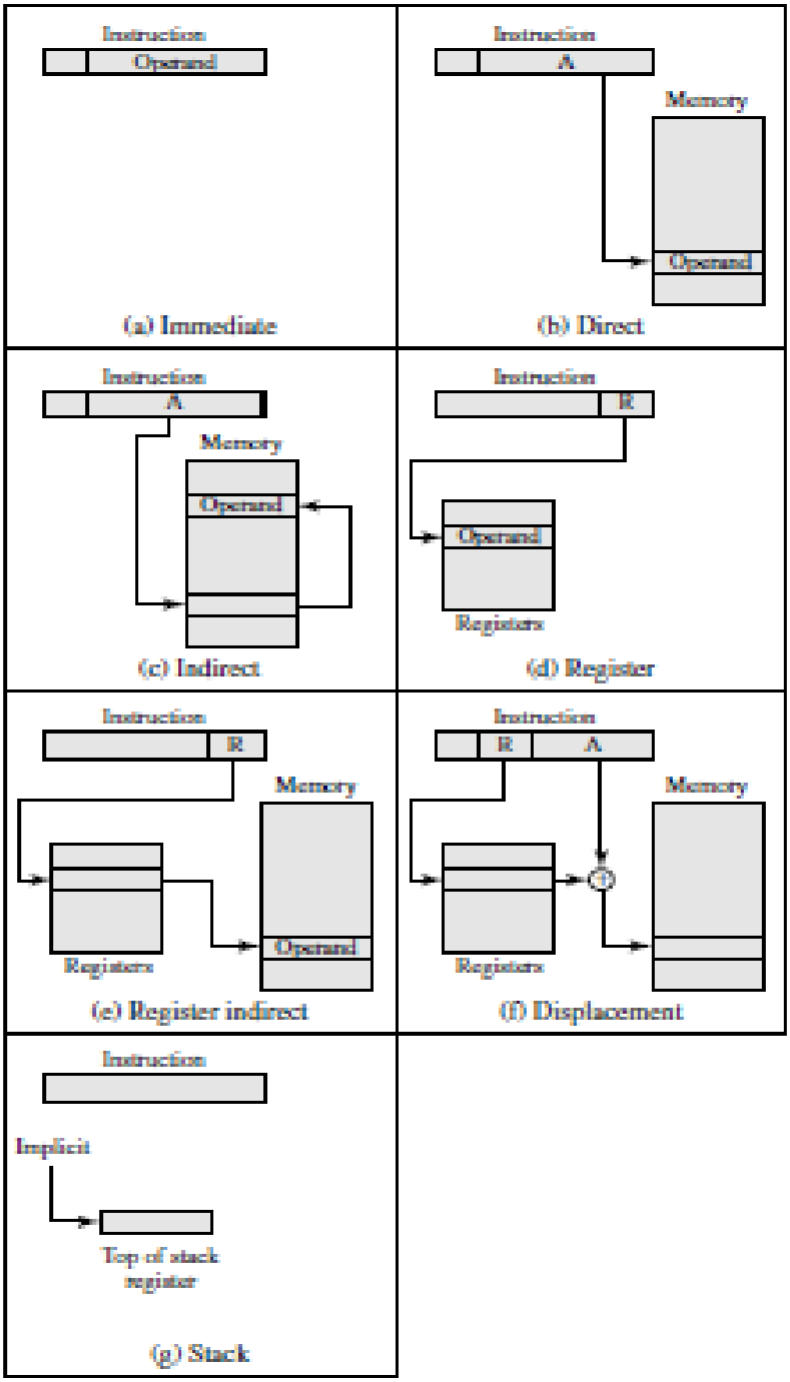
UNIT-2

MACHINE INSTRUCTION SET

❖ ADDRESSING MODES



These modes are illustrated in Figure 11.1. In this section, we use the following notation:

A = contents of an address field in the instruction
R = contents of an address field in the instruction that refers to a register
EA = actual (effective) address of the location containing the referenced operand
(X) = contents of memory location X or register X

Figure 2.1 Addressing Modes

The address field or fields in a typical instruction format are relatively small. We should be able to reference a large range of locations in main memory. For this, a variety of addressing techniques has been employed. The most common addressing techniques are:

- Immediate
- Direct
- Indirect
- Register
- Register indirect
- Displacement
- Stack

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

Table 2.1Basic Addressing Modes

Table 2.1 indicates the address calculation performed for each addressing mode. Different opcodes will use different addressing modes. Also, one or more bits in the instruction format can be used as a *mode field.* The value of the mode field determines which addressing mode is to be used.

Immediate Addressing

The simplest form of addressing is immediate addressing, in which the operand value is present in the instruction

$$\text{Operand} = A$$

This mode can be used to define and use constants or set initial values of variables

The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand, thus saving one memory or cache cycle in the instruction cycle. The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the word length.

Direct Addressing

A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

$$EA = A$$

The advantage is it requires only one memory reference and no special calculation. The disadvantage is that it provides only a limited address space.

Indirect Addressing

With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is known as *indirect addressing:*

$$EA = (A)$$

As defined earlier, the parentheses are to be interpreted as meaning *contents of*.

The obvious advantage of this approach is that for a word length of N, an address space of 2^N is now available. The disadvantage is that instruction execution requires two memory references to fetch the operand: one to get its address and a second to get its value.

A rarely used variant of indirect addressing is multilevel or cascaded indirect addressing:

$$EA = (Á (A) Á)$$

Register Addressing

Register addressing is similar to direct addressing. The only difference is that the address field refers to a register rather than a main memory address:

$$EA = R$$

To clarify, if the contents of a register address field in an instruction is 5, then register R5 is the intended address, and the operand value is contained in R5.

The advantages of register addressing are that (1) only a small address field is needed in the instruction, and (2) no time-consuming memory references are required because the memory access time for a register internal to the processor is much less than that for a main memory address. The disadvantage of register addressing is that the address space is very limited.

Register Indirect Addressing

Just as register addressing is analogous to direct addressing, register indirect addressing is analogous to indirect addressing. In both cases, the only difference is whether the address field refers to a memory location or a register.Thus,for register indirect address,

$$EA = (R)$$

The advantages and limitations of register indirect addressing are basically the same as for indirect addressing. In both cases, the address space limitation (limited range of addresses) of the address field is overcome by having that field refer to a word-length location containing an address. In addition, register indirect addressing uses one less memory reference than indirect addressing.

Displacement Addressing

A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing. We will refer to this as *displacement addressing:*

$$EA = A + (R)$$

Displacement addressing requires that the instruction have two address fields, at least one of which is explicit. The value contained in one address field (value = A) is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address.

We will describe three of the most common uses of displacement addressing:

- Relative addressing

- Base-register addressing
- Indexing

RELATIVE ADDRESSING For relative addressing, also called PC-relative addressing, the implicitly referenced register is the program counter (PC). That is, the next instruction address is added to the address field to produce the EA. Thus, the effective address is a displacement relative to the address of the instruction.

BASE-REGISTER ADDRESSING For base-register addressing, the interpretation is the following: The referenced register contains a main memory address, and the address field contains a displacement (usually an unsigned integer representation) from that address. The register reference may be explicit or implicit.

