classmate

Date _____

Page _____

16/7

- Instruction Set Architecture
  - Connection between hardware and software
    - hardware → Processor, memory, I/O
    - software → OS, assembler

17/7

- Both program and data are stored in memory.
- Aim of designer :- To reduce 'time per program'
- Performance $\propto \frac{1}{\text{Time}/\text{Program}}$.
- $\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \underbrace{\frac{\text{Time}}{\text{Instruction}}}_{\text{may be assumed to be constant}}$
- The no. of instructions will depend on which operations are supported by Instruction Set Architecture $(+, \times, -, \div)$
- Compiler converts program script into 0s and 1s.
  No. of instructions also depends on compiler used.
- In practice, Time/Instruction is different for different operations of ISA.
- $\frac{\text{Time}}{\text{Instruction}} = \frac{\text{Cycles}}{\text{Instruction}} \times \underbrace{\frac{\text{Time}}{\text{Cycle}}}_{\text{constant}}$.
- Instructions range in complexity from addition and logic gates to computation of Fourier Transform.

Time/Cycle will depend on digital design of the hardware.

INSTRUCTION SET ARCHITECTURE

Arithmetic and Logical operation

Any 'logical operation' can be performed by AND and NOT or OR and NOT

Any primitive 'arithmetic operation' (on integers) can be performed through addition only.

Memory Communication (Read/Write)

Branches - Decision making

Loops, conditional or unconditional

Assemby language does not require a compiler.

If memory is 4 GB (2^{32} bytes), there are 2^{32} memory locations available. Hence, we require 32 bits to point to one location (1 byte). eg

Instruction contains 1 operation, maximum 2 operands, 1 destination

3 address format :-

Operation	Operand 1	Operand 2	Destination
7 bits	32	32	32

(for 100 operations, say) eg

'Direct addressing mode'

i+j	Infix notation
+ij	Prefix
ij+	Postfix

- In earlier days, memory was slow. Accessing an address took ~1000 cycles. Even though processor is fast, efficiency is hindered by memory.

- More efficient ways of accessing memory, and ease of programming, are affected by addressing modes used.

→ To reduce demand for memory size,
- Reduce program size
  - Reduce no. of bits per instruction
- Cannot reduce no. of bits for operator.
- For statements like i = i + j; eliminate the need for 32 bits of destination.
- Define operands 'implicitly' - it is understood where they are located.
  - Store operands on LIFO stack. Every instruction is performed only on elements at the top. Stack is just a designated part of memory
  - Now, instruction requires only 7 bits for operator :)
  - Two new instructions pop and push required to transfer from memory to stack and vice versa.

eg

k = i + j	1) Push i & j to stack
p = k + r	2) Calculate k, store in stack
l = p + s	3) Push r,
	4) Calculate p, store in stack
	5) Push s
	6) Calculate l, store in stack

eg a = b + c + d × e = bcde*++

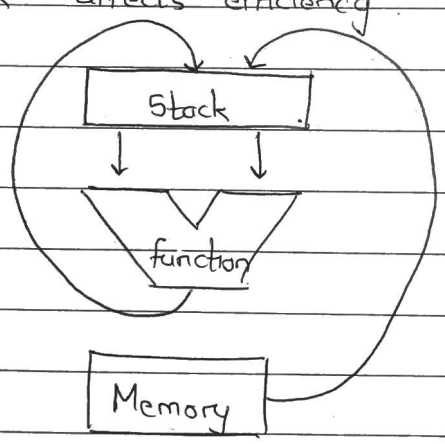
push b, push c, push d, push e, mult, add, add, pop a

eg Say, stack has only 2 addresses

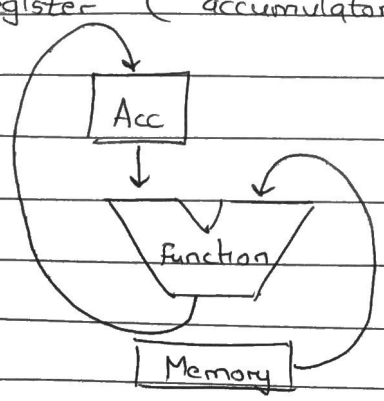
$a = b + c$	$a = b + c$
$d = a + e$	$d = f + e$
	$p = a + l$
push b, push c, add, push e, add, pop d	push b, push c, add, <u>pop a</u>, push f, push e, add, <u>pop d</u>, <u>push a</u>, push l, add, pop p

☆ - Operands deleted after one use?

- Depth of stack affects efficiency



→ Use a single register ("accumulator") instead of stack



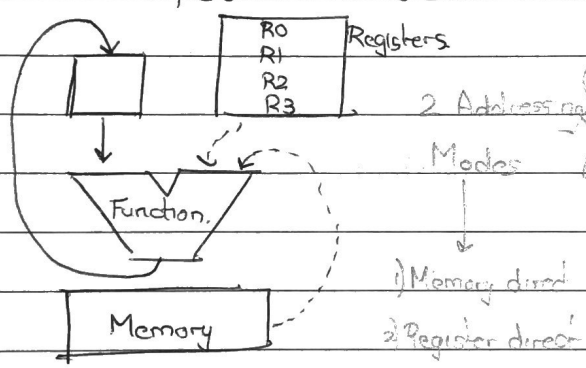
One operand is explicit, other is implicit

'1 operand format'

		k is in accumulator
eg	$i = j + k$	push j, add and store in acc, push r, add and
	$l = i + r$	store in acc, push t, add and store in acc,
	$p = l + t$	pop p

- For smaller microprocessors, accumulators are preferred because a register is easier to implement than a stack.

→ Improvement :- Operand 1 from accumulator.
Operand 2 ~~fro~~ either from other registers or from memory



- One bit to specify if operand 2 is from registers or memory
- If it is from registers, only 2 bits required for operand 2 :)

- 3rd addressing mode :- (will now require 2 bits to specify)
'Base + displacement'

Operation	1 1	0 0	Displacement

Addressing Mode (B + D)

Register which contains value of base address (say 100)

If R0 contains 100, and displacement is 20, then required address of operand 2 is 120.

Every address points to one byte.

classmate
Date
Page

19/7

- If you want to run a loop on arrays,

```
for (int i = 0 ; i < length(arr1) ; i++) {
        arr[i] = arr1[i] + arr2[i];
}
```

We can use Base + Displacement mode, where Base is the first element of array.

- 3 modes :- ~~1) M~~

00 1) Memory direct :- Get operand from memory.

01 2) Register direct :- " " " register

11 3) Register indirect :- " address of operand from register, then get operand from memory.
~~(B + D)~~

- ~~B + D~~ ~~requires more~~

→ Operating on constants instead of variables

eg. i++; i < 256 ; i == 0 ;

- Storing constant value in 32 bits as variable is wastage of memory
  - For constant values that take up 8 bits or 16 bits can be directly supplied within the instruction.
  - 4th addressing mode = 'Immediate'

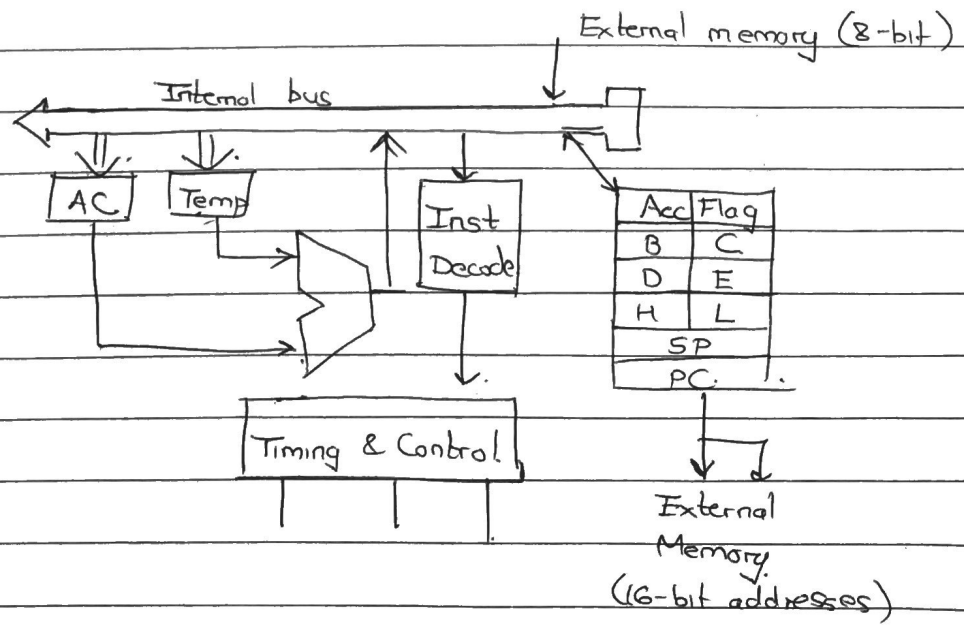
Operand	Mode	Const

→ An 8-bit processor, for example, can only operate on 8-bit numbers at a time.
If you want to operate on numbers with higher bits, it will have to be handled in software program.

PROCESSOR 8085

- 8-bit processor
- Accumulator, having 7 other registers called B, C, D, E, H, L.
- 64 kB memory ----- every address is 16 bits.
- Program Counter, which is one extra register, which acts a pointer to the next instruction to be executed.
  The program counter should be able to point to any address (where the instruction is stored). Hence, unlike other 8-bit registers, this has 16 bits.
- Stack Register - 16-bit for stack operation.
- ~~Accumulator~~
- Status Register - Tells status of execution at any instant.

24/7.

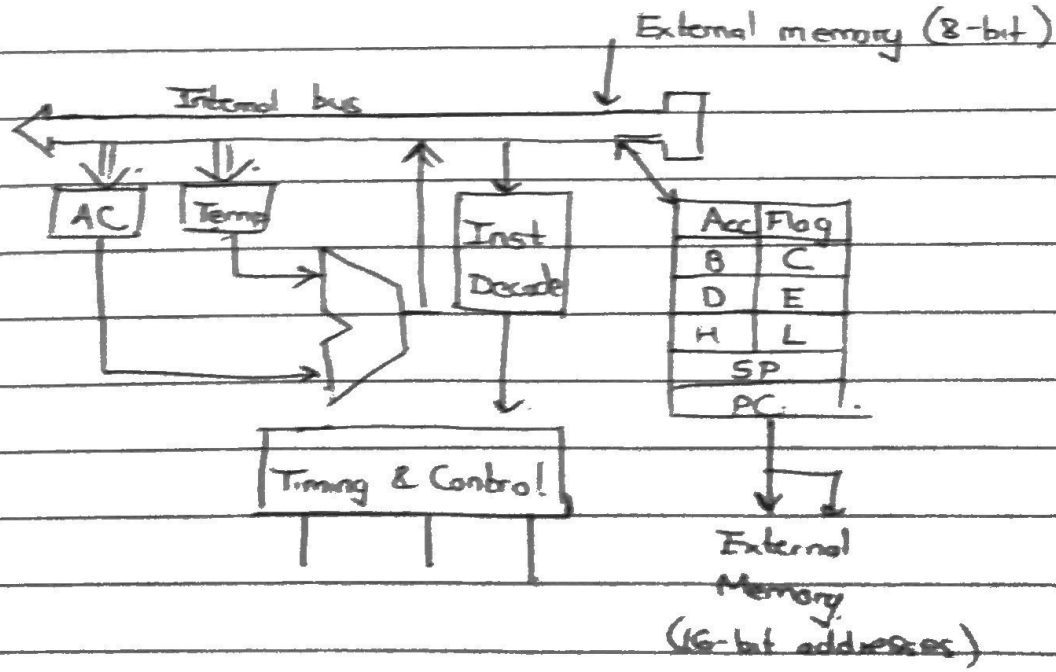


→ An 8-bit processor, for example, can only operate on 8-
numbers at a time
If you want to operate on numbers with higher bits, i
have to be handled in software program.

PROCESSOR 8085

- 8-bit processor
- Accumulator, having 7 other registers called B, C, D, E, H, L.
- 64 kB memory ..... every address is 16 bits
- Program Counter, which is one extra register, which acts a point
next instruction to be executed.
The program counter should be able to point to any address
the instruction is stored). Hence, unlike other 8-bit registers,
has 16 bits.
- Stack Register - 16-bit for stack operation.
- ~~Accumulator~~
- Status Register - Tells status of execution at any instant.

24/7



5 Flags

Carry	Parity	Aux carry	Zero	Sign
If carry is generated	odd/even	If there is carry from LS 4 bits to MS 4 bits	If result is zero	MSB shows sign of result

- Supports variable length instructions (1 byte, ~~to~~ 2 bytes or 3 bytes)
- Supports 5 addressing modes
  :- Memory direct, Register direct, Register Indirect, Immediate, Input & Output

- Memory direct :- Instruction must be 3 bytes (because 2 bytes (16 bits) are used to access memory anyway)
- Register direct :- Instruction can be 1 bytes, because there is just one explicit operand. (one comes from accumulator)
- Register indirect :- To retrieve 16 bit address of operand, we have to use a pair of registers, instead of just one
  ↳ BC, DE or HL
  ↳ default (implicit)
- Immediate - Can be 1 byte
  - Can be 2 bytes minimum
- I/O - For communicating with external world.

• For 8085, operation and addressing mode bits were combined. The total bits used for this is variable.

• Internal bus :- Transfer stored data within 8085.
  Width of bus is 8-bit (transfer 8-bit at a time)

A] Arithmetic and Logical Instructions

Add, subtract, AND, OR, XOR, compare, complement, increment, decrement
+1 -1

~~No. of bits for operation is variable~~.

Register encoded in 3-bits

INR reg/M :- reg/M += 1 | CMP reg/M :- Compare Acc & reg/M

DCR r/M :- reg/M -= 1 | CPI n :- " " n

INX r_p } Increment/decrement register pair
DCX r_p

1 byte | ADD register :- Add value in acc to value in reg, store in acc

2 byte | ADI data :- Immediate addition

1 byte | ADD M :- Register Indirect (HL by default)

ADC reg :- Add with carry (for > 8 bits)

Acc = Acc + (value in reg) + carry

↳ stored in carry flag 0 or 1

ACI data :- Add immediate with carry

ADC M. :- Add with carry (register indirect, HL pair)

⋮

2 SUB reg :- Acc = Acc − Value in register

SUB M :- Register indirect

SUI data :- Immediate

SBB reg/M :- Subtraction with borrow [Acc = Acc − reg − carry]

SBB B :- ?

3 ANA reg/M

ANI data.

4. ORA reg/M.

ORI data.

5 XRA reg/M

XRI data.

6 RAL :- Rotate left D7 D6 D5 D4 D3 D2 D1 D0

RLC Rotate left with carry

C D7 D6 D5 D4 D3 D2 D1

RAR

RRC

LHLD address } Load/Unload ~~to~~ to HL pair from addresses
SHDL address } and address + 1

LDAX r_p } Load/Unload ACC from register pair.
STAX r_p

25/7 XCHG } DE ↔ HL

PUSH
POP

- The pair of registers Accumulator + Flags = 'Program Status Word' (PSW) →

  - Tell current status of execution.

- All types of instructions need not utilize all the flags.

eg To check if all bits are 0,
Meth 1 :- Add with zero, check zero flag
Meth 2 :- Bitwise OR, check zero flag.

→ ALU Operations continued.

7 CMA :- Complement of value in Acc

8 NOP :- Chill macro

* Rotate L or R can have uses like
1. Checking MSB by rotating left and ~~carrying~~ check carry flag
2. Shift and add.

→ Operations for loading and unloading accumulator from memory

Load LDA 16-bit_address (source) } 3-byte
Unload STA 16-bit_address (destination)

LXI r_p 16-bit_address :- Load from address, address + 1 to register pair

1 byte
- MOV register_i register_j :- Copy data from j to i
- MOV register_i M. :- Copy data from memory whose address is stored by HL by default
- MOV M register_i :- r_i → memory of HL

2 bytes MVI r_i/M data :- Move immediate

PUSH reg
POP reg

→ Running different functional blocks

- CALL 16-bit_address :- Call a function (address where function is stored)

- For nested functions, recursions, etc, to shift execution to new address and return back in LIFO manner to main function.
The addresses of nested functional blocks are stored in memory in a stack. There is a pointer that points to current address, which is stored in 'Stack register'.
When some function is called, pointer += 2 bytes
is returned -=

- RET :- Return from a function.

- CC 16-bit address :- Conditional Call :- Call the function if carry was 1

- CZ 16-bit_address :- Call function if zero was 1

- CP 16-bit_address :- Call function if parity was 1

- RC, RZ, RP } 1 byte } Conditional Return :- Return if carry/parity/zero was 1

- JMP 16-bit_address :- Go to the destination with no intention of returning

- JC 16-bit, JZ 16-bit, JP 16-bit } Conditional Jump

classmate
Date
Page

A2

* Memory has three partitions
  - Code section, Data section, Stack section.
  - Stack section is where local variables of functions are stored.

* Program Counter stores location of next instruction to be executed.

Q1 Number n_1 is stored in location 2050, 2051 — n_1 & n_2 are
n_2 2052, 2053 — 16-bit
Add them and store in location 2054, 2055

Q2 10 8-bit numbers are stored in locations from 2050 to 2059
Add & store 16-bit answer in 2060, 2061.

* Microprocessor + Some memory + I/O ports = 'Microcontroller'
  - The instruction set of μC is made friendly for bit-operations & byte-operation.

A1

```
LDA  2051      // Load ½ of n1 to acc
  ?            // Load 2053 to HL
ADED M         // Add
STA  2055      // Unload
LDA  2050
  ?            // Load 2052 to HL
ADC  M
STA  2054
```

A2. LDA 2050 // Load n_1 to acc

? // Load n_2 to HL

ADD M

CC address

→ Address Bus and Data Bus

- 8 signals lines A8 - A15 of address bus are unidirectional
- Other 8 lines are multiplexed (time-shared) between address and data. These are bidirectional [AD0 - AD7]

- AD0 - AD7 act as address lines when Address Latch Enable (ALE) signal is high and data lines otherwise

* Instruction Size

1 byte :- Both operator and operand

2 bytes :- Operator :- 1st byte, Operand :- 2nd byte

3 bytes 2nd, 3rd byte

26/7.

