

UNIT 5

Central Processing Unit

(10 Hours) — *Computer Organization and Architecture*

The Central Processing Unit (CPU) is the part of the computer that fetches, decodes, and executes instructions. This unit studies how registers inside the CPU are organized and connected through internal buses, how a stack is used for temporary storage and subroutine handling, how instructions are formatted and how operands are addressed, and finally how the CPU transfers data, manipulates it, and controls the flow of a program through branching, subroutine calls, and interrupts.

5.1 General Register Organization

A CPU needs a set of registers to hold data temporarily during the execution of an instruction. Early computers used a single accumulator (AC) for this purpose, so every arithmetic or logic instruction referred implicitly to the AC. Modern CPUs instead provide a set of general-purpose registers ($R_1, R_2, \dots, R_n$) that the ALU can read from and write to. This is called a general register organization, and it reduces the number of memory references needed during program execution, since intermediate results can stay inside the CPU.

Bus-Organized CPU

A common way to connect several registers to one ALU is through a set of common buses. In the classic scheme, two buses (Bus A and Bus B) carry operands out of the selected registers into the ALU, and a third path carries the ALU result back into a destination register. Multiplexers (MUX A, MUX B) select which register drives each bus, and a decoder at the register bank selects which register accepts the incoming result.

General Register Organization (Bus-Connected)

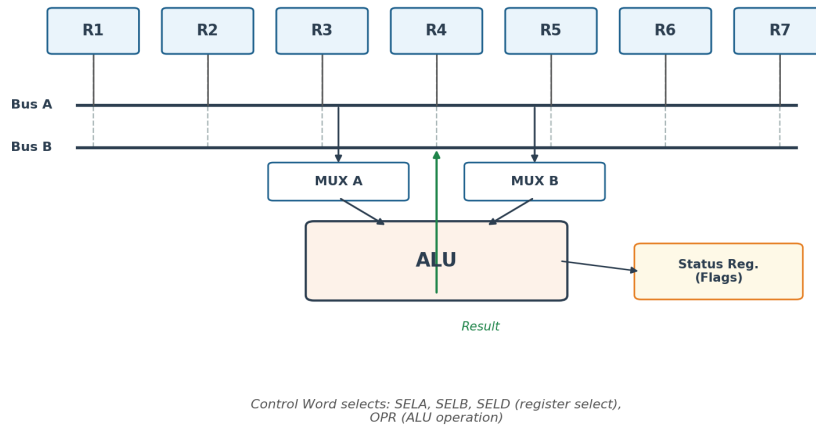


Fig. 5.1 — Registers R1–R7 connected to the ALU through Bus A and Bus B; the ALU result is written back through Bus B into the selected destination register.

The Control Word

Every micro-operation performed by this bus-organized CPU is specified by a binary control word. The control word has several fields, each controlling one part of the data path:

Field	Purpose	Typical Width
SELA	Selects the register that goes on Bus A (input to ALU)	3 bits for 7 registers
SELB	Selects the register that goes on Bus B (input to ALU)	3 bits for 7 registers
SELD	Selects the destination register that receives the ALU result	3 bits for 7 registers
OPR	Selects the micro-operation the ALU performs (ADD, SUB, AND, OR, ...)	5 bits typical

For example, to perform $R3 \leftarrow R1 + R2$, the control word sets SELA to R1, SELB to R2, OPR to the ADD operation, and SELD to R3. This single control word fully specifies one micro-operation cycle of the CPU.

Example 5.1 — Decoding a Control Word

A control word for a 7-register bus-organized CPU has the fields SELA(3) | SELB(3) | SELD(3) | OPR(5). If SELA = 001 (R1), SELB = 010 (R2), OPR = ADD, and SELD = 011 (R3), what micro-operation is performed?

Solution

SELA = R1 places R1 on Bus A; SELB = R2 places R2 on Bus B; the ALU is configured by OPR to add its two inputs; SELD = R3 routes the sum back into R3. The resulting micro-operation is $R3 \leftarrow R1 + R2$.

Note: If SELA or SELB selects a code with no corresponding register (e.g. all-zero code), the bus is often defined to carry the value 0, which allows the same hardware to also perform a direct transfer such as $R2 \leftarrow R1$ by adding R1 to 0.

