the registers

Execution Pipeline

- To enable high speed, RISC processors contain parallel hardware units that each perform one step listed above.
- The hardware is arranged in a multistage *pipeline*, which means the results from one hardware unit are passed to the next hardware unit.



Instructions move through the pipeline like an assembly line: at any time, the pipeline contains five instructions.

Figure 5.3 An example pipeline of the five hardware stages that are used to perform the fetch-execute cycle.

Execution Pipeline

- The speed of a pipeline arises because all stages can operate in parallel – while the fourth stage executes an instruction, the third stage fetches the operands for the next instruction. Thus, a stage never needs to delay because an instruction is always ready on each clock cycle.
- Although a RISC processor cannot perform all steps of the fetch execute cycle in a single clock cycle, an instruction pipeline with parallel hardware provides approximately the same performance:
  - Once the pipeline is full, one instruction completes on every clock cycle.

Time

clock	stage 1	stage 2	stage 3	stage 4	stage 5
1	inst. 1	-	-	-	-
2	inst. 2	inst. 1	-	-	-
3	inst. 3	inst. 2	inst. 1	-	-
4	inst. 4	inst. 3	inst. 2	inst. 1	-
5	inst. 5	inst. 4	inst. 3	inst. 2	inst. 1
6	inst. 6	inst. 5	inst. 4	inst. 3	inst. 2
7	inst. 7	inst. 6	inst. 5	inst. 4	inst. 3
8	inst. 8	inst. 7	inst. 6	inst. 5	inst. 4

Figure 5.4 Instructions passing through a five-stage pipeline. Once the pipeline is filled, each stage is busy on each clock cycle.

12. Pipelines and Instruction Stalls

- Consider a program that contains two successive instructions that perform an addition and subtraction on operands and results located in registers that we will label A, B, C, D, and E:

  Instruction K: C ← add A B

  Instruction K+1: D ← subtract E C

  Such a sequence causes a stall on a pipelined processor.

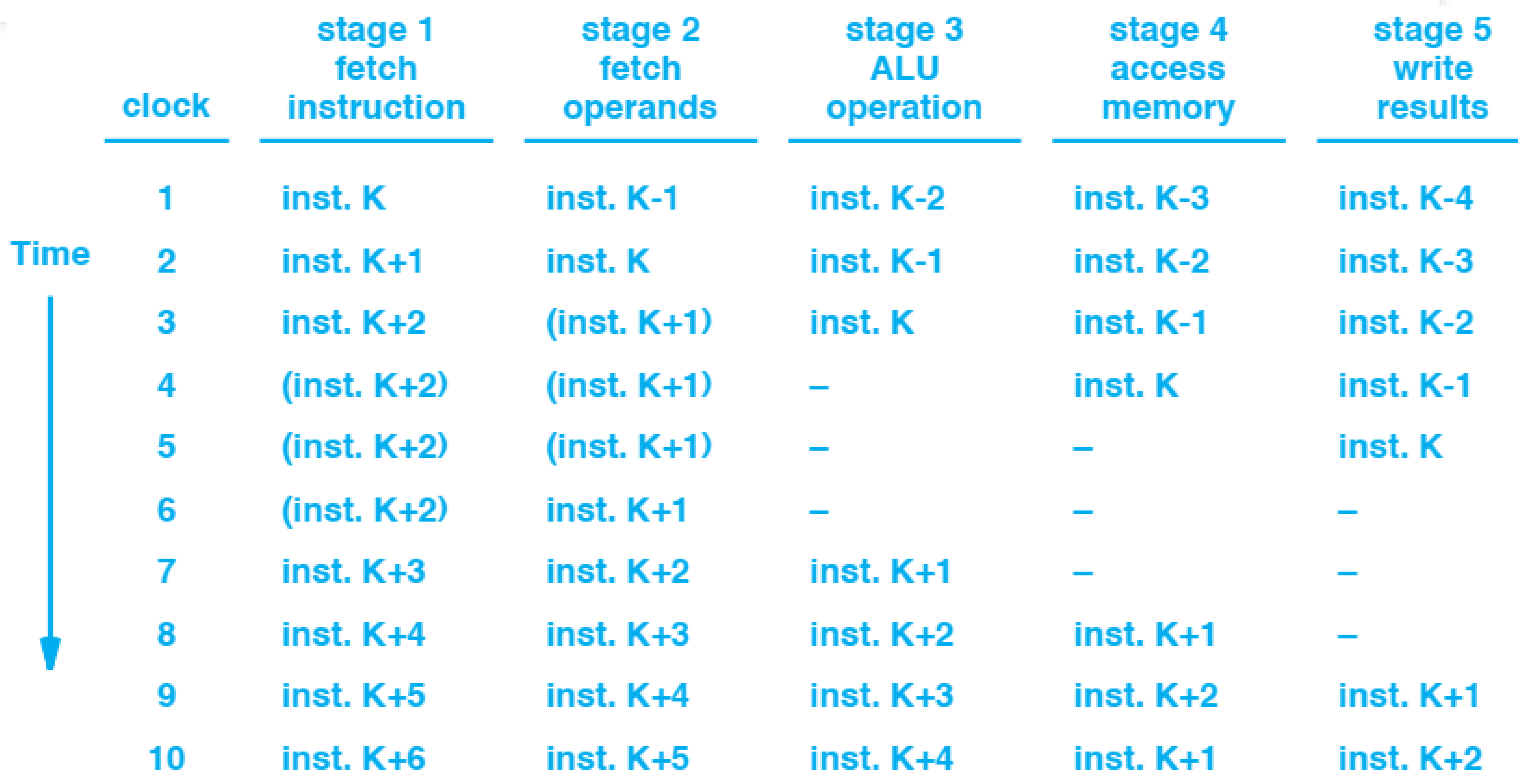
  - Instruction K+1 encounters a problem because operand C is not available in time. The hardware must wait for instruction K to finish before fetching the operands for instruction K+1.
  - We say that a stage of the pipeline *stalls* to wait for the operand to become available.

