B K Birla Institute of Engineering & Technology, Pilani, Rajasthan
Computer Architecture & Organization – 6CS4-04
Computer Science (CS B) – III Year/VI Semester
UNIT IV
Central Processing Unit

1. 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.

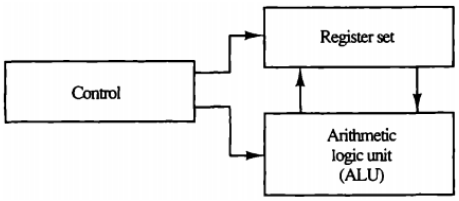


Figure 1 Major components of CPU.

- **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.

2. 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.

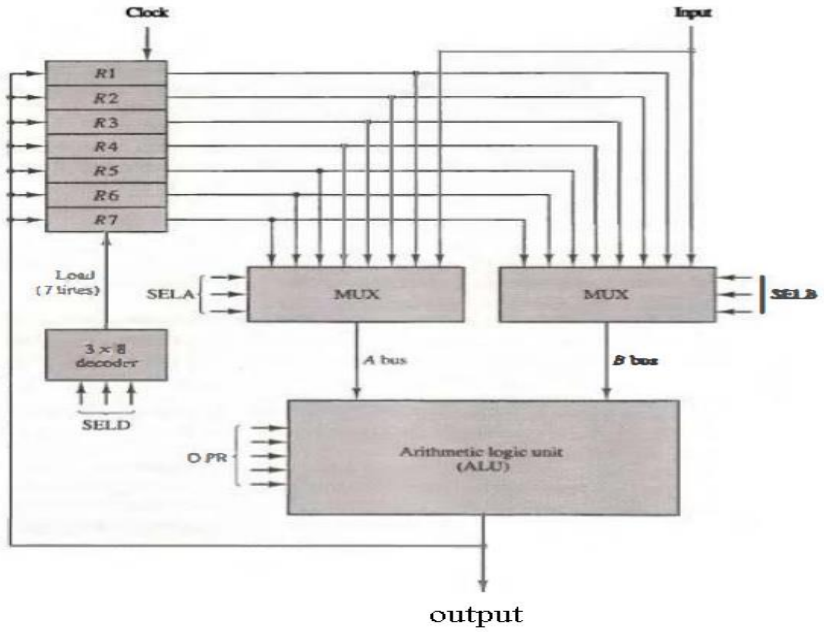


Figure 2 Register set with common ALU.

- **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.
    2. **MUX B selector (SELB):** to place the content o f R 3 into bus B.
    3. **ALU operation selector (OPR):** to provide the arithmetic addition A + B.
    4. **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.

3. 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.

TABLE 1 Encoding of Register Selection Fields

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

- **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.

TABLE 2 Encoding of ALU Operations

OPR Select	Operation	Symbol
00000	Transfer *A*	TSFA
00001	Increment *A*	INCA
00010	Add *A* + *B*	ADD
00101	Subtract *A* − *B*	SUB
00110	Decrement *A*	DECA
01000	AND *A* and *B*	AND
01010	OR *A* and *B*	OR
01100	XOR *A* and *B*	XOR
01110	Complement *A*	COMA
10000	Shift right *A*	SHRA
11000	Shift left *A*	SHLA

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 microoperation 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 microoperation is 010 011 001 00101 and is obtained as follows:

Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

- **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).**

TABLE 3 Examples of Microoperations for the CPU

Microoperation	Symbolic Designation SELA	SELB	SELD	OPR	Control Word
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	—	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	—	R7	TSFA	001 000 111 00000
Output $\leftarrow R2$	R2	—	None	TSFA	010 000 000 00000
Output $\leftarrow$ Input	Input	—	None	TSFA	000 000 000 00000
$R4 \leftarrow$ shl $R4$	R4	—	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

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

4. 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.

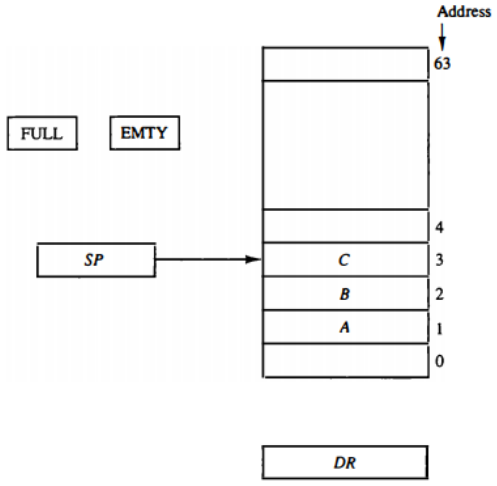


Figure 3 Block diagram of a 64-word stack.

- **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 $2^6 = 64$.
- **Since SP has only six bits**, it cannot exceed a number greater than 63 (111111 in binary). When63 is incremented by 1, the result is 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.

5. 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.

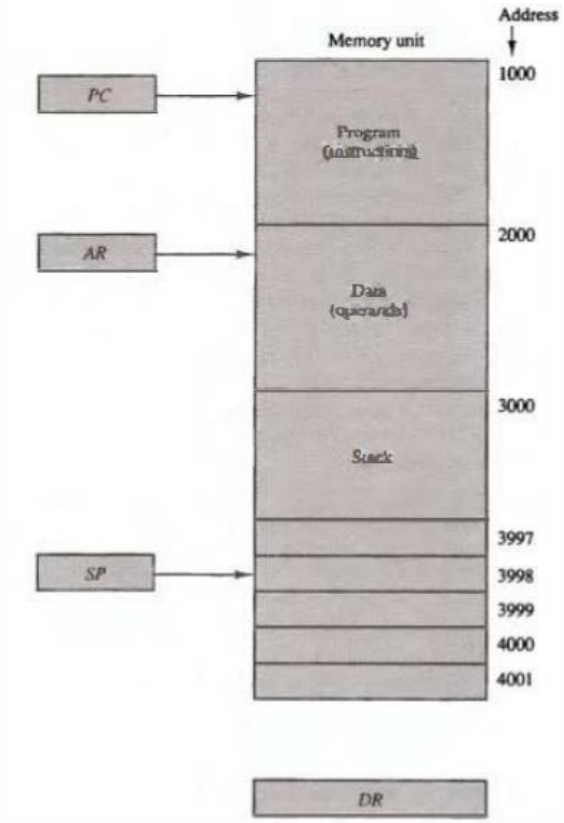


Figure 4 Computer memory with program, data, and stack segments.