INDEXING For indexing, the interpretation is typically the following: The address field references a main memory address, and the referenced register contains a positive displacement from that address. This usage is just the opposite of the interpretation for base-register

An important use of indexing is to provide an efficient mechanism for performing iterative operations. Consider, for example, a list of numbers stored starting at location A. Suppose that we would like to add 1 to each element on the list. We need to fetch each value, add 1 to it, and store it back. The sequence of effective addresses that we need is A,A + 1, A + 2, . . ., up to the last location on the list. With indexing, this is easily done. The value A is stored in the instruction's address field, and the chosen register, called an *index register,* is initialized to 0. After each operation, the index register is incremented by 1.

Because index registers are commonly used for such iterative tasks, it is typical that there is a need to increment or decrement the index register after each reference to it. Because this is such a common operation, some systems will automatically do this as part of the same instruction cycle.This is known as *autoindexing*

.. If general-purpose registers are used, the autoindex operation may need to be signaled by a bit in the instruction.Autoindexing using increment can be depicted as follows.

$$EA = A + (R)$$
$$(R) ; (R) + 1$$

In some machines, both indirect addressing and indexing are provided, and it is possible to employ both in the same instruction. There are two possibilities: the indexing is performed either before or after the indirection.

If indexing is performed after the indirection, it is termed *postindexing:*

$$EA = (A) + (R)$$

First, the contents of the address field are used to access a memory location containing a direct address. This address is then indexed by the register value.

With *preindexing,* the indexing is performed before the indirection:

$$EA = (A + (R))$$

An address is calculated as with simple indexing. In this case, however, the calculated address contains not the operand, but the address of the operand.

Stack Addressing

The final addressing mode that we consider is stack addressing. It is sometimes referred to as a *pushdown list* or *last-in-first-out queue.* The stack is a reserved block of locations.

Items are appended to the top of the stack so that, at any given time, the block is partially filled.

Associated with the stack is a pointer whose value is the address of the top of the stack. Alternatively, the top two elements of the stack may be in processor registers, in which case the stack pointer references the third element of the stack.

The stack pointer is maintained in a register. Thus, references to stack locations in memory are in fact register indirect addresses.

The stack mode of addressing is a form of implied addressing. The machine instructions need not include a memory reference but implicitly operate on the top of the stack

❖ X86 ADDRESSING MODES

The x86 address translation mechanism produces an address, called a virtual or effective address, that is an offset into a segment. The sum of the starting address of the segment and the effective address produces a linear address. If paging is being used, this linear address must pass through a page-translation mechanism to produce a physical address.

The x86 is equipped with a variety of addressing modes intended to allow the efficient execution of high-level languages. Figure 2.2 indicates the logic involved. The segment register determines the segment that is the subject of the reference. There are six segment registers. Each segment register holds an index into the segment descriptor table which holds the starting address of the corresponding segments. With each segment register is a segment descriptor register which records the access rights for the segment as well as the starting address and limit (length) of the segment. In addition, there are two registers that may be used in constructing an address: the base register and the index register.

Table 2.2 lists the x86 addressing modes.

- **Immediate mode**, the operand is included in the instruction. The operand can be a byte, word, or doubleword of data.
- **Register operand mode**, the operand is located in a register. For general instructions, such as data transfer, arithmetic, and logical instructions, the operand can be one of the 32-bit general registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP), one of the 16-bit general registers (AX, BX, CX, DX, SI, DI, SP, BP), or one of the 8bit general registers (AH, BH, CH, DH, AL, BL, CL, DL). There are also some instructions that reference the segment selector registers (CS, DS, ES, SS, FS, GS).
- **Displacement mode**, the operand's offset (the effective address of Figure 11.2) is contained as part of the instruction as an 8-, 16-, or 32-bit displacement. The displacement addressing mode is found on few machines because, as mentioned earlier, it leads to long instructions. In the case of the x86, the displacement value can be as long as 32 bits, making for a 6-byte instruction. Displacement addressing can be useful for referencing global variables.

The remaining addressing modes are indirect, in the sense that the address portion of the instruction tells the processor where to look to find the address.

- **Base mode** specifies that one of the 8-, 16-, or 32-bit registers contains the effective address. This is equivalent to what we have referred to as register indirect addressing.
- **Base with displacement mode**, the instruction includes a displacement to be added to a base register, which may be any of the general-purpose registers.

Examples of uses of this mode are as follows:

- Used by a compiler to point to the start of a local variable area
- Used to index into an array when the element size is not 1, 2, 4, or 8 bytes and which therefore cannot be indexed using an index register.
- Used to access a field of a record.

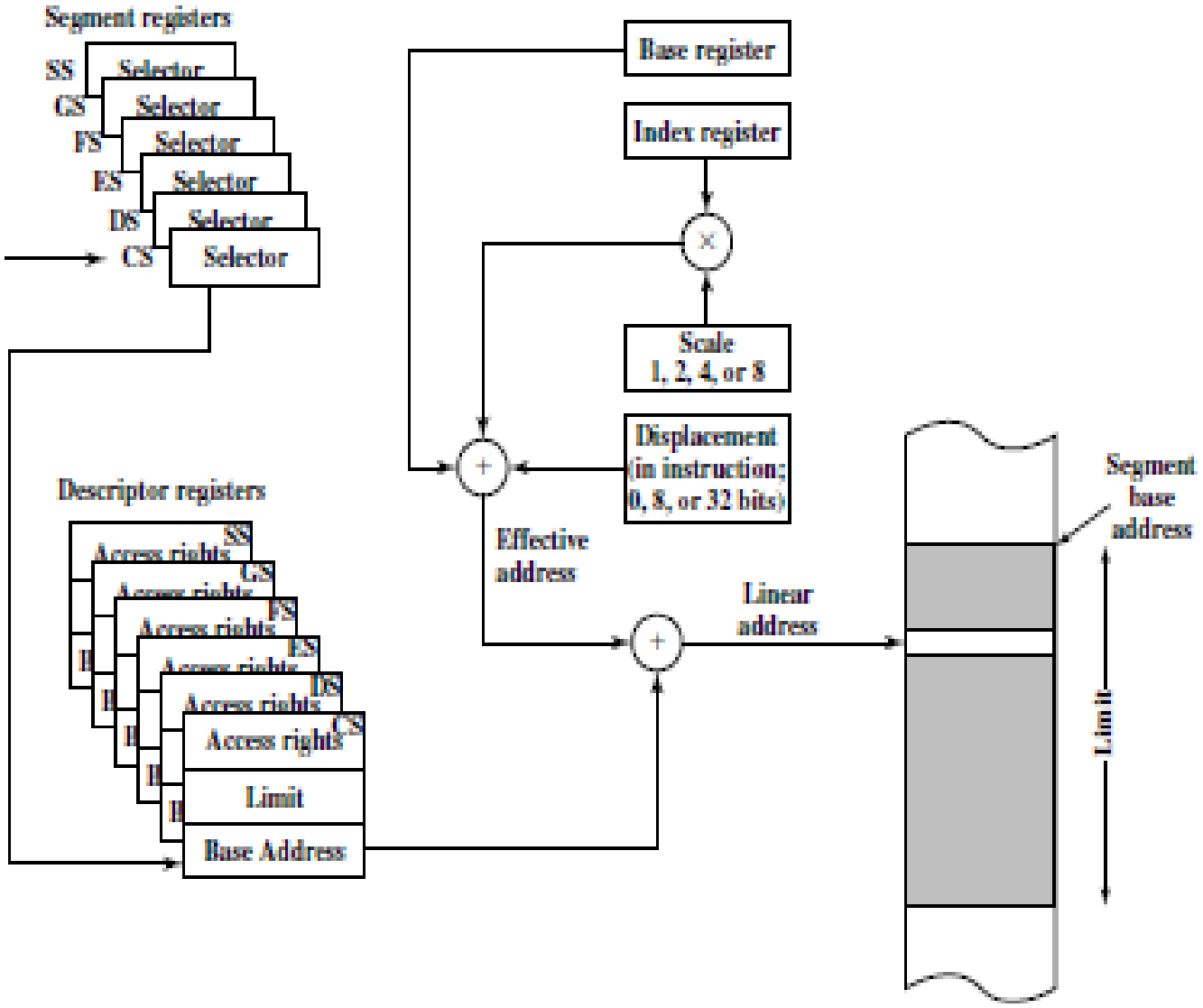


Figure 2.2 x86 Addressing mode caluclation

- **Scaled index with displacement mode**, the instruction includes a displacement to be added to a register, in this case called an index register. The index register may be any of the general-purpose registers except the one called ESP, which is generally used for stack processing. In calculating the effective address, the contents of the index register are multiplied by a scaling factor of 1, 2, 4, or 8, and then added to a displacement.

