Section 8.4 – Instruction Formats

- It is the function of the control unit within the CPU to interpret each instruction code.
- The bits of the instruction are divided into groups called fields.
- The most common fields are:
  - Operation code
  - Address field – memory address or a processor register
  - Mode field – specifies the way the operand or effective address is determined
- A register address is a binary number of \( k \) bits that defines one of \( 2^k \) registers in the CPU.
- The instructions may have several different lengths containing varying number of addresses.
- The number of address fields in the instruction format of a computer depends on the internal organization of its registers.
- Most computers fall into one of the three following organizations:
  - Single accumulator organization
  - General register organization
  - Stack organization
- Single accumulator org. uses one address field
  \[
  \text{ADD X: AC} \leftarrow \text{AC + M[X]}
  \]
- The general register org. uses three address fields
  \[
  \text{ADD R1, R2, R3: R1} \leftarrow \text{R2 + R3}
  \]
- Can use two rather than three fields if the destination is assumed to be one of the source registers.
- Stack org. would require one address field for PUSH/POP operations and none for operation-type instructions
  \[
  \text{PUSH X}\quad \text{ADD}
  \]
- Some computers combine features from more than one organizational structure.

Example: \( X = (A+B) \times (C + D) \)

Three-address instructions:

- \[
  \text{ADD R1, A, B R1} \leftarrow \text{M[A] + M[B]}
  \]
- \[
  \text{ADD R2, C, D R2} \leftarrow \text{M[C] + M[D]}
  \]
- \[
  \text{MUL X, R1, R2 M[X]} \leftarrow \text{R1 * R2}
  \]

Two-address instructions:

- \[
  \text{MOV R1, A R1} \leftarrow \text{M[A]}
  \]
- \[
  \text{ADD R1, B R1} \leftarrow \text{R1 + M[B]}
  \]
- \[
  \text{MOV R2, C R2} \leftarrow \text{M[C]}
  \]
ADD R2, D  R2 ← R2 + D  
MUL R1, R2  R1 ← R1 * R2  
MOV X, R1  M[X] ← R1

One-address instructions:

LOAD A  AC ← M[A]  
ADD B  AC ← AC + M[B]  
STORE T  M[T] ← AC  
LOAD C  AC ← M[C]  
ADD D  AC ← AC + M[D]  
MUL T  AC ← AC * M[T]  
STORE X  M[X] ← AC

Zero-address instructions:

PUSH A  TOS ← A  
PUSH B  TOS ← B  
ADD  TOS ← (A +B)  
PUSH C  TOS ← C  
PUSH D  TOS ← D  
ADD  TOS ← (C + D)  
MUL  TOS ← (C + D) * (A + B)  
POP X  M[X] ← TOS

RISC instructions:

LOAD R1, A  R1 ← M[A]  
LOAD R2, B  R2 ← M[B]  
LOAD R3, C  R3 ← M[C]  
LOAD R4, D  R4 ← M[D]  
ADD R1, R1, R2  R1 ← R1 + R2  
ADD R3, R3, R4  R3 ← R3 + R4  
MUL R1, R1, R3  R1 ← R1 * R3  
STORE X, R1  M[X] ← R1

Section 8.5 – Addressing Modes

• The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced
• The decoding step in the instruction cycle determines the operation to be performed, the addressing mode of the instruction, and the location of the operands
• Two addressing modes require no address fields – the implied mode and immediate mode
• Implied mode: the operands are specified implicitly in the definition of the instruction – complement accumulator or zero-address instructions
- Immediate mode: the operand is specified in the instruction
- Register mode: the operands are in registers
- Register indirect mode: the instruction specifies a register that contains the address of the operand
- Autoincrement or autodecrement mode: similar to the register indirect mode
- Direct address mode: the operand is located at the specified address given
- Indirect address mode: the address specifies the effective address of the operand
- Relative address mode: the effective address is the summation of the address field and the content of the PC
- Indexed addressing mode: the effective address is the summation of an index register and the address field
- Base register address mode: the effective address is the summation of a base register and the address field

![Diagram](image)

**Figure 8-7** Numerical example for addressing modes.
Section 8.6 – Data Transfer and Manipulation

- There is a basic set of operations that most computers include in their instruction set.
- The opcode and/or symbolic code may differ for the same instruction among different computers.

