CSA March 23rd to 28th

Practical 5:

Write a program to implement the following binary operations:

a. Addition

b. Subtraction using 2’s complement

```
import sys

def binAdd(s1, s2):

    if not s1 or not s2:

        return ''

    maxlen = max(len(s1), len(s2))

    s1 = s1.zfill(maxlen)

    s2 = s2.zfill(maxlen)

    result  = ''

    carry   = 0

    i = maxlen - 1

    while(i >= 0):

        s = int(s1[i]) + int(s2[i])

        if s == 2: #1+1

            if carry == 0:

                carry = 1

                result = "%s%s" % (result, '0')

            else:

                result = "%s%s" % (result, '1')
```

```
    elif s == 1: # 1+0
      if carry == 1:
        result = "%s%s" % (result, '0')
      else:
        result = "%s%s" % (result, '1')
    else: # 0+0
      if carry == 1:
        result = "%s%s" % (result, '1')
        carry = 0
      else:
        result = "%s%s" % (result, '0')
    i = i - 1;
  if carry>0:
    result = "%s%s" % (result, '1')
  return result[::-1]
def Complement(binarySequence):
  convertedSequence = [0] * len(binarySequence)
  carryBit = 1
  for i in range(0, len(binarySequence)):
    if binarySequence[i] == '0':
      convertedSequence[i] = 1
    else:
      convertedSequence[i] = 0
  if convertedSequence[-1] == 0:
      convertedSequence[-1] = 1
```

```
            return(''.join(str(x) for x in convertedSequence))
    else:
        for bit in range(0, len(binarySequence)):
            if carryBit == 0:
                break
            index = len(binarySequence) - bit - 1
            if convertedSequence[index] == 1:
                convertedSequence[index] = 0
                carryBit = 1
            else:
                convertedSequence[index] = 1
                carryBit = 0
        return(''.join(str(x) for x in convertedSequence))
while True:
    s1=input("Enter first binary number : ")
    s2=input("Enter second binary number : ")
    s3=binAdd(s1,s2)
    print(s3)
    binarySequence=s2
    s2=Complement(binarySequence)
    s4=binAdd(s1,s2)
    if len(s4)>len(s2):
        s4=s4[1:]
    if s4[0]=='0':
        print(s4)
```

```
  else:
    print("-" + Complement(s4))
  while True:
    ans=input("Would you like to continue : (Y/N) ")
    if ans=="Y":
      break
    elif ans=="N":
      sys.exit()
    else:
      print("\nNot Valid Choice Try again !!!")
```

CSA 5.6

https://www.youtube.com/watch?v=pqg8rgYC9Lw

CSA 5.7

https://www.youtube.com/watch?v=k2Z7EsUe7o0

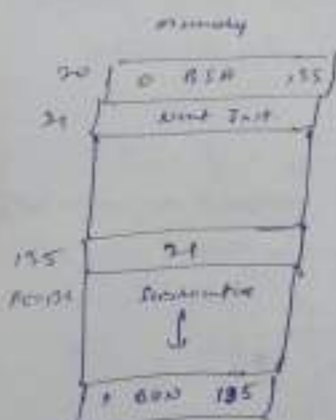
CSA 5.8

https://www.youtube.com/watch?v=k2Z7EsUe7o0&t=75s

https://www.youtube.com/watch?v=aZL4DsnLg_E

BSA perform

$M[135] \leftarrow 21$, $PC \leftarrow 135+1 = 136$



(b) memory and PC after execution

When BUN executed, effective address 21 is transferred to PC

BSA instruction must be executed in seq^n of two microoperations

$D_5 T_4$: $M[AR] \leftarrow PC$, $AR \leftarrow AR+1$

$D_5 T_5$: $PC \leftarrow AR$, $SC \leftarrow 0$

ISZ : Increment and skip if zero

Increment the word specified by effective address, and if the incremented value is 0 then $PC = PC+1$.

This prog^n usually stores the negative no and prog^n repeatedly inc. the value and reaches to zero

Since it is not possible to increment a word inside the memory, it is necessary to read the word in DR, increment DR and then store word back to memory.

