Some important quotations

|  |
| --- |
| Discuss various instruction formats. Give suitable example for each. |
| Describe stack organization in brief. |
| Discuss various addressing modes with the help of suitable example of each. |
| What is register indirect addressing mode? |
| What do you understand by LIFO & FIFO list? |
| Explain set-Associative mapping with the help of diagram. |
| Expalin various memory reference instructions with suitable example |
| What do you mean by 1- address, 2-address and 3 address instruction? |
| Discuss various addressing nodes with help of suitable example of each. |

Central Processing Unit - Introduction

* **The part of the computer** that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU.

* **The CPU is made up of three major parts**, as shown in Fig. 1. The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) performs the required microoperations for executing the instructions. The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

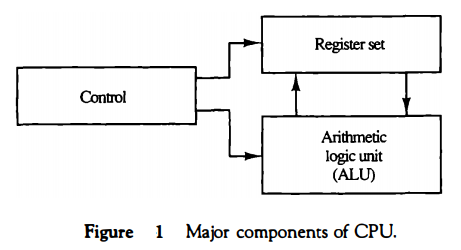
* **The CPU performs** a variety of functions dictated by the type of instructions that are incorporated in the computer.

* **Computer architecture** is sometimes defined as the computer structure and behavior as seen by the programmer that uses machine language instructions.

* **This includes** the instruction formats, addressing modes, the instruction set, and the general organization of the CPU registers.

* **One boundary** where the computer designer and the computer programmer see the same machine is the part of the CPU associated with the instruction set.

* **From the designer's point of view**, the computer instruction set provides the specifications for the design of the CPU.

* 

* **The design of a CPU**is a task that in large part involves choosing the hardware for implementing the machine instructions.

* **The user** who programs the computer in machine or assembly language must be aware of the register set, the memory structure, the type of data supported by the instructions, and the function that each instruction performs.

General Register Organization

* **Memory locations** are needed for storing pointers, counters, return addresses, temporary results, and partial products during multiplication.

* **Having to refer to memory locations** for such applications is time consuming because memory access is the most time-consuming, operation in a computer.

* **It is more convenient** and more efficient to store these intermediate values in processor registers.

* **When a large number** of registers are included in the CPU, it is most efficient to connect them through a common bus system. The registers communicate with each other not only for direct data transfers, but also while performing various microoperations.

* **Hence it is necessary** to provide a common unit that can perform all the arithmetic, logic, and shift microoperations in the processor.

* **A bus organization** for seven CPU registers is shown in Fig. 2. The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus.

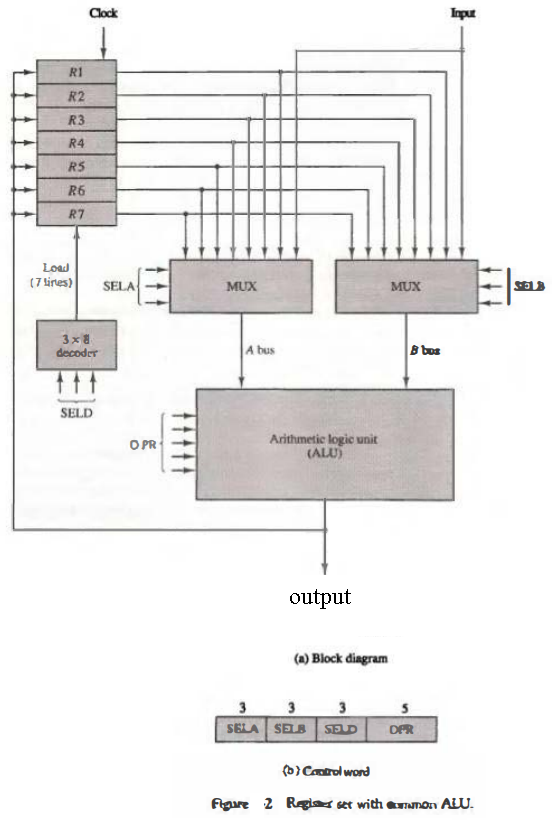
* **The A and B buses form** the inputs to a common arithmetic logic unit (ALU).

* **The operation** selected in the ALU determines the arithmetic or logic micro-operation that is to be performed.

* **The result of the microoperation** is available for output data and also goes into the inputs of all the registers.

* **The register** that receives the information from the output bus is selected by a decoder.

* **The decoder** activates one of the register load inputs, thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.

* 

* **The control unit** that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system. For example, to perform the operation R1 ← R2 + R3

* **the control must** provide binary selection variables to the following selector inputs:

* 1. **MUX A selector (SELA):** to place the content of R2 into bus A.

* 1. **MUX B selector (SELB):** to place the content o f R 3 into bus B.

* 1. **ALU operation selector (OPR):** to provide the arithmetic addition A + B.