- There are three main categories of computer instructions:
  - Data transfer
  - Data manipulation
  - Program control

- Data transfer instructions: transfer data from one location to another without changing the binary information content.

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Effective Address</th>
<th>Content of AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct address</td>
<td>500</td>
<td>800</td>
</tr>
<tr>
<td>Immediate operand</td>
<td>201</td>
<td>500</td>
</tr>
<tr>
<td>Indirect address</td>
<td>800</td>
<td>300</td>
</tr>
<tr>
<td>Relative address</td>
<td>702</td>
<td>325</td>
</tr>
<tr>
<td>Indexed address</td>
<td>600</td>
<td>900</td>
</tr>
<tr>
<td>Register</td>
<td>—</td>
<td>400</td>
</tr>
<tr>
<td>Register indirect</td>
<td>400</td>
<td>700</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>400</td>
<td>700</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>399</td>
<td>450</td>
</tr>
</tbody>
</table>

- Some assembly language conventions modify the mnemonic symbol to differentiate between addressing modes:
  - LDI – load immediate
- Some use a special character to designate the mode.
Data manipulation instructions: perform arithmetic, logic, and/or shift operation

Arithmetic instructions:

- Increment \( INC \)
- Decrement \( DEC \)
- Add \( ADD \)
- Subtract \( SUB \)
- Multiply \( MUL \)
- Divide \( DIV \)
- Add w/carry \( ADDC \)
- Sub. w/borrow \( SUBB \)
- Negate (2's comp) \( NEG \)

Some computers have different instructions depending upon the data type:

- ADDI Add two binary integer numbers
- ADDF Add two floating point numbers
- ADDD Add two decimal numbers in BCD

Logical and bit manipulation instructions:

- Clear \( CLR \)
- Complement \( COM \)
- AND \( AND \)
- OR \( OR \)
- Exclusive-OR \( XOR \)
- Clear carry \( CLRC \)
- Set carry \( SETC \)
- Comp. carry \( COMC \)
- Enable inter. \( EI \)
- Disable inter. \( DI \)

Clear selected bits – AND instruction
Set selected bits – OR instruction
Complement selected bits – XOR instruction

Shift instructions:

- Logical shift right \( SHR \)
- Logical shift left \( SHL \)
- Rotate right \( ROR \)
- Rotate left \( ROL \)
Section 8.7 – Program Control

- Program control instructions: provide decision-making capabilities and change the program path
- Typically, the program counter is incremented during the fetch phase to the location of the next instruction
- A program control type of instruction may change the address value in the program counter and cause the flow of control to be altered
- This provides control over the flow of program execution and a capability for branching to different program segments

| Branch   | Return   |  |  |
|----------|----------|  |  |
| Jump     | Compare  |  |  |
| Skip     | Test     |  |  |
| Call     |          |  |  |

- TST and CMP cause branches based upon four status bits: C, S, Z, and V

![Figure 8-8 Status register bits.](image-url)
A call subroutine instruction consists of an operation code together with an address that specifies the beginning of the subroutine

- **Execution of CALL:**
  - Temporarily store return address
  - Transfer control to the beginning of the subroutine – update PC

\[
\begin{align*}
\text{SP} & \leftarrow \text{SP} - 1 \\
\text{M}[\text{SP}] & \leftarrow \text{PC} \\
\text{PC} & \leftarrow \text{effective address}
\end{align*}
\]

- **Execution of RET:**
  - Transfer return address from the temporary location to the PC
  - Control returns to the original program after the service program is executed

**Program interrupt** refers to the transfer of program control to a service routine as a result of interrupt request

### Table 8-11 Conditional Branch Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Branch condition</th>
<th>Tested condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BZ</td>
<td>Branch if zero</td>
<td>( Z = 1 )</td>
</tr>
<tr>
<td>BNZ</td>
<td>Branch if not zero</td>
<td>( Z = 0 )</td>
</tr>
<tr>
<td>BC</td>
<td>Branch if carry</td>
<td>( C = 1 )</td>
</tr>
<tr>
<td>BNC</td>
<td>Branch if no carry</td>
<td>( C = 0 )</td>
</tr>
<tr>
<td>BP</td>
<td>Branch if plus</td>
<td>( S = 0 )</td>
</tr>
<tr>
<td>BM</td>
<td>Branch if minus</td>
<td>( S = 1 )</td>
</tr>
<tr>
<td>BV</td>
<td>Branch if overflow</td>
<td>( V = 1 )</td>
</tr>
<tr>
<td>BNV</td>
<td>Branch if no overflow</td>
<td>( V = 0 )</td>
</tr>
</tbody>
</table>