Stack Organization

A stack is a storage structure that allows data to be inserted and removed only from one end, called the top of stack (TOS). It follows the Last-In-First-Out (LIFO) principle: the last item pushed onto the stack is the first one popped off. A stack is essential for evaluating arithmetic expressions, for handling subroutine calls and returns, and for saving the CPU state during an interrupt.

- **PUSH:** a new item is placed on top of the stack; the stack pointer (SP) is incremented (or decremented, depending on convention) first, and then the item is stored at the new SP location.
- **POP:** the item at the top of the stack is read out, and the stack pointer is then decremented (or incremented) to point to the new top of stack.
- **Overflow:** occurs when a PUSH is attempted but the stack is completely full.
- **Underflow:** occurs when a POP is attempted but the stack is already empty.

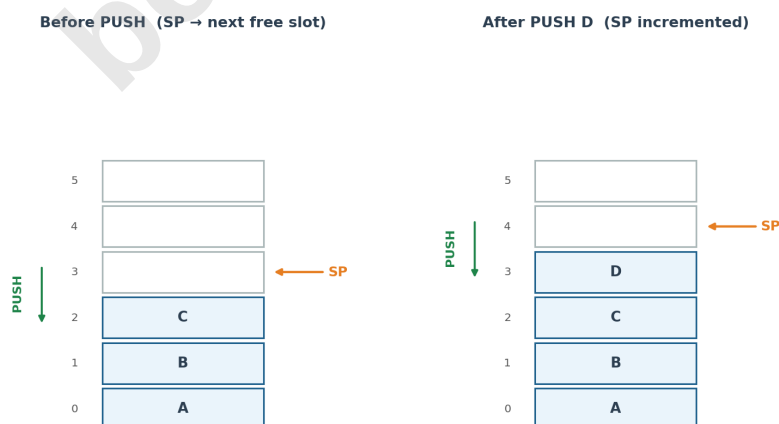


Fig. 5.2 — A register stack before and after a PUSH operation. SP always points to the next free slot; PUSH stores the new value and then advances SP.

Register Stack vs. Memory Stack

Aspect	Register Stack	Memory Stack
Storage	A small set of dedicated registers	A reserved block of main memory
Speed	Very fast (no memory access)	Slower (needs a memory cycle)
Size	Fixed, limited by hardware	Limited only by available memory
SP implementation	A counter register within the stack unit	A CPU register holding a memory address
Typical use	Small, fast scratch storage (e.g. expression evaluation)	Subroutine return addresses, saved registers, large data

In a memory stack, two special locations often bound the stack: the SP register itself (pointing to the current top) and a fixed memory address that marks the bottom of the stack. Overflow and underflow are detected by comparing SP against these limit values.

Example 5.2 — Reverse Polish (Postfix) Evaluation Using a Stack

Evaluate the postfix expression $6\ 3\ 2\ *\ +\ 5\ -$ using a stack, one symbol at a time.

Solution

Scan left to right, pushing operands and popping two operands for every operator:

PUSH 6 → stack: 6

PUSH 3 → stack: 6, 3

PUSH 2 → stack: 6, 3, 2

* → POP 2, POP 3, push $3 \times 2 = 6$ → stack: 6, 6

+ → POP 6, POP 6, push $6 + 6 = 12$ → stack: 12

PUSH 5 → stack: 12, 5

- → POP 5, POP 12, push $12 - 5 = 7$ → stack: 7

Final result on top of stack = 7.

Note: Postfix (Reverse Polish) notation removes the need for parentheses or operator precedence rules, which is exactly why a stack-based (zero-address) CPU can evaluate arithmetic expressions so efficiently.

Instruction Formats

An instruction format specifies how many bits are given to the opcode and how many operand (address) fields follow it. The number of address fields a CPU uses reflects its internal organization — stack, accumulator, or general register.