A scaling factor of 2 can be used for an array of 16-bit integers. A scaling factor of 4 can be used for 32-bit integers or floating-point numbers. Finally, a scaling factor of 8 can be used for an array of double-precision floating-point numbers.

- **Base with index and displacement mode** sums the contents of the base register, the index register, and a displacement to form the effective address. Again, the base register can be any general-purpose register and the index register can be any general-purpose register except ESP.

This mode can also be used to support a two-dimensional array; in this case, the displacement points to the beginning of the array and each register handles one dimension of the array.

- **Based scaled index with displacement mode** sums the contents of the index register multiplied by a scaling factor, the contents of the base register, and the displacement.This is useful if an array is stored in a stack frame. This mode also provides efficient indexing of a two-dimensional array when the array elements are 2, 4, or 8 bytes in length.
- **Relative addressing** can be used in transfer-of-control instructions. A displacement is added to the value of the program counter, which points to the next instruction.

Mode	Algorithm
Immediate	Operand = A
Register Operand	LA = R
Displacement	LA = (SR) + A
Base	LA = (SR) + (B)
Base with Displacement	LA = (SR) + (B) + A
Scaled Index with Displacement	LA = (SR) + (I) × S + A
Base with Index and Displacement	LA = (SR) + (B) + (I) + A
Base with Scaled Index and Displacement	LA = (SR) + (I) × S + (B) + A
Relative	LA = (PC) + A

LA = linear address
(X) = contents of X
SR = segment register
PC = program counter
A = contents of an address field in the instruction
R = register
B = base register
I = index register
S = scaling factor

Table 2.2 x86 Addressing modes

❖ **ARM Addressing Modes**

In the ARM[1] architecture the addressing modes are most conveniently classified with respect to the type of instruction.

1. ***LOAD/STORE ADDRESSING*** Load and store instructions are the only instructions that reference memory. This is always done indirectly through a base register plus offset. There are three alternatives with respect to indexing (Figure 2.3):

• **Offset:** For this addressing method, indexing is not used. An offset value is added to or subtracted from the value in the base register to form the memory address.

As an example Figure 2.3a illustrates this method with the assembly language instruction

STRB r0,[r1,#12].

This is the store byte instruction. In this case the base address is in register r1 and the displacement is an immediate value of decimal 12. The resulting address (base plus offset) is the location where the least significant byte from r0 is to be stored.

• **Preindex:** The memory address is formed in the same way as for offset addressing.The memory address is also written back to the base register.In other words, the base register value is incremented or decremented by the offset value. Figure 2.3b illustrates this method with the assembly language instruction

STRBr0,[r1,#12]!.

The exclamation point signifies preindexing.

• **Postindex:** The memory address is the base register value.An offset is added to or subtracted from the base register value and the result is written back to the base register. Figure 2.3c illustrates this method with the assembly language instruction

STRB r0, [r1], #12.

The value in the offset register is scaled by one of the shift operators: Logical Shift Left, Logical

Shift Right, Arithmetic Shift Right, Rotate Right, or Rotate Right Extended (which includes the carry bit in the rotation). The amount of the shift is specified as an immediate value in the instruction.

1. ***DATA PROCESSING INSTRUCTION ADDRESSING*** Data processing instructions use either register addressing of a mixture of register and immediate addressing. For register addressing, the value in one of the register operands may be scaled using one of the five shift operators defined in the preceding paragraph.

2. ***BRANCH INSTRUCTIONS*** The only form of addressing for branch instructions is immediate addressing. The branch instruction contains a 24-bit value. For address calculation, this value is shifted left 2 bits, so that the address is on a word boundary. Thus the effective address range is ;32 MB from the program counter.

3. ***LOAD/STORE MULTIPLE ADDRESSING*** Load multiple instructions load a subset of the general-purpose registers from memory. Store multiple instructions store a subset of the general-purpose registers to memory.