* 1. **Decoder destination selector (SELD):** to transfer the content of the output bus into R1.
* **The four control** selection variables are generated in the control unit and must be available at the beginning of a clock cycle.

* **The data** from the two source registers propagate through the gates in the multiplexers and the ALU, to the output bus, and into the inputs of the destination register, all during the clock cycle interval.

* **Then**, when the next clock transition occurs, the binary information from the output bus is transferred into R1.

* **To achieve** a fast response time, the ALU is constructed with high-speed circuits.

Control Word

* **There are 14 binary selection** inputs in the unit, and their combined value specifies a control word. The 14-bit control word is defined in Fig. 2(b).

* **It consists of four fields**. Three fields contain three bits each, and one field has five bits.

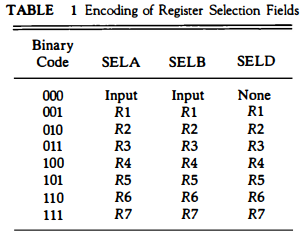
* **The three bits of SELA** select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU.

* **The three bits of SELD** select a destination register using the decoder and its seven load outputs.

* **The five bits of OPR** select one of the operations in the ALU.

* **The 14-bit control word**when applied to the selection inputs specify a particular microoperation.

* **The encoding of the register**selections is specified in Table 1.

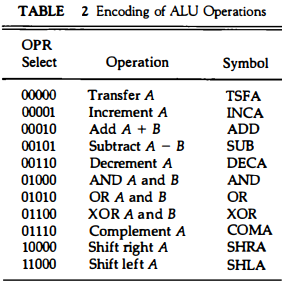
* 

* **The 3-bit binary code** listed in the first column of the table specifies the binary code for each of the three fields.

* **The register** selected by fields SELA, SELB, and SELD is the one whose decimal number is equivalent to the binary number in the code. When SELA or SELB is 000, the corresponding multiplexer selects the external input data.

* **When SELD = 000**, no destination register is selected but the contents of the output bus are available in the external output. The ALU provides arithmetic and logic operations.

* **In addition**, the CPU must provide shift operations. The shifter may be placed in the input of the ALU to provide a preshift capability, or at the output of the ALU to provide postshifting capability. In some cases, the shift operations are included with the ALU.

* **The function table for this ALU** is listed in Table 8. The encoding of the ALU operations for the CPU is specified in Table 2. The OPR field has five bits and each operation is designated with a symbolic name.

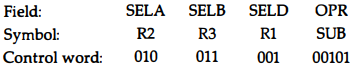
Examples of Microoperations

* **A control word of 14 bits** is needed to specify a microoperation in the CPU. The control word for a given microoperation can be derived from the selection variables.

* **For example**, the subtract rnicrooperation given by the statement R1 ← R2 - R3 specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and an ALU operation to subtract A - B.

* **Thus the control word** is specified by the four fields and the corresponding binary value for each field is obtained from the encoding listed in Tables 1 and 2.

* **The binary control word** for the subtract rnicrooperation is 010 011 001 00101 and is obtained as follows:

* 

* **The control word** for this rnicrooperation and a few others are listed in Table 3.

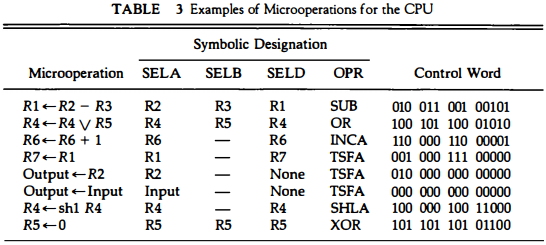
* **The increment and transfer microoperations** do not use the B input of the ALU.

* **For these cases**, the B field is marked with a dash. We assign 000 to any unused field when formulating the binary control word, although any other binary number may be used.

* **To place the content of a register** into the output terminals we place the content of the register into the A input of the ALU, but none of the registers are selected to accept the data.

* **The ALU operation TSFA** places the data from the register, through the ALU, into the output terminals.

* **The direct transfer** from input to output is accomplished with a control word

* 

* **of all 0's (making the B field 000).**

* **A register** can be cleared to 0 with an exclusive-OR operation. This is because x ⊕ x = 0.

* **It is apparent** from these examples that many other microoperations can be generated in the CPU.

* **The most efficient way** to generate control words with a large number of bits is to store them in a memory unit.

* **A memory unit** that stores control words is referred to as a control memory.

* **By reading consecutive control words from memory**, it is possible to initiate the desired sequence of microoperations for the CPU.

* **This type of control** is referred to as microprogrammed control.