Instruction Formats by Number of Addresses

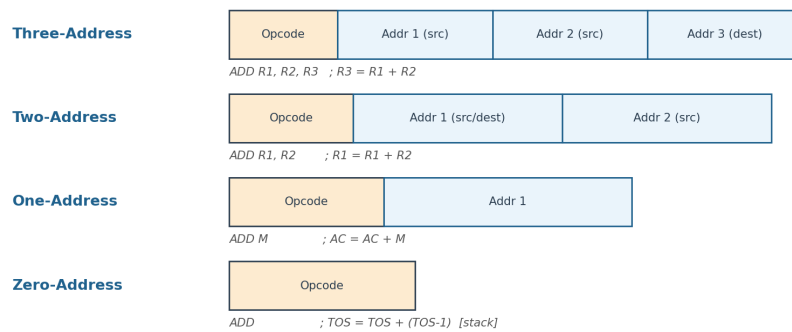


Fig. 5.3 — Three-address, two-address, one-address, and zero-address instruction formats, each shown with an equivalent ADD instruction.

Format	Organization	Description
Three-address	General register	Opcode + 2 source addresses + 1 destination address; flexible but needs longer instructions
Two-address	General register	Opcode + 2 addresses; one address doubles as both a source and the destination
One-address	Accumulator-based	Opcode + 1 address; the accumulator (AC) is always the implied second operand and destination
Zero-address	Stack-based	Opcode only; operands are implicitly the top two items of the stack

Example 5.3 — Evaluating $X = (A + B) \times (C - D)$ in Each Format

Show how the expression $X = (A + B) \times (C - D)$ is evaluated using (a) a three-address format and (b) a zero-address (stack) format.

Solution

(a) Three-address:

ADD R1, A, B ; $R1 = A + B$

SUB R2, C, D ; $R2 = C - D$

MUL X, R1, R2 ; $X = R1 \times R2$ (3 instructions)

(b) Zero-address (stack), using postfix $A B + C D - \times$:

PUSH A ; PUSH B ; ADD ; PUSH C ; PUSH D ; SUB ; MUL ; POP X (8 instructions, but each instruction is very short since it has no address field).

Addressing Modes

An addressing mode is the way an instruction specifies where to find an operand. Addressing modes give programs flexibility without needing a new opcode for every possible way of locating data.

Mode	Effective Address	Example / Remarks
Immediate	Operand is given directly in the instruction (no memory reference)	ADD R1, #5 → adds the constant 5
Register	Operand is inside a CPU register	ADD R1, R2 → operand is in R2
Register Indirect	Register holds the address of the operand	R2 holds the address; operand is in memory[R2]
Direct (Absolute)	Address field holds the operand's memory address directly	ADD R1, [1000] → operand at address 1000
Indirect	Address field points to a memory word that holds the operand's address	One extra memory access to resolve the address
Displacement	Effective address = base register + address field (offset)	Used for arrays: EA = R_base + offset
Relative	Effective address = Program Counter (PC) + address field	Used for short, position-independent branches
Autoincrement / Autodecrement	Register is used as a pointer and is automatically incremented or decremented after (or before) access	Common in loops that step through an array

Example 5.4 — Computing the Effective Address

A CPU has PC = 800, and a base register R1 = 400. An instruction at address 800 uses displacement addressing with address field = 50 and is 2 words long. Find the effective address (a) using displacement addressing and (b) using relative addressing.

Solution

(a) Displacement addressing: $EA = R1 + \text{address field} = 400 + 50 = 450$.

(b) Relative addressing: $EA = PC + \text{address field}$. After fetching this 2-word instruction, PC has already advanced to 802, so $EA = 802 + 50 = 852$.

Key Takeaways — Section 5.1

- ✓ A bus-organized CPU connects several general registers to the ALU using shared buses; a control word (SELA, SELB, SELD, OPR) specifies each micro-operation.
- ✓ A stack is a LIFO structure accessed only at its top; PUSH inserts and advances SP, POP removes and retreats SP.
- ✓ Register stacks are fast and small; memory stacks are large but slower, and need overflow/underflow checks against limit addresses.
- ✓ Instruction formats (three-, two-, one-, and zero-address) trade off instruction length against the number of instructions needed.
- ✓ Addressing modes (immediate, register, indirect, displacement, relative, autoincrement/decrement, etc.) determine how the effective address of an operand is computed.

bcalink.com

5.2 Data Transfer and Manipulation

Every computer needs an instruction set rich enough to carry out any computational task. Most instruction sets fall into three functional groups: data transfer instructions that move information without changing it, data manipulation instructions that perform arithmetic, logic, and shift operations, and program control instructions that alter the normal sequence of execution (covered in Section 5.3).