* Auxilliary Carry Flag.

If data is $D_7 \cdots D_0$, AC flag is 1 if D_3 sends a carry to D_4

eg BCD addition of digits

eg - 99

1001 1001.

1 Form 8-bit numbers from the digits

i) 99 AND 0F = 09

ii) 99 AND F0 = 90 $\xrightarrow{\text{Right shift}}$ 09

2 Add 09 + 09 $\longrightarrow$ 08 (12 in hex stored in address)

Aux Carry = 1

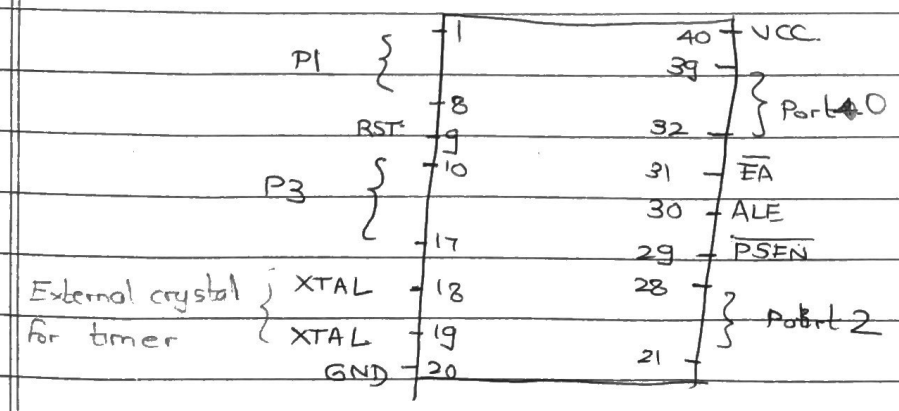
26/7.

8051 MICROCONTROLLER

Microprosessor + Memory + I/O Interface + Timer system .... = Microcontroller
+ Interrupts
↳ like Alt+F4
Changing execution midway

→ 8051 :- (8-bit processor)

- Different memory for program, different memory for data.
  - No secondary memory.
    - Memory must be non-volatile for program. Memory is erased only when overwritten (not erased when powered off)
    - Memory for data can be volatile.
- 4kB non-volatile instruction (program) memory (expandable to 64kB)
  128 B volatile data memory (expandable to 64kB)
- 2 timers
- 5 interrupts.
- Unlike 8085, supports operations on individual bits along with bytes.
  - For bit operations, uses carry bit as one operand.
- 64kB :- Maximum accessible (external) memory
- Address and data pins are multiplexed (like 8085)
  - ALE pin is present.
- 4 ports (8 bit) :- P0 – P3
- 40 pins



Port 3 has 3 functionalities :- Port, Address, Data

Ports 2 and 4 have 2 functionalities

- All ports can be used for input or output. They can even be partitioned into an I/P port and an O/P port.

- Port 3 is also used as following (multiplexed)

Pin	10	11	12				17.
	RxD	TxD	$\overline{INT0}$	$\overline{INT1}$	T0	T1	$\overline{WR}$	$\overline{RD}$

- Port 4 is multiplexed to be used as address/data paths (AD0 to AD7)

- Port 2 " " " " " " address paths (A8 to A15)

→ Memory.

- 4kB internal Read only memory (0000 to 0FFF in hex - 12 bits)
- External ~~60~~ 60kB ROM (1000 to FFFF)

- $\overline{PSEN}$:- ~~When 0, read from external memory~~ For reading from external code memory every time (output signal that goes to memory)
  Program Status Enable
- $\overline{EA}$:- External enable input.
  When $\overline{EA} = 0$, ~~#~~ only external ~~memory~~ PCP is used. (64 kB)
  When $\overline{EA} = 1$, 4 kB internal, 60 kB external.
  $\overline{EA}$ is external, cannot be set through instructions

- $\overline{WR}$, $\overline{RD}$
  Write or read data memory. (upto 64 kB) (RAM)

27/7

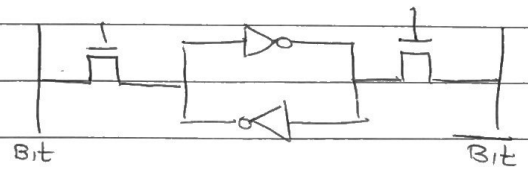
- Code memory, once burnt, can only be read from (ROM) during execution.

- SFR can be only accessed directly only
- 8052 has 256 byte RAM, so 80 to FF of RAM is addressed indirectly, SFR is addressed directly

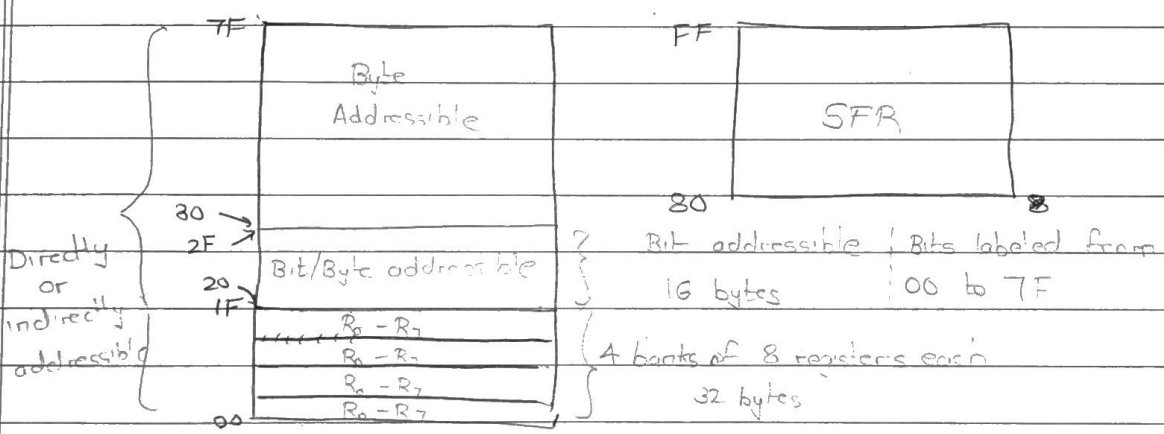
→ RAM.

- 128 B internal.
- Unlike 8085, RAM is like memory instead of registers in construction.

* SRAM.



•



- 20 to 2F :: Can be used to access 16 8-bit registers (labelled 20 to 2F) or 128 bits (labelled 00 to 7F)

SFR ↓ - RAM = ~~20~~ 00 to 7F (128 bytes) --- "General Function Register"

Special Function Registers = 80 to FF (128 bytes)

- 4 bytes in SFR keep track of ports

P0 = 80, P1 = 90, P2 = A0, P3 = B0

- 81 = Stack Pointer

- 82, 83 = DPTR (Data Pointer) (16-bit)

DPH DPL

Verify - E0 = Accumulator

P.T.O.

Only R0 and R1 can be addressed indirectly

Verify – Register 'B' = F0

On multiplying 2 8-bit numbers :- 8 LSBs in Acc,
8 MSBs in B

On dividing 2 numbers :- Quotient in ~~Acc B~~ Acc
Remainder in ~~B Acc~~ B

– The two counters can each count till 16-bit nos.
Their count is store in two pairs in SFR (TL0-TH0, TL1-TH1)
16 bit 16 bit

– D0 = PSW :- Program status word

Carry	Aux Carry	–	RS1	RS0	Overflow	–	Parity

Comes from MSB – 1th bit because MSB is sign bit

Refers to which of the 4 register banks is being used (default 00)

Comes from MSB

– Indirect memory access uses RS1, RS0 and R0-R7 to refer to regs
Direct memory access can directly use 00 to 1F

• 4 addressing modes (for the 4 register banks)
– Register, Direct, Indirect (Register), Immediate

Register	Direct	Indirect (Register)	Immediate
R0-R7 depending on RS1-RS0, or 00 to 1F In general for RAM USER	00 to 1F for banks, or 00 to 7F	@Ri i=0 or 1 only Contents of register are the address to be found in RAM USER. @R0 means R0 of currently selected bank	#n

All hex numbers must begin with a digit (Write FF as 0FFH)

5 INC R_n / @R_i / address Increment
6 DEC " " " Decrement
~~7 XCH A, R_n / @R_i / address, Exchange~~
8 SWAP A Swap the two nibbers in A

⟹ Instructions

→ Arithmetic

1 Addition.

ADD A, R_n. eg – ADD A, R0 :- Acc = Acc + R0
ADD A, add. ADD A, 4 :- Acc = Acc + R4
ADD A, @R_i ADD A, @R2 :- Acc = Acc + M(R2)
ADD A, #n. ADD A, #4 :- Acc = Acc + 4
ADDC A, R_n / @R_i / address / #n.

2 Subtraction

There is no subtraction without borrow. Make carry bit zero for no borrow

SUBB.

3 Multiplication.

MUL AB B = 8MSBs of A*B , A = 8 LSBs of A*B
correct use other registers

4 Division

DIV AB A = Quotient (A/B) B = Rem (A/B)

→ Logical

1 AND.

ANL A, R_n
ANL A, Add
ANL A, @R_i
ANL A, #Imm
ANL Add, A
ANL Add, #Imm

} AND of Address with A or #Imm and result is stored in Address.

Only logical operations can have results stored not in accumulat

2 OR

ORL :- All 6 variants as with AND

3 XOR

XRL :- All 6 variants as with AND

→ Other

1 Rotation

RL
RLC } Only usable on A.
RR
RRC

2 CLR A Clear Accumulator (all zeros)

(right

3 CPL A Complement Accumulator (0 ↔ 1)

→ Data Transfer

1 MOV

MOV A, address A = address

MOV address, A

MOV address, R_n

MOV address, address'

X MOV R_m, R_n --- Use MOV add, add for achieving same eff

MOV address, @R_i

MOV address, #n. Store n in address

MOV DPTR, #16-bit data

MOV transfers with internal RAM or SFR.

A + DPTR means
add value in A to 16-bit no. of DPTR
and access its ROM

classmate
Date
Page

- MOVX transfers from external memory (RAM) to RAM or SFR, or vice versa
  MOVX address, @DPTR
  MOVX @DPTR, address
  (both 16-bit)

- MOVC transfers from ROM to Acc only (X vice versa) generates PSEN signal and not RD signal
  MOV C A, @(A + DPTR)
  MOV C A, @(A + PC)
  (PC: Program Counter)

2 Exchange.
  XCH A, R_n
  XCH A, Address
  XCH A, @R_p

→ Changing Control Flow

1 Calling a function
  ACALL 11-bit-address ---- 5 MSBs are 00000, for compact instructions
  LCALL 16-bit address
  RET

Continued ahead

- When first function of a nest of functions is called, ~~stack pointer is initialized to 07 (R7 of bank 1), and~~ the PC is stored in 08, 09. Further nested functions will be stored in 0A, 0B, 0C-0D and so on. Returning will return in reverse order (LIFO), and decrement stack pointer by 2.
  - While executing any function, do not overwrite registers of stack, or you won't be able to return.
  - The starting register of stack pointer can be manually changed from 07.

Bits of PSW :- SETB PSW.4
CLR PSW.5

JB bit, rel - Jump if bit = 1
JNB " - ≠
JBC " - = , then clear bit

60/8

* Chips like 8052 have 256 bytes RAM (00 to FF)
The addresses ~~from 80 to FF are~~ of 80 to FF in RAM are indirectly addressed (@R), and 80 to FF in special function registers are directly addressed.

→ Bit-oriented Operations
- Carry bit is one of the operands
- Only bit addressible regions of RAM can be accessed

ANL C, bit
ORL C, bit

CLR C
CLR bit
SETB C
SETB bit
CPL C
CPL bit

→ Stack Operations
PUSH address
POP address

→ Control Flow Operations (continued)
- AJMP 11-bit address
- LJMP 16-bit address
SJMP rel (8-bit-number)

RETI

- JZ rel Jump relative to PC if A == 0
  JNZ rel !=
  JC/JNC rel Jump relative to PC based on carry bit
- CJNE ~~rel~~ A, address, rel " " " " if A != value at address
  CJNE R_n/@R_i/address, #n, rel

- DJNZ R_n, rel R_n-=1, jump rel to PC if R_n != 0

Q Add 10 values stored in consecutive locations from 50H.

```
      MOV R3, #10      // no. of additions to perform
      MOV R0, #50H
      MOV A, @R0
Loop: INC R0
      DEC R3
      ADD A, @R0
      JNC Next
      INC R5
Next: CJNE R3, #0, loop
      MOV R4, A
Stop: SJMP Stop   ... // to waste time
```

Q 10 values stored in locations from 50H.
Find if a number is present there, and return its index if yes.

```
MOV R0, #50H
MOV R2, #40
MOV A, @R0
CJNE A, #25, Forward
```

31/7/18

R0, because MOV R0, R0 is not allowed

```
MOV   R3, 0
SJMP  STOP
Forward:
    INC  R0
    DJNZ R3, Loop
    CLR  A
STOP: SJMP STOP
```

→ ORG address Assembler Directive

Write in beginning of code to ask the instructions to begin from written address, instead of default 00H.

• END

An assembler directive that ends the program.

→ LJMP v/s SJMP

LJMP can be used to jump to any memory address

SJMP can only move 256 bytes in neighborhood of current PC.

→ ORG is not an instruction. So there ~~any~~ program must begin at the memory 00H.

```
ORG 00H
LJMP MAIN
  :
ORG 100H
MAIN: MOV SP, #60H
```

→ EQU nam

~~Ex~~ Using the

in the prog

Q Sort 50 num

```
MOV  R2
MOV  R0
MOV  R3
DEC  R3
MOV  A, @
INC  R0
MOV  B, A
CLR  C
SUBB A, @
JC   Next
MOV  A, B
XCH  A, @R0
DEC  R0
MOV  @R0, A
INC  R0
Next: DJNZ
      DJNZ
Stop: SJMP
```

→ EQU name #25

~~Us~~ Using this assembler directive, we can write 'name' everywhere in the program in place of #25

Q Sort 50 numbers of an array starting from 50H.

```
        MOV  R2, #40
        MOV  R0, #50H
        MOV  R3, 2
        DEC  R3
Loop:   MOV  A, @R0
        INC  R0
        MOV  R5, A
        CLR  C
        SUBB A, @R0
        JC   Next
        MOV  A, R5
        MOV  A, @R0
        DEC  R0          (Exchanging
        MOV  @R0, A
        INC  R0
Next:   DJNZ R3, Loop
        DJNZ R2, Loop2
Stop:   SJMP STOP
```

Stack Pointer points to the top of stack
(already filled memory)

classmate
Date
Page

A] Subroutine

= Function, but can return multiple values, and can actually change memory outside its scope.

eg SWAP :
⋮
RET

To main program, use LCALL/ACALL SWAP

- By default, Stack pointer is pointing to 07H.
Calling a function stores a new address and 08H and 09H, and SP now points to 09H.
RET returns to memory pointed to by SP, SP-1 and decrements SP by 2.

- Changes made by a subroutine to memory will stay after returning.

eg Subroutine for waiting.

```
WAIT :  PUSH 3
        MOV R4, #50
WAIT1 : DJNZ R4, WAIT1
        POP 3
        RET
```

Pushing and popping is done so that value of R3 is not lost

2/8

eg Periodically lighting up ~~one of~~ 8 LEDS

```
LCALL DELAY
MOVA A, #FFH
MOV P1, A                    // turn on
LCALL DELAY
SJMP LOOP
```

→ Passing arguments by Stack.

PUSH to stack before calling subroutine.
The subroutine acquires those arguments by popping.

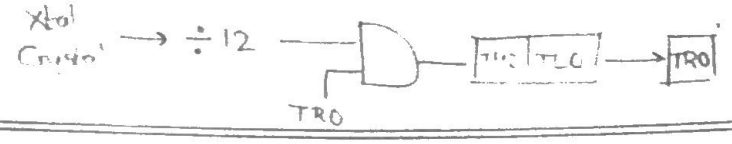
```
MOV R1, #50
MOV R2, #60
PUSH 01                      // push to 08, SP = 08
LCALL DELAY                  // SP = 0A
DEC SP                       // required later to reset PC to 07
DELAY: DEC SP
DEC SP                       // SP = 08
POP 3                        // SP = 07
Loop: DJNZ R3, Loop
INC SP
INC SP
INC SP                       // SP = 0A
RET                          // SP = 08
```

In addition, you can push R3 and pop it later to not lose value stored in it. (seen before)

- If you want to ~~build a bigger stack~~, ~~or to~~ utilize the different register banks, initialize Stack Pointer to 20, so as to use ~~20~~ 21 to 7F as byte-addressible locations.

```
MOV SP, #20
```

MMC = 01



6/8

B] Timers

12 cycles = 1 Machine Cycle

* The minimum time taken by instructions is 12 ~~machine~~ cycles. Some instructions can take 24/36/48 cycles.
- ISA does not specify time to be taken, actual implementation does

* 12 MHz is actually a 11.0592 MHz crystal, because this is very ~~M~~ Time for every instruction ~ 1μs, 2μs and so on.
- Crystal is connected across pins 18 and 19

→ TIMER0 and TIMER1
- Mundane tasks (like LED on-off) can be offloaded to these hardware programmable timers. While these timers work, microcontroller will no longer be busy.
- Both timers are 16-bit (TH0-TL0, TH1-TL1)
- Both timers count up. They raise a flag when they wrap around 0FFFFH
- The register TCON (Timer control) shows these flags.