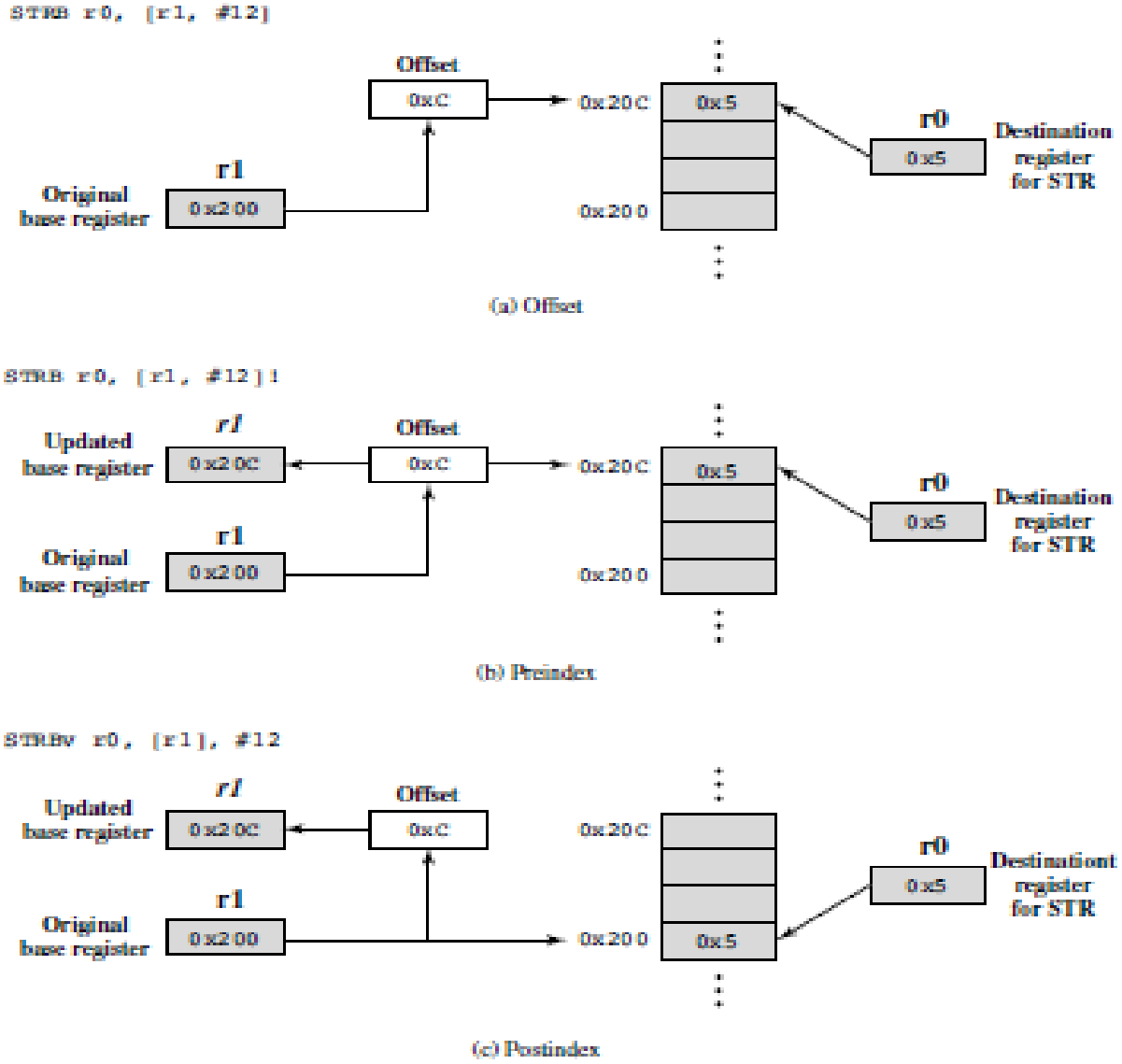


Figure 2.3 ARM Indexing Methods

The list of registers for the load or store is specified in a 16-bit field in the instruction with each bit corresponding to one of the 16 registers. Load and Store Multiple addressing modes produce a sequential range of memory addresses. The lowest-numbered register is stored at the lowest memory address and the highest-numbered register at the highest memory address. Four addressing modes are used

(Figure 2.4): increment after, increment before, decrement after, and decrement before. A base register specifies a main memory address where register values are stored in or loaded from in ascending (increment) or descending (decrement) word locations. Incrementing or decrementing starts either before or after the first memory access.

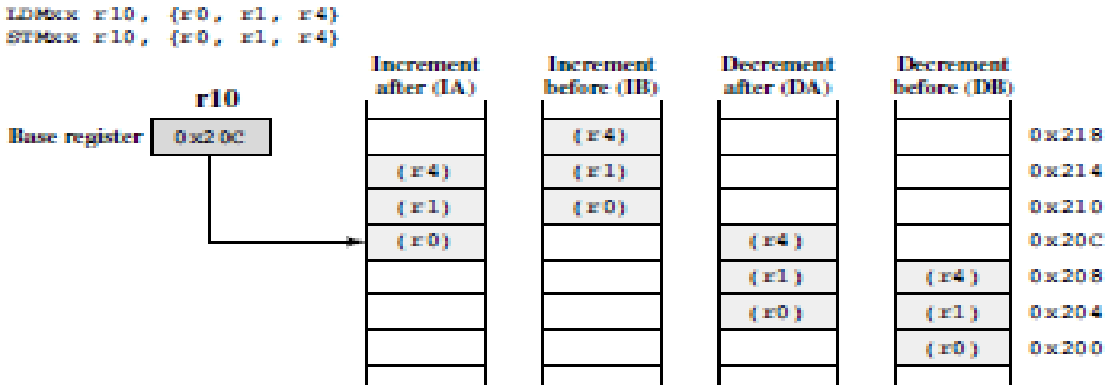


Figure 2.4 ARM Load/Store Multiple addressing

❖ INSTRUCTION FORMATS

An instruction format defines the layout of the bits of an instruction. An instruction format must include an opcode and, implicitly or explicitly, zero or more operands. Each explicit operand is referenced using one of the addressing modes. Key design issues in X86 instruction formats are:

Instruction Length

The most basic design issue to be faced is the instruction format length which is affected by, memory size, memory organization, bus structure, processor complexity, and processor speed.

Beyond this basic trade-off, there are other considerations.

- Either the instruction length should be equal to the memory-transfer length or one should be a multiple of the other.
- Memory transfer rate has not kept up with increases in processor speed.
- Memory can become a bottleneck if the processor can execute instructions faster than it can fetch them. One solution to this problem is to use cache memory and another is to use shorter instructions.
- Instruction length should be a multiple of the character length, which is usually 8 bits, and of the length of fixed-point numbers.

Allocation of Bits

An equally difficult issue is how to allocate the bits in that format. For a given instruction length, more opcodes obviously mean more bits in the opcode field. For an instruction format of a given length, this reduces the number of bits available for addressing. There is one interesting refinement to this trade-off, and that is the use of variable-length opcodes.

In this approach, there is a minimum opcode length but, for some opcodes, additional operations may be specified by using additional bits in the instruction. For a fixed-length instruction, this leaves fewer bits for addressing. Thus, this feature is used for those instructions that require fewer operands and/or less powerful addressing.

The following interrelated factors go into determining the use of the addressing bits.

- **Number of addressing modes:** Sometimes an addressing mode can be indicated implicitly. In other cases, the addressing modes must be explicit, and one or more mode bits will be needed.
- **Number of operands:** Typical instructions on today's machines provide for two operands. Each

operand address in the instruction might require its own mode indicator, or the use of a mode indicator could be limited to just one of the address fields.

- **Register versus memory:** A machine must have registers so that data can be brought into the processor for processing. With a single user-visible register (usually called the accumulator), one operand address is implicit and consumes no instruction bits. However, single-register programming is awkward and requires many instructions. Even with multiple registers, only a few bits are needed to specify the register. The more that registers can be used for operand references, the fewer bits are needed
- **Number of register sets:** Most contemporary machines have one set of general-purpose registers, with typically 32 or more registers in the set. These registers can be used to store data and can be used to store addresses for displacement addressing
- **Address range:** For addresses that reference memory, the range of addresses that can be referenced is related to the number of address bits. Because this imposes a severe limitation, direct addressing is rarely used. With displacement addressing, the range is opened up to the length of the address register
- **Address granularity:** For addresses that reference memory rather than registers, another factor is the granularity of addressing. In a system with 16- or 32-bit words, an address can reference a word or a byte at the designer's choice. Byte addressing is convenient for character manipulation but requires, for a fixed-size memory, more address bits. Thus, the designer is faced with a host of factors to consider and balance.

❖ **x86 Instruction Formats**

The x86 is equipped with a variety of instruction formats. Figure 2.5 illustrates the general instruction format. Instructions are made up of from zero to four optional instruction prefixes, a 1- or 2-byte opcode, an optional address specifier (which consists of the ModR/m byte and the Scale Index byte) an optional displacement, and an optional immediate field.