Data Transfer Instructions

Data transfer instructions move data from one location to another — between registers, between memory and a register, or between the CPU and an I/O device — without changing the value being moved.

Mnemonic	Name	Description
MOV	Move	Transfers data from a source to a destination
LD / LOAD	Load	Loads a value from memory into a register
ST / STORE	Store	Stores a register's value into memory
MOVI	Move Immediate	Loads a constant, given in the instruction itself, into a register
PUSH	Push	Copies a value onto the top of the stack
POP	Pop	Removes the top value of the stack into a register
XCH	Exchange	Swaps the contents of two registers, or a register and a memory word
IN	Input	Transfers data from an input device into a register
OUT	Output	Transfers data from a register to an output device

Note: Data transfer instructions never set the ALU's status (condition-code) flags, because they do not perform any arithmetic or logic computation — they simply copy bits from one place to another.

Data Manipulation Instructions

Data manipulation instructions actually process data. They are usually grouped into three categories: arithmetic, logical and bit manipulation, and shift instructions.

(a) Arithmetic Instructions

Mnemonic	Operation	Effect
ADD	Addition	$R \leftarrow R1 + R2$

Mnemonic	Operation	Effect
SUB	Subtraction	$R \leftarrow R1 - R2$
MUL	Multiplication	$R \leftarrow R1 \times R2$
DIV	Division	$R \leftarrow R1 \div R2$
INC	Increment	$R \leftarrow R + 1$
DEC	Decrement	$R \leftarrow R - 1$
NEG	Negate (2's complement)	$R \leftarrow 0 - R$
ADC / SBB	Add / Subtract with carry	Includes the previous carry/borrow — used for multi-word arithmetic

Example 5.5 — Multi-Precision Addition Using ADD-WITH-CARRY

Add the two 8-bit numbers 11110101 and 00110110 stored across the AC and a carry flag, and show the result and final carry.

Solution

11110101 (245) + 00110110 (54):

```

1 1 1 1 0 1 0 1
+ 0 0 1 1 0 1 1 0
-----

```

1 0 0 1 0 1 0 1 1 → result byte = 00101011, Carry-out = 1

So AC = 00101011 (43) with Carry = 1, consistent with $245 + 54 = 299 = 256 + 43$.

(b) Logical and Bit Manipulation Instructions

Logical instructions operate on data bit-by-bit and are commonly used to test, set, clear, or complement individual bits or bit fields — something arithmetic instructions cannot do directly.

Mnemonic	Operation	Typical Use
AND	Bitwise AND	Clears selected bits (masking)
OR	Bitwise OR	Sets selected bits
XOR	Bitwise Exclusive-OR	Complements selected bits; XOR of a value with itself clears it to 0
NOT	Bitwise complement	Inverts every bit of the operand (1's complement)
CLR	Clear	Sets a register or bit to all 0s
COM	Complement	Same effect as NOT — 1's complement of the operand

Example 5.6 — Masking Bits with AND

Register R1 = 10110110. Use a logical instruction to clear (zero out) the lower 4 bits of R1 while keeping the upper 4 bits unchanged.

Solution

Use a mask with 1s in the bits to keep and 0s in the bits to clear: mask = 11110000.

R1 AND mask: 1 0 1 1 0 1 1 0

AND 1 1 1 1 0 0 0 0

= 1 0 1 1 0 0 0 0

Result R1 = 10110000 — the lower 4 bits are cleared, the upper 4 bits are preserved.

Note: AND is used to clear bits (mask with 0), OR is used to set bits (mask with 1), and XOR is used to selectively complement bits (mask with 1) or to test for equality — XOR of two equal values always produces 0.

(c) Shift Instructions

Shift instructions move the bits of a register left or right. They are used to align data, to implement fast multiplication/division by powers of two, and to serially transfer bits in or out of a register.