TF1	TR1	TF0	TR0

- Timer clocks upward if TR = 1
- TF becomes 1 when timer wraps around.
- 1 timer count = 12 clock cycles = 1 machine cycle

9/8
- These 16 bit timers can count up to 65536 machine cycles
- TCON register is bit addressible

• Second register = TMOD

	T1				T0		
GATE	C/T	M1	M0	GATE	C/T	M1	M0

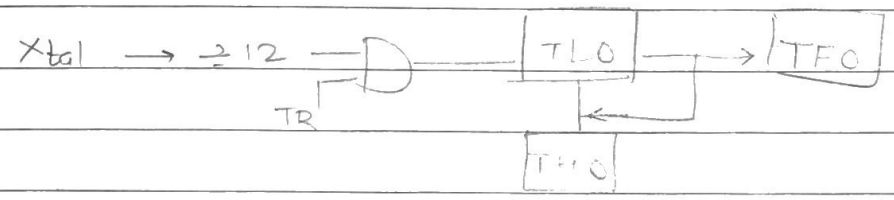
TMOD is byte addressible, not bit addressible

- 4 modes (M1,M0)
- 00 - Timer - 13 bits (5 from LSB, 8 from MSB)
- 01 - Timer - 16 bits

  The registers can be initialized to any number.

  eg - Initialize to FFFF if you want to count only once.
- 10 - Autoload - 8 bits

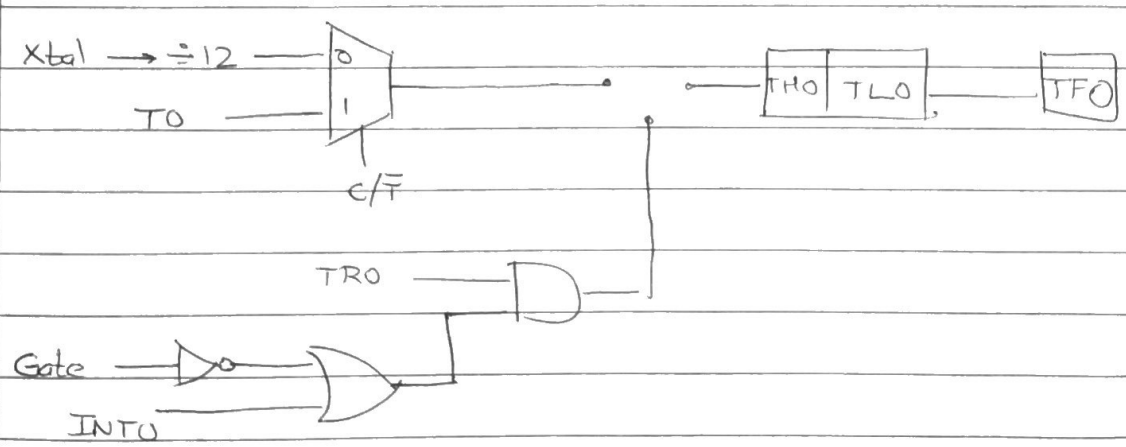
  Can count upto 256. Automatically loads after wrap-around



When count rolls over, value from TH0 is loaded into TL0

- 11 - Split timer (3 timers) ---- Later.

- Circuit



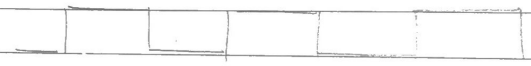
- Gate is used to control timers via interrupts. It counts the number of events at interrupt flag.

  When Gate = 1, count up if TR0 · INT0 = 1

LOOP:

```
MOV   TMOD, #1H      ; Makes gate=0, C/F=0, M1M0=01 for T0
MOV   TH0, #0FEH
MOV   TL0, #0F0H
CPL   P1.0           ; generating pulse
SETB  TR0            ; timer starts counting
CALL  DELAY
STMP  LOOP

DELAY :  JNB  TF0  DELAY
         CLR  TF0
         CLR  TR0
         RET
```

Final P1.0 :-



eg Program to count execution time of a program

```
MOV   TMOD, #01H
MOV   TH0, #0H
MOV   TL0, #0H
SETB  TR0
      { Program
CLR   TR0            ; TH0-TL0 will give execution time in
                       machine cycles (upto 65536)
```

Timer 0 can be used in

classmate
Date
Page

13/8

* Quiz Question :-

LJMP is used instead of LCALL to call a function

- RET will return to the location SP, SP − 1 and decrements SP twice
- If not assigned a value, any bit can be initalized to 0 or 1 when program starts.
- Not all memory locations in SFR exist. Program will hang if you jump there.
- If SP = 01, and RET is executed, SP will change to FF.

→ Timer (Cntd)

- Mode 11 (Split timer) of Timer 0 only
  3 timers are run simultaneously. (Useful only when Timer 1 is already running in mode 10 eg - Serial communication)
- ~~Timer 1 is run in mode 3 10~~
- TH0 and TL0 run as two independent 8-bit timers.
  TH0 will use TF1 as flag, TL0 will use TF0 as flag

eg

```
Delay :  SETB  TR0
Loop  :  JNZ   TF0, Loop
         CLR   TR0
         CLR   TF0
         RET
```

This program waits for TF0 to become 1 to take an action.
Time is wasted while waiting for TF0.
Use interrupts instead.

Using interrupt's can affect delay loops that give specific delay of times

Interrupts

When some event occurs, jump to another location where a sequence of statements are written, then return to main program where you left off.

Reset button.
INT_1, INT_0 :- External interrupts
T_0, T_1 :- Timers
Serial Communication.

The code which is executed after an interrupt occurs is called 'Interrupt service routine'

Default locations for ISRs :-

Reset	00H
INT1	03H
T1	0BH
INT0	13H
T0	1BH
Serial Communication	23H

Eight addresses for every ISR might not be sufficient. Write a further elsewhere and call it from main

Command 'RETI' :- Return from interrupt.

Because of ISRs, we don't usually start main program from 00H.

IE (Interrupt Enable) Register in SFR
Directly addressible.

Used to enable/disable all interrupts.
Reset cannot be disabled.

Using interrupts can affect delay loops that give specific delay of times

C] Interrupts

When some event occurs, jump to another location where a sequence of statements are written, then return to main program where you left off.

eg Reset button.
eg INT1, INT0 :- External interrupts
eg T0, T1 :- Timers
eg Serial Communication.

- The code which is executed after an interrupt occurs is called 'Interrupt service routine'
- Default locations for ISRs :-

Reset	00H	
INT1	03H	Eight addresses for every ISR might not be sufficient. Write a function elsewhere and call it from here.
T1	0BH	
INT0	13H	
T0	1BH	
Serial Communication	23H	

eg

- Command 'RETI' :- Return from interrupt.
- Because of ISRs, we don't usually start main program from 00H.

→ IE (Interrupt Enable) Register in SFR
- Directly addressible.

- Used to enable/disable all interrupts.
- Reset cannot be disabled.

14/8

- IE Register — Bit addressible

EA			ES	T1	EX1	T0	EX0

Global enable
Interrupts are disabled if EA = 0.

Interrupt's when TF1 = 1

- When one interrupt occurs, from timer, its timer-flag is automatically made zero by 8051.
- When any interrupt occurs, current instruction address is stored in SP, SP+1 and SP is incremented twice, so as to return on RETI

eg Square wave at P1.0 using interrupts.

```
MOV   IE, #82H          ; EA=1, T0=1
MOV   TMOD, #02H        ; Timer 0 in mode 2 (Auto load)
MOV   TH0, #number
SETB  TR0               ; will jump to 0BH when TF0=1
---------
; At 0BH
CPL   P1.0
RETI
```

- If more than one interrupts get triggered at the same time, then the one with highest priority occurs first (order on prev page decreasing from Reset to serial communication)

- If an interrupt of higher priority occurs while executing one interrupt, we first execute the newer interrupt.

P.T.O.

- Hence, 8051 has to remember all the 'pending' interrupts at all times.

ITI

- Hence, RETI is different from RET, because it also updates the pending interrupts status.

- The default priority order can be changed by ~~TCON~~ IP register.

~~TF1~~	~~TR1~~	~~TF0~~	~~TR0~~

Serial comm

				T1	INT1	T0	INT0
–	–	–	PS	PT1	PX1	PT0	PX0

All with value '1' are of higher priority.
Those with same IP values have default priority order between them.

→ External Interrupts .... INT1 & INT0
- Level-triggered by default (triggered when becoming low)
- Can be made edge-triggered (negative edge) with IT1 or IT0 bits

TCON Register

Timer part — exchange

TF1	TR1	TF0	TR0	IT1	IE1	IT0	IE0

↳ 0 – Level triggered INT1
1 – Edge

16/8

- Level-triggered interrupt :-

INT 1 or INT 0 should be low for 4 machine cycles to safely register an event for the interrupt
(4 MCs = maximum time for an instruction)
Instruction can only take ~~12~~, 1, 2, 3 or 4 MCs

- Edge-triggered interrupts.-

Whenever there is a negative edge, IE 0 or IE 1 becomes 1, which triggers the interrupt. No minimum time condition
After interrupt, ISR makes IE 0.

eg Code for showing no. of interrupts ~~(Gate = 1, timer counts pulses in INT0)~~

```
ORG 0003H
    LJMP 100H
ORG 100H
    INC R1
    MOV A, R1
    MOV P1, A          ; Print out the count
    RETI
MAIN: SETB. TCON. 0    ; Make INT0 edge triggered
    MOV IE, #81H       ; Enable INT0.
```

→ $\overline{CE}$:- Chip enable (make low to enable chip) Can be permanently 0 if only 1 chip present.

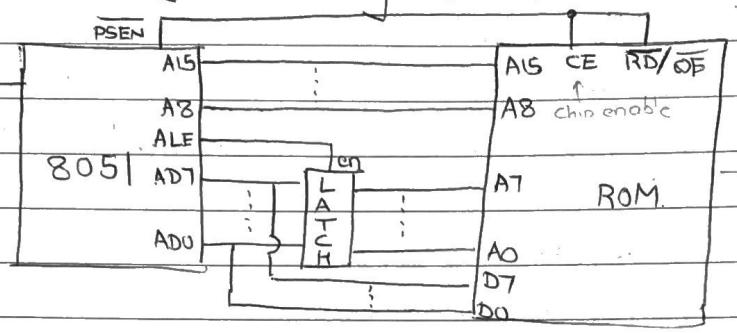
→ In the beginning, PC = 0000

→ When you want to read make $\overline{PSEN}$ 0. This will make $\overline{RD}$ of external chip

→ $\overline{EA}$ is external input signal. It is sampled only once initially to set the configuration.

D] ✳ External Memory Interfacing (ROM)

→ Every instruction can make $\overline{PSEN}$ 0 (multiple times) whenever it it is to be read from ext ROM or it wants to fetch data from ext ROM



- If $\overline{EA} = 1$, use 4 kB from internal ROM, and remaining 60 kB from external ROM. First 4 kB of external ROM are unusable.

  entire 64 kB from external ROM

- Instead of 64 kB ROM, if you want to use 32 kB only along with 32 kB extnl, you will need only 15 bits for addressing.

  Make A15 low :- Lower 32 kB are used.

  high :- Higher 32 kB (external) and 4 kB (internal) are used.

→ When an instruction is completed, (12/24/36/45 cycles), $\overline{PSEN}$ is made 0 again to read next instruction.

→ An n-byte instruction (n = 1,2,3) will make $\overline{PSEN}$ 0 [illegible] times. [illegible] stays high for 2 cycles

→ Reading an instruction :- Make ALE high. Set address on AD [illegible]. Then make $\overline{PSEN}$ 0 to read. Thus, reading one byte of the instruction takes one cycle

→ It will as 17th If P3.

20/8

→ Apart from reading instruction, $\overline{PSEN}$ will be made 0 after MOVC is executed, and the data is stored in A via data bus D7-D0

- $\overline{RD}$ (or $\overline{OE}$) signal on ROM has three possible values :- 0, 1, High impedance
  - High Z position, ensures that ROM does not load the data bus when D7 to D0 are being used.

MOVC A, @A+PC , MOVC A, @A+DPTR

Step 1 :- Instruction MOVC is read from code ROM.
Step 2 :- Data is read from data ROM

} Both steps make $\overline{PSEN}$ 0.

21/8

$\overline{RD}$ →

$\overline{PSEN}$ →

✳ You can use RAM as code memory by interfacing it with $\overline{PSEN}$ and writing instructions to it after every boot. If actual code memory, & in the end, set PC to first instruction of RAM so that it will continue execution. Here, MOVX == MOVC

→ SP is indirectly addressible. So, you cannot move it to SFR

→ Code ROM interfaced with $\overline{PSEN}$ } connect to $\overline{RD}/\overline{OE}$ of
Data ROM $\overline{RD}$ } ext chip
RAM $\overline{RD}$, $\overline{WR}$

→ Whenever $\overline{RD}$ is high, ROM memory byte is read via data bus D7 – D0

→ $\overline{CE}$:- Chip enable (make low to enable chip) Can be permanently 0 if only 1 chip is present

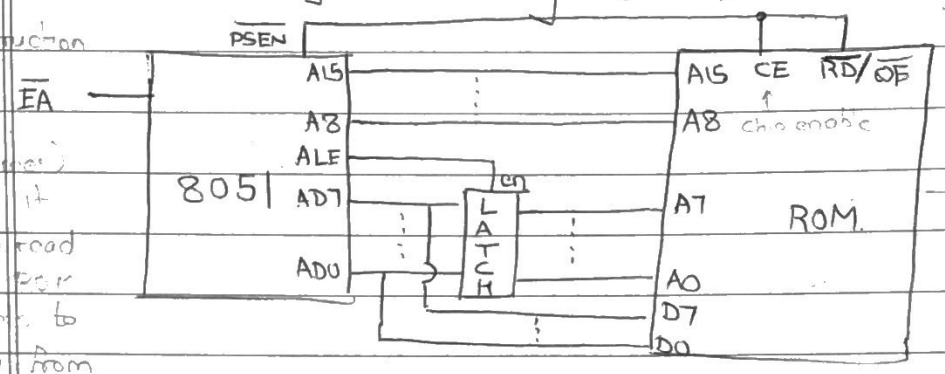
D] * External Memory Interfacing (ROM)

→ In the beginning, PC = 0000

→ When you want to read, make $\overline{PSEN}$ 0. This will make $\overline{RD}$ of external chip

→ $\overline{EA}$ is external input signal. It is sampled only once initially to set the configuration.

→ Every instruction can make $\overline{PSEN}$ 0 (multiple times) whenever it it is to be read from ext ROM or it wants to fetch data from ext ROM



- If $\overline{EA}$ = 1, use 4 kB from internal ROM, and remaining 60kB from external ROM. First 4kB of external ROM are unusable.

- Instead of 64 kB ROM, if you want to use 32 kB only along with 32kB ext, you will need only 15 bits for addressing.
  Make A15 low :- Lower 32 kB are used.
  high :- Higher 32 kB (external) and 4kB (internal) are used.

→ When microcontroller is completed (i.e. 24 bit address), $\overline{PSEN}$ is made 0 again to read next instruction.

→ An mcode instruction [illegible] 0 [illegible] takes 2 cycles

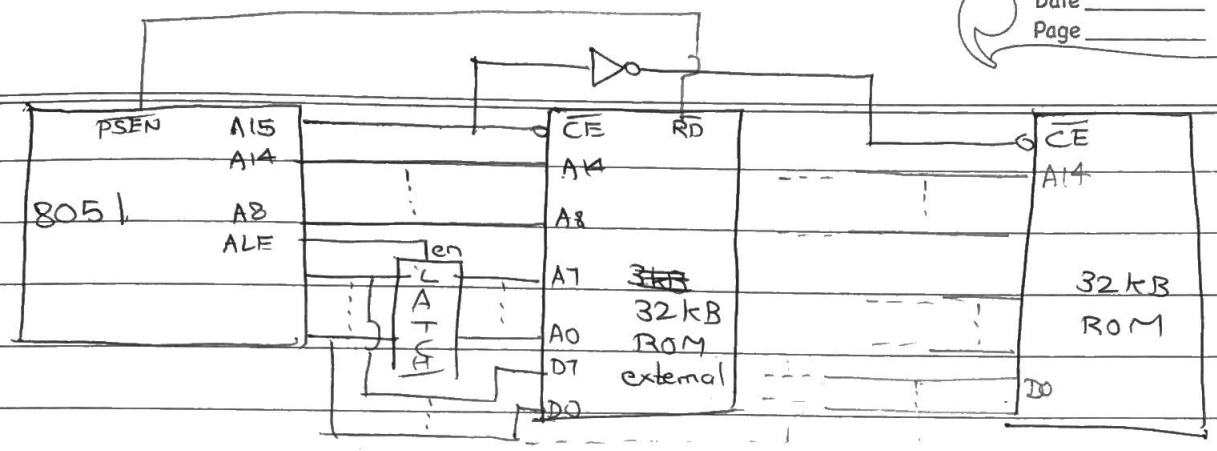
→ Reading a register :- Make ALE high. Set address [illegible] make $\overline{PSEN}$ 0 to read. Thus, reading a register takes 1½ cycles

20/8

→ Practicum • $\overline{RD}$ (or $\overline{OE}$) signal on ROM has three possible values :- 0, 1, High impedance
- High Z position, ensures that ROM does not load the data bus when D7 to D0 are being used.

→ Practicum makes instruction $\overline{PSEN}$ 0 to read after MOVC is executed, and the two instructions for displacement are:
MOVC A, @A+PC , MOVC A, @A+DPTR
Step 1 :- Instruction MOVC is read from code ROM. } Both steps make $\overline{PSEN}$ 0.
Step 2 :- Data is read from ROM

* You can use CPU as code memory by interfacing it with $\overline{PSEN}$ and writing instructions to other memory [illegible]. If actual code memory, & in the end, set PC to first instruction of CPU so that it will continue execution. ∋ Here, MOVX == MOVC



When A15 is 0, 0000H to 7FFFH are used

When A15 is 1, 8000H to FFFFH are used
and 0000H to 0FFFH (internal 4kB)

- Similarly, if you want to use 4 16kB memories, use A13 to A0 for addressing, and A15 & A14 to choose which memory is being used

A15	A14	Memory (Chip)
0	0	0000H to 3FFFH (1st)
0	1	4000H to 7FFFH (2nd)
1	0	8000H to CFFFH (3rd)
1	1	D000H to FFFFH (4th)

- Internal ROM (4kB) can be accessed using 0000H to 0FFFH only when 1st chip is not being used.