**Unsigned compare conditions** \((A - B)\)

- BHI: Branch if higher \( A > B \)
- BHE: Branch if higher or equal \( A \geq B \)
- BLO: Branch if lower \( A < B \)
- BLOE: Branch if lower or equal \( A \leq B \)
- BE: Branch if equal \( A = B \)
- BNE: Branch if not equal \( A \neq B \)

**Signed compare conditions** \((A - B)\)

- BGT: Branch if greater than \( A > B \)
- BGE: Branch if greater or equal \( A \geq B \)
- BLT: Branch if less than \( A < B \)
- BLE: Branch if less or equal \( A \leq B \)
- BE: Branch if equal \( A = B \)
- BNE: Branch if not equal \( A \neq B \)
• An interrupt procedure is similar to a subroutine call except:
  o The interrupt is usually initiated by an internal or external signal rather than an instruction
  o The address of the interrupt service routine is determined by the hardware rather than the address field of an instruction
  o All information necessary to define the state of the CPU is stored rather than just the return address

• The interrupted program should resume exactly as if nothing had happened
• The state of the CPU at the end of the execute cycle is determined from:
  o The content of the PC
  o The content of all processor registers
  o The content of certain status conditions

• The *program status word* (PSW) is a register that holds the status and control flag conditions
• Not all computers store the register contents when responding to an interrupt
• The CPU does not respond to an interrupt until the end of an instruction execution
• The control checks for any interrupt signals before entering the next fetch phase
• Three types of interrupts:
  o External interrupts
  o Internal interrupts
  o Software interrupts

• *External interrupts* come from I/O devices, timing devices, or any other external source
• *Internal interrupts* arise from illegal or erroneous use of an instruction or data, also called traps
• Internal interrupts are synchronous while external ones are asynchronous
• Both are initiated from signals that occur in the hardware of the CPU
• A *software interrupt* is initiated by executing an instruction

**Section 8.8 – Reduced Instruction Set Computer (RISC)**

• An important aspect of computer architecture is the design of the instruction set for the processor
• The instruction set determines the way that machine language programs are constructed
• Many computers have instructions sets of about 100 - 250 instructions
• These computers employ a variety of data types and a large number of addressing modes – complex instruction set computer (CISC)
• A RISC uses fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often
• The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement that is written in a high-level language
• The major characteristics of CISC architecture are:
  o Large number of instructions
  o Some instructions that perform specialized tasks and are used infrequently
  o Large variety of addressing modes
  o Variable length instruction formats
  o Instructions that manipulate operands in memory
• The goal of RISC architecture is to reduce execution time by simplifying the instructions set
• The major characteristics of RISC architecture are:
  o Relatively few instructions
  o Relatively few addressing modes
  o Memory access limited to load and store instructions
  o All operations done within the registers of the CPU
  o Fixed-length, easily decoded instruction format
  o Single-cycle instruction execution
  o Hardwired rather than microprogrammed control
  o Relatively large number of registers in the processor unit
  o Use of overlapped register windows to speed-up procedure call and return
  o Efficient instruction pipeline
  o Ability to execute one instruction per clock cycle
  o Compiler support for efficient translation of high-level language programs into machine language programs

• Overlapped register windows are used to pass parameters and avoids the need for saving and restoring register values during procedure calls
• Each procedure call activates a new register window by incrementing a pointer, while the return statement decrements the pointer and causes the activation of the previous window
• Windows for adjacent procedures have overlapping registers that are shared to provide the passing of parameters and results

• Example: system with 74 registers and four procedures
  o Each procedure has a total of 32 registers while active
  o 10 global registers
  o 10 local registers
  o 6 low overlapping registers
  o 6 high overlapping registers

• Relationships of register windows
  o # of global registers = G
  o # of local registers in each window = L
  o # of common registers to two windows = C
  o # of windows = W
  o window size = L + 2C + G
  o total # of registers = (L + C)W + G