- **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.

6. 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 registers.
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 '2^k' 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

7. 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.

8. 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.

9. 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.

10. 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.

11. Addressing Modes

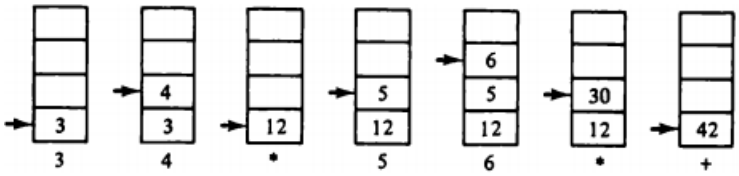
- **The operation field** of an instruction specifies the operation to be performed.
- **This operation** must be executed on some data stored in computer registers or memory words.
- **The way the operands** are chosen during program execution is dependent on the addressing mode of the instruction. 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.
    2. **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.

Figure 5 Stack operations to evaluate 3 • 4 + 5 • 6.



12. 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.

13. 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.

14. 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 2^k registers.

15. 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.

16. Auto increment or Auto decrement 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.

17. 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.

18. 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.

19. 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.

20. 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.

21. 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.

22. 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.

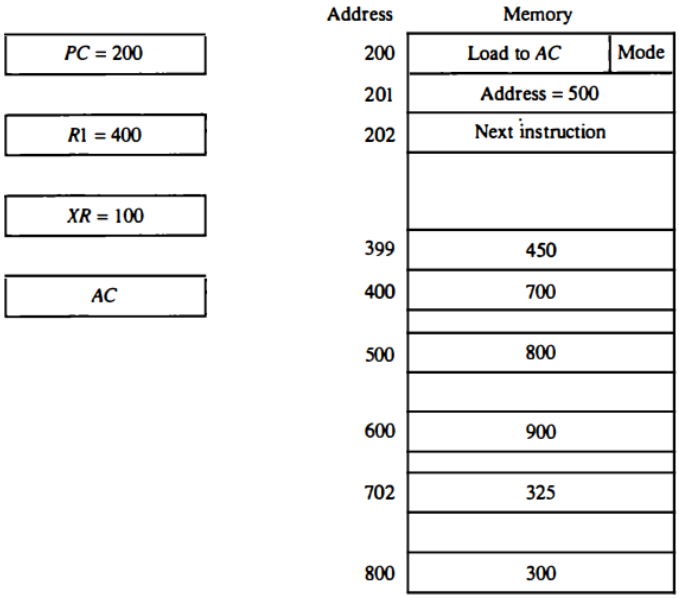


Figure 7 Numerical example for addressing modes.

- **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.

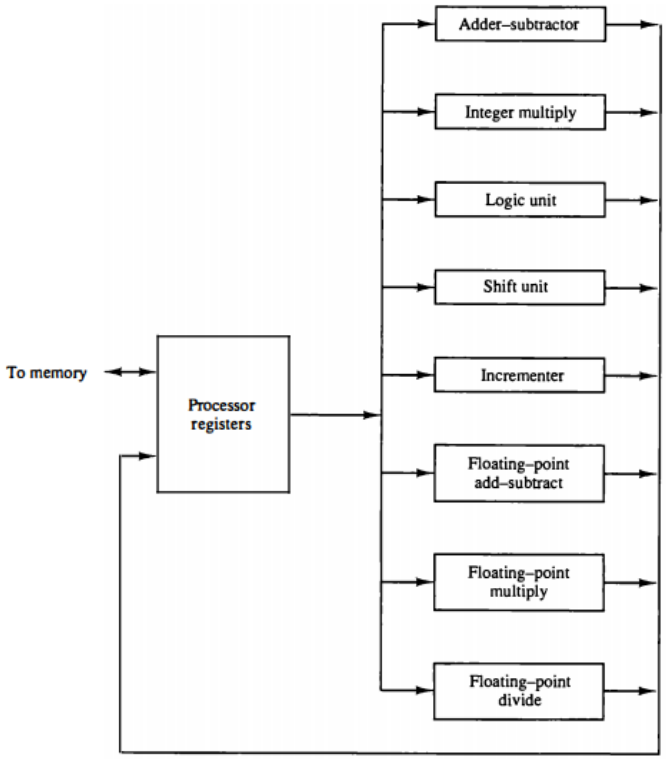
TABLE 4 Tabular List of Numerical Example

Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

23. Parallel Processing

- **Parallel processing** is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of inaeasing the computational speed of a computer system.
- **Instead of processing each instruction sequentially** as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time.
- **For example**, while an instruction is being executed in the ALU, the next instruction can be read from memory.
- **The system** may have two or more ALUs and be able to execute two or more instructions at the same time.
- **Furthermore**, the system may have two or more processors operating concurrently.
- **The purpose** of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time.
- **The amount** of hardware increases with parallel processing and with it, the cost of the system increases.
- **However**, technological developments have reduced hardware costs to the point where parallel processing techniques are economically feasible.
- **We distinguish** between parallel and serial operations by the type of registers used. Shift registers operate in serial fashion one bit at a time, while registers with parallel load operate with all the bits of the word simultaneously
- **Figure 1** shows one possible way of separating the execution unit into eight functional units operating in parallel. The operands in the registers are applied to one of the units depending on the operation specified by the instruction associated with the operands

Figure 1 Processor with multiple functional units.



- **All units** are independent of each other, so one number can be shifted while another number is being incremented.
- **A multifunctional organization** is usually associated with a complex control unit to coordinate all the activities among the various components.