→ It will be possible to attach 64kB more memory if you use a pin in a port as 17th bit of address. (say P3.0)

If P3.0 = 0, access to first 64kB,
1 other

At the beginning, P3.0 can take any value (0 or 1).
If $\overline{EA}$ is 1, make sure you set required value of P3.0 in initial few instructions.
If $\overline{EA}$ is 0, in the beginning of both 64kB memory, set the required value of P3.0

21/8

$\overline{RD}$ → MOVX → External RAM

$\overline{PSEN}$ → MOVC → Internal / External ROM

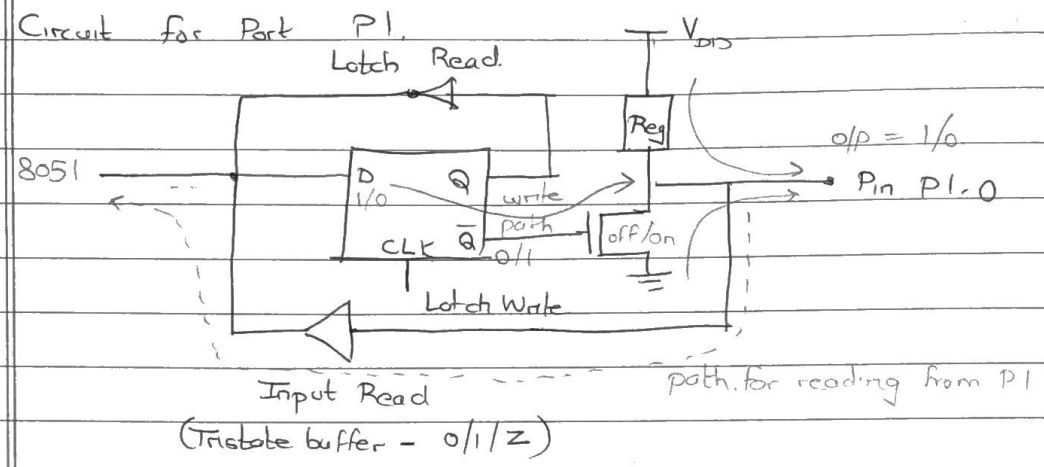
But make sure after 0FFFF is executed, PC should move to 10000

At end of 0FFFF, jump to 00100, where you will make P3.0 = 1. Then you will move to 10100 and not have to execute initial few instructions of 10000

P.T.O

If you execute XRL P1, #n, and P1 is configured as input
FF will be read (because P1 is I/P port)
n will be XORed with FF
n ⊕ FF will be written to P1 ⇒ P1 is 'reconfigured'

→ Circuit for Port P1.



- In 8051, for all operations, A is one of the ~~accumulators~~ operands and the destination except for logical operations.

eg - XRL P1, A ; P1 ⊕ A, send to P1
'Read P1, modify, write to P1'
all actions at ...

- For this statement, the 'reading' part is of signal 'D', not the Pin P1.0 directly (because port is configured as O/P port)
Read D, modify, Change D.

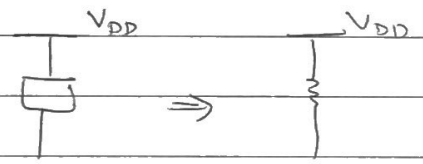
- ~~Cannot take input from P~~ The above instruction is valid only when P1 is configured as ~~an~~ output port.

- 8051 can only supply limited current.
Value of current depends on register Reg.
If current is high, 8051 will burn.

- If you wont to read from P1,
  MOV P1, #0FFH ; This ~~it~~ makes D = 1. Transistor is turned off.
  MOV A, P1

Now, Latch read is on high Z mode.
Input is active.

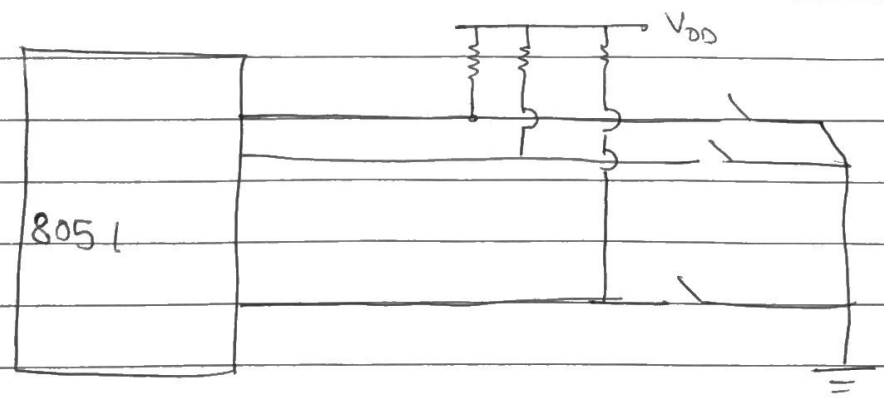
- To ensure high current is not drawn from 8051, replace internal register (Reg) ~~out~~ by an external resistor instead.



Choose ~~pull~~ this pull-up resistor appropriately.

- As this resistor is external, current is not being drawn from 8051.
- Only port P0 has an open-drain transistor.
  External R can be connected only to pins of P0.
  If there is a demand for current, interfacing needs to be done only on P0, with appropriate pull-up resistor.

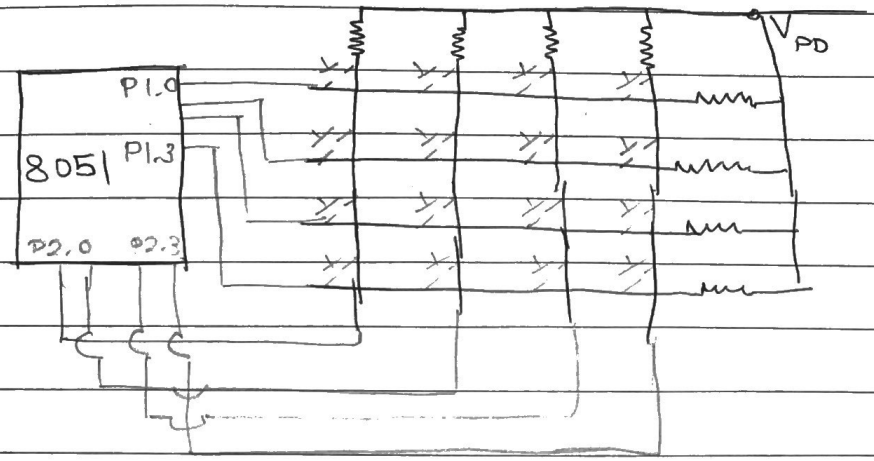
→ eg - Input from keyboard.



- You is raise when number of keys > no. of pos
  hence the comes 2-or values corresponding to a' mapping
  [illegible]

[illegible]

23.2 • Keyboard has 6 keys (say), but we want to use 4 mod pos



P1 is O/P port. P2 is I/P port

1 Make all pins of P1 zero.

If no push-button is pressed, all pins of P2 are 1.

If P2 ≠ 1111, (one of the pins is zero), some key has been pressed

2 Scan rows one-by-one (if some key has been pressed)

To check row 0, make P1 = 0111.

If no key in row 0 is pressed, P2 = 1111

0th key P2 = 0111

1st 1011

- Repeat for other rows (if no key was found till now)

3 Suppose, while scanning a row, you get 1101

How does 8051 know the switch in which column was pressed?

Use ~~carry~~ rotate with carry, and count the no. of rotations until carry becomes zero.

4 Now that you know row and column, access corresponding character from ROM database.

Row 0 DB 'a', 'b', 'c', 'd'

Row 1 DB 'e', 'f', 'g', 'h'

⋮

- The keyboard key may have jitters. Hence, ~~to~~ implement hardware debouncing (astable multivibrator) ~~or~~ and/or software debouncing (wait for some delay before observing change)

- Code

```
        MOV  P2,  #0FFH          ; Set as I/P
Step 1  MOV  P1,  #00H           ; Make P1 = 00000000

L1:     MOV  A, P2
        ANL  A, #0FH
        CJNE  A, #0FH, Scan      ; Compare with 00001111
        SJMP  L1

Scan:   ACALL  DELAY             ; few milliseconds, debouncing
        MOV  A, P2               ; Just to confirm a curr key was pressed
        ANL  A, #0FH
        CJNE  A, #0FH, Read
        SJMP  L1

Read:   MOV P1, #11111110B
        MOV  A, P2
        ANL  A, #0FH
        CJNE, #0FH, ROW0         ; Key is from 0th row

        MOV P1, #11111101B
        ~~MO~~  :
        :

        CJNE          ROW1       ; Key is from 1st row
        :
        :
        ; for row 2 & row 3
```

```
ROW0:   MOV   DPTR,   #Row0        ; Make DPTR = Starting location
        SJMP  Match                  of 0th row in ROM
ROW1:   MOV   DPTR,   #Row1
        SJMP  Match
ROW2:   :
        :
        :

Match:  ; To check which column.
        RRC   A                    ; Rotate right and left because
        JNC   final                  1st four bits of A are 0000
        INC   DPTR
        SJMP  Match
final:  CLR A
        MOVC  A, @A+DPTR
        SJMP  L1
```

- After going to L1, if the same key is still pressed, and you don't want to register same key again, edit L1 loop accordingly to wait until key is lifted.

- Implementing using interrupts.

Port P1 = 0000.

Take AND of all pins of P2 and feed it into INT0

When a key is pressed, INT0 will become 0.

28/8

→ Quiz Question

Wrongly connected :- Code ROM connected to $\overline{RD}$

Data ROM $\overline{PSEN}$

- Does not work
- In the beginning, PC = 0000.
- Because $\overline{EA}$ is low, we are using only external memories (ROM, RAM)

  Code ROM :- 0000 - 3FFF

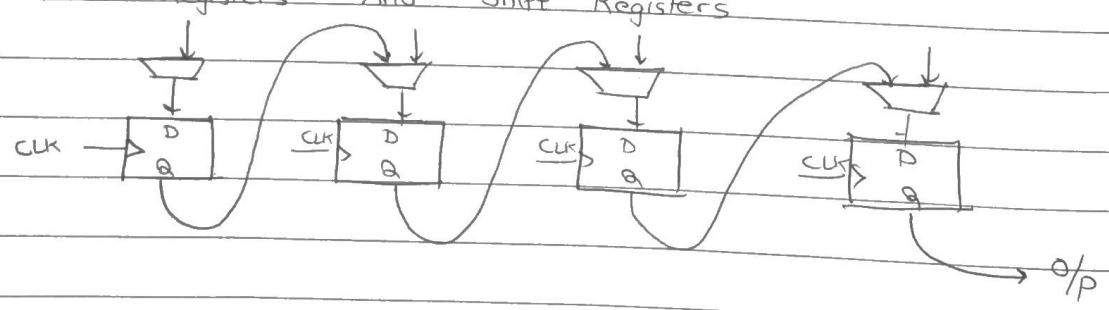
  Data ROM :- 4000 - 7FFF

  wrongly exchange
- Now, code ROM will not begin at 0000.

  ∴ 8051 will not find the first instruction
- If $\overline{EA}$ were high (using both internal & external memories), 0000 would have been in internal ROM and it would have worked well (using Jump whenever required)

E] Serial Communication

* Parallel Registers And Shift Registers



- Data is loaded parallelly (1 machine cycle)
- Then MUX is changed to serial mode

  Data is read serally (4 machine cycles)

- This method wastes more time reading, but saves on cost of wires.

→ 2 pins for Serial Communication :- RxD (P3.0)
TxD (P3.1)

- Because of parallel-in-serial-out, ~~B~~ we don't need 8 bits for Rx and 8 bits for Tx.
- We also need P/S and S/P converters
  Parallel → Serial Serial → Parallel

→ 2 types of communication :- Synchronous, Asynchronous.

- Synchronous :- Clocks of transmitter and receiver have same frequency and are synchronized
  - Clock Data Recovery (CDR) circuit :-
    At receiver, try to recover the clock ~~as~~ to be synchronized with transmitter from the data received.
    - Transmitter initially sends 01010101 - twice (2 bytes)
      Receiver notes the transitions to infer the clock at transmitter.

30/8 • Asynchronous
- ~~Has huge overhead~~ Synchronous has overhead of 2 bytes ☹
- Set frequency beforehand for receiver and transmitter
- RxD (P3.0) is high by default.
  Send a 0 to signal initiation of transmission.
  End every byte transmission with a 1.

Receiver knows beginning of ~~B~~ message by negative edge

eg 1 $\underbrace{11000011}_{\text{Data byte}}$ 0 10 bits
End bit Start bit

- Every byte's transmission begins with 0, ends with 1.
  - 2-bit overhead per byte
  - Fine for infrequent, small messages.

- There may be errors during transmission,
To check for errors, add a parity bit in the end

1 1 11000011 0 — 11-bits

end — Data — Start

parity of sum of bits of data

Parity bit can surely detect only 1-bit errors.

- Computing and comparing parity has to be done in software by programmer.
- If parity does not match, programmer chooses further steps to take based on his requirement

→ Transmission

- 'High' and 'low' voltages defined in RS232 standard

Zero bit :- 3V to 25V
One bit :- -3V to -25V

- At the output of 8051, we ~~need~~ place a 'level translator' that converts output of 8051 to voltage in suitable range.

- Datarate = Number of bits transmitted per second
- Effective datarate = Number of useful bits transmitted per second (not counting header bits)
- Baud Rate = Number of symbols transmitted per second
'voltage level'

- If one voltage level marks one bit, BR = datarate

SCS [illegible] 'symbol' for each bit → BR = DR

→ There are actually two physical addresses (two registers), both of them having address 99H., both called SBUF. Appropriate one is automatically used
One is used for transmission (MOV SBUF, #n)
other is used for reception (MOV A, SBUF)

eg- Transmit two bits of information with one voltage level

eg- +10V :- 00
+5V :- 01
-5V :- 10
-10V :- 11

One symbol = 2 bits

~~Baud~~ Data rate = 2 × Baud Rate.

- Inbuilt protocol to help in Tx/Rx = UART (having start bit, stop bit)
  = Universal Asynchronous Receiver/Transmitter

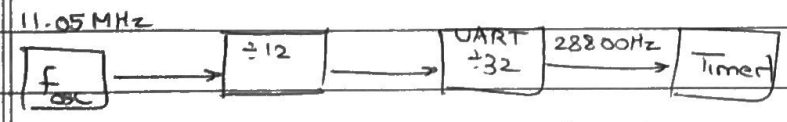
- Three registers in SFR that assist in serial communication
  i) SBUF - Serial buffer - Holds data to be transmitted or that was received.
  ii) SCON - Serial control - Controls how you want to transmit
  - Defines Baud Rate

Bit Accessible

S0	S1	S2	REN	RB8	TB8	TI	RI

S0, S1: Modes (1 = Multiprocessor, 0 = Single)
REN: Receive Enable — High when it detects it is being transmitted to.
RB8, TB8: Contains the 9th bit received/transmitted. Do with them as you please
TI: Transmission Interrupt
RI: Reception Interrupt
TI, RI: Become high when Tx/Rx of current byte is completed (STOP bit is received/transmitted). Have to be cleared in software

Modes :- 00 :- Shift register mode at $f_{osc}/12$ = 921.6 kHz ✗ UART
01 :- UART 8-bit transmission of variable frequency.
Frequency is defined by Timer 1 running in mode 2



Timer 1 can divide 28800 Hz further
For 9600 bps = $\frac{28800}{3}$, initialize TH1 to # -3

When you are using Variable Baud rate transmission, if you want to use two timers use timer 0 in mode ~~3~~ 11. We can do this because UART does not require to check the flag TF1 to work.

Maximum rate = 28800

10 :- UART 9 :- 9-bit transmission at $f_{osc}/32$

11 :- UART 9 at variable frequency defined by Timer 1 in mode 2

eg MOV SCON, #01000000B

~~eg~~ MOV TMOD, #20H ; Timer 1 in mode 2

MOV TH1, #0FDH ; For transmission at 9600 bps

SETB TR1

Send: MOV SBUF, A ; ~~The~~ starts transmission after this instruction

Here: JNB TI, Here ; 'Polling' :- Keep checking TI

CLR TI

RET

Receive: ; P.T.O

3/9/18

→ Transmission

SBUF
↓
TI ← UART → ... to ~~&~~ TxD (Baud rate is defined by mode in SCON and Timer 1)

→ Reception

SBUF
↑
RI ← UART ↔ ... from RxD

- Most commonly used Baud Rate = 9600 symbols/second

- For this, ~~MOV TH1, #3~~

MOV SCON, #010xdd00B ; d = don't care about 9th bit in first mode

x = 1 if you also want to receive

MOV TMOD, #20H

MOV TH1, #0FDH

SETB TR1

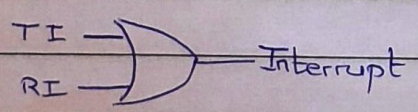
```
Receive :   JNB RI, Receive
            MOV A, SBUF
            CLR RI
            RET
```

- Transmission begins as soon as MOV SBUF, A is executed.

- The above method for Send and Receive is 'Polling'.
  8051 has to be engaged in checking reception & transmission
  Use interrupts instead.
  - There is a single interrupt for both Tx and Rx.
  - IE

EA	-	-	SI	TI	EX1	T0	EX0

  - ISR location :- 023 H ---- ISR has codes for both Tx & Rx interrupts
  - When enabled, either of TI or RI can trigger interrupt.