12. Pipelines and Instruction Stalls

- What happens during a pipeline stall

  Instruction K: C ← add A B

  Instruction K+1: D ← subtract E C



clock	stage 1 fetch instruction	stage 2 fetch operands	stage 3 ALU operation	stage 4 access memory	stage 5 write results
1	inst. K	inst. K-1	inst. K-2	inst. K-3	inst. K-4
2	inst. K+1	inst. K	inst. K-1	inst. K-2	inst. K-3
3	inst. K+2	(inst. K+1)	inst. K	inst. K-1	inst. K-2
4	(inst. K+2)	(inst. K+1)	–	inst. K	inst. K-1
5	(inst. K+2)	(inst. K+1)	–	–	inst. K
6	(inst. K+2)	inst. K+1	–	–	–
7	inst. K+3	inst. K+2	inst. K+1	–	–
8	inst. K+4	inst. K+3	inst. K+2	inst. K+1	–
9	inst. K+5	inst. K+4	inst. K+3	inst. K+2	inst. K+1
10	inst. K+6	inst. K+5	inst. K+4	inst. K+1	inst. K+2

Figure 5.5 Illustration of a pipeline stall. Instruction K+1 cannot proceed until an operand from instruction K becomes available.

13. Other Causes of Pipeline Stalls

- In addition to waiting for operands, a pipeline can stall when the processor executes any instruction that delays processing or disrupts the normal flow.
  - Accesses external storage
  - Invokes a coprocessor
  - Branches to a new location
  - Calls a subroutine
- The most sophisticated processors contain additional hardware to avoid stalls. For example, some processors contain two copies of a pipeline, which allows the processor to start decoding the instruction that will be executed if a branch is taken as well as the instruction that will be executed if a branch is not taken.
  - The two copies operate until a branch instruction can be executed. At that time, the hardware knows which copy of the pipeline to follow; the other copy is ignored.
- Other processors contain special shortcut hardware that passes a copy of a result back to a previous pipeline stage.

13. Other Causes of Pipeline Stalls

- Consequences for programmers
    - To achieve maximum speed, a program for a RISC architecture must be written to accommodate an instruction pipeline.
    - For example, a programmer should avoid introducing unnecessary branch instructions. Similarly, instead of referencing a result register immediately in the following instruction, the reference can be delayed.

```
C ← add A B
D ← subtract E C
F ← add G H
J ← subtract I F
M ← add K L
P ← subtract M N
```

(a)

```
C ← add A B
F ← add G H
M ← add K L
D ← subtract E C
J ← subtract I F
P ← subtract M N
```

(b)

Figure 5.6 (a) A list of instructions, and (b) the instructions reordered to run faster in a pipeline. Reducing pipeline stalls increases speed.

Rearranging code sequences can increase the speed of processing when the hardware uses an instruction pipeline; programmers view the reordering as an optimization that can increase speed without affecting correctness.