Stack Organization

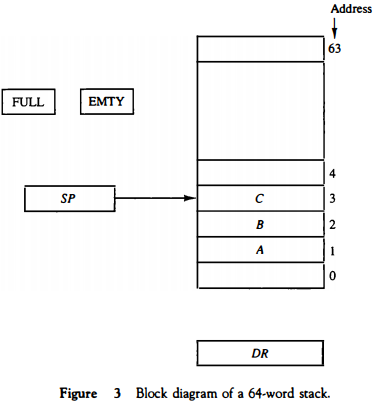
* **A useful feature that is included in the CPU** of most computers is a stack or last-in, first-out (UFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.

* **The stack in digital computers** is essentially a memory unit with an address register that can count only (after an initial value is loaded into it).

* **The register that holds** the address for the stack is called a stack pointer (SP) because its value always points at the top item in the stack.

* **The two operations** of a stack are the insertion and deletion of items. However, nothing is pushed or popped in a computer stack. These operations are simulated by incrementing or decrementing the stack pointer register.

* **Register Stack**A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Figure 3 shows the organization of a 64-word register stack. The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: A, B, and C, in that order. Item C is on top of the stack so that the content of SP is now 3.

* 

* **To remove the top item**, the stack is popped by reading the memory word at address 3 and decrementing the content of SP.

* **Item B is now on top of the stack since SP holds address 2**. To insert a new item, the stack is pushed by incrementing SP and writing a word in the next-higher location in the stack. Note that item C has been read out but not physically removed.

* **This does not matter because when the stack is pushed**, a new item is written in its place. In a 64-word stack, the stack pointer contains 6 bits because 26 = 64.

* **Since SP has only six bits**, it cannot exceed a number greater than 63 (111111 in binary). When63 is incrementedby 1, the resultis 0 since 111111 + 1 = 1000000 in binary, but SP can accommodate only the six least significant bits.

* **Similarly, when 000000 is decremented by 1**, the result is 111111. The one-bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty of items. DR is the data register that holds the binary data to be written into or read out of the stack.

* **Initially**, SP is cleared to 0, EMTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if FULL = 0), a new item is inserted with a push operation.

* **The push operation** is implemented with the following sequence of microoperations;

* SP ← SP + 1 Increment stack pointer
* M[SP] ← DR Write item on top of the stack
* If (SP = 0) then (FULL ←1) Check if stack is full
* EMTY ← 0 Mark the stack not empty

* **The stack pointer** is incremented so that it points to the address of the next-higher word. A memory write operation inserts the word from DR into the top of the stack. Note that SP holds the address of the top of the stack and that M[SP] denotes the memory word specified by the address presently available in SP.

* **The first item stored** in the stack is at address L The last item is stored at address 0.

* **If SP reaches 0**, the stack is full of items, so FULL is set to L This condition is reached if the top item prior to the last push was in location 63 and, after incrementing SP, the last item is stored in location 0.

* **Once an item is stored** in location 0, there are no more empty registers in the stack. If an item is written in the stack, obviously the stack cannot be empty, so EMTY is cleared to 0.

* **A new item**is deleted from the stack if the stack is not empty (if EMTY = 0). The pop operation consists of the following sequence of microoperations:

* DR ← M[SP] Read item from the top of stack
* SP ← SP - 1 Decrement stack pointer
* If (SP = 0) then (EMTY ← 1) Check if stack is empty
* FULL ← 0 Mark the stack not full

* **The top item** is read from the stack into DR . The stack pointer is then decremented. If its value reaches zero, the stack is empty, so EMTY is set to 1.

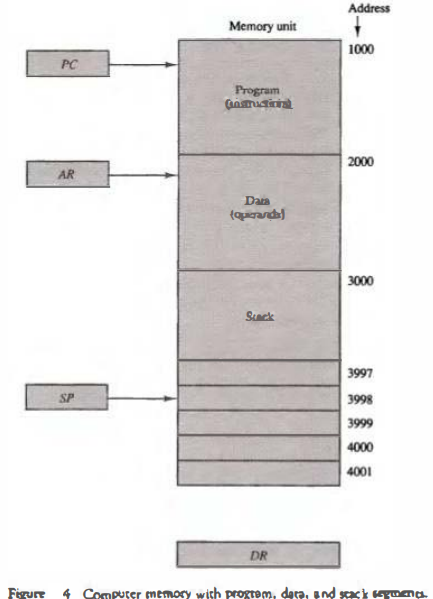
* **This condition** is reached if the item read was in location 1. Once this item is read out, SP is decremented and reaches the value 0, which is the initial value of SP. Note that if a pop operation reads the item from location 0 and then SP is decremented, SP changes to 111111, which is equivalent to decimal 63.

* **In this configuration**, the word in address 0 receives the last item in the stack. Note also that an erroneous operation will result if the stack is pushed when FULL = 1 or popped when EMTY = 1.

Memory Stack

* **A stack can exist** as a stand-alone unit as in Fig. 3 or can be implemented in a random-access memory attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer.