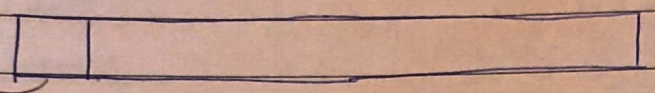



```
Main:
    MOV IE, #90H
    MOV SCON, #50H
    MOV TMOD, #20H
    MOV TH1, #FDH
    SETB TR1

    MOV SBUF, A

ORG 0023 :-
```

→ It is possible to change Baud Rate dynamically.

Not bit addressible

- Power Control (PCON) Register



0: Baud Rate

1: 2 (Baud Rate)

MICROPROCESSOR DESIGN

- Time required for execution of instruction is not defined by ISA. The implementation (design) will decide the time that it takes.

- Minimize :- $\frac{\text{Time}}{\text{Program}} = \underbrace{\frac{\#\text{ of instructions}}{\text{Program}}}_{\text{depends on ISA}} \times \underbrace{\frac{\#\text{ of cycles}}{\text{Instruction}}}_{\text{depends on ISA and machine organization}} \times \underbrace{\frac{\text{Time}}{\text{Cycle}}}_{\text{depends on digital design technology and skeels}}$

  not MCs

  CS CE EE

∴ Revision 3-address format, Stack-based architecture, Accumulator architecture, Accumulator + Register architecture.

4/9

* Software
- Apps
- Compilers
- OS

Hardware
- Digital Logic Design
- Transistor Level Design
- Placement and Layout
- Fabrication

ISA :- Defines programmer's model
- Define instructions – Operations, Operands.

- Instructions – Change the 'state' of the program $(S \rightarrow S')$
- State ≜ Values stored in memory ∪ registers.

ARM gives ISA
Intel/AMD produce μP

→ 3 address format :-

7 + 32 + 32 + 32

OP	OPR1	OPR2	DEST

For 4GB RAM, every address takes 32 bits to ~~ad~~ write.

- Need to reduce size of instruction.

→ Take operand from register (4 bits for 16 registers) or memory (32 bits) followed by 1 bit to say which among those is used.

→ Fix the size of instruction.
Use variable length encoding.
'Complex Instruction Set Computers'
- All x86 computers follow CISC.

- Registers are easy to build. (collection of flip flops)
- Compiler assigns registers to variables based on lifetime of variables
  - Lifetime = No. of instructions b/w definition of variable and its last use.
  - Variables having longest lifetime are given a register, others are stored in memory.
- If you have ~~5 reg~~ 32 registers (5-bit address), say, we can devise 3-operand instructions, with all three operands coming from registers only.
  'Reduced Instruction Set Computer'

OP	R1	R2	R3
7	5	5	5

- Load & Store instructions to transfer data between memory and registers. In RISC, only these two instructions are permitted to directly use memory.
  RISC = 'Load/Store Architecture'

- RISC uses fixed-length encoding to ease decoder's job

→ Performance $\triangleq \frac{1}{\text{Time/Program}}$

→ Instructions can take single cycle or multiple cycles

- RISC may use instructions of both kinds.
- CISC must use multiple cycles for most of its instructions
  - For example, simply accessing memory takes 1 cycle.

- Multiple cycle instructions :- $S \to T \to T' \to T'' \to S'$

  Single $S \to S'$

- Every instruction takes fixed time. Different instructions take different time

  eg - 12/24/36/48 cycles in 8051

- TFT :- An estimate of performance, Time/Program.
  - Need to estimate Cycles/Instruction
    - Take weighted sum of cycles required for all instructions, with weights being the frequency with which that instruction is executed.

      Based on what program(s) would you decide weights ?

- Minimize :- Cycles/Instruction = CPI

  CPI depends on ISA, and machine organization.

  People are too lazy to modify ISA at this point.

6/9

- The three factors of Time/Program have common dependencies, and may not be optimized independently.

* Cycles/Instruction = 10 in 1980s
  = 0.2 ~ 0.3 today. (simultaneous instructions)
- How to decrease CPI?

- RISC has a smaller instruction set as compared to CISC.
- CISC (x86) :- Instructions can take [1, 17] cycles to execute.

→ Estimating CPI :-
- Identifying most common programs and find CPI by taking weighted mean of cycles required for those instructions.
  - Done by SPEC (Standard Performance Evaluation Corporation)

  eg - In 2000, 26 programs were chosen, 12 of which were floating point operations based, and remaining were based on ~~decision making~~ ~~conditional~~ ~~loops~~ integer operations

- MIPS ≜ Million instructions per second.

$$= \frac{\text{No. of instructions}}{\text{Time} \times 10^6}$$

$$= \frac{\text{No. of instructions}}{\text{No. of instructions} \times \text{CPI} \times \text{Cycle time} \times 10^6}$$

$$= \frac{\text{Frequency of clock} \times 10^{-6}}{\text{CPI}}$$

- For comparing performance in MIPS, the two machines must have same ISA and ~~for~~ ~~com~~ compiler.
- Lower bound on MIPS can be calculated by substituting maximum CPI for the machine.

- MFLOPS ≜ Million Floating Point Operations per second.
  - More reasonable than MIPS, because while no. of instructions might be different for different machines, no. of operations would remain roughly same, because algorithm is same.
  - If program has integer operations or decision making (conditional loops), MFLOPS makes no sense and IPS is used.

- Comparing your machine with a reference machine.

eg :- Sparc III has SPEC score 100.

Program	P1	P2	P3	P4 ...
Speedup	S1	S2	S3	S4 ...

S1 times faster than Sparc III

Final score = (GM of S_i's) × 100

I CISC DESIGN

→ CISC

- Variable length encoding
- Supports many addressing modes → Reg direct, mem direct, reg indirect, immediate, B+D, auto INC/DEC, mem indirect
- Large number of instructions in ISA.
- Multiple-cycle instructions.

→ Components of machine

1. Datapath
2. Control path
3. Phase locked loop
4. Bus controller :- Interfaces with memory
5. Memory

6/9
द्वितीय

* A high number of instructions use immediate addressing. Pretty useful
* Base + Displacement :- Useful for iterating over arrays
* Auto increment/decrement :- Push/pop of stack, etc.
* Memory indirect :- Memory contains address of operand.

- Begin designing system by defining ISA.
  - Specify no. of registers, etc.
  - Specify no. of instructions, addressing formats, addressing mode

- There is a systematic way of converting your ISA to a real circuit
  = 'Hardware Flow Chart'
  - Start by building logic gates.
  - Your operations should be performable via smaller functional units
    eg- Multiplication by repeated addition
    Subtraction by addition of 2's complement.
  - Need more storage which stores partially computed values.
    - Registers, which are accessible to programmer.
    - Temp storage, which is invisible to programmer
  - Steering Logic (Transport system)
    - Point-to-point data transfer and Broadcast (bus)
    - Transferring data from ~~register to r~~ storage to storage, storage to functional units, functional units to storage (P2P)
    - Only one set of data should be transferred at a time on the bus
  - Controller
    - Controls which instruction is executed when
      - Maintains an FSM for state transitions
        - No FSM would have been required if all instructions were of the single-cycle kind

$\longrightarrow$ Input $\longrightarrow$ Next-State Logic $\longrightarrow$ State Register $\longrightarrow$ Output

* Mealy machine :- O/P depends on current I/P and current state
Moore current state only

* Implementing FSMs by hard-coding, as opposed to ~~FS~~ Mealy/Moore.
- Simpler, no next-state logic required.
- Stores states of every instruction in ROM by computing beforehand.

Add	T_1	100100
	T_2	101101
	T_3	100001
Sub	P_1	001000
	P_2	100101

T_i's, P_i's are states.

- Instruction will say which state to move to at given instant

6/9
द्वितीय

* A high number of instructions use immediate addressing. Pretty useful
* Base + Displacement :- Useful for iterating over arrays.
* Auto increment/decrement :- Push/pop of stack, etc.
* Memory indirect :- Memory contains address of operand.

- Begin designing system by defining ISA.
  - Specify no. of registers, etc.
  - Specify no. of instructions, addressing formats, addressing mode

- There is a systematic way of converting your ISA to a real circuit
  = 'Hardware Flow Chart'
  - Start by building logic gates.
  - Your operations should be performable via smaller functional units
    eg- Multiplication by repeated addition
    Subtraction by addition of 2's complement.
  - Need more storage which stores partially computed values.
    - Registers, which are accessible to programmer.
    - Temp storage, which is invisible to programmer
  - Steering Logic (Transport system)
    - Point-to-point data transfer and Broadcast (bus)
    - Transferring data from ~~register to~~ storage to storage, storage to functional units, functional units to storage (P2P)
    - Only one set of data should be transferred at a time on the bus
  - Controller
    - Controls which instruction is executed when
      - Maintains an FSM for state transitions.
        - No FSM would have been required if all instructions were of the single-cycle kind.

→ Input → Next-State Logic → State Register → Output

* Mealy machine :- O/p depends on current I/p and current state
Moore current state only

* Implementing FSMs by hard-coding, as opposed to ~~FE~~ Mealy/Moore
- Simpler, no next-state logic required.
- Stores states of every instruction in ROM by computing beforehand.

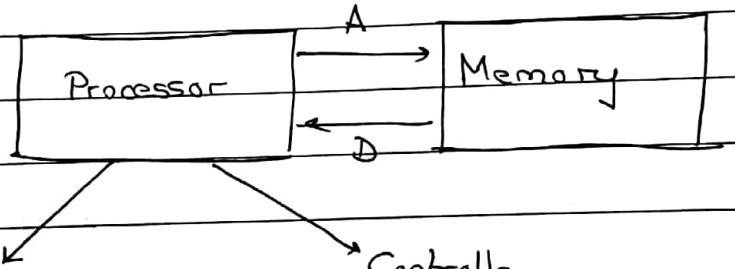
Add	T_1	100100
	T_2	101101
	T_3	100001
Sub	P_1	001000
	P_2	100101

T_i's, P_i's are states.

- Instruction will say which state to move to at given instant

17/9

CISC Architecture



Datapath
- → Functional units
- → Architecture registers
- Temporary registers
- → Steering Logic
  - → Point-to-point
  - → Bus

Controller
- → FSM
  - → Random Logic
  - → ROM based.

- To do :- Convert ISA into Implementation (Gate-level 'netlist')
- Make hardware flow chart.

Scanned by CamScanner

→ Say, you want to design a microprocessor with following specs:-

1. 3 Addressing modes :-

Register Direct, Register Indirect, Based Displacement

AR	AI	AB
00	01	10

2. Instruction format :- | Operation | R_x | A/M | R_y | ---- 1 'word'

R_y → Destination

eg :- ADD R1 AR R2 ; R2 = R1 + R2

ADD R1 AI R2 ; [R2] = R1 + [R2]

ADD R1 AB R2 Disp ; [R2 + Disp] = R1 + [R2 + Disp]

- R_x is always directly addressible.
- If you are using B+D, the displacement will be given after the instruction in the next word.

• Steps to follow while execution :- (examples 1 & 2 :- AR, AI)

1. Fetch instruction
2. Understand instruction, find locations of control signals in ROM.
3. Add
4. Write result in memory.

- Steps 1, 2, 4 are generic. Step 3 is particular to the instruction.

• Steps for B+D instructions -

1. Fetch instruction
2. Understand instruction
3. Fetch second word (displacement)
4. Compute operand 2 :- R_y + disp
5. Fetch second operand
6. Add
7. Write result in memory

Scanned by CamScanner

Only ADD, SUB, NAND, LOAD can modify zero flag

- The steps written above are states of the controller FSM

- Steps 1 and 2 are generic in all 3 examples.
  - Two ways to proceed
    1) Execute in the order written

       OR

    2) Execute the generic steps at the end of the loop
       - The first instruction is fetched and understood by Power ON sequence.

- Instructions we want to implement:- (ISA)

$\underbrace{\text{ADD, SUB, NAND}}_{\text{3 addressing modes as discussed}}$, TEST, LOAD, STORE, PUSH, POP, BZ

LOAD: Load to R_x from memory indirectly addressed by R_y

$R_x \leftarrow [R_y]$

STORE: $[R_y] \leftarrow R_x$

PUSH: Build stack downwards

$R_y = R_y - 1$

$[R_y] \leftarrow R_x$

POP: $R_x \leftarrow [R_y]$

$R_y = R_y + 1$

BZ: If the result of previous instruction is zero, jump to a particular location. Whether previous result is zero is specified as a flag

TEST: Sets the zero flag if $R_x = 0$

We will build a hardware flow chart to implement this.

- Building the datapath:-

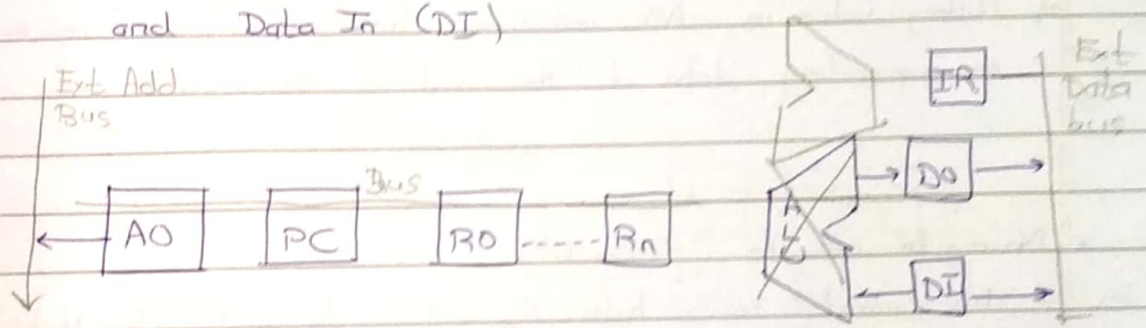
  - Functional units :- ALU (Add, subtract, ~~etc~~ NAND) is sufficient.

  - Registers :- R0 to Rn for architecture registers
    :- One register to act as instruction pointer (program counter)

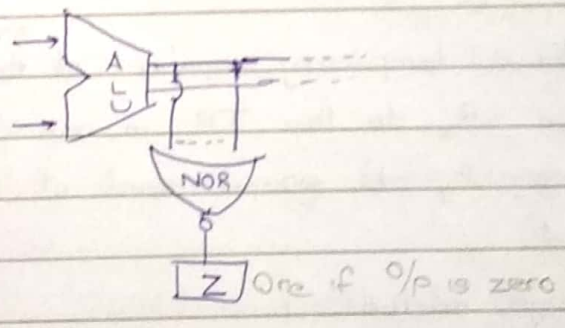
Scanned by CamScanner

- As an engineer, start with minimum no. of registers required. Add later if desired.

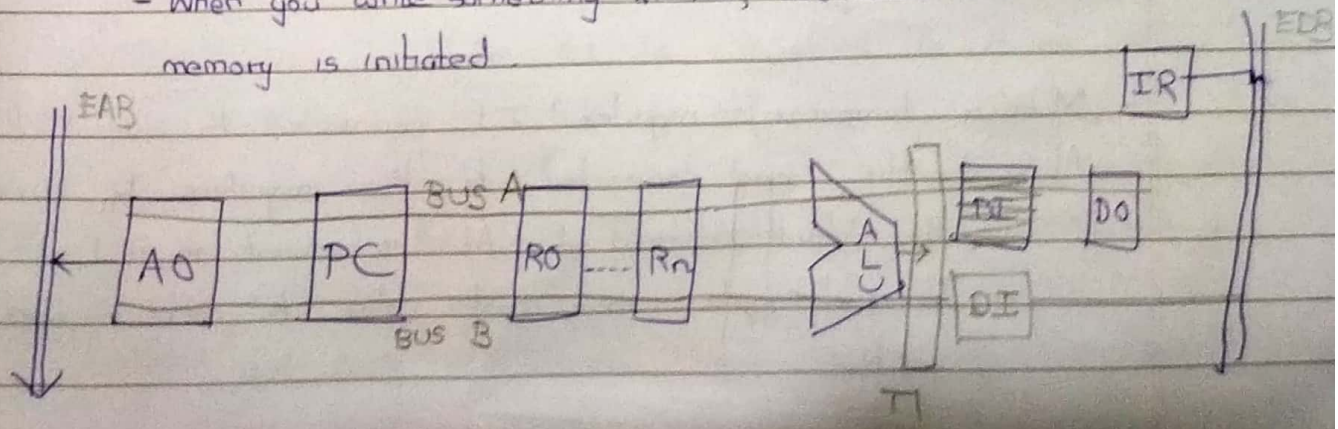
- Steering Logic :- Broadcast Bus.
  - To communicate with external world, we need an external address bus and external data bus.
  - The external address bus is interfaced via a buffer register known as 'Address out (AO)'. Similarly, external data bus is connected via Data out (DO) and Data In (DI)



* Zero flag implemented at output of ALU.



18/9 - When you write something in AO, the transmission to external memory is initiated.



Scanned by CamScanner

- Instruction Register (IR) - Holds the instruction currently being executed.

- Manufacturing comes with restrictions.

- Rules of operation as a result of restrictions :-
  1) In one cycle, bus can be used once (for transport from one source to destination only - eg - R1 to ALU)
  2) Putting value in A0 initializes transmission, and we receive corresponding data at the end of that cycle.

- Now, to execute $R_x + R_y$

☹ - We cannot move both R_x and R_y to inputs of ALU within one cycle. We cannot move R_x & R_y one after another because ALU inputs are not memorized. ALU cannot hold values. It is combinational logic. Do what you want within that clock cycle.

Meth 1 - Make a temporary register at ~~the~~ one input of ALU.
Move R_x to this TR in one cycle.
Move R_y to second input of ALU in second cycle.
Add.