24. Classification of Parallel Processing

- **One classification introduced by M. J. Flynn** considers the organization of a computer system by the number of instructions and data items that are manipulated simultaneously. The normal operation of a computer is to fetch instructions from memory and execute them in the processor.
- **The sequence of instructions** read from memory constitutes an instruction stream . The operations performed on the data in the processor constitutes a data stream .
- **Parallel processing** may occur in the instruction stream, in the data stream, or in both. Flynn's classification divides computers into four major groups as follows:

Single instruction stream, single data stream (SISD)

Single instruction stream, multiple data stream (SIMD)

Multiple instruction stream, single data stream (MISD)

Multiple instruction stream, multiple data stream (MIMD)

- **SISD represents** the organization of a single computer containing a control unit, a processor unit, and a memory unit.
- **Instructions** are executed sequentially and the system may or may not have internal parallel processing capabilities.
- **Parallel processing** in this case may be achieved by means of multiple functional units or by pipeline processing.
- **SIMD represents** an organization that includes many processing units under the supervision of a common control unit.
- **All processors** receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously.
- **MISD structure** is only of theoretical interest since no practical system has been constructed using this organization.
- **MIMD organization** refers to a computer system capable of processing several programs at the same time. Most multiprocessor and multicomputer systems can be classified in this category.
- **Flynn's classification** depends on the distinction between the performance of the control unit and the data-processing unit.
- **It emphasizes** the behavioral characteristics of the computer system rather than its operational and structural interconnections.

25. Pipelining

- **Pipelining is a technique** of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments.
- **A pipeline** can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned.
- **The result** obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments.
- **The name** "pipeline" implies a flow of information analogous to an industrial assembly line.
- **It is characteristic** of pipelines that several computations can be in progress in distinct segments at the same time. The overlapping of computation is made possible by associating a register with each segment in the pipeline.
- **The registers** provide isolation between each segment so that each can operate on distinct data simultaneously.
- **Perhaps the simplest way** of viewing the pipeline structure is to imagine that each segment consists of an input register followed by a combinational circuit. The register holds the data and the combinational circuit performs the suboperation in the particular segment.

- **The output** of the combinational circuit in a given segment is applied to the input register of the next segment. A clock is applied to all registers after enough time has elapsed to perform all segment activity. In this way the information flows through the pipeline one step at a time.
- **The pipeline organization** will be demonstrated by means of a simple example. Suppose that we want to perform the combined multiply and add operations with a stream of numbers. $A_i * B_i + C_i$ for i = 1, 2, 3, . .. , 7
- **Each suboperation** is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Fig. 2. R1 through R5 are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits. The suboperations performed in each segment of the pipeline are as follows:
- **R1** ← A_i, R2 ← B_i Input A_i and B_i R3 ← R1 * R2, R4 ← C_i Multiply and input C_i R5 ← R3 + R4 Add C_i to product
- **The five registers** are loaded with new data every clock pulse. The effect of each clock is shown in Table 1. The first clock pulse transfers A1 and B1 into R1 and R2.

Figure 2 Example of pipeline processing.

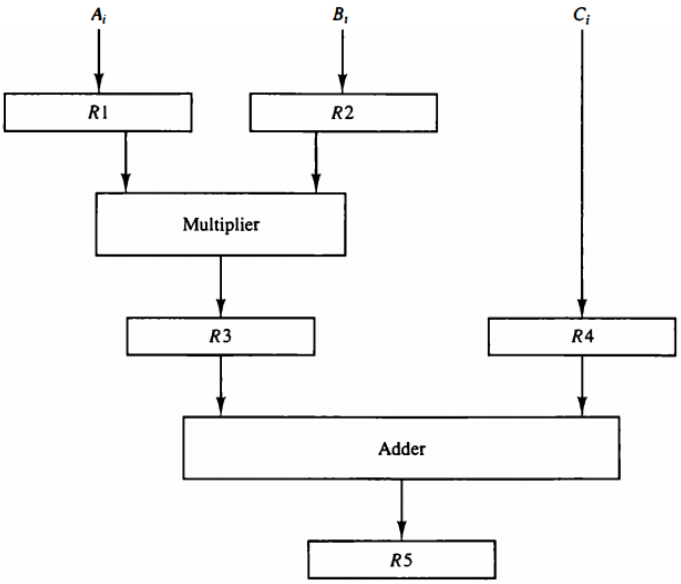


TABLE 1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

- **The second dock pulse** transfers the product of R1 and R2 into R3 and C1 into R4. The same clock pulse transfers A2 and B2 into R1 and R2. The third clock pulse operates on all three segments simultaneously.
- **It places A, and B,** into R1 and R2, transfers the product of R1 and R2 into R3, transfers C, into R4, and places the sum of R3 and R4 into RS. It takes three clock pulses to fill up the pipe and retrieve the first output from RS. From there on, each dock produces a new output and moves the data one step down the pipeline.
- **This happens as long** as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.

26. Mathematical Notations for Pipelining

- **The general structure** of a four-segment pipeline is illustrated in Fig. 3. The operands pass through all four segments in a fixed sequence.
- **Each segment** consists of a combinational circuit S_i that performs a suboperation over the data stream flowing through the pipe.
- **The segments** are separated by registers R_i that hold the intermediate results between the stages.
- **Information flows** between adjacent stages under the control of a common clock applied to all the registers simultaneously.
- **We task define a task** as the total operation performed going through all the segments in the pipeline.
- **The behavior of a pipeline** can be illustrated with a space-time diagram. This is a diagram that shows the segment utilization as a function of time. The space-time diagram of a four-segment pipeline is demonstrated in Fig. 4. The horizontal axis displays the time in clock cycles and the vertical axis gives the segment number.

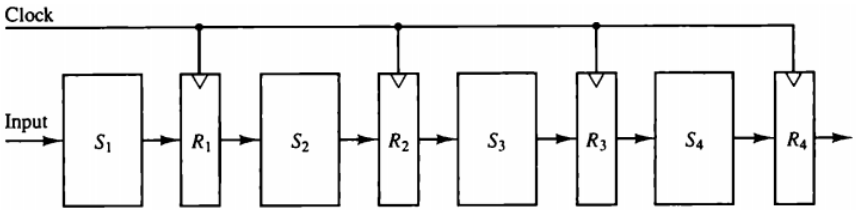


Figure 3 Four-segment pipeline.

- **The diagram** shows six tasks T1 through T6 executed in four segments. Initially, task T1 is handled by segment 1. After the first clock, segment 2 is busy with T1, while segment 1 is busy with task T2.
- **Continuing in this manner,** the first task T1 is completed after the fourth clock cycle.
- **From then on**, the pipe completes a task every clock cycle. No matter how many segments there are in the system, once the pipeline is full, it takes only one clock period to obtain an output.
- **Now consider** the case where a k-segment pipeline with a clock cycle time t_p is used to execute n tasks.