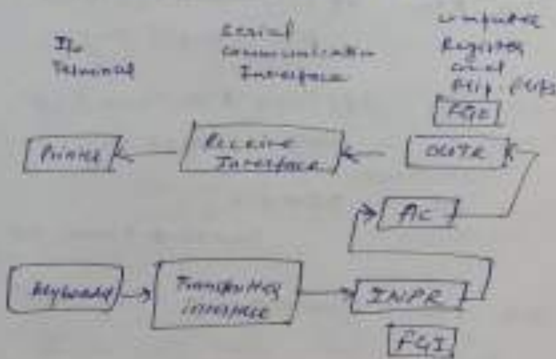
$D_6 T_4$: $DR \leftarrow M[AR]$

$D_6 T_5$: $DR \leftarrow DR+1$

$D_6 T_6$: $M[AR] \leftarrow DR$, if $(DR=0)$ then $(PC \leftarrow PC+1)$, $SC \leftarrow 0$

Control Flow chart [Refer in Book]

5-7 Input-Output Interrupt



INPR consists of 8 bits

FGI " " 1 bit flag

1 = 1 when info is available at input device

Input-Output Instruction

Op-code - 1111

When $D_7 = 1$ and $I = 1$

$D_7 I T_3$ will be denoted as p

B_i ; i = IR (6-11)

P: SC ← 0

INP pB11 : AC(0-7) ← INPR, FGI ← 0 Input char

OUT pB10 : OUTR ← AC(0-7), FGO ← 0 output character

SKI pB9 : If (FGI = 1) then (PC ← PC+1) skip on Input flag

SKO pB8 : If (FGO = 1) then (PC ← PC+1) skip on output flag

ION pB7 : IEN ← 1 Interrupt enable on

IOF pB6 : IEN ← 0 " " off

Program Interrupt

Computer has to check flag for transfer is called programmed control transfer

There is a huge diff. b/w instruction cycle rate and I/O transfer rate so computer is wasting time in programmed control transfer

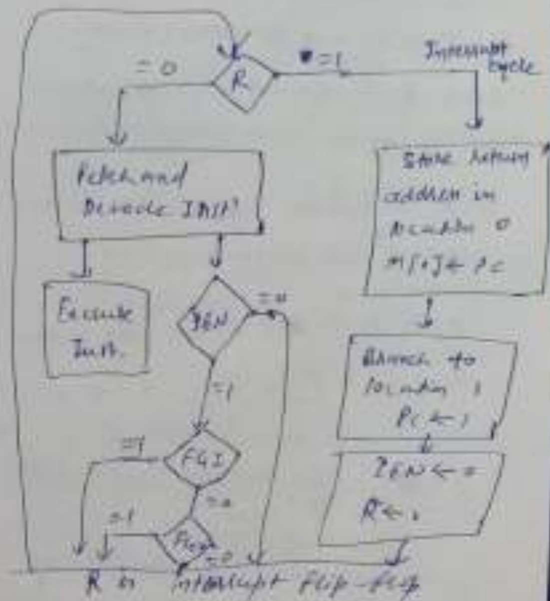
An alternative is to let the external device inform the computer when it is ready to transfer. This is called interrupt

IEN ← 0 [Flag can't interrupt the computer]

and vice-versa.

Fig - 5.13

It is a BAS



R is interrupt flip-flop

Interrupt cycle

$T_0' T_1' T_2' (IEN)(FGI + FGO)$:

$R \leftarrow 1$

Fetch and decode phase will be recognized from three control functions $R'T_0$, $R'T_1$, $R'T_2$

Interrupt cycle

RT_0 : $AR \leftarrow 0$, $TR \leftarrow PC$

RT_1 : $M[AR] \leftarrow TR$, $PC \leftarrow 0$

RT_2 : $PC \leftarrow PC+1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$

TR [Temp Register]

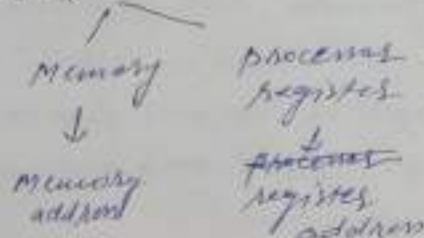
8.4 Instruction formats

Mode	Opcode	Address

other fields in some special instruction.

i.e. Shift type instruction which also define no. of Shifts.

In this section we are concerned with the address field of an instruction format and consider the effect of including multiple address fields in an instruction.



A Register address is a binary no. of k-bits that defines one of 2^k registers in the CPU.

Thus a CPU with 16 processor registers R0 to R15 have a address field of 4 bits.

R5 — 0101

Computers may have inst of several diff lengths containing varying no. of addresses. The no. of address fields in inst. format of a computer depends upon internal organisation of its registers. Most computers fall into one of the three types of CPU organisations.

1. Single Accumulator organization
2. General Register Organisation
3. Stack Organisation

(1) In an accumulator-type organisation, all operations are performed with an implied accumulator register. The inst format in this type of computer uses one address field

ADD X

X : Address of operand

Result :

$AC \leftarrow AC + M[X]$

(2) Computer needs three register address fields.