* **Figure 4** shows a portion of computer memory partitioned into three segments: program, data, and stack. The program counter PC points at the address of the next instruction in the program. The address register AR points at an array of data.

* 

* **The stack pointer**SP points at the top of the stack. The three registers are connected to a common address bus, and either one can provide an address for memory.

* **PC is used during the fetch phase**to read an instruction. AR is used during the execute phase to read an operand.

* **SP is used to push or pop items** into or the stack. As shown in Fig. 4, the initial value of SP is 4001 and the stack grows with decreasing addresses.

* **Thus the first item** stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack Is 3000.

* **No provisions** are available for stack limjt checks.

* **We assume that the items** in the stack communicate with a data register DR . A new item is inserted with the push operation as follows:

* SP ← SP - 1
* M[SP] ← DR

* **The stack pointer**is decremented so that it points at the address of the next word. A memory write operation inserts the word from DR into the top of the stack. A new item is deleted with a pop operation as follows:

* DR ← M[SP]
* SP ← SP + 1

* **The top item** is read from the stack into DR. The stack pointer is then incremented to point at the next item in the stack.

* **Most computers** do not provide hardware to check for stack overflow (full stack) or underflow (empty stack).

* **The stack limits** can be checked by using two processor registers: one to hold the upper limit (3000 in this case), and the other to hold the lower limit (4001 in this case).

* **After a push operation**, SP is compared with the upper-limit register and after a pop operation, SP is compared with the lower-limit register.

* **The two microoperations** needed for either the push or pop are (1) an access to memory through SP, and (2) updating SP. Which of the two microoperations is done first and whether SP is updated by incrementing or decrementing depends on the organization of the stack.

* **In Fig. 4 the stack grows** by decreasing the memory address. The stack may be constructed to grow by increasing the memory address as in Fig. 3.

* **In such a case, SP is incremented** for the push operation and decremented for the pop operation. A stack may be constructed so that SP points at the next empty location above the top of the stack.

* **In this case the sequence** of microoperations must be interchanged. A stack pointer is loaded with an initial value. This initial value must be the bottom address of an assigned stack in memory. Henceforth, SP is automatically decremented or incremented with every push or pop operation.

* **The advantage of a memory stack** is that the CPU can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.

Instruction Formats

* **A computer will usually** have a variety of instruction code formats. It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction.

* **The format of an instruction** is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

* 1. An operation code field that specifies the operation to be performed.
* 2. An address field that designates a memory address or a processor
* register.
* 3. A mode field that specifies the way the operand or the effective address
* is determined.

* **Other special fields** are sometimes employed under certain circumstances, as for example a field that gives the number of shifts in a shift-type instruction. The operation code field of an instruction is a group of bits that define various processor operations, such as add, subtract, complement, and shift.

* **The bits that define the mode** field of an instruction code specify a variety of alternatives for choosing the operands from the given address.

* **Operations specified by computer** instructions are executed on some data stored in memory or processor registers. Operands residing in memory are specified by their memory address. Operands residing in processor registers are specified with a register address.

* **A register address** is a binary number of k bits that defines one of '2k' registers in the CPU. Thus a CPU with 16 processor registers R0 through R15 will have a register address field of four bits.

* **The binary number 0101**, for example, will designate register RS. Computers may have instructions of 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 three types of CPU organizations:

* 1. Single accumulator organization.
* 2. General register organization.
* 3. Stack organization.

* **In an accumulator-type** organization all operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field.

* **For example**, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as ADD X where X is the address of the operand. The ADD instruction in this case results in the operation AC ← AC + M [X].

* **AC is the accumulator register** and M [X] symbolizes the memory word located at address X.

* **In a general register**type of organization the instruction format in this type of computer needs three register address fields.

* **Thus the instruction** for an arithmetic addition may be written in an assembly language as ADD R1 , R2 , R3 to denote the operation R1 ← R2 + R3.

* **The number of address fields** in the instruction can be reduced from three to two if the destination register is the same as one of the source registers.

* **Thus the instruction** ADD R1 , R2 would denote the operation R1 ← R1 + R2. Only register addresses for R1 and R2 need be specified in this instruction.

* **Computers** with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction. Thus the instruction MOV R1 , R2 denotes the transfer R1 ← R2 (or R2 ← R1, depending on the particular computer).

* **Thus transfer-type instructions** need two address fields to specify the source and the destination.

* **General register-type** computers employ two or three address fields in

Three-Address Instructions

* **Computers with three-address** instruction formats can use each address field to specify either a processor register or a memory operand. The program in assembly language that evaluates X = (A + B) \* (C + D) is shown below, together with comments that explain the register transfer operation of each instruction.