Mnemonic	Name	Description
SHL	Logical Shift Left	Shifts all bits left by one position; a 0 enters at the LSB; the MSB shifted out often goes into the Carry flag
SHR	Logical Shift Right	Shifts all bits right by one; a 0 enters at the MSB
SAL	Arithmetic Shift Left	Same bit pattern as SHL, but signals overflow if the sign bit changes
SAR	Arithmetic Shift Right	Shifts right while duplicating the sign bit at the MSB, preserving the operand's sign
ROL	Rotate Left	Bits shifted out of the MSB re-enter at the LSB (circular shift)
ROR	Rotate Right	Bits shifted out of the LSB re-enter at the MSB (circular shift)

Example 5.7 — Multiplying by 4 Using Shifts

Register R1 holds the unsigned value 00000101 (decimal 5). Show how two logical left shifts (SHL) compute 5×4 .

Solution

Start: 00000101 (5)

After 1st SHL: 00001010 (10) — equivalent to $\times 2$

After 2nd SHL: 00010100 (20) — equivalent to $\times 2$ again, so overall $\times 4$

Two logical left shifts multiply the unsigned value by $2^2 = 4$, giving 20.

Note: Arithmetic shifts (SAL/SAR) are used with signed numbers because SAR preserves the sign bit while dividing by 2; logical shifts (SHL/SHR) are used with unsigned data or for pure bit manipulation.

Key Takeaways — Section 5.2

- ✓ Data transfer instructions (MOV, LOAD, STORE, PUSH, POP, IN, OUT) move data without altering it or affecting status flags.
- ✓ Arithmetic instructions (ADD, SUB, MUL, DIV, INC, DEC, NEG) perform numeric computation and typically update status flags.
- ✓ Logical and bit manipulation instructions (AND, OR, XOR, NOT, CLR) operate bit-by-bit and are used for masking, setting, and testing bits.
- ✓ Shift instructions (SHL, SHR, SAL, SAR, ROL, ROR) move bits within a register and provide a fast way to multiply or divide by powers of two.

5.3 Program Control

Normally, the CPU executes instructions in strict sequential order, incrementing the Program Counter (PC) after every fetch. Program control instructions are the instructions that deliberately break this sequence — branching to a different part of the program, calling and returning from subroutines, or responding to an interrupt. Program control depends heavily on the CPU's status register, which records the outcome of the most recent arithmetic or logic operation.

Status Bit Conditions

The status register (also called the flags register or condition-code register) holds a set of individual bits, each set or cleared automatically by the ALU after every arithmetic/logic operation. Conditional branch instructions later test these bits to decide whether to branch.

Flag	Name	Set to 1 When...
C	Carry	An addition produces a carry out of the MSB, or a subtraction requires a borrow
Z	Zero	The result of the operation is exactly zero
S (or N)	Sign	The result's most significant bit is 1 (i.e., the result is negative in signed representation)
V (or O)	Overflow	A signed arithmetic operation produces a result too large to fit in the destination — the sign bit is not a reliable sign
P	Parity	The result contains an even (or odd, by convention) number of 1 bits

Example 5.8 — Detecting Signed Overflow

Add the two 4-bit signed numbers 0111 (+7) and 0001 (+1) and determine the values of the Carry (C) and Overflow (V) flags.

Solution

0111 (+7)

+ 0001 (+1)

1000 → this is -8 in 4-bit two's complement, which is clearly wrong (7+1 should be 8, not representable in 4-bit signed range -8..+7).

Carry out of the MSB position = 0, so C = 0.

But the sign of the result (1000, negative) differs from what two positive operands should produce, so V = 1 (signed overflow) even though C = 0. This shows why overflow must be checked separately from carry for signed arithmetic.

Note: For unsigned arithmetic, the Carry flag alone correctly signals an out-of-range result. For signed arithmetic, only the Overflow flag can be trusted; Carry can be 0 or 1 independently of whether a signed overflow actually happened.

Conditional Branch Instructions

A conditional branch instruction transfers control to a target address only if a specified condition (built from the status flags) is true; otherwise execution simply continues with the next instruction in sequence. An unconditional branch (JMP) always transfers control.