- **The first task T1 requires** a time equal to kt_p to complete its operation since there are k segments in the pipe. The remaining n - 1 tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to $(n - 1)t_p$.
- **Therefore**, to complete n tasks using a k-segment pipeline requires k + (n - 1) clock cycles. For example, the diagram of Fig. 4 shows four segments and six tasks. The time required to complete all the operations is 4 + (6 - 1) = 9 clock cycles, as indicated in the diagram.
- **Next consider a nonpipeline** unit that performs the same operation and takes a time equal to t_n. to complete each task. The total time required for n tasks is nt_n.
- **The speedup** of a pipeline processing over an equivalent nonpipeline processing is defined by the ratio S=ntn(k+n−1)tp
- **As the number** of tasks increases, n becomes much larger than k - 1, and k + n - 1 approaches the value of n. Under this condition, the speedup becomes S=tntp
- **If we assume** that the time it takes to process a task is the same in the pipeline and non pipeline circuits, we will have $t_n = kt_p$. Including this assumption, the speedup reduces to S=ktptp=k

Figure 4 Space-time diagram for pipeline.

Segment:	1	2	3	4	5	6	7	8	9	→ Clock cycles
1	T_1	T_2	T_3	T_4	T_5	T_6				
2		T_1	T_2	T_3	T_4	T_5	T_6			
3			T_1	T_2	T_3	T_4	T_5	T_6		
4				T_1	T_2	T_3	T_4	T_5	T_6	

- **This shows that the theoretical** maximum speedup that a pipeline can provide is k, where k is the number of segments in the pipeline.
- **To clarify the meaning** of the speedup ratio, consider the following numerical example. Let the time it takes to process a suboperation in each segment be equal to t_p = 20 ns.
- **Assume that the pipeline** has k = 4 segments and executes n = 100 tasks in sequence. The pipeline system will take $(k + n - 1)t_p = (4 + 99) \times 20 = 2060$ ns to complete. Assuming that $t_n = kt_p = 4 \times 20 = 80$ ns, a non pipeline system requires $nkt_p = 100 \times 80 = 8000$ ns to complete the 100 tasks.
- **The speedup** ratio is equal to 8000/2060 = 3.88. As the number of tasks increases, the speedup will approach 4, which is equal to the number of segments in the pipeline. If we assume that t_n = 60 ns, the speedup becomes 60/20 = 3.
- **To duplicate the theoretical** speed advantage of a pipeline process by means of multiple functional units, it is necessary to construct k identical units that will be operating in parallel.
- **The implication** is that a k-segment pipeline processor can be expected to equal the performance of k copies of an equivalent nonpipeline circuit under equal operating conditions. This is illustrated in Fig. 5, where four identical circuits are connected in parallel. Each P circuit performs the same task of an equivalent pipeline circuit.
- **Instead of operating with the input data** in sequence as in a pipeline, the parallel circuits accept four input data items simultaneously and perform four tasks at the same time. As far as the speed of operation is concerned, this is equivalent to a four segment pipeline.

- **Note that the four-unit circuit** of Fig. 5 constitutes a single-instruction multiple-data (SIMD) organization since the same instruction is used to operate on multiple data in parallel.
- **There are various reasons** why the pipeline cannot operate at its maximum theoretical rate. Different segments may take different times to complete their suboperation. The clock cycle must be chosen to equal the time delay of the segment with the maximum propagation time.
- **This causes** all other segments to waste time while waiting for the next clock. Moreover, it is not always correct to assume that a nonpipe circuit has the same time delay as that of an equivalent pipeline circuit.
- **Many of the intermediate** registers will not be needed in a single-unit circuit, which can usually be constructed entirely as a combinational circuit.
- **Nevertheless**, the pipeline technique provides a faster operation over a purely serial sequence even though the maximum theoretical speed is never fully achieved.
- **There are two areas of computer** design where the pipeline organization is applicable. An arithmetic pipeline divides an arithmetic operation into suboperations for execution in the pipeline segments.
- **An instruction pipeline** operates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle.

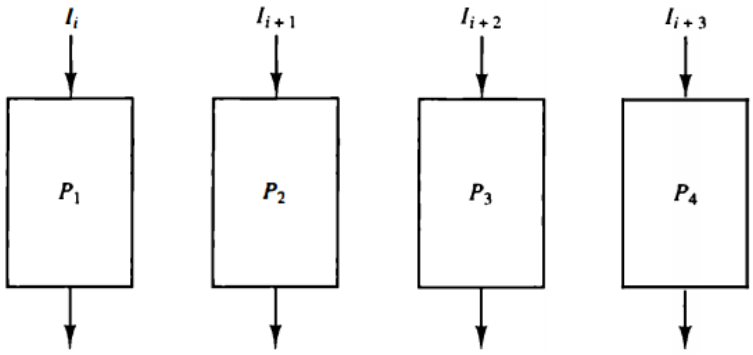


Figure 5 Multiple functional units in parallel.

27. Arithmetic Pipeline

- **Pipeline arithmetic** units are usually found in very high speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems.
- **A pipeline multiplier** is essentially an array multiplier, with special adders designed to minimize the carry propagation time through the partial products.
- **Floating-point** operations are easily decomposed into suboperations.
- **The inputs** to the floating-point adder pipeline are two normalized floating-point binary numbers.
- **$X = A * 2^a$; $Y = B * 2^b$**

- **A and B are two fractions** that represent the mantissas and a and b are the exponents. The floating-point addition and subtraction can be performed in four segments, as shown in Fig. 6. The registers labeled R are placed between the segments to store intermediate results. The suboperations that are performed in the four segments are:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

- **The exponents** are compared by subtracting them to determine their difference.
- **The larger exponent** is chosen as the exponent of the result. The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right.
- **This produces an alignment of the two mantissas**. It should be noted that the shift must be designed as a combinational circuit to reduce the shift time. The two mantissas are added or subtracted in segment 3. The result is normalized in segment 4.
- **When an overflow occurs**, the mantissa of the sum or difference is shifted right and the exponent incremented by one.
- **If an underflow occurs**, the number of leading zeros in the mantissa determines the number of left shifts in the mantissa and the number that must be subtracted from the exponent.
- **The following numerical example** may clarify the suboperations performed in each segment. For simplicity, we use decimal numbers, although Fig. 6 refers to binary numbers. Consider the two normalized floating-point numbers:

$X = 0.9504 * 10^3$

$Y = 0.8200 * 10^2$

- **The two exponents** are subtracted in the first segment to obtain 3 - 2 = 1. The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of Y to the right to obtain

$X= 0.9504 * 10^3$

$Y = 0.0820 * 10^3$

- **This aligns** the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum $Z = 1 .0324 * 10^3$

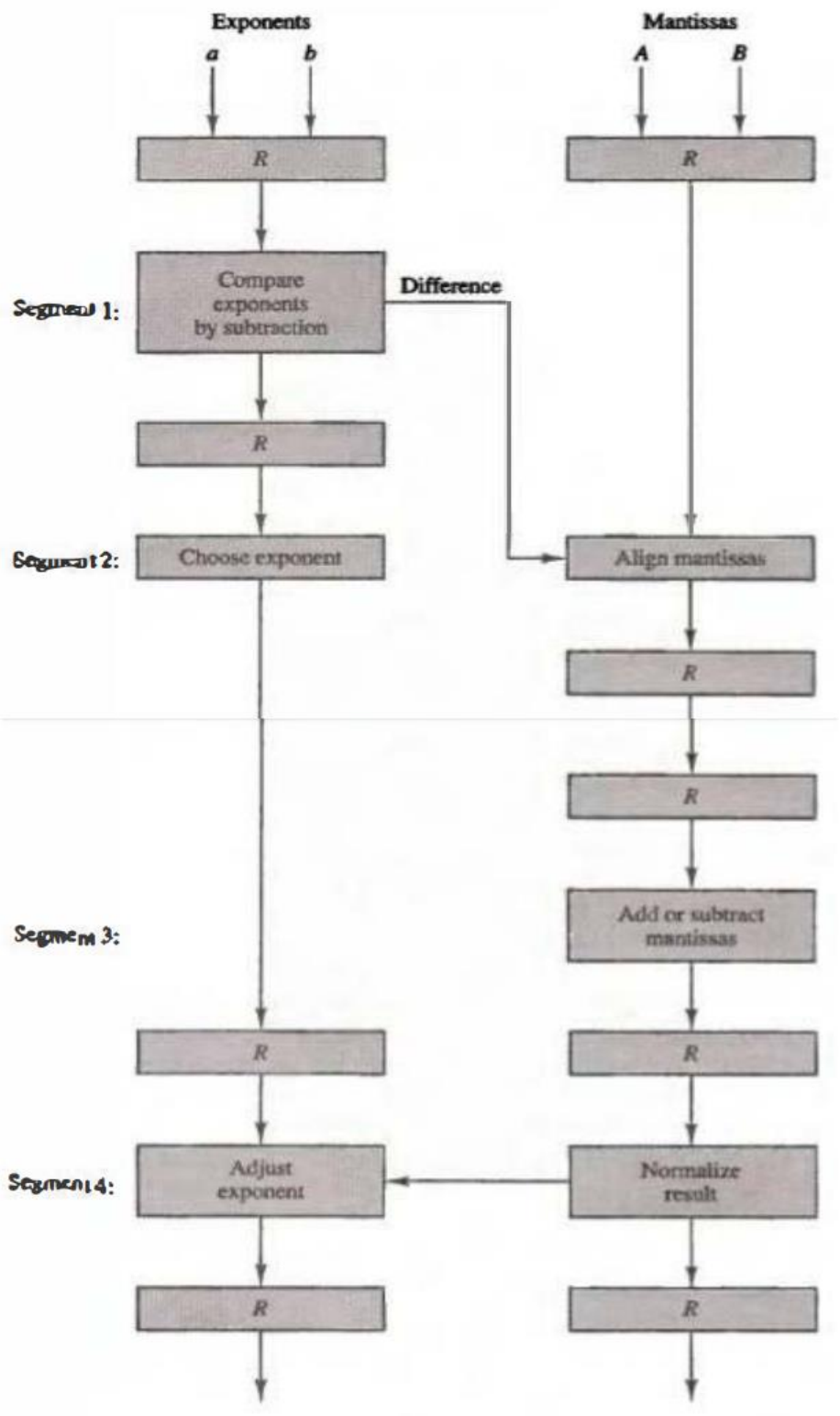


Figure 6 Pipeline for floating-point addition and subtraction.

- **The sum is adjusted** by normalizing the result so that it has a fraction with a nonzero first digit. This is done by shifting the mantissa once to the right and incrementing the exponent by one to obtain the normalized sum.
- **$Z = 0.10324 * 10^4$**
- **The comparator,** shifter, adder-subtractor, incrementer, and decrementer in the floating-point pipeline are implemented with combinational circuits. Suppose that the time delays of the four segments are $t_1 = 60$ ns, $t_2 = 70$ ns, $t_3 = 100$ ns, $t_4 = 80$ ns, and the interface registers have a delay of $t_r = 10$ ns.
- **The clock cycle** is chosen to be $t_p = t_3 + t_r = 110$ ns.
- **An equivalent** nonpipeline floating point adder-subtractor will have a delay time $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 320$ ns.
- **In this case** the pipelined adder has a speedup of 320/110 = 2.9 over the nonpipelined adder.

28. Instruction Pipeline

- **Pipeline processing** can occur not only in the data stream but in the instruction stream as well.
- **An instruction** pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and execute phases to overlap and perform simultaneous operations.
- **One possible** digression associated with such a scheme is that an instruction may cause a branch out of sequence. In that case the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded.
- **Consider a computer** with an instruction fetch unit and an instruction execution unit designed to provide a two-segment pipeline. The instruction fetch segment can be implemented by means of a first-in, first-out (FIFO) buffer. This is a type of unit that forms a queue rather than a stack.
- **Whenever the execution unit** is not using memory, the control increments the program counter and uses its address value to read consecutive instructions from memory. The instructions are inserted into the FIFO buffer so that they can be executed on a first-in, first-out basis.
- **Thus an instruction stream** can be placed in a queue, waiting for decoding and processing by the execution segment. The instruction stream queuing mechanism provides an efficient way for reducing the average access time to memory for reading instructions.
- **Whenever there is space in the FIFO buffer**, the control unit initiates the next instruction fetch phase.
- **The buffer acts** as a queue from which control then extracts the instructions for the execution unit.
- **Computers** with complex instructions require other phases in addition to the fetch and execute to process an instruction completely. In the most general case, the computer needs to process each instruction with the following sequence of steps.