* ADD R1, A, B R1 ← M[A] + M[B]
* ADD R2, C, D R2 ← M[C] + M[D]
* MOL X, R1, R2 M[X] ← R1 \* R2

* **It is assumed** that the computer has two processor registers, R1 and R2. The symbol M[A] denotes the operand at memory address symbolized by A.

* **The advantage** of the three-address format is that it results in short programs when evaluating arithmetic expressions.

* **The disadvantage** is that the binary-coded instructions require too many bits to specify three addresses.

Two-Address Instructions

* **Two-address instructions** are the most common in commercial computers. Here again each address field can specify either a processor register or a memory word. The program to evaluate X = (A + B) \* (C + D) is as follows:

* MOV R1, A R1 ← M[A]
* ADD R1, B R1 ← R1 + M[B]
* MOV R2, C R2 ← M[C]
* ADD R2, D R2 ← R2 + M[D]
* MOL R1, R2 R1 ← R1 \* R2
* MOV X, R1 M[X] ← R1

* **The MOV instruction** moves or transfers the operands to and from memory and processor registers.

* **The first symbol**listed in an instruction is assumed to be both a source and the destination where the result of the operation is transferred.

One-Address Instructions

* **One-address instructions** use an implied accumulator (AC) register for all data manipulation. For multiplication and division there is a need for a second register. However, here we will neglect the second register and assume that the AC contains the result of all operations.

* **The program to evaluate** X = (A + B) \* (C + D) is

* LOAD A AC ← M[AJ
* ADD B AC ← AC + M[B]
* STORE T M[T] ← AC
* LOAD c AC ← M[C]
* ADD D AC ← AC + M[D]
* MOL T AC ← AC\*M[T]
* STORE X M[X] ← AC

* **All operations are done** between the AC register and a memory operand. T is the address of a temporary memory location required for storing the intermediate result.

Zero-Address Instructions

* **A stack-organized computer** does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack. The following program shows how X = (A + B) \* (C + D) will be written for a stack organized computer. (TOS stands for top of stack.)

* PUSH A TOS ← A
* PUSH B TOS ← B
* ADD TOS ← (A + B)
* PUSH C TOS ← C
* PUSH D TOS ← D
* ADD TO S ← (C + D)
* MOL TOS ← (C + D)\*(A + B)
* POP X M[X] ← TOS

* **To evaluate arithmetic expressions** in a stack computer, it is necessary to convert the expression into reverse Polish notation. The name "zero-address" is given to this type of computer because of the absence of an address field in the computational instructions.

Addressing Modes

* **The operation field** of an instruction specifies the operation to be performed.

* **This operation** must be executed on some data stored in computer registers or memory words.

* **The way the operands** are chosen during program execution is dependent on the addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

* **Computers** use addressing mode techniques for the purpose of accommodating one or both of the following provisions:

* 1. **To give programming versatility**to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.

* 1. **To reduce the number of bits** in the addressing field of the instruction.
* **The availability** of the addressing modes gives the experienced assembly language programmer flexibility for writing programs that are more efficient with respect to the number of instructions and execution time.

* **To understand** the various addressing modes to be presented in this section, it is imperative that we understand the basic operation cycle of the computer.

* **The control unit**of a computer is designed to go through an instruction cycle that is divided into three major phases:

* 1. Fetch the instruction from memory.
* 2. Decode the instruction.
* 3. Execute the instruction.

* **There is one register** in the computer called the program counter or PC that keeps track of the instructions in the program stored in memory. PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory.

* **The decoding done in step 2** determines the operation to be performed, the addressing mode of the instruction, and the location of the operands. The computer then executes the instruction and returns to step 1 to fetch the next instruction in sequence.

* **In some computers** the addressing mode of the instruction is specified with a distinct binary code, just like the operation code is specified. Other computers use a single binary code that designates both the operation and the mode of the instruction.

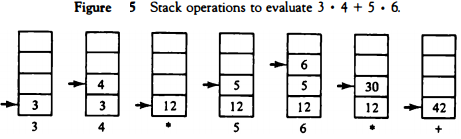
* **Instructions** may be defined with a variety of addressing modes, and sometimes, two or more addressing modes are combined in one instruction.

* **An example of an instruction format** with a distinct addressing mode field is shown in Fig. 6. The operation code specifies the operation to be permode field formed. The mode field is used to locate the operands needed for the operation.

* **There may or may not** be an address field in the instruction. If there is an address field, it may designate a memory address or a processor register.

* **Moreover**, as discussed in the preceding section, the instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

* **Although most addressing modes** modify the address field of the instruction, there are two modes that need no address field at all. These are the implied and immediate modes.

* 

Implied Mode

* **In this mode the operands** are specified implicitly in the definition of the instruction.

* **For example**, the instruction "complement accumulator" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction.

* **In fact**, all register reference instructions that use an accumulator are implied-mode instructions.

