Three address instruction

- Three address instruction-ADD A,B,C;[A]+[B] → C
- Operation Source1,Source2,Destination.
- No GPR or accumulator needed.
- No of three address instructions=No of arithmetic operations.
- Instruction length is very long.

- Problem:WAP to realize the following expression using 3 address instruction.

  A+B*C+D

- START:MPY B,C,T1; [B]*[C] → T1
- ADD A,T1,T2; [A]+[T1] → T2
- ADD T2,D,T3; [T2]+[D] → T3
- HLT

Two address instruction

- Two operands
- No GPR or accumulator needed
- No of two address instruction=No of arithmetic operations +one more instruction.
- Instruction length smaller.

Eg: MPY X,Y;[X]+[Y] → [Y]

ADD X,Y;[X]+[Y] → [Y]

MOV X,Y;[X] → [Y]

1 A*B+C*D

START:MPY A,B;[A]*[B] → [B]

MPY C,D;[C]*[D] → [D]

ADD B,D;[B]+[D] → [D]

2.A=B*[C+D*E-F/G]

- 2. A=B*[C+D*E-F/G]

DIV F,G

MPY D,E

ADD E,C

SUB C,G

MPY B,G

MOV G,A

HLT

Contd..

- One address instruction-only one operand.

  Opcode Source/Destination
- Accumulator is a must & destination is also the accumulator.
- Instructions are much faster.
- Instruction length is less so execution speed is more.
- Eg: LOAD X;[ACC] ⟵ [X]

  STORE X;[X] ⟵ [ACC]

  ADD X;[ACC] ⟵ [X]+[ACC]

  MPY X;[ACC] ⟵ [ACC]*[X]

- Problem 1 A*B+C*D

Contd..

- Problem 1 A*B+C*D

  LOAD A;ACC ⟵ [A]

  MPY B;[ACC] ⟵ [ACC]*[B]

  STORE X;X ⟵ [ACC]

  LOAD C;[ACC] ⟵ [C]

  MPY D;[ACC] ⟵ [ACC]*[D]

  ADD X;[ACC] ⟵ A*B+C*D

  STORE X;[X] ⟵ [ACC]

  HLT

- Problem 2:A=B*(C+D*E-F/G)

- Problem 2:A=B*(C+D*E-F/G)

```
START:LOAD F;
        DIV G
        STORE X;
        LOAD D
         MPY E
         ADD C
         SUB X
         MPY B
         STORE A
```

Example: Evaluate (A+B) * (C+D)

- Zero-Address

1. PUSH A ; TOS ← A
2. PUSH B ; TOS ← B
3. ADD ; TOS ← (A + B)
4. PUSH C ; TOS ← C
5. PUSH D ; TOS ← D
6. ADD ; TOS ← (C + D)
7. MUL ; TOS ← (C+D)*(A+B)
8. POP X ; M[X] ← TOS

Addressing Modes

- Different ways of specifying address of operands in the instruction.

Variable and Constant representations.

- Register mode-operands are registers holding contents

  Eg:MOVE R1,R2

- Absolute or Direct mode:Adrress of location holding content is specified explicitly

  Eg:MOVE LOC,R1

- Immediate AM-Value is specified explicitly in instruction

  Eg:MOVE #200,R1

Indirection and Pointers.

•Indirect Mode

- The EA of the operand is the contents of a register(or memory-location).
- The register (or memory-location) that contains the address of an operand is called a **Pointer**.
- We denote the indirection by
  - → name of the register or
  - → new address given in the instruction.

  E.g: *Add (R1),R0* ;The operand is in memory. Register R1 gives the effective-address (B) of the operand. The data is read from location B and added to contents of register R0.

- To execute the Add instruction in fig 2.11 (a), the processor uses the value which is in register R1, as the EA of the operand.
- It requests a read operation from the memory to read the contents of location B. The value read is the desired operand, which the processor adds to the contents of register R0.
- Indirect addressing through a memory-location is also possible as shown in fig 2.11(b). In this case, the processor first reads the contents of memory-location A, then requests a second read operation using the value B as an address to obtain the operand.

INDEXING AND ARRAYS

A different kind of flexibility for accessing operands is useful in dealing with lists and arrays.

Index mode

- The operation is indicated as X(Ri)
    - where X=the constant value which defines an offset(also called a displacement).
    - Ri=the name of the index register which contains address of a new location.
- The effective-address of the operand is given by EA=X+[Ri]
- The contents of the index-register are not changed in the process of generating the effective-address.
- The constant X may be given either
    - → as an explicit number or
    - → as a symbolic-name representing a numerical value.