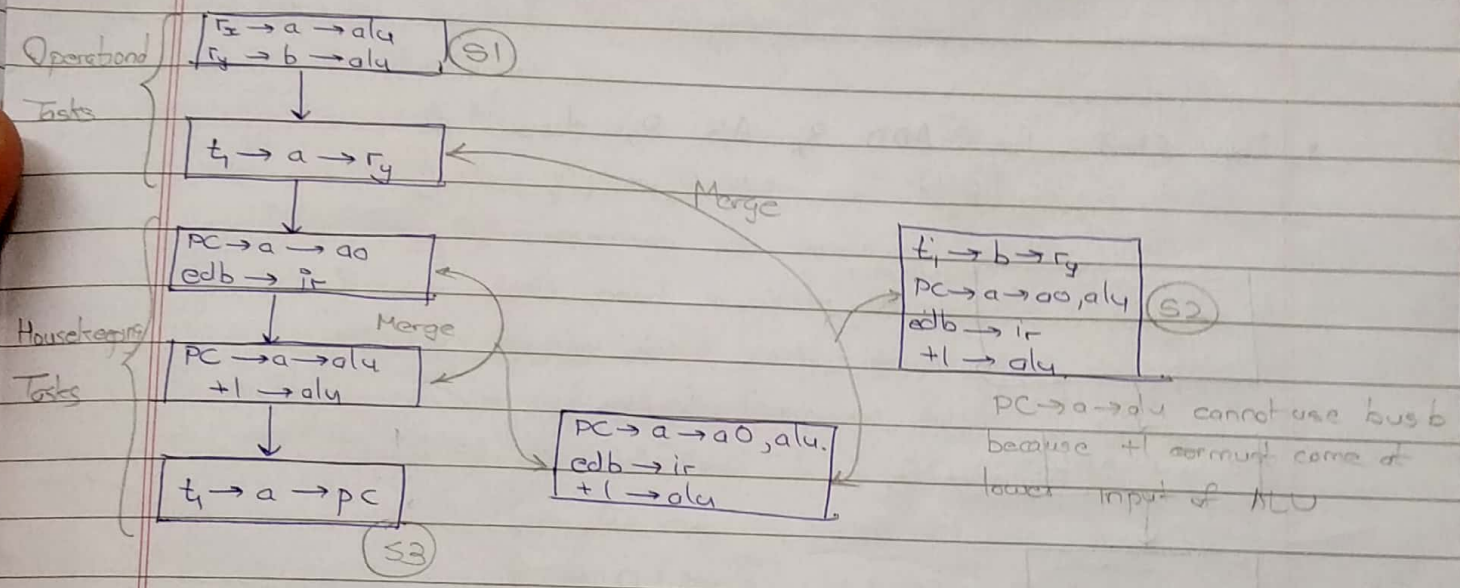
Meth 2 - Design another bus
☺ Move R_x and R_y simultaneously on different buses in one cycle

- Make a temporary register T1 connected to output of ALU directly, and connected to other registers by buses.
  - By default, output of ALU is stored in T1
  - In next cycle, move that value to correct destination via bus

Scanned by CamScanner

- In order to process addition with constants (-1, 0, 1) for INC or DEC, we have a new register at input of ALU that can take one of the above three

Step 1 • Flow chart for addition instruction
One block = 1 state (execution takes 1 cycle)



Step 2 • Write operational and house-keeping tasks separately. Merge them as well as you can
• One block contains concurrent tasks, to be executed parallelly

Step 3 • Give extra information about operation (flag updates)
- ALU operation in state S1 is supposed to update zero flag
S2 is not

Step 4 • Now, if you find that your flow chart has identical states, combine them
eg - ADD and SUB instructions have same S2, S3. Only S1 is different

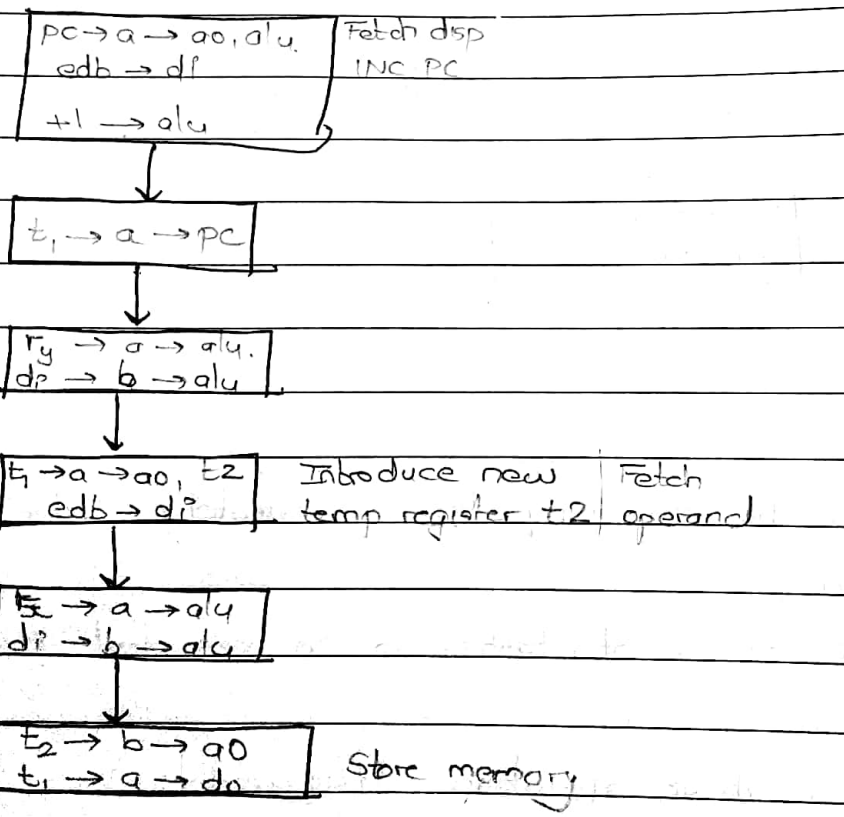
Scanned by CamScanner

- Overall Steps :-

Level 1	→	Merged Level 1	→	Level 2	→	Merged Level 2
Flow Chart		FC		FC		FC
Write operations & house keeping separately		Merge as many states as possible		Encode flag updates		Combine identical blocks across instructions (like ADD, SUB, ANL)

- Flow Chart for 'ADD R_x AB R_y disp'

$$[R_y + disp] \leftarrow R_x + [R_y + disp]$$



- If your μP supports 'm' type of instructions (~100), 'n' Addressing modes (~10), 'p' states per instruction per addressing mode on average, (~10)

Total no. of states = $m n p$ --- Very high :(

Scanned by CamScanner

- Assume instructions and addressing modes to be orthogonal
  - Any A/M can be used for any instruction.

- Operational Tasks

Addressing mode sequence (Fetching the operand(s))

Execution sequence (Execute)

- We want to make both sequences distinct.
  - Standardize :-
    1) Always place operand in di
    2) Always place address of destination in t2.
    - Standardization will make $O(m+n)$ instead of $O(mn)$
  - But we don't need to waste one cycle moving operand to di in Register direct mode, as registers are directly accessible
    - Do not standardize for register direct.
    - ∴ We will have to write separate execution sequences (one for register direct and one otherwise)
      ☆ ~ $O(2m+n)$

eg :- Addressing mode sequence for AI and AB

AI:

$ry \rightarrow a \rightarrow ao, t2$
$edb \rightarrow di$

AB:

$PC \rightarrow a \rightarrow ac, alu$
$ti \rightarrow alu$
$edb \rightarrow di$

$t1 \rightarrow a \rightarrow PC$

$ry \rightarrow a \rightarrow alu$
$di \rightarrow b \rightarrow alu$

$t4 \rightarrow a \rightarrow ao, t2$
$edb \rightarrow di$

Register direct does not have addressing mode test

Scanned by CamScanner

eg Execution sequence for AR and AI/AB

$r_x \rightarrow a \rightarrow alu$
$r_y \rightarrow b \rightarrow alu$
↓
$t1 \rightarrow a \rightarrow r_y$

$r_x \rightarrow a \rightarrow alu$
$di \rightarrow b \rightarrow alu$
↓
$t1 \rightarrow a \rightarrow do$
$t2 \rightarrow b \rightarrow oc$

- Now that you have separate addressing mode sequence and execution sequence, which of those will you preferentially use to merge with housekeeping tasks?

Ans - Execution sequence, because ~~all~~ register direct addressing mode does not have addressing mode sequence.

eg Addressing mode sequence for AM:- Memory indirect (double pointer)

$$[[R_y]] \leftarrow R_x + [[R_y]]$$

$r_y \rightarrow a \rightarrow oo$
$edb \rightarrow di$
↓
$di \rightarrow b \rightarrow oo, t2$
$edb \rightarrow di$

HW Write for other instructions from slide

HW Study level 2 flow charts with zero flag updation augmented

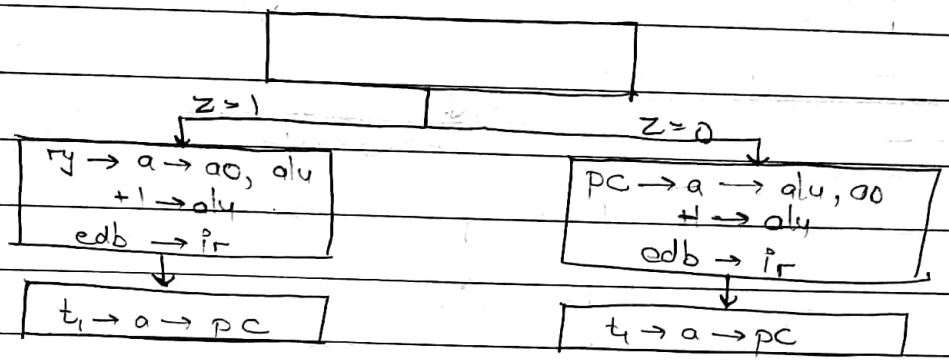
Scanned by CamScanner

All registers are n-bit

assuming R_y & PC have same no. of bits

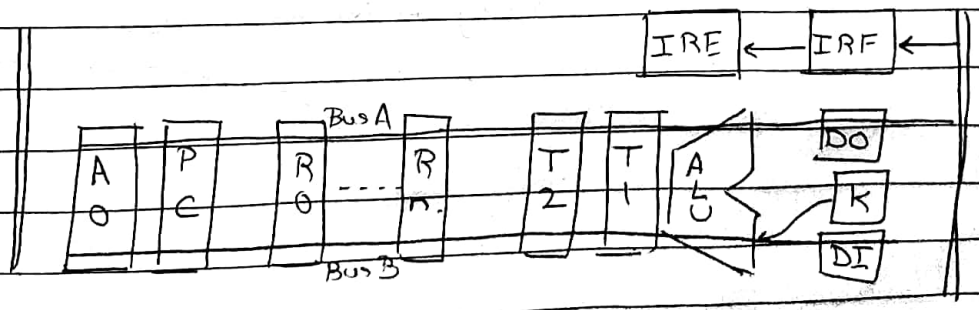
- BZ $[R_y]$:- If Z (flag) = 1, then PC ← R_y

  else PC = PC + 1



- This is execution sequence. There is no addressing mode sequence.
- BZ just chooses which instruction to execute next. No 'useful work' is done.

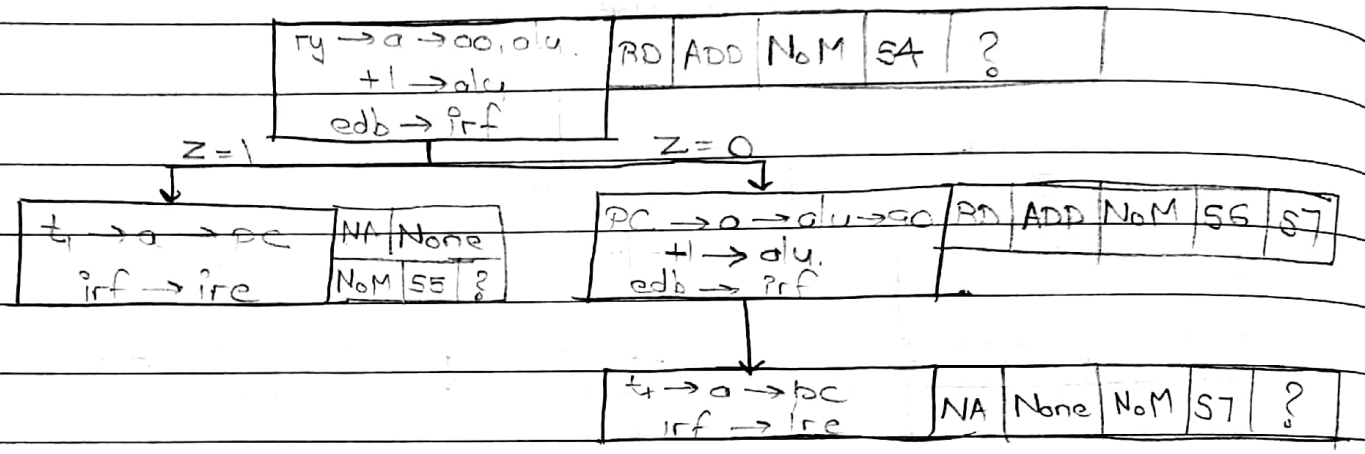
- Restriction on merging housekeeping tasks with execution tasks :-
  IR register should be updated only after previous instruction is completely executed.
  - Introduce IRF (Instruction register for fetching) and IRE (Instruction register for execution) in place of IR.
    - Move IRF to IRE when execution of previous instruction is complete.



P.T.O.

Scanned by CamScanner

- BZ (Modified) Level 2 flowchart.



- This is better because it has one state less.
- Observe :- ~70% of conditional jumps are actually taken. That is why, we moved Z = 1 state above and shortened time for Z = 1 branch preferentially.

→ Level 2 Flowchart.

Encode information about every state.

1 - Memory Access :- No access / Read / Write
2 - ALU operation :- OP / Add → operation as defined by instruction; addition
3 - Zero flag updation :- Modify / No modification
4 - Unique ID (of state)
5 - Next state ID

Scanned by CamScanner

eg ADD R_x AR R_y

$r_x \to a \to alu.$ $r_y \to b \to alu.$	NA, OP, Mod, S1, S2				
$t_1 \to b \to r_y$ $pc \to a \to ao, alu.$ $t_1 \to alu$ $edb \to irf$	RD Read	ADD	No-mod	S2	S3
$t_1 \to a \to pc$ $irf \to ire.$	NA	None	No-mod	S3	?

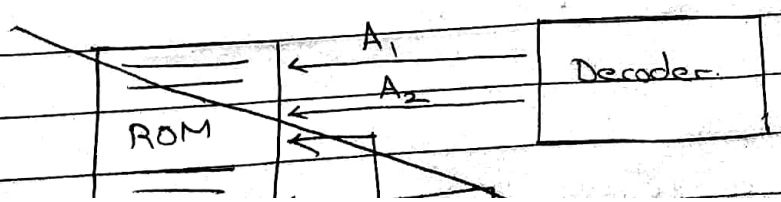
- See modified BZ for second example.

→ Decoder

Decoder of instruction must generate two addresses – one for first state of addressing mode sequence and one for first state of execution sequence.

- Assume, for every instruction, all states have consecutive unique IDs (S_i)

Better – Many states, like last instruction in both the above examples (S3, S5, S7) are identical and redundant.
Merge all such 'equivalent states'
Now, for every instruction, states will not have consecutive IDs.



Scanned by CamScanner

- Decoder gives :-
  - A1 :- ID of first state of addressing mode sequence execution.
  - A2 :-

- Possible values 'Next state' of each state can take :-

  IB SB DB BC.

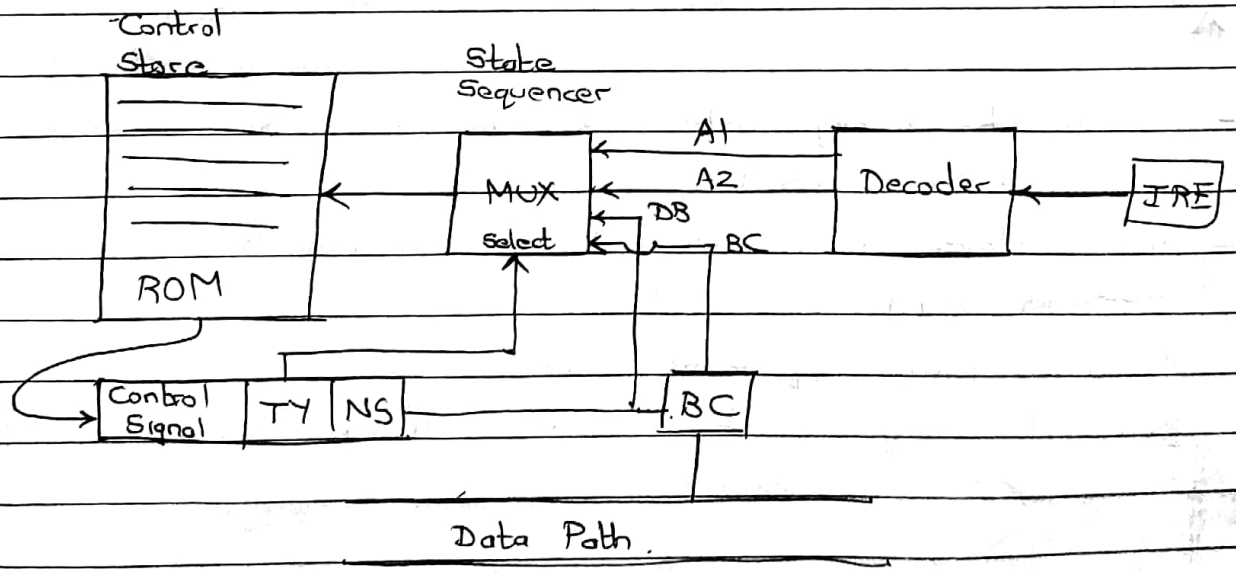
- IB = Instruction Branch

  Current instruction is completed, at last state of execution sequence Now fetch first state of next instruction (A1 of next instruction)

- SB = Sequence Branch

  'Next state' of last state of addressing mode sequence. Fetch the next state by A2.