* **Zero-address instructions** in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

Immediate Mode

* **In this mode the operand** is specified in the instruction itself. In other words, an immediate-mode instruction has an operand field rather than an address field.

* **The operand field** contains the actual operand to be used in conjunction with the operation specified in the instruction. Immediate-mode instructions are useful for initializing registers to a constant value.

* **It was mentioned previously** that the address field of an instruction may specify either a memory word or a processor register.

* **When the address**field specifies a processor register, the instruction is said to be in the register mode.

Register Mode

* **In this mode** the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction.

* **A k-bit field** can specify any one of 2k registers.

Register Indirect Mode

* **In this mode the instruction** specifies a register in the CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself.

* **Before using a register** indirect mode instruction, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction.

* **A reference** to the register is then equivalent to specifying a memory address.

* **The advantage** of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

Autoincrement or Autodecrement Mode

* **This is similar** to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.

* **When the address** stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table.

* **This can be achieved**by using the increment or decrement instruction.

* **However**, because it is such a common requirement, some computers incorporate a special mode that automatically increments or decrements the content of the register after data access. The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory.

* **Sometimes the value** given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated.

* **To differentiate** among the various addressing modes it is necessary to distinguish between the address part of the instruction and the effective address used by the control when executing the instruction.

* **The effective address** is defined to be the memory address obtained from the computation dictated by the given addressing mode. The effective address is the address of the operand in a computational- type instruction.

* **It is the address** where control branches in response to a branch-type instruction.

Direct Address Mode

* **In this mode the effective address** is equal to the address part of the instruction.

* **The operand resides** in memory and its address is given directly by the address field of the instruction.

* **In a branch-type** instruction the address field specifies the actual branch address.

Indirect Address Mode

* **In this mode the address field** of the instruction gives the address where the effective address is stored in memory.

* **Control fetches the instruction** from memory and uses its address part to access memory again to read the effective address.

* **A few addressing modes** require that the address field of the instruction be added to the content of a specific register in the CPU.

* **The effective address** in these modes is obtained from the following computation: effective address = address part of instruction + content of CPU register

* **The CPU register** used in the computation may be the program counter, an index register, or a base register.

* **In either case we have a different addressing** mode which is used for a different application.

Relative Address Mode

* **In this mode the content of the program counter** is added to the address part of the instruction in order to obtain the effective address.

* **The address part of the instruction** is usually a signed number (in 2' s complement representation) which can be either positive or negative. When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction.

* **To clarify with an example**, assume that the program counter contains the number 825 and the address part of the instruction contains the number 24.

* **The instruction at location 825** is read from memory during the fetch phase and the program counter is then incremented by one to 826. The effective address computation for the relative address mode is 826 + 24 = 850.

* **This is 24 memory locations** forward from the address of the next instruction. Relative addressing is often used with branch-type instructions when the branch address is in the area surrounding the instruction word itself.

* **It results**in a shorter address field in the instruction format since the relative address can be specified with a smaller number of bits compared to the number of bits required to designate the entire memory address.

Indexed Addressing Mode

* **In this mode the content** of an index register is added to the address part of the instruction to obtain the effective address.

* **The index register** is a special CPU register that contains an index value.

* **The address field of the instruction** defines the beginning address of a data array in memory.

* **Each operand in the array** is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the index register.

* **Any operand in the array** can be accessed with the same instruction provided that the index register contains the correct index value.

* **The index register** can be incremented to facilitate access to consecutive operands. Note that if an indextype instruction does not include an address field in its format, the instruction converts to the register indirect mode of operation.

* **Some computers** dedicate one CPU register to function solely as an index register.

* **This register** is involved implicitly when the index-mode instruction is used.

* **In computers** with many processor registers, any one of the CPU registers can contain the index number.

* **In such a case** the register must be specified explicitly in a register field within the instruction format.

Base Register Addressing Mode

* **In this mode the content of a base register** is added to the address part of the instruction to obtain the effective address.

* **This is similar to the indexed addressing mode** except that the register is now called a base register instead of an index register.

* **The difference** between the two modes is in the way they are used rather than in the way that they are computed. An index register is assumed to hold an index number that is relative to the address part of the instruction.

* **A base register** is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address. The base register addressing mode is used in computers to facilitate the relocation of programs in memory.

* **When programs** and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of instructions must reflect this change of position.

* **With a base register,** the displacement values of instructions do not have to change. Only the value of the base register requires updating to reflect the beginning of a new memory segment.