Mnemonic	Condition Tested	Meaning
BZ / JZ	Z = 1	Branch if the previous result was zero
BNZ / JNZ	Z = 0	Branch if the previous result was not zero
BC / JC	C = 1	Branch if carry occurred
BNC / JNC	C = 0	Branch if no carry occurred
BN / JN	S = 1	Branch if the result was negative
BV / JV	V = 1	Branch if signed overflow occurred
JG / BG	Result > 0 (Z=0 and S=0, no overflow)	Branch if greater than
JLE / BLE	Result ≤ 0	Branch if less than or equal

Example 5.9 — Using a Conditional Branch to Find the Maximum of Two Numbers

Write a short instruction sequence that stores the larger of A and B into MAX, using a compare and a conditional branch.

Solution

```
LOAD R1, A ; R1 = A
LOAD R2, B ; R2 = B
SUB R3, R1, R2 ; R3 = A - B (sets flags)
BN B_IS_MAX ; if R3 is negative, A < B, so branch
STORE MAX, R1 ; else A ≥ B, so MAX = A
JMP DONE
B_IS_MAX: STORE MAX, R2 ; MAX = B
DONE: ...
```

Subroutine Call and Return

A subroutine (procedure) is a self-contained block of code that performs a specific task and can be invoked (called) from many places in a program. A **CALL** instruction transfers control to the subroutine while saving the return address (the address of the instruction right after the **CALL**); a **RETURN** instruction at the end of the subroutine restores that address to the PC, resuming the calling program exactly where it left off.

- **CALL** — pushes the current PC (the return address) onto the stack, then loads the PC with the subroutine's starting address.
- **RETURN** — pops the saved return address off the stack back into the PC.
- Because return addresses are pushed and popped in strict LIFO order, a stack naturally supports nested and even recursive subroutine calls — each call pushes a new return address on top of the previous ones.

Example 5.10 — Tracing Nested Subroutine Calls

Main calls subroutine A at address 500 (return address 100). Subroutine A, in turn, calls subroutine B at address 800 (return address 520). Show the stack contents at the deepest point of execution, and the order in which control returns.

Solution

After **CALL A** (from address 99, return addr 100 pushed): stack (top→bottom): 100

After **CALL B** (from within A, return addr 520 pushed): stack (top→bottom): 520, 100

When B executes **RETURN**: pop 520 → PC = 520 (back inside A). Stack: 100

When A executes **RETURN**: pop 100 → PC = 100 (back inside Main). Stack: empty.

Control unwinds in exactly the reverse order of the calls, which is guaranteed by the LIFO nature of the stack.

Note: Because the stack also stores return addresses, some architectures deliberately place a bound on stack depth to catch runaway recursion before it exhausts memory (stack overflow).

Program Interrupt

An interrupt is a mechanism that lets the CPU respond immediately to an event, external or internal, that needs urgent attention, without the programmer having to write explicit code to keep checking for that event. When an interrupt occurs, normal instruction execution is temporarily suspended, the CPU state is saved, and control transfers to a special routine called the Interrupt Service Routine (ISR), also known as an interrupt handler.