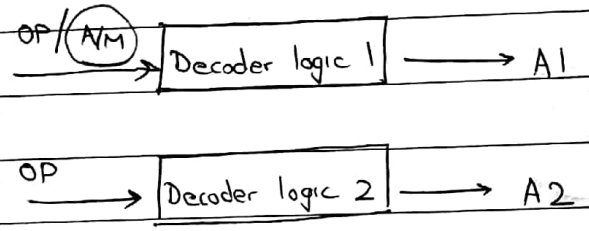
*



- Every state has its own (Control Signal, TY, NS) encoded.
  - TY :- 2-bits to choose from IB, SB, DB, BC.
  - NS :- ID of next state if ~~use~~ use DB or BC

Scanned by CamScanner

- If instruction does not have addressing mode sequence (register direct), A1 contains ID of execution sequence. A2 is 'don't care'. IB will directly lead to execution sequence.



These two logics can be combinationally

- DB = Direct Branch
Simply move to consecutively next state. The address (ID) of next consecutive state ~~st~~ is stored in NS.

- BC

→ Control Signal :-

A0	PC	Reg	T2	T1	ALU	D0	DT

Parts for

- Consider part for PC.
  - Through "all" the states, what all jobs does PC do?
    - NONE, pc → a, a → pc, b → pc
  - Since there are 4 jobs, part for PC is described in 2 bits.

Observe :- a → pc is used only once (addressing mode seq of B+D)

P.T.O.

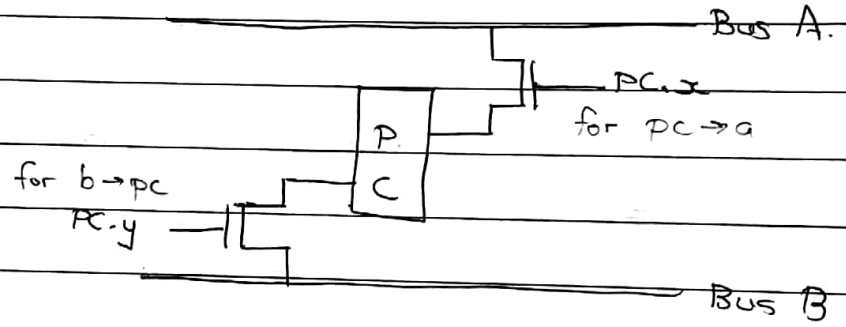
Scanned by CamScanner

Even in that state, bus b was lukkha.
Do not use $a \to pc$. Use $b \to pc$ only.

- If part for PC is xy

y \ x	0	1
0	NONE	$pc \to a$
1	$b \to pc$	X

- Since you have only 3 possible functions, use the (1,1) combination to simplify the decoder



- Consider T2

- All possible functions :-
  NONE, $t_2 \to a$, $t_2 \to b$, $a \to t_2$, $b \to t_2$

- $a \to t_2$ is used only once, when bus b is free.
  Use $b \to t_2$ instead
  - ∴ 2 bits of encoding are sufficient

-

y \ x	0	1
0	NONE	$t_2 \to a$ → $x\bar{y}$
1	$t_2 \to b$ → $\bar{x}y$	$b \to t_2$ → xy

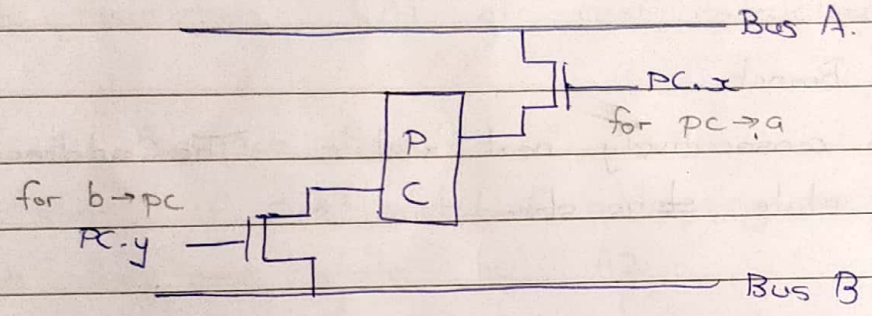
Scanned by CamScanner

Even in that state, bus b was lukkha.
Do not use $a \to pc$. Use $b \to pc$ only.

- If part for PC is xy

$y \backslash x$	0	1
0	NONE	$pc \to a$
1	$b \to pc$	X

- Since you have only 3 possible functions, use the (1,1) combination to simplify the decoder logic by K-Map



- Consider T2

- All possible functions :-
  NONE, $t_2 \to a$, $t_2 \to b$, $a \to t_2$, $b \to t_2$

- $a \to t_2$ is used only once, when bus b is free
  Use $b \to t_2$ instead
  - ∴ 2 bits of encoding are sufficient

-

$y \backslash x$	0	1
0	NONE	$t_2 \to a$ → $x\bar{y}$
1	$t_2 \to b$ → $\bar{x}y$	$b \to t_2$ → xy

~~Push :- Pre~~
Push :- Pre DEC/INC
Pop : Post INC/DEC

27/9

ALU Design.

Condition code register (Zero flag) is not modified

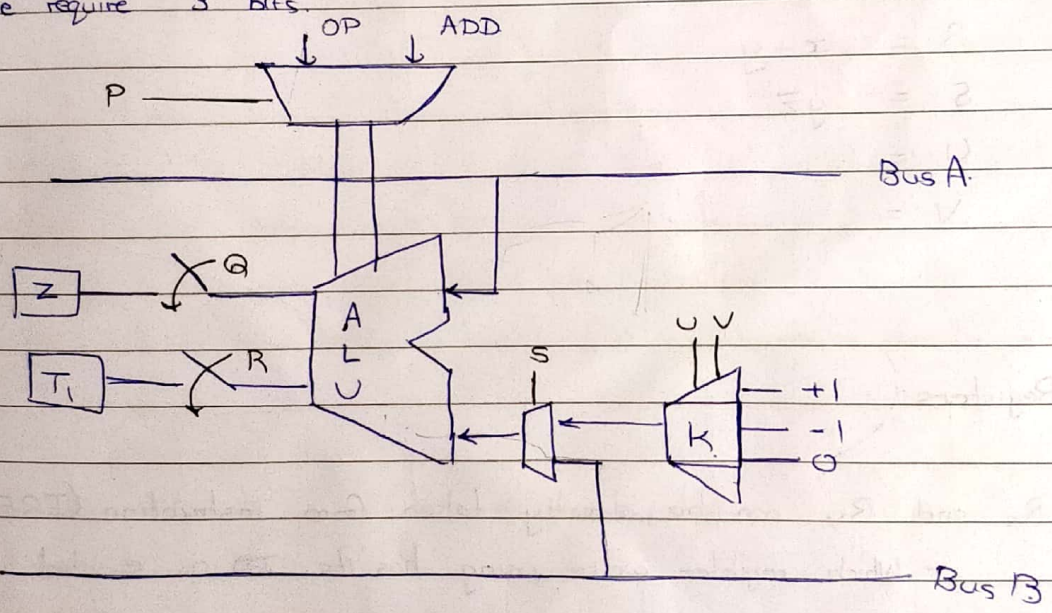
operation to be performed

Encoding (PPO) Functions :-

101	1)	a → alu,	+1 → alu,	add - n,	alu → t,	eg:- PC = PC + 1
111	2)		-1			eg :- SP = SP - 1
100	3)		0	- m		eg:- TEST, LOAD
010	4)		b	- n		eg:- Calculate B + D
110	5)			op - m		OP ∈ {ADD, SUB, NAND}
000	6) NONE					OP is taken from IRE

* Shift taken from IRE ≡ R_x, R_y, OP.

- We require 3 bits.



- Six control ~~signals~~ points
  - P chooses operation
  - Q decides if zero flag is to be updated.
  - R is open for NONE operation and closed otherwise
  - S chooses b/w bus B and constant k
  - U, V decide value of k.

- We can combine S, U, V and reduce one control signal.

- Encoding :- 3 bits (x, y, z)

x \ yz	00	01	11	10
0	NONE	d	d	b→alu add-n
1	0→alu add-m	+1→alu add-n	-1→alu add-n	b→alu op-m

Place functions in K-Map in a way to minimize encoding of control points

- Encoding of control points

$P =$

$Q = x\bar{z}$

$R = x + y$

$S = y\bar{z}$

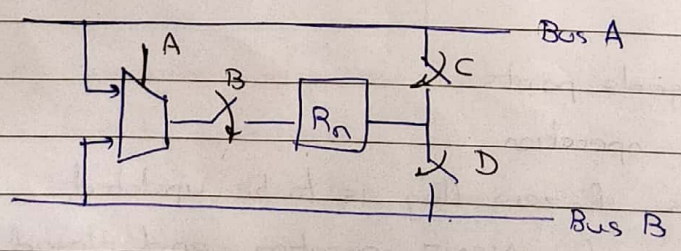
$U =$

$V =$

- Registers

- R_x and R_y can be directly taken from instruction (IRE)
  - Which register we're using has its ~~ID~~ no. encoded in instruction

-



- A chooses ~~wheth~~ which bus to write from
  B chooses whether to read
  C write to bus A
  D B

1/10

→ ISA ⟶ Implementation (Gate-level netlist)

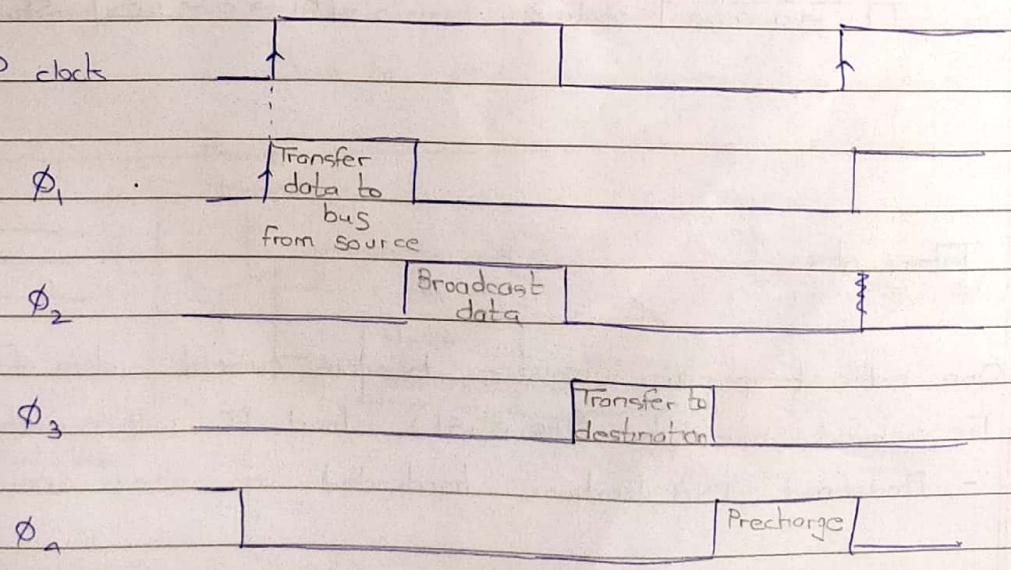
1) HFC
2) Level 2 ~~mta~~ merged FC
3) Controller development using microcode ROM.

⟶ Time taken to transfer signal from one point on bus to another.
- Find R, C, find $\tau = RC$, use length of bus between the two points

* We don't need to worry about time taken to broadcast if steering logic is point-to-point.

- Charging takes longer than discharging
  - 'Precharge' :- Make all lines of bus '1' in steady state. Whenever you want to transfer a '0' on any line, give low voltage to that line.
  eg 8 bit bus has 8 lines (1111 1111 ~~b~~ in steady state)

- Multiphase Clocking :- 4 phase clocks



Scanned by CamScanner

- If, in the given cycle, we want to fetch from memory, address will reach 'ao' at end of ϕ_3. We want data to reach 'di' at end of ϕ_2.
  - Memory performs a 'handshake' when data successfully arrives at di.
  - Sometimes memory might be slow.
    - Only for memory accessing states,
      - Meth 1 - Shift rising edge of system clock ahead by ϕ_2' until handshake is done (asynchronous)
      - Meth 2 - Keep system clock as it is. ~~(synchronous)~~ Perform memory-using states in 2 clock cycles.

pc → a → ao, alu +1 → alu alu → t_1 edb → d	⇒	pc → a → ao, alu +1 → alu alu → t_1 — Cycle 1 edb → di — Cycle 2

eg B+D N/M.

Fast memory

pc → a → alu, ao +1 → alu alu → t1 edb → di	State 1
t1 → a → pc	State 2

Slow memory (takes 2 cycles)

pc → a → alu, ao +1 → alu alu → t1	State 1
edb → di t1 → a → pc	State 2

→ Interrupts.

- One extra temporary register to store current value of PC.
- For vectored interrupts (like 8051), load PC with correct location of ISR.
  - Predefined ISR locations hardcoded in other (new) registers.

Scanned by CamScanner

Overdesign :- Using more resources than minimum required

T/F • Controller efficiency does not indicate optimal use of resources

3/10 • 100% CE implies overdesign – False

• Sharing of states cannot decrease CPI.

* Controller Efficiency $\triangleq$ $\frac{\text{No. of states ~~in an instruction~~ accessing memory}}{\text{No. of states ~~in the instruction~~}}$

• If an ~~state~~ instruction accesses memory 'n' times, it cannot run in less than 'n' cycles.

– No. of memory accesses provides lower bound on no. of cycles.

—x—x—

* If, at any point in execution, you come across an erroneous instruction (eg – divide by zero, jump to non-existent address, etc), the μP raises an exception.

Two possible solutions :- 1) Reset and restart execution. (→ 0)

2) Ignore current instruction, move ahead. (JMP +3)

—x—x—

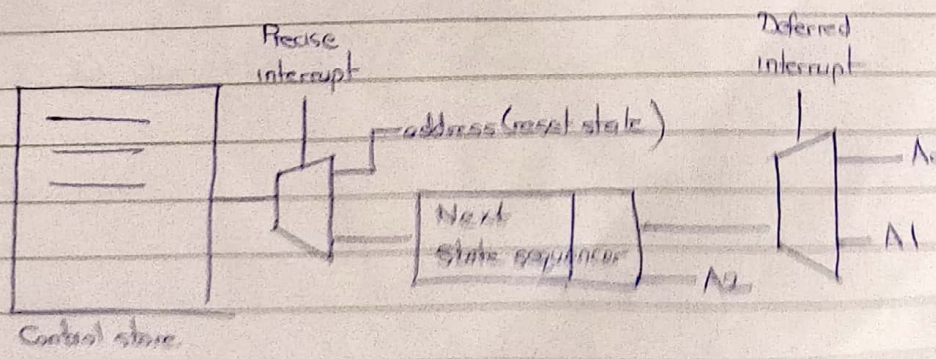
* Apart from all discussed states, there is one 'Reset State', which is executed after boot-up. Its job is to bring first instruction (from a predecided location (0000 ~~b~~ say)) into IRE.

—x—x—

Interrupts CNTD

• Precise Interrupts :- Pause current instruction. Move to ISR in next cycle

Deferred :- Move to ISR after current instruction is done.



Scanned by CamScanner

- Assume no nested interrupts
- Assume Reset is the only precise interrupt.

- Precise interrupt :- Address contains hard-coded ID of Reset State
- Deferred interrupt :- Address contains ID of state that handles interrupts :-

$$\boxed{\begin{array}{l} pc \rightarrow a \rightarrow t3 \\ Intr \rightarrow b \rightarrow pc \end{array}}$$

Introduce new temporary register to store pc

Intr will contain address of appropriate ISR.

Addresses of all ISR are hard-coded in new inaccessible registers. Based on the interrupt that occurs, appropriate address is moved to Intr.

Scanned by CamScanner

II RISC DESIGN

* Application specific instruction set processor :-
- Strikes a balance between flexibility of general μP and speed of a circuit.
- Application-specific :- eg - Fast implementation of Fourier Transform for image processing.

→ RISC
- Small no. of instructions
- Only few addressing modes
- Simple design
- All ~~arithmetic~~ instructions other than arithmetic, logical and LOAD/STORE instructions must use register direct addressing ~~or immediate~~.
- Only Load and Store instructions can access memory
- We can afford to have fixed length encoding.

  eg - If there are 32 registers (5 bits),

OP	OPR1	OPR2	DEST
7	5	5	5

- Arithmetic & logical instructions can also use immediate addressing
  Load/Store can use any addressing mode.

• If 32-bit ~~dt~~ instructions are decided,

Register direct (R)

OP	OPR1	OPR2	DEST	Wasted
6	5	5	5.	

Immediate (I)

OP	OPR1	DEST	16-bit number
6	5	5	

Scanned by CamScanner

II RISC DESIGN

* Application specific instruction set processor :-
  - Strikes a balance between flexibility of general μP and speed of a circuit.
  - Application-specific :- eg - Fast implementation of Fourier Transform for image processing.

→ RISC
- Small no. of instructions
- Only few addressing modes
- Simple design
- All ~~arithmetic~~ instructions other than arithmetic, logical and LOAD/STORE instructions must use register direct addressing ~~or immediate~~.
- Only Load and Store instructions can access memory
- We can afford to have fixed length encoding.

  eg - If there are 32 registers (5 bits),

OP	OPR1	OPR2	DEST
7	5	5	5

- Arithmetic & logical instructions can also use immediate addressing
  Load/Store can use any addressing mode.

• If 32-bit ~~dt~~ instructions are decided,

Register direct (R)

OP	OPR1	OPR2	DEST	Wasted
6	5	5	5.	