Numerical Example

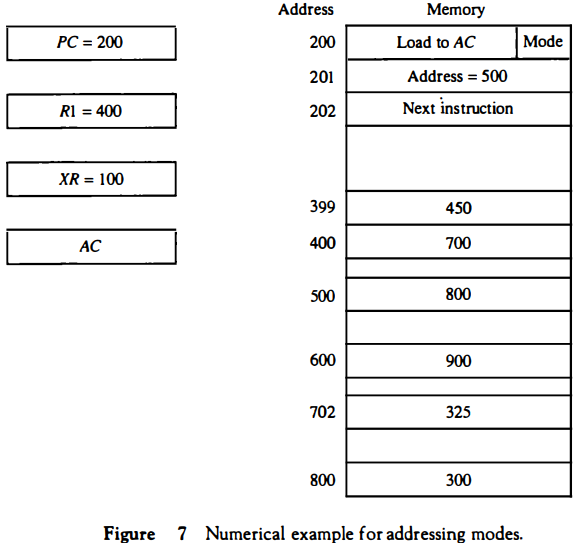
* **To show the differences** between the various modes, we will show the effect of the addressing modes on the instruction defined in Fig. 7.

* **The two-word instruction at address 200 and 201** is a "load to AC" instruction with an address field equal to 500.

* **The first word of the instruction** specifies the operation code and mode, and the second word specifies the address part.

* **PC has the value 200 for fetching this instruction**. The content of processor register R1 is 400, and the content of an index register XR is 100.

* **AC receives the operand** after the instruction is executed. The figure lists a few pertinent addresses and shows the memory content at each of these addresses.

* 

* **The mode field of the instruction** can specify any one of a number of modes. For each possible mode we calculate the effective address and the operand that must be loaded into AC.

* **In the direct address mode** the effective address is the address part of the instruction 500 and the operand to be loaded into AC is 800. In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC. (The effective address in this case is 201 .)

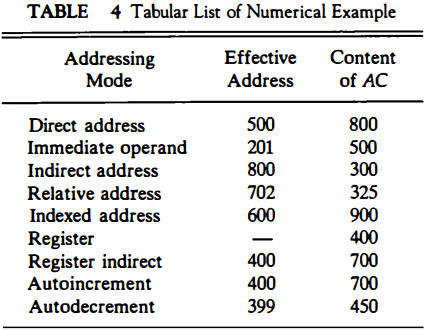
* **In the indirect mode** the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300. In the relative mode the effective address is 500 + 202 = 702 and the operand is 325.

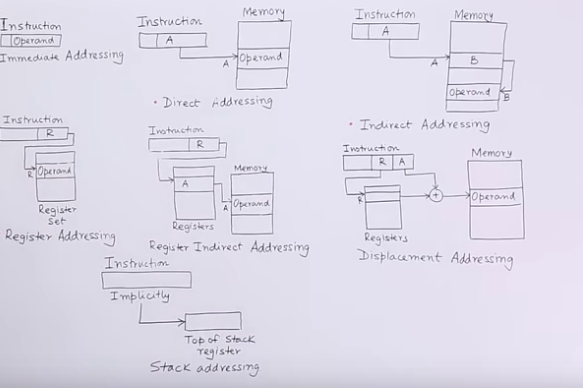
* **(Note that the value in PC after the fetch phase and during the execute phase is 202.)** In the index mode the effective address is XR + 500 = 100 + 500 = 600 and the operand is 900. In the register mode the operand is in R1 and 400 is loaded into AC.

* **(There is no effective address in this case.)** In the register indirect mode the effective address is 400, equal to the content of R1 and the operand loaded into AC is 700.

* **The autoincrement** mode is the same as the register indirect mode except that R1 is incremented to 401 after the execution of the instruction. The autodecrement mode decrements R1 to 399 prior to the execution of the instruction.

* **The operand loaded** into AC is now 450. Table 4 lists the values of the effective address and the operand loaded into AC for the nine addressing modes.

* 



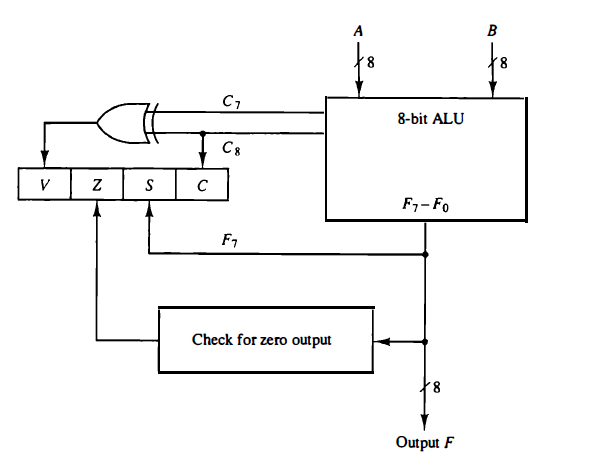
data transfer and manipulation in computer architecture

## Program Control

### 8.7.1. Status Bit Conditions

Most CPU architectures maintain a number of status bits that indicate the results from the most recent ALU operation. These bits are usually stored in a status register, which is not directly accessible as an argument in machine instructions. The bits are set automatically by many instructions, and used by conditional branch instructions that follow.