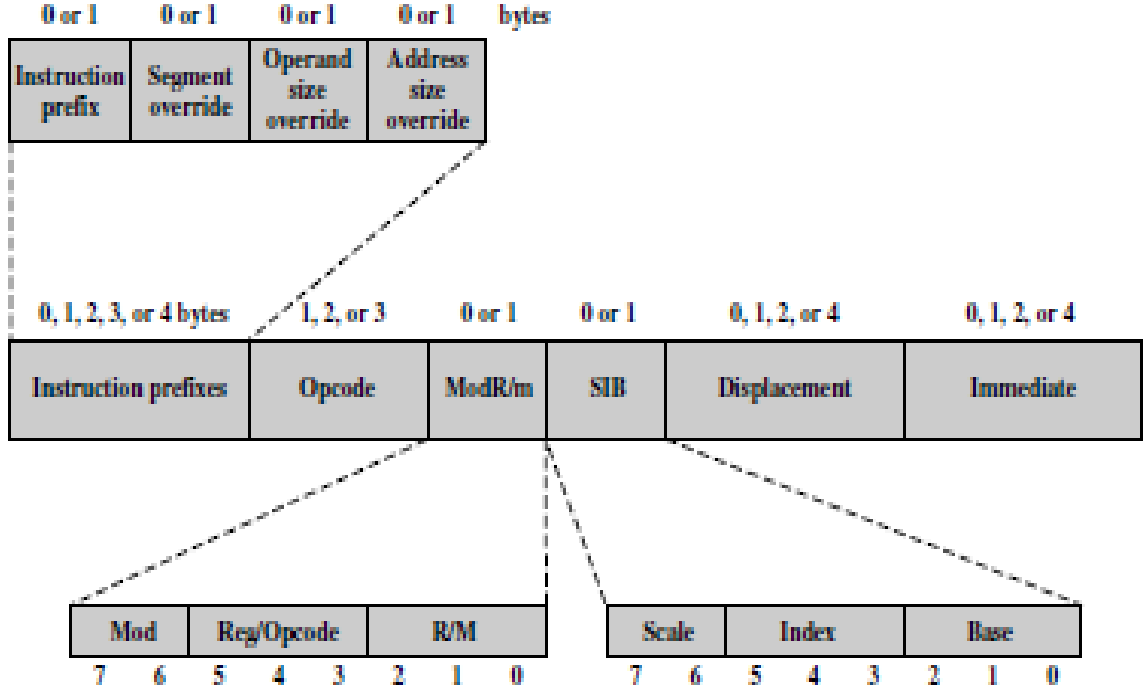


Figure 2.5 X86 Instruction Format

- **Instruction prefixes:** The instruction prefix, if present, consists of the LOCK prefix or one of the repeat prefixes. The LOCK prefix is used to ensure exclusive use of shared memory in multiprocessor environments. The repeat prefixes specify repeated operation of a string, which enables the x86 to process strings much faster than with a regular software loop.

There are five different repeat prefixes: REP, REPE, REPZ, REPNE, and REPNZ. When the absolute REP prefix is present, the operation specified in the instruction is executed repeatedly on successive elements of the string; the number of repetitions is specified in register CX.

- **Segment override:** Explicitly specifies which segment register an instruction should use, overriding the default segment-register selection generated by the x86 for that instruction.
- **Operand size:** An instruction has a default operand size of 16 or 32 bits, and the operand prefix switches between 32-bit and 16-bit operands.
- **Address size:** The processor can address memory using either 16- or 32-bit addresses. The address size determines the displacement size in instructions and the size of address offsets generated during effective address calculation.
- **Opcode:** The opcode field is 1, 2, or 3 bytes in length. The opcode may also include bits that specify if data is byte- or full-size (16 or 32 bits depending on context), direction of data operation (to or from memory), and whether an immediate data field must be sign extended.
- **ModR/m:** This byte, and the next, provide addressing information. The ModR/m byte specifies whether an operand is in a register or in memory; if it is in memory, then fields within the byte specify the addressing mode to be used. The ModR/m byte consists of three fields:

The Mod field (2 bits) combines with the r/m field to form 32 possible values: 8 registers and 24 indexing modes;

the Reg/Opcode field (3 bits) specifies either a register number or three more bits of opcode information; the r/m field (3 bits) can specify a register as the location of an operand, or it can form part of the addressing-mode encoding in combination with the Mod field.

- **SIB:** Certain encoding of the ModR/m byte specifies the inclusion of the SIB byte to specify fully the addressing mode.The SIB byte consists of three fields: The Scale field (2 bits) specifies the scale factor for scaled indexing; the Index field (3 bits) specifies the index register; the Base field (3 bits) specifies the base register.
- **Displacement:** When the addressing-mode specifier indicates that a displacement is used, an 8-, 16-, or 32-bit signed integer displacement field is added.
- **Immediate:** Provides the value of an 8-, 16-, or 32-bit operand

Several comparisons may be useful here.

In the x86 format, the addressing mode is provided as part of the opcode sequence rather than with each operand. Because only one operand can have address-mode information, only one memory operand can be referenced in an instruction. In contrast, the VAX carries the address-mode information with each operand, allowing memory-to-memory operations. The x86 instructions are therefore more compact. However, if a memory-to-memory operation is required, the VAX can accomplish this in a single instruction.

The x86 format allows the use of not only 1-byte, but also 2-byte and 4-byte offsets for indexing. Although the use of the larger index offsets results in longer instructions, this feature provides needed flexibility.

❖ PROCESSOR ORGANISATION

To understand the organization of the processor, let us consider the requirements placed on the processor, the things that it must do:

- **Fetch instruction:** The processor reads an instruction from memory (register, cache, main memory).
- **Interpret instruction:** The instruction is decoded to determine what action is required.
- **Fetch data:** The execution of an instruction may require reading data from memory or an I/O module.
- **Process data:** The execution of an instruction may require performing some arithmetic or logical operation on data.

- **Write data:** The results of an execution may require writing data to memory or an I/O module.

To do these things, it should be clear that the processor needs to store some data temporarily. In other words, the processor needs a small internal memory.

Figure 2.6 is a simplified view of a processor, indicating its connection to the rest of the system via the system bus. The major components of the processor are an *arithmetic and logic unit* (ALU) and a *control unit* (CU). The ALU does the actual computation or processing of data. The control unit controls the movement of data and instructions into and out of the processor and controls the operation of the ALU. In addition, the figure shows a minimal internal memory, consisting of a set of storage locations, called *registers.*

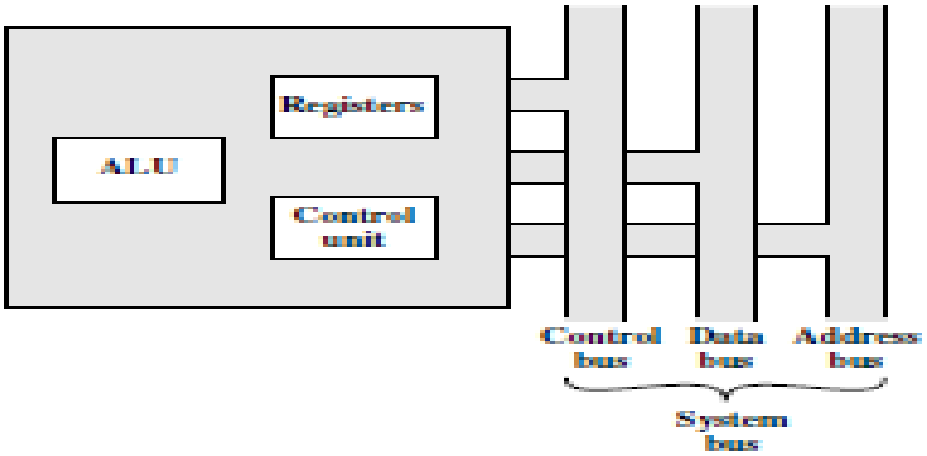
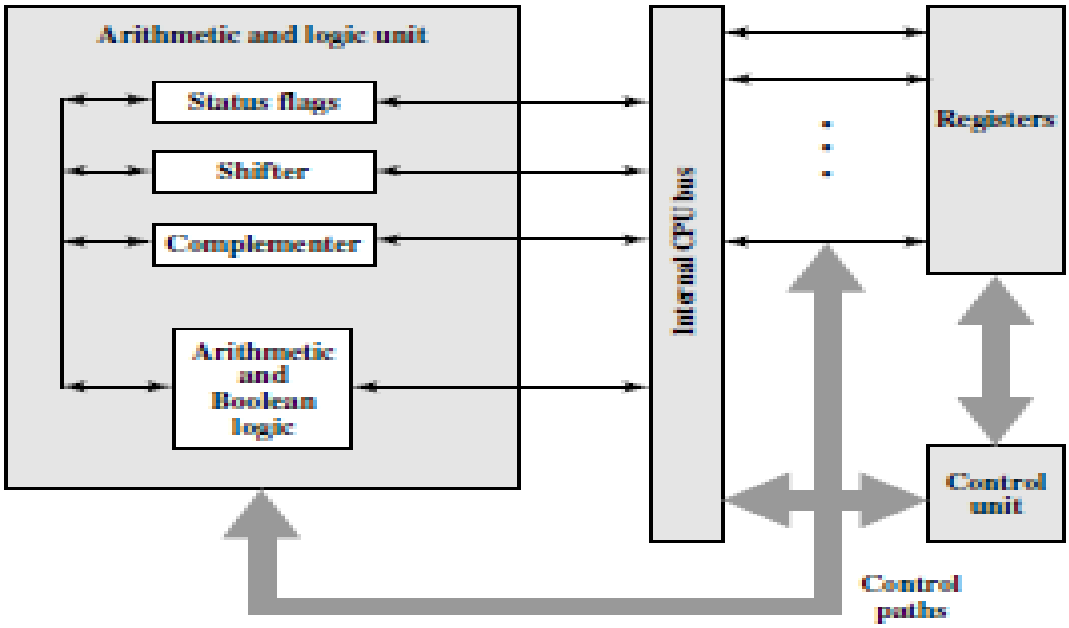