14. no-op Instructions

- In some cases, the instructions in a program cannot be rearranged to prevent a stall.
    - In such cases, programmers can document stalls so anyone reading the code will understand that a stall occurs. Such documentation is especially helpful if a program is modified because the programmer who performs the modification can reconsider the situation and attempt to reorder instructions to prevent a stall.
- In places where stalls occur, insert extra *no-op* instructions in the code.
    - no-op instruction: An instruction that does absolutely nothing except occupy time

15. Forwarding

- Consider the example of two instructions where operands A, B, C, D, and E are in registers:

  Instruction K: C ← add A B

  Instruction K+1: D ← subtract E C

  Such a sequence causes a stall on a pipelined processor.

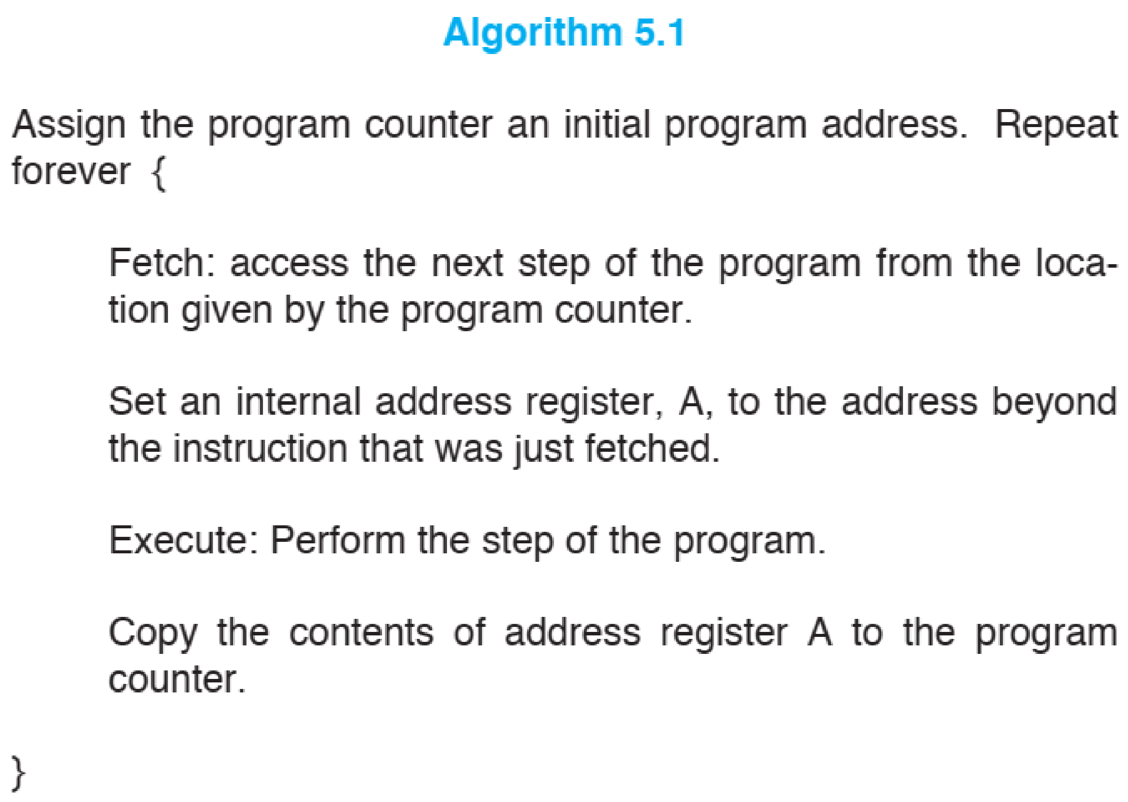
- Processor that implements *forwarding* can avoid the stall by arranging for the hardware to **detect the dependency and automatically pass the value for C from instruction K directly to instruction K + 1**.
  - That is, a copy of the output from the ALU in instruction K is forwarded directly to the input of the ALU in instruction K + 1. As a result, instructions continue to fill the pipeline, and no stall occurs.