Figure 8-8



V (overflow) indicates overflow in 2's complement.

C (carry) indicates unsigned overflow or shift out.

S (sign) indicates a negative result. (Also called N)

Z (zero) indicates a result of 0. ( all bits are 0 )

1111111

A 11111111 A 11111111

B 01111111 B'+1 10000001

--------------------------------

A-B 1 10000000

V = 0 (c7 xor c8: clever way of predicting wrong sign on result)

Z = 0 (o7'o6'...o0')

S = 1 (o7)

C = 1 (c8)

Other examples of comparisons:

Status bits are set by most instructions, but especially by compare instructions. Many architectures have branch instructions that use the status bits set by the previous instruction in order to make their decisions.

movl #10, r0

loop: # some code

decl r0

cmpl r0, #0 # Unnecessary!

bgt loop

Since the decl (decrement) instruction above also affects the status bits, the cmpl (compare) instruction is redundant, and the loop can be optimized by simply removing it:

movl #10, r0

loop: # some code

decl r0 # Sets z if r0 = 0

bgt loop

Some architectures, such as the MIPS, combine compare and branch into single instructions:

bgt $t0, $t2, label

Since these instructions can do the comparison and load the PC in a single clock cycle, there is no advantage to separating them and having other instructions set the status bits.

Table 8-11

Arithmetic comparison is done by subtracting B from A, discarding the result, and checking the status bits.

Table 8-11 fails to tie status bits to arithmetic comparisons.

Branch Control function

------------------------

bhi cz': PC ← effective address

bhe c: PC ← effective address

blo c': PC ← effective address

bloe c' + z: PC ← effective address

be z: PC ← effective address

bne z': PC ← effective address

bgt n'z': PC ← effective address

bge n' + z: PC ← effective address

blt n: PC ← effective address

ble n + z: PC ← effective address

be z: PC ← effective address

bne z': PC ← effective address

## RISC and CISC

Bear in mind that these are just categories. Words are just labels and essentially empty.

### 8.8.1. CISC

* Large instruction set
* Usually memory-to-memory or register-memory
* Sophisticated instructions
* Instruction cycle takes many clock cycles
* Many addressing modes

Examples

* [x86](http://en.wikipedia.org/wiki/X86) Register-memory, 2-operand, 80 instructions on 8086/8088, many more added to 80186, 80286, 80386, 80486, Pentium (80586)
* [68k series](http://en.wikipedia.org/wiki/68k) Register-memory, 2-operand, 56 instructions on 68000, more added to 68010, etc.
* [VAX](http://en.wikipedia.org/wiki/Vax) Memory-to-memory, 3-operand, 240 instructions, 16 addressing modes, 16 general registers. The polyf instruction evaluates a polynomial given a pointer to an array of coefficients, the order of the polynomial, and a value for x.

### 8.8.2. RISC

Reduced Instruction Set Computer

* Small instruction set
* Few addressing modes
* Only load and store instructions can access memory
* Simple, fixed-length instruction code format
* Most instructions execute in 1 clock cycle (common exceptions are load, store, mul, div)
* Hardwired control unit
* Many registers

Examples:

* [Alpha](http://en.wikipedia.org/wiki/DEC_Alpha)
* [ARM](http://en.wikipedia.org/wiki/ARM_architecture)
* [MIPS](http://en.wikipedia.org/wiki/MIPS_architecture) 60 instructions, load-store architecture, 3 addressing modes (register-direct, immediate, and offset). Most instructions limited to register direct, while a few use immediate for one operand. Load and store instructions have one register direct and one offset operand.
* [PowerPC](http://en.wikipedia.org/wiki/Power_Architecture)
* [Sun Sparc](http://en.wikipedia.org/wiki/Sun_Sparc)

#### 8.8.2.1. Overlapped Register Windows

Large number of registers

Each subprogram can only access a subset

Subprogram call instruction updates pointer to indicate which subset are currently accessible.

* R0 - R9 are global to all subprograms.
* Each subprogram has access to 22 more.
* Top-level (main): R0 - R9, R10 - R31
* Called by main: R0 - R9, R26 - R47

Since only 32 of 74 registers are accessible at any moment, only 5 bits are needed to indicate a register in the instruction code.

+-------+

| R73 |

| ... |

| R63 |--+

| ... | |

| R58 | |

| ... | |

+--| R47 | |

| | ... | |

| | R42 |--+-- p2

| | ... |

| | R31 |--+

| | ... | |

p1 --+--| R26 | |

| ... | |

| R10 |--+-- main

+-------+

| R9 |--+

| ... | |

| R0 |--+-- Global

+-------+