Figure 2.6 The CPU With System Bus

Figure 2.7 is a slightly more detailed view of the processor. The data transfer and logic control paths are indicated, including *internal processor bus which* is needed to transfer data between the various registers and the ALU because the ALU in fact operates only on data in the internal processor memory.



2.7Internal Structure of the CPU

❖ REGISTER ORGANISATION

A computer system employs a memory hierarchy. At higher levels of the hierarchy, memory is faster, smaller, and more expensive (per bit). Within the processor, there is a set of registers that function as a level of memory above main memory and cache in the hierarchy. The registers in the processor perform two roles:

- **User-visible registers:** Enable the machine- or assembly language programmer to minimize main memory references by optimizing use of registers.
- **Control and status registers:** Used by the control unit to control the operation of the processor and by privileged, operating system programs to control the execution of programs.

User-Visible Registers

A user-visible register is one that may be referenced by means of the machine language that the processor executes. We can characterize these in the following categories:

- General purpose
- Data
- Address
- Condition codes

General-purpose registers can be assigned to a variety of functions by the programmer. Sometimes their use within the instruction set is orthogonal to the operation. That is, any general-purpose register can contain the operand for any opcode. This provides true general-purpose register use. There may be dedicated registers for floating-point and stack operations.

In some cases, general-purpose registers can be used for addressing functions (e.g., register indirect, displacement).

Data registers may be used only to hold data and cannot be employed in the calculation of an operand address.

Address registers may themselves be somewhat general purpose, or they may be devoted to a particular addressing mode. Examples include the following:

- **Segment pointers:** In a machine with segmented addressing, a segment register holds the address of the base of the segment.
- **Index registers:** These are used for indexed addressing and may be auto indexed.
- **Stack pointer:** If there is user-visible stack addressing, then typically there is a dedicated register that points to the top of the stack.

There are several design issues to be addressed here.

- An important issue is whether to use completely general-purpose registers or to specialize their use.
- Another design issue is the number of registers, general purpose or data plus address, to be provided. Again, this affects instruction set design because more registers require more operand specifier bits.
- Finally, there is the issue of register length. Registers that must hold addresses obviously must be at least long enough to hold the largest address. Data registers should be able to hold values of most data types. Some machines allow two contiguous registers to be used as one for holding double-length values.

Condition codes (also referred to as *flags*): Condition codes are bits set by the processor hardware as the result of operations. For example, an arithmetic operation may produce a positive, negative, zero, or overflow result. In addition to the result itself being stored in a register or memory, a condition code is also set. The code may subsequently be tested as part of a conditional branch operation.

Table 2.3, lists key advantages and disadvantages of condition codes

Advantages	Disadvantages
1. Because condition codes are set by normal arithmetic and data movement instructions, they should reduce the number of COMPARE and TEST instructions needed. 2. Conditional instructions, such as BRANCH are simplified relative to composite instructions, such as TEST AND BRANCH. 3. Condition codes facilitate multiway branches. For example, a TEST instruction can be followed by two branches, one on less than or equal to zero and one on greater than zero.	1. Condition codes add complexity, both to the hardware and software. Condition code bits are often modified in different ways by different instructions, making life more difficult for both the microprogrammer and compiler writer. 2. Condition codes are irregular; they are typically not part of the main data path, so they require extra hardware connections. 3. Often condition code machines must add special non-condition-code instructions for special situations anyway, such as bit checking, loop control, and atomic semaphore operations. 4. In a pipelined implementation, condition codes require special synchronization to avoid conflicts.

Table 2.3 Condition code Advantages and Disadvantages

Control and Status Registers

There are a variety of processor registers that are employed to control the operation of the processor. Most of these, on most machines, are not visible to the user. Some of them may be visible to machine instructions executed in a control or operating system mode.

Four Registers are essential for instruction Execution

- **Program counter (PC):** Contains the address of an instruction to be fetched
- **Instruction register (IR):** Contains the instruction most recently fetched
- **Memory address register (MAR):** Contains the address of a location in memory
- **Memory buffer register (MBR):** Contains a word of data to be written to memory or the word most recently read

Many processor designs include a register or set of registers, often known as the *program status word* (PSW), that contain status information. The PSW typically contains condition codes plus other status information. Common fields or flags include the following:

- **Sign:** Contains the sign bit of the result of the last arithmetic operation.
- **Zero:** Set when the result is 0.
- **Carry:** Set if an operation resulted in a carry (addition) into or borrow (subtraction) out of a high-order bit. Used for multiword arithmetic operations.
- **Equal:** Set if a logical compare result is equality.
- **Overflow:** Used to indicate arithmetic overflow.
- **Interrupt Enable/Disable:** Used to enable or disable interrupts.
- **Supervisor:** Indicates whether the processor is executing in supervisor or user mode. Certain privileged instructions can be executed only in supervisor mode, and certain areas of memory can be accessed only in supervisor mode.