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

- **There are certain difficulties** that will prevent the instruction pipeline from operating at its maximum rate. Different segments may take different times to operate on the incoming information.
- **Some segments** are skipped for certain operations. For example, a register mode instruction does not need an effective address calculation.
- **Two or more segments** may require memory access at the same time, causing one segment to wait until another is finished with the memory.
- **Memory access conflicts** are sometimes resolved by using two memory buses for accessing instructions and data in separate modules. ln this way, an instruction word and a data word can be read simultaneously from two different modules.
- **The design of an instruction** pipeline will be most efficient if the instruction cycle is divided into segments of equal duration. The time that each step takes to fulfill its function depends on the instruction and the way itis executed.

29. Example: Four-Segment Instruction Pipeline

- **Assume that the decoding** of the instruction can be combined with the calculation of the effective address into one segment.
- **Assume further** that most of the instructions place the result into a processor register so that the instruction execution and storing of the result can be combined into one segment. This reduces the instruction pipeline into four segments.
- **Figure 7 shows** how the instruction cycle in the CPU can be processed with a four-segment pipeline. While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3.
- **The effective address** may be calculated in a separate arithmetic circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions can be fetched and placed in an instruction FIFO.
- **Thus up to four suboperations** in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.
- **Once in a while**, an instruction in the sequence may be a program control type that causes a branch out of normal sequence.
- **In that case** the pending operations in the last two segments are completed and all information stored in the instruction buffer is deleted. The pipeline then restarts from the new address stored in the program counter.

- **Similarly**, an interrupt request, when acknowledged, will cause the pipeline to empty and start again from a new address value.

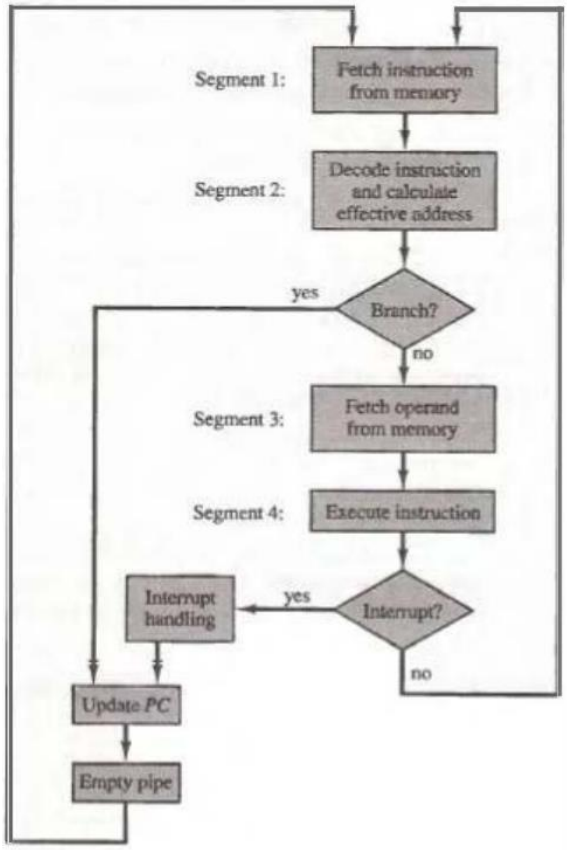


Figure 7 Four-segment CPU pipeline.

Figure 8 shows the operation of the instruction pipeline. The time in the horizontal axis is divided into steps of equal duration. The four segments are represented in the diagram with an abbreviated symbol.

1. F1 is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

- **It is assumed** that the processor has separate instruction and data memories so that the operation in Fl and FO can proceed at the same time. In the absence of a branch instruction, each segment operates on different instructions.

Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction:	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	–	–	FI	DA	FO	EX			
	5					–	–	–	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

Figure 8 Timing of instruction pipeline.

- **Thus, in step 4**, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched in segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FL Assume now that instruction 3 is a branch instruction.
- **As soon as this instruction** is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions is halted until the branch instruction is executed in step 6. If the branch is taken, a new instruction is fetched in step 7.
- **If the branch is not taken**, the instruction fetched previously in step 4 can be used. The pipeline then continues until a new branch instruction is encountered. Another delay may occur in the pipeline if the EX segment needs to store the result of the operation in the data memory while the FO segment needs to fetch an operand. In that case, segment FO must wait until segment EX has finished its operation.
- **In general**, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

30. Data Dependency

- **A difficulty** that may caused a degradation of performance in an instruction pipeline is due to possible collision of data or address.
- **A collision** occurs when an instruction cannot proceed because previous instructions did not complete certain operations.
- **A data dependency** occurs when an instruction needs data that are not yet available.
- **For example**, an instruction in the FO segment may need to fetch an operand that is being generated at the same time by the previous instruction in segment EX.
- **Therefore**, the second instruction must wait for data to become available by the first instruction. Similarly, an address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available.
- **For example**, an instruction with register indirect mode cannot proceed to fetch the operand if the previous instruction is loading the address into the register.
- **Therefore**, the operand access to memory must be delayed until the required address is available. Pipelined computers deal with such conflicts between data dependencies in a variety of ways.
- **The most straightforward** method is to insert hardware interlocks. An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline.

- **Detection of this situation** causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict. This approach maintains the program sequence by using hardware to insert the required delays.
- **Another technique** called operand forwarding uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments.
- **For example**, instead of transferring an ALU result into a destination register, the hardware checks the destination operand, and if it is needed as a source in the next instruction, it passes the result directly into the ALU input, bypassing the register.
- **This method** requires additional hardware paths through multiplexers as well as the circuit that detects the conflict.
- **A procedure** employed in some computers is to give the responsibility for solving data conflicts problems to the compiler that translates the high-level programming language into a machine language program.
- **The compiler** for such computers is designed to detect a data conflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting no-operation instructions. This method is referred to as delayed load.