16. Types of Operations

- Instructions are divided into basic categories
  - Integer arithmetic instructions
  - Floating point arithmetic instructions
  - Logical instructions (also called Boolean operations)
  - Data access and transfer instructions
  - Conditional and unconditional branch instructions
  - Processor control instructions
  - Graphics instructions

17. Program Counter, Fetch-Execute, and Branching

- To implement the fetch-execute cycle and a move to the next instruction, the processor uses a special-purpose internal register known as an *instruction pointer* or *program counter*.
- When a fetch-execute cycle begins, the program counter contains the address of the instruction to be executed. After an instruction has been fetched, the program counter is updated to the address of the next instruction.
- The update of the program counter during each fetch-execute cycle means the processor will automatically move through successive instructions in memory.



Algorithm 5.1

```
Assign the program counter an initial program address.  Repeat
forever {

        Fetch: access the next step of the program from the loca-
        tion given by the program counter.

        Set an internal address register, A, to the address beyond
        the instruction that was just fetched.

        Execute: Perform the step of the program.

        Copy the contents of address register A to the program
        counter.

}
```

17. Program Counter, Fetch-Execute, and Branching

- *Absolute branch*
  - Computes a memory address, and the address specifies the location of the next instruction to execute.
  - Typically, an absolute branch instruction is known as a *jump*.
    - For example: **jump 0x05DE**
      - The processor loads 0x05DE into the internal address register, which is copied into the program counter before the next instruction is fetched. In other words, the next instruction fetch will occur at memory location 0x05DE.
- *Relative branch*
  - Does not specify an exact memory address. Instead, a relative branch computes a positive or negative increment for the program counter.
  - For example, the instruction: **br +8**
    - The processor branches to a location that is eight bytes beyond the current location (i.e., beyond the current value of the program counter).

17. Program Counter, Fetch-Execute, and Branching

- Most processors also provide an instruction to invoke a subroutine, typically *jsr* (*jump subroutine*).
  - A jsr instruction operates like a branch instruction with a key difference: before the branch occurs, the jsr instruction saves the value of the address register, A. When it finishes executing, a subroutine returns to the caller. To do so, the subroutine executes an absolute branch to the saved address. Thus, when the subroutine finishes, the fetch-execute cycle resumes at the instruction immediately following the jsr.

18. Subroutine Calls, Arguments, and Register Windows

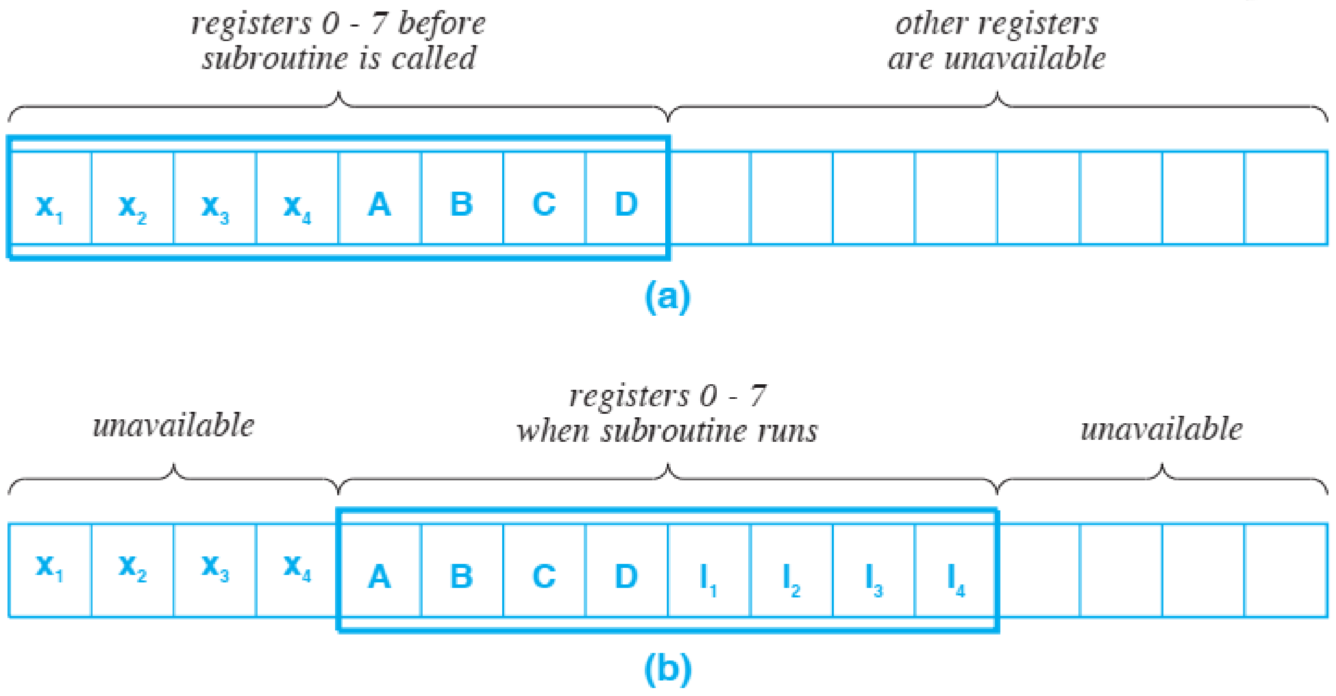
- The calling program supplies a set of arguments that the subroutine uses in its computation.
  - For example, the function call cos(3.14159) has the floating point constant 3.14159 as an argument.
- One of the principal differences among processors arises from the way the underlying hardware passes arguments to a subroutine.
  - Uses memory to pass arguments
    - Arguments are stored on the stack in memory before the call, and the subroutine extracts the values from the stack when they are referenced.
  - Uses either general-purpose or special-purpose registers to pass arguments.
    - Much faster than using stack in memory
    - Few processors provide special-purpose registers for argument passing, general-purpose registers are typically used.
    - (An ARM architecture passes some arguments in registers and some in memory)