Example Microprocessor Register Organizations

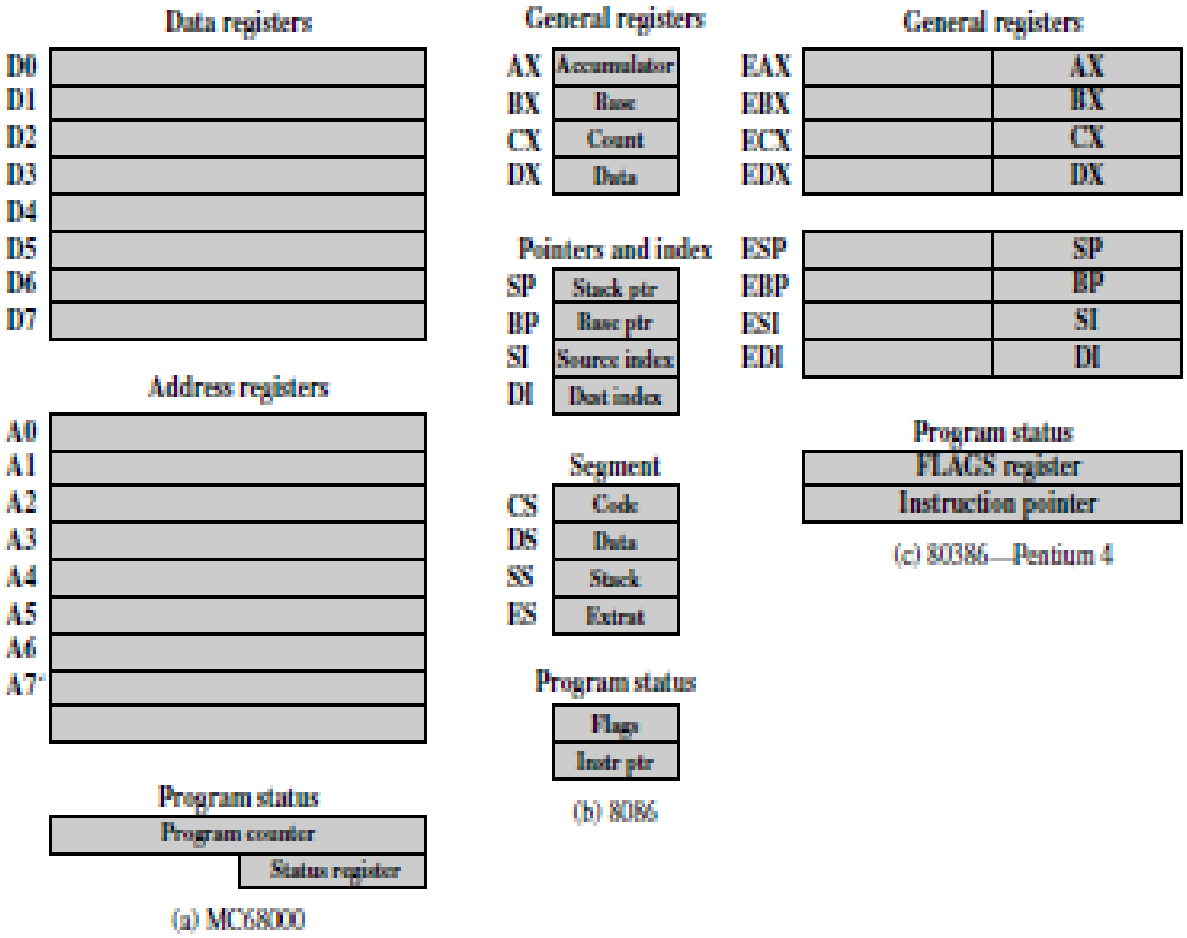


Figure 2.8 Example Microprocessor Register Organisation

• It is instructive to examine and compare the register organization of comparable systems. In this section, we look at two 16-bit microprocessors that were designed at about the same time: the Motorola MC68000 and the Intel 8086. Figures 2.8 a and b depict the register organization of each; purely internal registers, such as a memory address register, are not shown.

• The Motorola team wanted a very regular instruction set, with no special-purpose registers. The MC68000 partitions its 32-bit registers into eight data registers and nine address registers. The eight data registers are used primarily for data manipulation and are also used in addressing as index registers. The width of the registers allows 8-, 16-, and 32-bit data operations,determined by opcode. .The address registers contain 32-bit (no segmentation) addresses; two of these registers are also used as stack pointers, one for users and one for the operating system, depending on the current execution mode. Both registers are numbered 7, because only one can be used at a time. The MC68000 also includes a 32-bit program counter and a 16-bit status register.

• The Intel 8086 takes a different approach to register organization. Every register is special purpose, although some registers are also usable as general purpose. The 8086 contains four 16-bit data registers that are addressable on a byte or 16-bit basis, and four 16-bit pointer and index registers. The data registers can be used as general purpose in some instructions. The four pointer registers are also used implicitly in a number of operations; each contains a segment offset. There are also four 16-bit segment registers. Three of the four segment registers are used in a dedicated, implicit fashion, to point to the segment of the current instruction (useful for branch instructions), a segment containing data, and a segment containing a stack, respectively. The 8086 also includes an instruction pointer and a set of 1-bit status and control flags.

❖ INSTRUCTION CYCLE

An instruction cycle includes the following stages:

- **Fetch:** Read the next instruction from memory into the processor.
- **Execute:** Interpret the opcode and perform the indicated operation.
- **Interrupt:** If interrupts are enabled and an interrupt has occurred, save the current process state and service the interrupt.

We now elaborate instruction cycle. First, we must introduce one additional stage, known as the indirect cycle.

The Indirect Cycle

The execution of an instruction may involve one or more operands in memory, each of which requires a memory access. Further, if indirect addressing is used, then additional memory accesses are required. We can think of the fetching of indirect addresses as one more instruction stages. The result is shown in Figure 2.9.

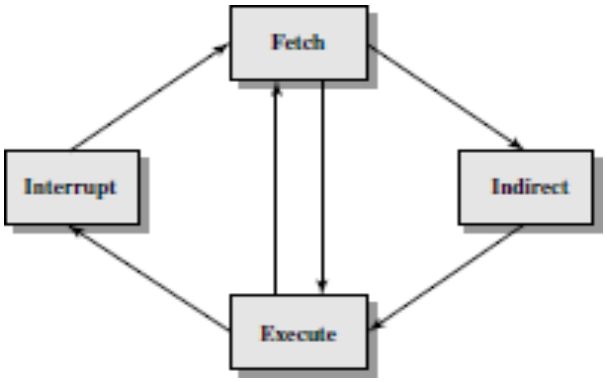


Figure 2.9 Instruction Cycle

After an instruction is fetched, it is examined to determine if any indirect addressing is involved. If so, the required operands are fetched using indirect addressing. Following execution, an interrupt may be processed before the next instruction fetch.