31. Handling of Branch Instructions

- **One of the major problems** in operating an instruction pipeline is the occurrence of branch instructions.
- **A branch instruction** can be conditional or unconditional. An unconditional branch always alters the sequential program flow by loading the program counter with the target address.
- **In a conditional branch**, the control selects the target instruction if the condition is satisfied or the next sequential instruction if the condition is not satisfied. As mentioned previously, the branch instruction breaks the norrnal sequence of the instruction stream, causing difficulties in the operation of the instruction pipeline.
- **Pipelined computers** employ various hardware techniques to minimize the performance degradation caused by instruction branching. One way of handling a conditional branch is to prefetch the target instruction in addition to the instruction following the branch. Both are saved until the branch is executed. If the branch condition is successful, the pipeline continues from the branch target instruction. An extension of this procedure is to continue fetching instructions from both places until the branch decision is made. At that time control chooses the instruction stream of the correct program flow.
- **Another possibility** is the use of a branch target buffer or BTB.
- **The BTB is an associative memory** included in the fetch segment of the pipeline. Each entry in the BTB consists of the address of a previously executed branch instruction and the target instruction for that branch.
- **It also stores** the next few instructions after the branch target instruction. When the pipeline decodes a branch instruction, it searches the associative memory BTB for the address of the instruction.
- **If it is in the BTB**, the instruction is available directly and prefetch continues from the new path.

- **If the instruction** is not in the BTB, the pipeline shifts to a new instruction stream and stores the target instruction in the BTB. The advantage of this scheme is that branch instructions that have occurred previously are readily available in the pipeline without interruption. A variation of the BTB is the loop buffer.
- **This is a small very high speed** register file maintained by the instruction fetch segment of the pipeline. When a program loop is detected in the program, it is stored in the loop buffer in its entirety, including all branches.
- **The program loop** can be executed directly without having to access memory until the loop mode is removed by the final branching out. Another procedure that some computers use is branch prediction. A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed.
- **The pipeline then begins prefetching** the instruction stream from the predicted path. A correct prediction eliminates the wasted time caused by branch penalties. A procedure employed in most ruse processors is the delayed branch.
- **In this procedure**, the compiler detects the branch instructions and rearranges the machine language code sequence by inserting useful instructions that keep the pipeline operating without interruptions.
- **An example** of delayed branch is the insertion of a no-operation instruction after a branch instruction. This causes the computer to fetch the target instruction during the execution of the no operation instruction, allowing a continuous flow of the pipeline.

32. RISC Pipeline

- **Among the characteristics** attributed to ruse is its ability to use an efficient instruction pipeline.
- **The simplicity** of the instruction set can be utilized to implement an instruction pipeline using a small number of suboperations, with each being executed in one clock cycle.
- **Because of the fixed-length** instruction format, the decoding of the operation can occur at the same time as the register selection. All data manipulation instructions have register-to register operations.
- **Since all operands** are in registers, there is no need for calculating an effective address or fetching of operands from memory. Therefore, the instruction pipeline can be implemented with two or three segments. One segment fetches the instruction from program memory, and the other segment executes the instruction in the ALU.
- **A third segment** may be used to store the result of the ALU operation in a destination register. The data transfer instructions in RISC are limited to load and store instructions. These instructions use register indirect addressing.
- **They usually need three or four stages in the pipeline.** To prevent conflicts between a memory access to fetch an instruction and to load or store an operand, most RISC machines use two separate buses with two memories: one for storing the instructions and the other for storing the data. The two memories can sometime operate at the same speed as the CPU clock and are referred to as cache memories.

- **One of the major advantages of RISC is its ability** to execute instructions at the rate of one per clock cycle. It is not possible to expect that every instruction be fetched from memory and executed in one clock cycle.
- **What is done**, in effect, is to start each instruction with each clock cycle and to pipeline the processor to achieve the goal of single-cycle instruction execution.
- **The advantage of RISC over OSC** (complex instruction set computer) is that RISC can achieve pipeline segments, requiring just one clock cycle, while OSC uses many segments in its pipeline, with the longest segment requiring two or more clock cycles. Another characteristic of RISC is the support given by the compiler that translates the high-level language program into machine language program.
- **Instead of designing** hardware to handle the difficulties associated with data conflicts and branch penalties, RISC processors rely on the efficiency of the compiler to detect and minimize the delays encountered with these problems.

33. Example: Three-Segment Instruction Pipeline

- **The data manipulation** instructions operate on data in processor registers. The data transfer instructions are load and store instructions that use an effective address obtained from the addition of the contents of two registers or a register and a displacement constant provided in the instruction.
- **The program control** instructions use register values and a constant to evaluate the branch address, which is transferred to a register or the program counter PC.
- **Now consider the hardware operation** for such a computer. The control section fetches the instruction from program memory into an instruction register. The instruction is decoded at the same time that the registers needed for the execution of the instruction are selected.
- **The processor unit** consists of a number of registers and an arithmetic logic unit (ALU) that performs the necessary arithmetic, logic, and shift operations. A data memory is used to load or store the data from a selected register in the register file. The instruction cycle can be divided into three suboperations and implemented in three segments:

I: Instruction fetch

A: ALU operation

E: Execute instruction

- **The I segment** fetches the instruction from program memory. The instruction is decoded and an ALU operation is performed in the A segment.
- **The ALU** is used for three different functions, depending on the decoded instruction. It performs an operation for a data manipulation instruction, it evaluates the effective address for a load or store instruction, or it calculates the branch address for a program control instruction.
- **The E segment** directs the output of the ALU to one of three destinations, depending on the decoded instruction. It transfers the result of the ALU operation into a destination register in the register file, it transfers the effective address to a data memory for loading or storing, or it transfers the branch address to the program counter.