- **Base with Index Mode**

- Another version of the Index mode uses 2 registers which can be denoted as (Ri, Rj)
- Here, a second register may be used to contain the offset X.
- The second register is usually called the *base register*.
- The effective-address of the operand is given by EA=[Ri]+[Rj]
- This form of indexed addressing provides more flexibility in accessing operands because both components of the effective-address can be changed.

- **Base with Index & Offset Mode**

- Another version of the Index mode uses 2 registers plus a constant, which can be denoted as X(Ri, Rj)
- The effective-address of the operand is given by EA=X+[Ri]+[Rj]
- This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri, Rj) part of the addressing-mode. In other words, this mode implements a 3-dimensional array.

.

- **Relative Mode**
- This is similar to index-mode with one difference:
  - The effective-address is determined using the PC in place of the general purpose register Ri.
- The operation is indicated as X(PC).
- X(PC) denotes an effective-address of the operand which is X locations above or below the current contents of PC.
- Since the addressed-location is identified "relative" to the PC, the name Relative mode is associated with this type of addressing.
- This mode is used commonly in conditional branch instructions.
- An instruction such as
  - *Branch > 0 LOOP* ;Causes program execution to go to the branch target location identified by name LOOP if branch condition is satisfied.

ADDITIONAL ADDRESSING MODES

1) Auto Increment Mode

- Effective-address of operand is contents of a register specified in the instruction.
- After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.
- Implicitly, the increment amount is 1.
- This mode is denoted as
  - (Ri)+ ;where Ri=pointer-register.

1) Auto Decrement Mode

- The contents of a register specified in the instruction are first automatically decremented and are then used as the effective-address of the operand.
- This mode is denoted as
  - -(Ri) ;where Ri=pointer-register.
- These 2 modes can be used together to implement an important data structure called a stack.

Basic Input/Output Operations

I/O

- The data on which the instructions operate are not necessarily already stored in memory.
- Data need to be transferred between processor and outside world (disk, keyboard, etc.)
- I/O operations are essential, the way they are performed can have a significant effect on the performance of the computer.

Program-Controlled I/O Example

- Read in character input from a keyboard and produce character output on a display screen.

- Rate of data transfer (keyboard, display, processor)
- Difference in speed between processor and I/O device creates the need for mechanisms to synchronize the transfer of data.
- A solution: on output, the processor sends the first character and then waits for a signal from the display that the character has been received. It then sends the second character. Input is sent from the keyboard in a similar way.

Program-Controlled I/O Example

- Registers
- Flags
- Device interface

Program-Controlled I/O Example

- Machine instructions that can check the state of the status flags and transfer data:

```
READWAIT  Branch to READWAIT if SIN = 0
          Input from DATAIN to R1

WRITEWAIT Branch to WRITEWAIT if SOUT = 0
          Output from R1 to DATAOUT
```

Queues

- https://www.youtube.com/watch?v=UpvDOm3prfI

QUEUE

- Data are stored in and retrieved from a queue on a FIFO basis.
- Difference between stack and queue?

1) One end of the stack is fixed while the other end rises and falls as data are pushed and popped.

2) In stack, a single pointer is needed to keep track of top of the stack at any given time.

- In queue, two pointers are needed to keep track of both the front and end for removal and insertion respectively.

3) Without further control, a queue would continuously move through the memory of a computer in the direction of higher addresses. One way to limit the queue to a fixed region in memory is to use a circular buffer.

Subroutines

- **SUBROUTINES**
- A subtask consisting of a set of instructions which is executed many times is called a **Subroutine**.
- A Call instruction causes a branch to the subroutine (Figure: 2.16).
- At the end of the subroutine, a return instruction is executed
- Program resumes execution at the instruction immediately following the subroutine call
- The way in which a computer makes it possible to call and return from subroutines is referred to as its **Subroutine Linkage** method.
- The simplest subroutine linkage method is to save the return-address in a specific location, which may be a register dedicated to this function. Such a register is called the **Link Register**.
- When the subroutine completes its task, the Return instruction returns to the calling-program by branching indirectly through the link-register.
- The **Call Instruction** is a special branch instruction that performs the following operations:
- → Store the contents of PC into link-register.
- → Branch to the target-address specified by the instruction.
- The **Return Instruction** is a special branch instruction that performs the operation:
- → Branch to the address contained in the link-register.
-

- **SUBROUTINE NESTING AND THE PROCESSOR STACK**
- **Subroutine Nesting** means one subroutine calls another subroutine.
- In this case, the return-address of the second call is also stored in the link-register, destroying its previous contents.
- Hence, it is essential to save the contents of the link-register in some other location before calling another subroutine. Otherwise, the return-address of the first subroutine will be lost.
- Subroutine nesting can be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it.
- The return-address needed for this first return is the last one generated in the nested call sequence. That is, return-addresses are generated and used in a LIFO order.
- This suggests that the return-addresses associated with subroutine calls should be pushed onto a stack. A particular register is designated as the SP(Stack Pointer) to be used in this operation.
- SP is used to point to the processor-stack.
- Call instruction pushes the contents of the PC onto the processor-stack.
- Return instruction pops the return-address from the processor-stack into the PC.