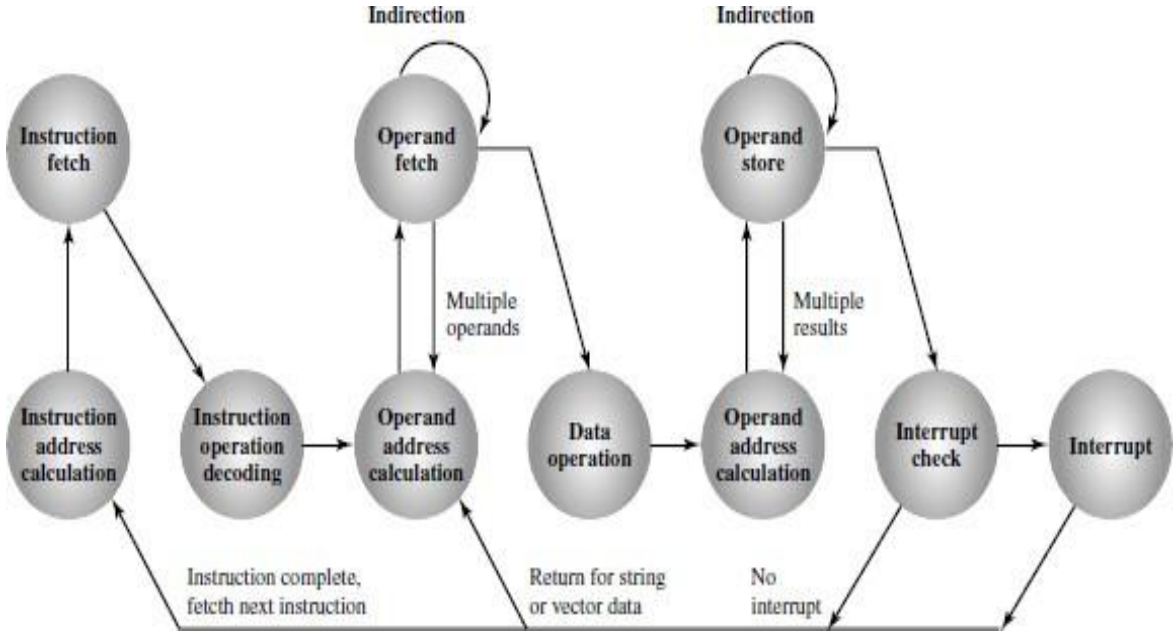


Figure 2.10 Instruction cycle State Diagram

Another way to view this process is shown in Figure 2.10. Once an instruction is fetched, its operand specifiers must be identified. Each input operand in memory is then fetched, and this process may require indirect addressing. Register-based operands need not be fetched. Once the opcode is executed, a similar process may be needed to store the result in main memory.

Data Flow

The exact sequence of events during an instruction cycle depends on the design of the processor Let us assume that a processor that employs a memory address register (MAR), a memory buffer register (MBR), a program counter (PC), and an instruction register (IR).

During the *fetch cycle*, an instruction is read from memory. Figure 2.11 shows the flow of data during this cycle. The PC contains the address of the next instruction to be fetched.This address is moved to the MAR and placed on the address bus.

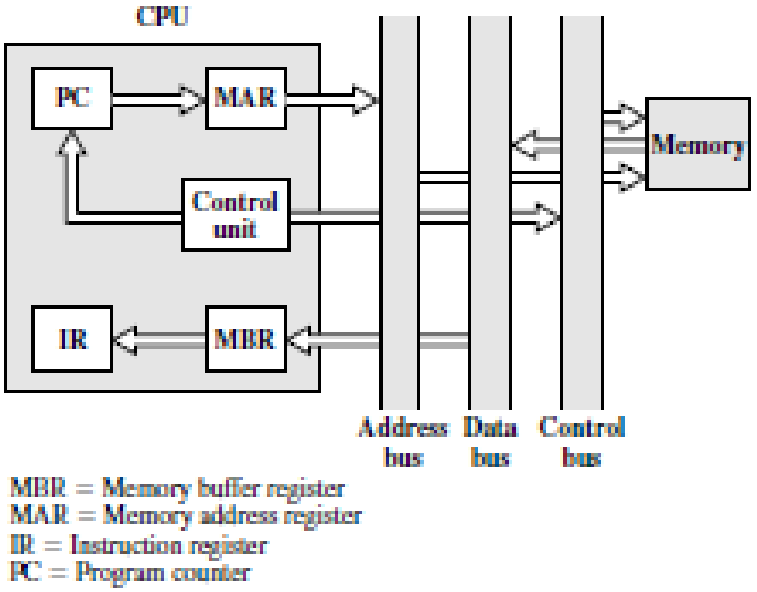


Figure 2.11 Data Flow, Fetch cycle

The control unit requests a memory read, and the result is placed on the data bus and copied into the MBR and then moved to the IR. Meanwhile, the PC is incremented by 1, preparatory for the next fetch.

Once the fetch cycle is over, the control unit examines the contents of the IR to determine if it contains an operand specifier using indirect addressing. If so, an *indirect cycle* is performed. As shown in Figure 2.12, this is a simple cycle. The right-most N bits of the MBR, which contain the address reference, are transferred to the MAR. Then the control unit requests a memory read, to get the desired address of the operand into the MBR.

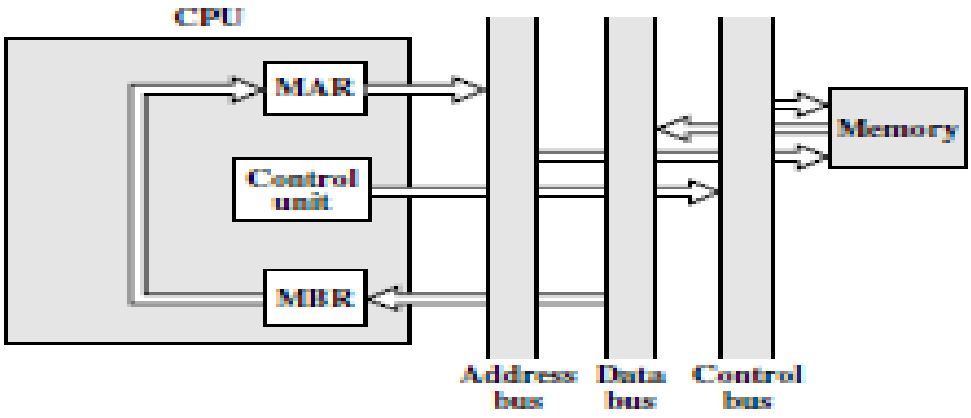


Figure 2.12 Data Flow, Indirect cycle

The fetch and indirect cycles are simple and predictable. The *execute cycle* takes many forms; the form depends on which of the various machine instructions is in the IR. This cycle may involve transferring data among registers, read or write from memory or I/O, and/or the invocation of the ALU. Like the fetch and indirect cycles, the *interrupt cycle* is simple and predictable (Figure 2.13). The current contents of the PC must be saved so that the processor can resume normal activity after the interrupt. Thus, the contents of the PC are transferred to the MBR to be written into memory. The special memory location reserved for this purpose is loaded into the MAR from the control unit. It might, for example, be a stack pointer. The PC is loaded with the address of the interrupt routine.As a result,the next instruction cycle will begin by fetching the appropriate instruction.

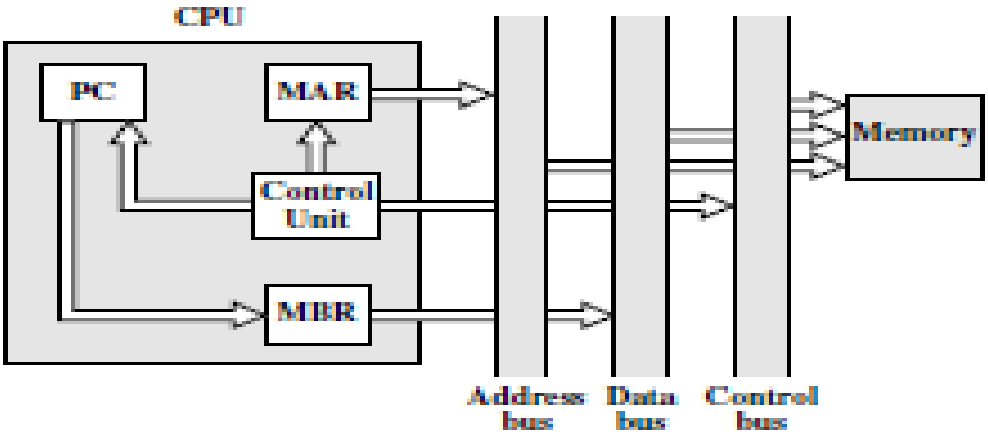


Figure 2.13 Data Flow, Interrupt Cycle