34. Delayed Load

- **Consider** now the operation of the following four instructions:

1. LOAD: R1 ← M [address 1]

LOAD: R2 ← M [address 2]

ADD: R3 ← R1 + R2

STORE: M[address 3] ← R3

- **If the three-segment pipeline** proceeds without interruptions, there will be a data conflict in instruction 3 because the operand in R2 is not yet available in the A segment.
- **This can be seen** from the timing of the pipeline shown in Fig. 9(a). The E segment in clock cycle 4 is in a process of placing the memory data into R2.
- **The A segment** in clock cycle 4 is using the data from R2, but the value in R2 will not be the correct value since it has not yet been transferred from memory.
- **It is up to the compiler** to make sure that the instruction following the load instruction uses the data fetched from memory. If the compiler cannot find a useful instruction to put after the load, it inserts a no-op (no-operation) instruction.
- **This is a type** of instruction that is fetched from

Clock cycles:	1	2	3	4	5	6
1. Load *R*1	I	A	E			
2. Load *R*2		I	A	E		
3. Add *R*1 + *R*2			I	A	E	
4. Store *R*3				I	A	E

(a) Pipeline timing with data conflict

Clock cycle:	1	2	3	4	5	6	7
1. Load *R*1	I	A	E				
2. Load *R*2		I	A	E			
3. No-operation			I	A	E		
4. Add *R*1 + *R*2				I	A	E	
5. Store *R*3					I	A	E

(b) Pipeline timing with delayed load

Figure 9 Three-segment pipeline timing.

- **memory but has no operation**, thus wasting a clock cycle. This concept of delaying the use of the data loaded from memory is referred to as delayed load.

- **Figure 9(b)** shows the same program with a no-op instruction inserted after the load to R2 instruction. The data is loaded into R2 in clock cycle 4.
- **The add instruction** uses the value of R2 in step 5.
- **Thus the no-op instruction** is used to advance one clock cycle in order to compensate for the data conflict in the pipeline. (Note that no operation is performed in segment A during clock cycle 4 or segment E during clock cycle 5.)
- **The advantage** of the delayed load approach is that the data dependency is taken care of by the compiler rather than the hardware.
- **This results** in a simpler hardware segment since the segment does not have to check if the content of the register being accessed is currently valid or not.

35. Vector (Array) Processor and its Types

Array processors are also known as multiprocessors or vector processors. They perform computations on large arrays of data. Thus, they are used to improve the performance of the computer.

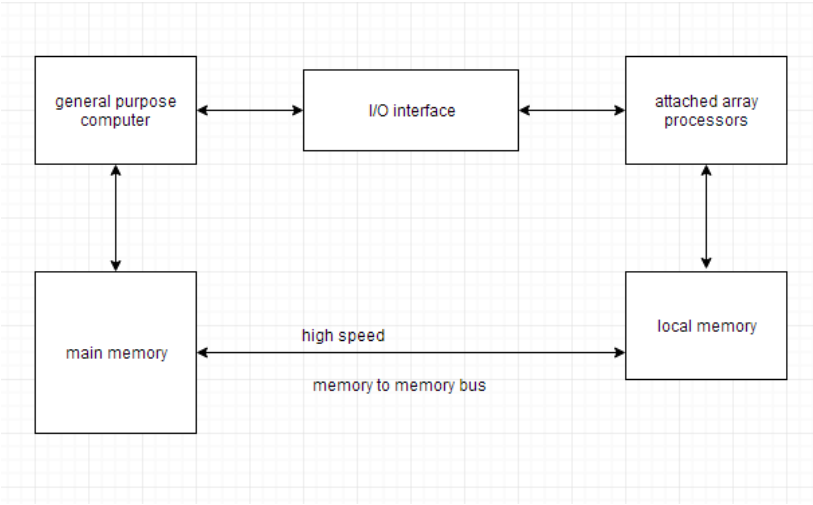
36. Types of Array Processors

There are basically two types of array processors:

1. Attached Array Processors
2. SIMD Array Processors

37. Attached Array Processors

An attached array processor is a processor which is attached to a general purpose computer and its purpose is to enhance and improve the performance of that computer in numerical computational tasks. It achieves high performance by means of parallel processing with multiple functional units.



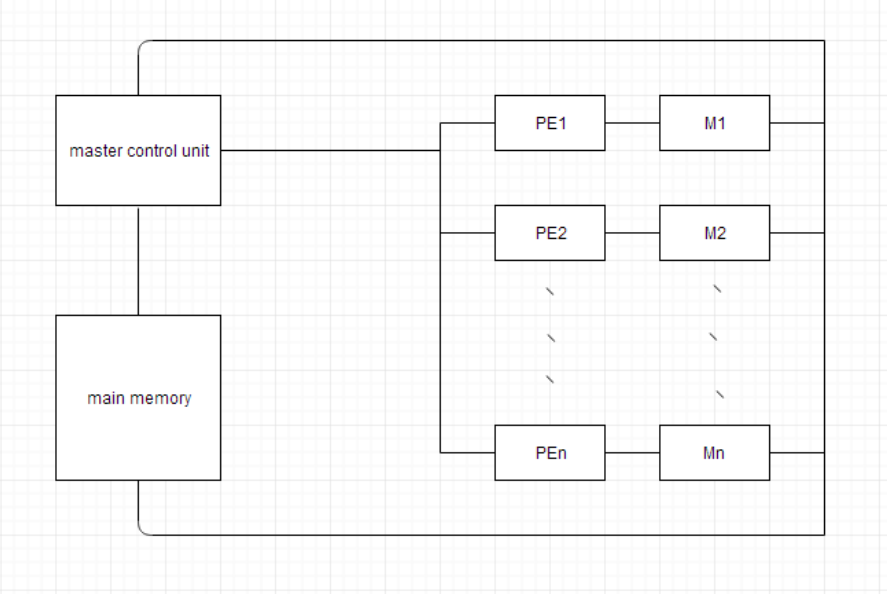
38. SIMD Array Processors

SIMD is the organization of a single computer containing multiple processors operating in parallel. The processing units are made to operate under the control of a common control unit, thus providing a single instruction stream and multiple data streams.

A general block diagram of an array processor is shown below. It contains a set of identical processing elements (PE's), each of which is having a local memory M. Each processor element includes an ALU and registers. The master control unit controls all the operations of the processor elements. It also decodes the instructions and determines how the instruction is to be executed.

The main memory is used for storing the program. The control unit is responsible for fetching the instructions. Vector instructions are send to all PE's simultaneously and results are returned to the memory.

The best known SIMD array processor is the ILLIAC IV computer developed by the Burroughs corps. SIMD processors are highly specialized computers. They are only suitable for numerical problems that can be expressed in vector or matrix form and they are not suitable for other types of computations.



39. Why use the Array Processor

- Array processors increases the overall instruction processing speed.
- As most of the Array processors operates asynchronously from the host CPU, hence it improves the overall capacity of the system.
- Array Processors has its own local memory, hence providing extra memory for systems with low memory.