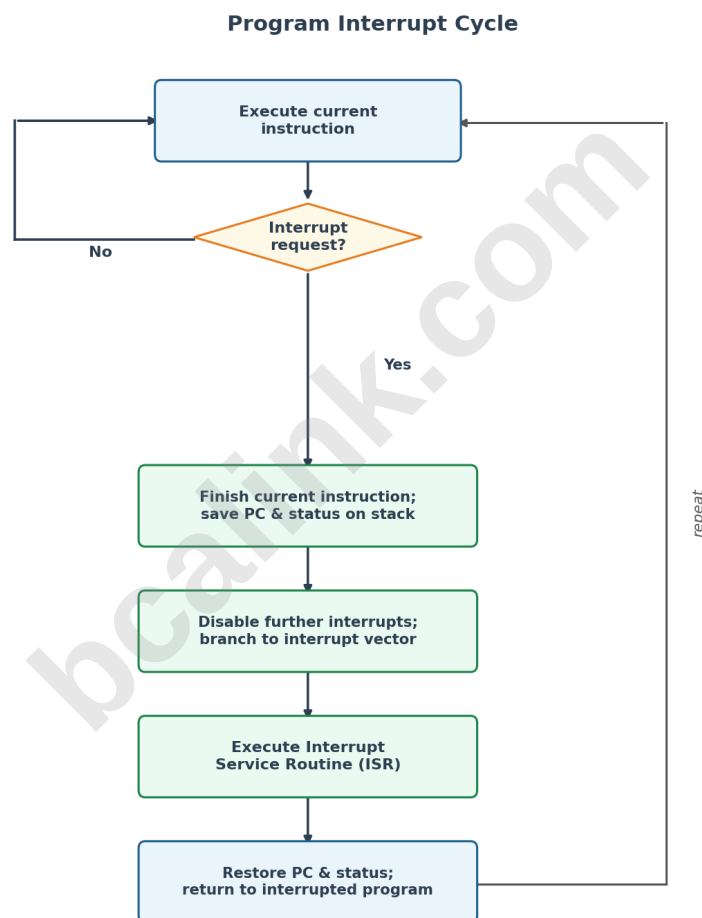


Fig. 5.4 — The program interrupt cycle: after each instruction the CPU checks for a pending interrupt request; if found, it saves state, services the interrupt, and resumes the interrupted program.

The typical sequence of steps when an interrupt is accepted is as follows:

- 1. The CPU finishes the instruction currently in progress (interrupts are usually recognized only between instructions, not in the middle of one).
- 2. The current Program Counter and status register are saved, typically by pushing them onto the stack.

- 3. Further interrupts are disabled (or masked) so the ISR itself is not interrupted unexpectedly, unless the design specifically allows nested interrupts.
- 4. Control branches to the address of the Interrupt Service Routine, usually found through an interrupt vector table.
- 5. The ISR executes, performing whatever task the interrupt requires (e.g., reading a keystroke, handling a timer tick, reporting a fault).
- 6. A special Return from Interrupt instruction restores the saved PC and status, re-enables interrupts, and resumes the interrupted program exactly where it left off.

Types of Interrupts

Interrupts are commonly classified by their source into three broad categories:

Type	Source	Examples
Hardware (External) Interrupt	An external device or signal, independent of the currently running program	Keyboard key press, timer tick, disk I/O completion, power-failure signal
Software Interrupt	An explicit instruction in the program itself	A TRAP or INT instruction used to request an operating-system service (a system call)
Internal Interrupt (Trap / Exception)	An error or unusual condition detected inside the CPU while executing an instruction	Divide-by-zero, illegal opcode, arithmetic overflow, page fault, invalid memory access

Hardware interrupts are asynchronous — they can occur at any time, unrelated to the instruction currently executing. Software interrupts and internal traps/exceptions are synchronous — they are a direct, immediate, and repeatable consequence of executing a particular instruction.

Example 5.11 — Classifying Interrupts

Classify each of the following as a hardware, software, or internal interrupt: (a) a printer signaling that it has run out of paper; (b) a program deliberately executing a TRAP instruction to open a file through the OS; (c) an instruction attempting to divide an integer by zero.

Solution

- (a) Hardware (external) interrupt — the event originates in an I/O device, unrelated to the instruction currently executing.
- (b) Software interrupt — the program itself explicitly requests OS service via TRAP.
- (c) Internal interrupt (exception/trap) — the CPU itself detects an illegal operation (division by zero) while executing the instruction.

Note: When several interrupts occur at once, the CPU resolves the conflict using a priority scheme: each interrupt source is assigned a priority level, and if multiple requests are pending, the highest-priority request is serviced first.

Key Takeaways — Section 5.3

- ✓ Status flags (Carry, Zero, Sign, Overflow, Parity) record the outcome of the last ALU operation and are tested by conditional branch instructions.
- ✓ CALL saves the return address on the stack and jumps to a subroutine; RETURN pops that address back into the PC, which naturally supports nested and recursive calls.
- ✓ A program interrupt suspends the current program, saves the CPU state, runs an Interrupt Service Routine, and then resumes execution exactly where it left off.
- ✓ Interrupts are classified as hardware (external, asynchronous), software (explicit TRAP/INT instruction), or internal (exceptions/traps detected by the CPU itself).

— *End of Unit 5* —