ADD R1, R2, R3

$R1 \leftarrow R2 + R3$

OR

ADD R1, R2

$R1 \leftarrow R1 + R2$

MOV R1, R2

$R1 \leftarrow R2$ or $R2 \leftarrow R1$

ADD R1, X

$R1 \leftarrow R1 + M[X]$

(3) Stack organisation require PUSH and POP instructions.

PUSH X

Push the word at address X to the top of the stack. The stack pointer is updated automatically.

Operation type instruction do not need address field in stack-organised computer.

ADD

pop two topmost elements of stack and do ADD.

most computer fall into one of three types. But some may be combine the features i.e Intel 8080 microprocessor has 7 CPU registers, one of which is accumulator register.

Symbols used in Instructions

ADD, SUB, MUL, DIV, MOV, LOAD, STORE

$X = (A+B) * (C+D)$

We assume operands are in memory addresses A, B, C, D and result must be stored in memory address of X.

Three-Address Instruction

$X = (A+B) * (C+D)$

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

Advantage :- Short prog. for evaluating an instruction

Disadvantage :- binary-coded inst require too many bits to specify three addresses.

Two address Instructions

$X = (A+B) * (C+D)$

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]
MUL	R1, R2	R1 ← R1 * R2
MOV	X, R1	M[X] ← R1

One-address Instructions

$X = (A+B) * (C+D)$

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

Zero Address Instructions

Stack does not need operands for ADD and MUL. However PUSH and POP needs operand address that communicate with the stack.

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

RISC - Reduced Instruction Set Computer.

Process is restricted to use of load and store instructions when communicating b/w CPU and memory. All other instructions are executed within the registers of the CPU without referring to the memory.

LOAD	R1, A	$R1 \leftarrow M[A]$
LOAD	R2, B	$R2 \leftarrow M[B]$
LOAD	R3, C	$R3 \leftarrow M[C]$
LOAD	R4, D	$R4 \leftarrow M[D]$
ADD	R1, R1, R2	$R1 \leftarrow R1+R2$
ADD	R3, R3, R4	$R3 \leftarrow R3+R4$
MUL	R1, R1, R3	$R1 \leftarrow R1 \times R3$
STORE	X, R1	$M[X] \leftarrow R1$

8-5 Addressing modes

The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

Benefits of Addressing mode

1. To provide programming versatility to the user by providing facilities such as - pointer to memory
   - counters for loop control
   - indexing of data
   - Program relocation
2. To reduce the no of bits in the addressing field of instruction.

Two modes have no address fields
① Implied mode
② Immediate "

① Implied mode : operands are specified implicitly in the defination of the instruction.
i.e. ① complement accumulator
② zero-address inst. in stack-organised.
③ All register reference inst. that uses an accumulator

② Immediate mode :- operand is specified in the instruction itself. In other words inst. has an operand field rather than an address field.

Ex :- Initialisation of register to a constant value

Register mode :- operands are in register that reside within CPU. Addresses are in processor registers

B

Register Indirect Mode :-

Inst specify a register whose contents give the address of the operand in memory

Effective Address :- is the memory address obtained from the computation dictated by the given addressing mode.

- Direct Addressing mode
- Indirect " "

Effective address may be

Effective address = address part of instruction + content of CPU register

Relative Address mode

= address part of inst + content of program counter

= ~~24 + 825~~

Fetch at read location 825 and PC = PC+1 = 826

relative address = 24 + 826 = 850

Relative address used in branch-type instructions.

Indexed Address Mode

address part of inst. + content of index register

Index register is a special CPU register that contains an index value.

Base Register Mode.

address part of inst. + content of base register

Used for relocation of prog in memory

Address	Memory
200	Load to AC / Mode
201	Address = 500
	Next inst.
399	450
400	700
500	800
600	900
702	325
800	300

PC = 200	Fetch
R1 = 400	Processor Register
XR = 100	Index Register
AC	

(1) Direct Address Mode

effective address = 500

and the content is 800 loaded into AC

(2) Immediate Mode :

Inst. is taken as operand rather than address

AC = 500

(3) Indirect mode : effective address stored in memory at address 500

effective address = 800

AC = 300

(4) Relative mode

Effective address = 505 + 202 = 702

AC = 325

(5) Index mode : XR + 500 = 600

AC = 900

(6) Register mode

effective address equal to content of R1

AC = 700

(7) Auto increment mode is same as Register indirect mode except that R1 is incremented to 401 after the execution of the inst. AC = 700 x++

(8) Auto Decrement :- same as above but decrement to 399 before execution [--x]

AC = 450

8.8 - CISC characteristics

An important aspects of computer architecture is the design of the instruction set for the processor.

Earlier computer had small and simple inst. sets, forced mainly be need to minimize the hardware used to implement them.

As digital hardware become cheaper with the advent of IC, computer inst. tends to inc. both in nos. and complexity