Additional instructions

- **LOGIC INSTRUCTIONS**
- Logic operations such as AND, OR, and NOT applied to individual bits.
- These are the basic building blocks of digital-circuits.
- This is also useful to be able to perform logic operations is software, which is done using instructions that apply these operations to all bits of a word or byte independently and in parallel.

- **SHIFT AND ROTATE INSTRUCTIONS**
- There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions.
- The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information.
- For general operands, we use a logical shift.
- For a number, we use an arithmetic shift, which preserves the sign of the number.
- **LOGICAL SHIFTS**
- Two logical shift instructions are

  Shifting left (LShiftL) &

  Shifting right (LShiftR).
- These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction

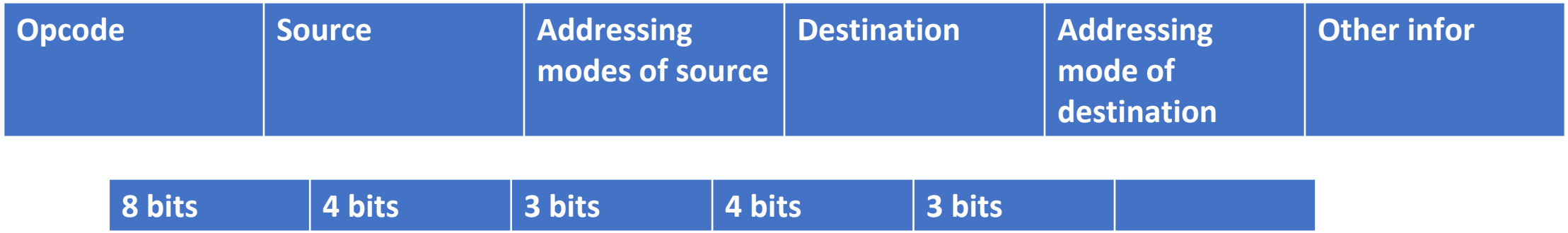
- **ROTATE OPERATIONS**
- In shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry-flag C.
- To preserve all bits, a set of rotate instructions can be used.
- They move the bits that are shifted out of one end of the operand back into the other end.
- Two versions of both the left and right rotate instructions are usually provided. In one version, the bits of the operand is simply rotated.
- In the other version, the rotation includes the C flag.
-

- 0 01110…….011
- RotateL #2,R0
- 0 1110……0110
- 1 110……01101

ENCODING OF MACHINE INSTRUCTIONS

- To be executed in a processor, an instruction must be encoded in a binary-pattern. Such encoded instructions are referred to as **Machine Instructions**.
- The instructions that use symbolic-names and acronyms are called *assembly language instructions*.
- We have seen instructions that perform operations such as add, subtract, move, shift, rotate, and branch. These instructions may use operands of different sizes, such as 32-bit and 8-bit numbers.
- Let us examine some typical cases. The instruction
- *Add R1, R2* ;Has to specify the registers R1 and R2, in addition to the OP code. If the processor has 16 registers, then four bits are needed to identify each register. Additional bits are needed to indicate that the Register addressing-mode is used for each operand.
- The instruction
- *Move 24(R0), R5* ;Requires 16 bits to denote the OP code and the two registers, and some bits to express that the source operand uses the Index addressing mode and that the index value is 24.

- In all these examples, the instructions can be encoded in a 32-bit word (Fig 2.39).
- The OP code for given instruction refers to type of operation that is to be performed.
- Source and destination field refers to source and destination operand respectively.
- The "Other info" field allows us to specify the additional information that may be needed such as an index value or an immediate operand.
- Using multiple words, we can implement complex instructions, closely resembling operations in high- level programming languages. The term complex instruction set computers (CISC) refers to processors that use
- CISC approach results in instructions of variable length, dependent on the number of operands and the type of addressing modes used.
- In RISC (reduced instruction set computers), any instruction occupies only one word.
- The RISC approach introduced other restrictions such as that all manipulation of data must be done on operands that are already in registers.
- *Ex: Add R1,R2,R3*
- In RISC type machine, the memory references are limited to only Load/Store operations.
-


Opcode
Source
Addressing modes of source
Destination
Addressing mode of destination
Other infor
8 bits
4 bits
3 bits
4 bits
3 bits

ADD R1,R2