18. Subroutine Calls, Arguments, and Register Windows

- A programmer faces a tradeoff:
  - Using a general-purpose register to pass an argument can increase the speed of a subroutine call
  - Using the register to hold a data value can increase the speed of general computation.
  - Thus, a programmer must choose which arguments to keep in registers and which to store in memory.
- Some processors include an optimization for argument passing known as a *register window*.
  - Although a processor has a large set of general-purpose registers, the register hardware only exposes a subset of the registers at any time. The subset is known as a *window*.
  - The window moves automatically each time a subroutine is invoked, and moves back when the subroutine returns.

18. Subroutine Calls, Arguments, and Register Windows

- In the figure, the hardware has 16 registers, but only 8 registers are visible at any time; others are unavailable.
- When a subroutine is called, the hardware changes the set of registers that are visible by sliding the window.
- The calling program places arguments A through D in registers 4 through 7, and the subroutine finds the arguments in registers 0 through 3.
    - Registers with values xi are only available to the calling program.
- The Sparc architecture has 128 or 144 physical registers and a window size of 32 registers; however, only 8 of the registers in the window overlap (i.e., only eight registers can be used to pass arguments).



registers 0 - 7 before subroutine is called
other registers are unavailable
x_1 x_2 x_3 x_4 A B C D
(a)
unavailable
registers 0 - 7 when subroutine runs
unavailable
x_1 x_2 x_3 x_4 A B C D l_1 l_2 l_3 l_4
(b)

Figure 5.8 Illustration of a register window (a) before a subroutine call, and (b) during the call. Values *A*, *B*, *C*, and *D* correspond to arguments that are passed.

19. The Principle of Orthogonality

- In addition to the technical aspects of instruction sets discussed above, an architect must consider the aesthetic aspects of a design.
- In particular, an architect strives for *elegance*.
  - Elegance relates to human perception:
    - How does the instruction set appear to a programmer?
    - How do instructions combine to handle common programming tasks?
    - Are the instructions balanced (if the set includes right-shift, does it also include left-shift)?
  - One particular aspect of elegance, known as *orthogonality*, concentrates on eliminating unnecessary duplication and overlap among instructions.
    - We say that an instruction set is orthogonal if each instruction performs a unique task.
    - **The principle of orthogonality specifies that each instruction should perform a unique task without duplicating or overlapping the functionality of other instructions.**

20. Condition Codes and Conditional Branching

- On many processors, executing an instruction results in a *status*, which the processor stores in an internal hardware mechanism. A later instruction can use the status to decide how to proceed.
  - For example, when it executes an arithmetic instruction, the ALU sets an internal register known as a *condition code* that contains bits to record whether the result is positive, negative, zero, or an arithmetic overflow occurred. A *conditional branch* instruction that follows the arithmetic operation can test one or more of the condition code bits, and use the result to determine whether to branch.

```
        cmp   r4, r5    # compare regs. 4 & 5, and set condition code
        be    lab1      # branch to lab1 if cond. code specifies equal
        mov   r3, 0     # place a zero in register 3
lab1:   ...program continues at this point
```

Figure 5.11 An example of using a condition code. An ALU operation sets the condition code, and a later *conditional branch* instruction tests the condition code.