Immediate (I)

OP	OPR1	DEST	16-bit number
6	5	5	

Scanned by CamScanner

- No memory direct in RISC because too many bits required for memory address
- Say all registers are ~~32 bit~~ n-bit. n could be 32

- Load/Store supports Base + Index and Base + Displacement

LOAD Rd, (RS1) RS2
≡ Rd ← [RS1 + RS2]
'R-type instruction'

LOAD Rd, (RS1) Imm
≡ Rd ← [RS1 + Imm]
'I-type instruction'
Imm is 16-bit. Max displacement = 65,536

→ 3-types of load/store instructions :- LW :- Load word (32-bits)
LH :- half-word (16-bits)
LB :- byte (8-bits)

→ Change in control flow instructions

We have studied branching on flag values
Other possibility :- No flags at all. #different flavour
Condition is computed by same instruction and decision is taken accordingly

eg Branch if not equal :-

```
bneq RS1, RS2, offset
≡ if (RS1 == RS2) {
        PC = PC+4 }
  else {
        PC = ~~PC+4+offset*4~~ PC+4+offset*4 }
```

one instruction is 4 bytes

Jump 'offset' no. of instructions ahead.

offset is corresponding to next instruction, not current instruction

- Similarly ~~the~~ Branch if equal beq

- Jump
  - J loc ← 26 bit

Design new type of instruction :- J-type

OP	Imm (26-bit)
6	

$PC = PC + 4 + loc * 4$

- Register direct jumping is also possible :- JR R1 ---- R-type instruction
  $PC \leftarrow R1$

- Calling a subroutine :- 'Jump and link' :- JAL loc
  - Uses J-type instruction.

→ $R31 \leftarrow PC + 4$:- Store location of next instruction in R31. Define R31 as return register.

$PC = PC + 4 + loc * 4$

- All jumping, offsetting is done wrt next instruction.

- Jump and link to register :- JALR R1.

$\equiv$ $R31 \leftarrow PC + 4$

$PC \leftarrow R1$

→ SLE :- Set if less than equal to

SLE R1 R2 R3

if $(R1 <= R2)$: $R3 = 01$

else $R3 = 00$

SLEI R1 R2 I

if $R1 \leq I$ ----

Similarly we have SLT, SLTI, SGET, SGET, SGT, SGTI

- These instructions compensate for lack of flags.

→ Structured Instruction format.

x	
6	

- If $x = 000000$ → R-type instruction.
  OP is stored as last 6 bits (bits 27 to 32)
  ∴ 64 R-type instructions possible.

~~eg =~~

- If $x \neq 000000$ → I or J type instruction
  Total 63 I or J type instructions possible.

4/10

- Add/Sub/NAND/shift :- R-type
  LW, SW, BEQ :- I-type
  J :- J-type.

000000	RS1	RS2	DEST		OP
OP	RS1	DEST	Imm 16		
OP	Imm 26				

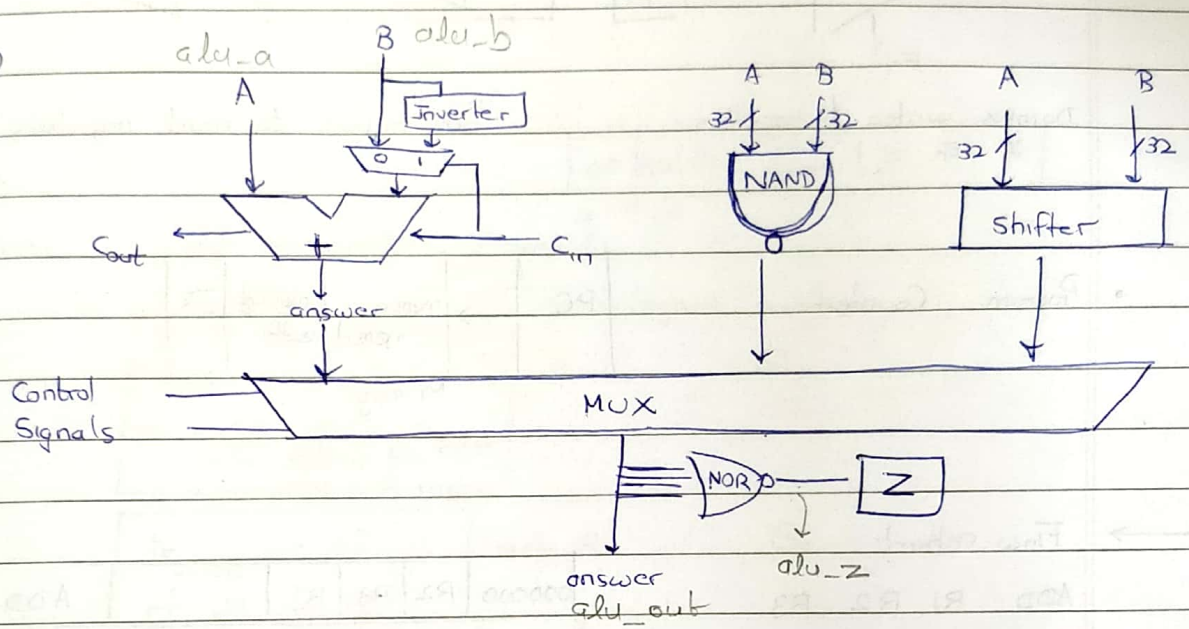
- In RISC, there is almost no possibility of merging HKT with operational tasks, because all instructions do similar stuff and resources are already utilized.
  - Either of HKT or OT can be performed first.
    - We will do HKT first, unlike what we did in CISC
      #different flavour

Scanned by CamScanner

→ Implementation ← HFC — ISA

→ Minimal Datapath
- 32 registers (programmer-accessible)
- Steering Logic can be point-to-point or broadcast
  - Point-to-point transport # different flavour

• ALU



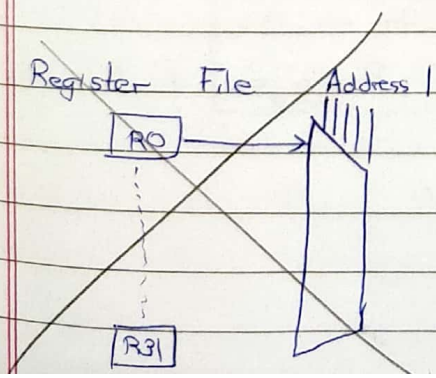
- Add / Subtract :- $C_{in} = 0$:- Add
  $C_{in} = 1$:- Subtract (by 2's complement)

* Add with carry is not supported

- Shifter :- 5 LSBs of B will tell how much to shift A by.

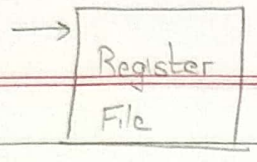
- NOR :- Checks if output is 0 (NOR = 1)
  Z bit is NOT visible to programmer

• Register File

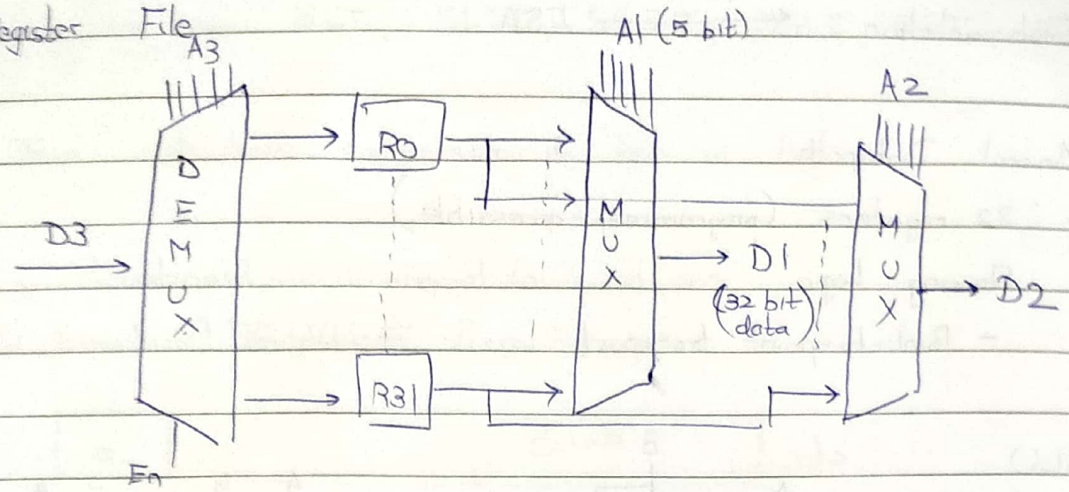


Scanned by CamScanner

All 'writing to registers' happens at clock edge



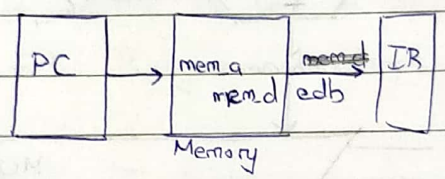
Register File



Demux writes to registers if En = 1

Two muxes to read registers

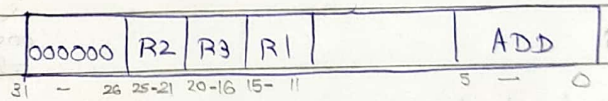
Program Counter



mem_a = a0
mem_d = edb

Flow chart

ADD R1 R2 R3



HKT	PC → mem_a/a0/~~edb~~, alu_a mem_d/edb → ir +4 → alu_b alu_out → pc	S1 RD, Add, S1, S2 alu_out is not just connected to T1
CT	ir_{21-25} → rf_a1 ir_{16-20} → rf_a2 rf_d1 → t1 rf_d2 → t2	S2 Need temporary registers t1 & t2, because we cannot serially use ALU in the same state we read from register file in. S2,
	t1 → alu_a t2 → alu_b alu_out → t1	S3 S3
	t1 → rf_d3 ir_{11-15} → rf_a3	S4 S4

In one state,
one use of ALU
or
one use of memory
or
one use of register file
in series

Parallel use of ALU, RF, memory is allowed.

Scanned by CamScanner

8/10

- Time-consuming tasks (only one possible (serially) in one state) :-
  ALU use, Register File RD/WR, Memory RD/WR.

- Clock Cycle time = ~~Clock~~ max (time for individual states)
  - Hence we do not want to perform timeconsuming tasks serially in one state.

31 - 26	20 - 25	16 - 21	
op	RS1	RD	Imm 16

→ LOAD

B+D :- RD ← [RS1 + Imm16]

Signed number

HKT	PC → mem_a, alu_a mem_d → ir +4 → alu_b alu_out → pc	S5	S1
OT	ir_{21-25} → rf_a1 rf_d1 → t1	Fetch RS	S6 S2

Now, RS is 32-bit. ∴ Use sign extender to extend Imm16 to 32 bits ⇒ If MSB is 0, pre-append sixteen '0's
1 '1's

	t1 → alu_a ir_{0-15} → SE16 → alu_b alu_out → t1	S7 Find RS1 + Imm16	S5
	t1 → mem_a mem_d → t1	Fetch operand	S8 S6
	t1 → rf_d3 ir_{16-20} → rf_a3	Write to RD	S9 S4

Scanned by CamScanner

→ STORE

HKT

PC → mem_a, alu_a +4 → alu_b mem_d → ir alu_out → pc	S10	S1

OT

ir_{21-25} → rf_a1 ir_{20-16} → rf_a2 rf_d1 → t1 rf_d2 → t2	S11	S2 Store RS in t1 and RD in t2

t1 → alu_a. ir_{0-15} → SE16 → alu_b alu_out → t1	S12	S5

t1 → mem_a t2 → mem_d	S13	S7

→ BEQ R1, R2, Im16

if (R1==R2) { PC = PC+4 + 4* Imm16 }
else { PC = PC+4 }

HKT

	S14	Stores PC+4 in PC S1

OT

ir_{21-25} → rf_a1 ir_{20-16} → rf_a2 rf_d1 → t1 rf_d2 → t2	S15	S2

t1 → alu_a t2 → alu_b alu_z → Z	S16	S8

PC → alu_a ir_{0-15} → SE16 → 25 → alu_b If (z=1) { alu_out → PC }	S17	S9

Shift left twice to multiply by 4

Use some concoction of MUX to execute if statements

Scanned by CamScanner

→ JUMP

OP	Imm 26

HKT [----] S1

OT [blank] S2 One state is blank for time to decode the instruction. In other instructions, decoding was done in parallel

~~ad~~ [pc → alu_a
ir_{0-25} → SE26 → alu_b
alu_out → pc] S10

9/10

→ Level 2 flowchart :- Memory Access, ALU operation, current state, next state.

- Merge ~~tasks~~ states doing similar tasks

Note that second state of LOAD and JUMP is a subset of second state of ADD which is same as second states of STORE and BEQ. ~~and JUM~~
Merge all these (S2, S6 .....)
Even for LOAD, use same second state as ADD.

- States 1 and 2 have become common to all instructions.
  The last state of each instruction has S1 as next state, which has S2 as next state.
  After S2, next state is different.

- We can merge S4 and S9, and choose between ir_{15-11} and ir_{16-20} using a MUX, at the cost of an additional control signal.

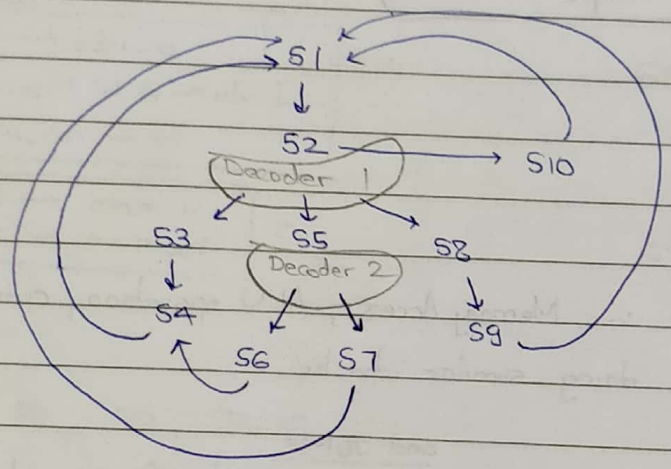
- Merge S12 in STORE and S7 in LOAD.

Scanned by CamScanner

- Wherever there are multiple possible 'next states' (S2, S5, etc), we require a decoder that decodes current instruction

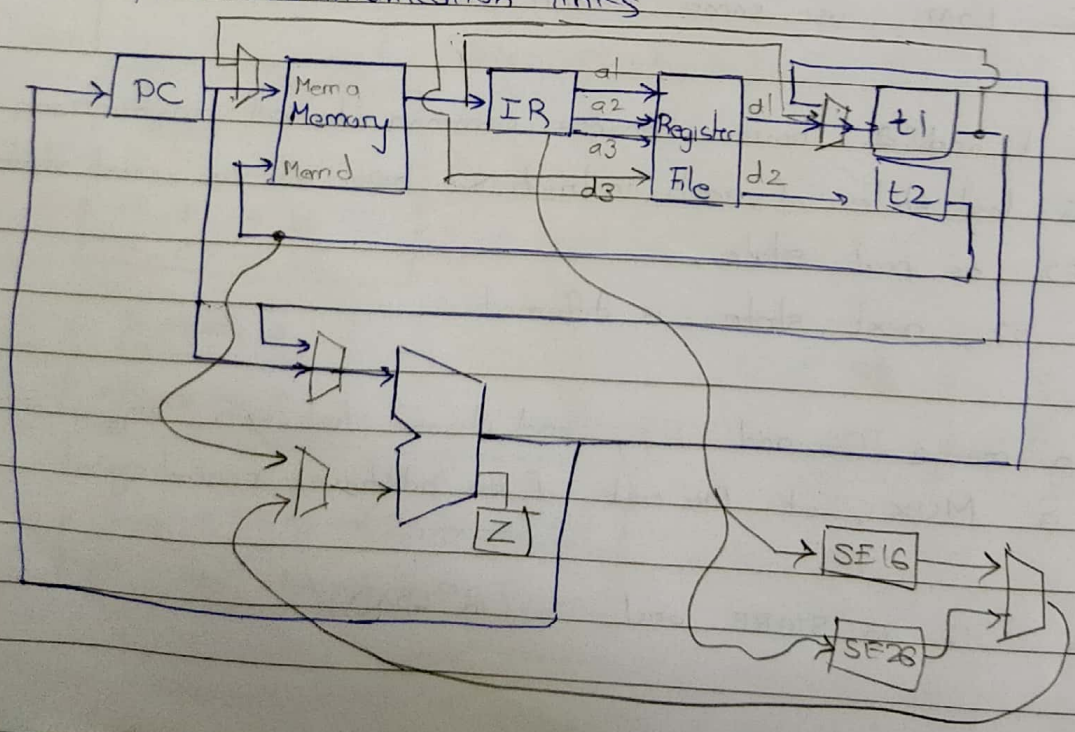
- We can even merge S10 and S2, with a modification that pc will be updated only if the instruction decoder says that current instruction is indeed JUMP.
  This can be done because alu is idle in S2

S3 - R type
S5 - I type
S8 - J type



We will use a non-microcode style FSM #different flavour

→ Point to point communication links



Scanned by CamScanner

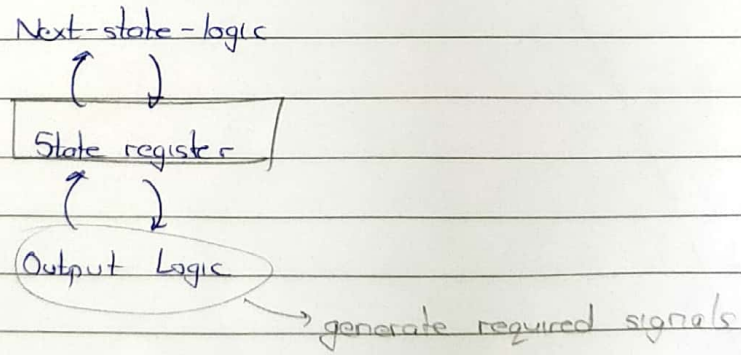
Basically → Go over all states. Draw the lines that you need.
Use MUXes at whichever inputs you need.

11/10

→ Control signals

1. Memory :- $\overline{RD}$, $\overline{WR}$ (Active low)
2. Register File :- WR
3. IR :- Enable bit
4. MUX at memory input :- 1 bit
5. " " T1 " :- 2 bits
6. " " ALU input 1 :- 1 bit
7. 2 :- 1 bit
8. SE output :- 1 bit
9. Choose ALU operation :- 2 bits
10. PC :- Enable bit ?
11. Updating Z flag :- 1 bit

- 10 states encoded in 4 bits



Scanned by CamScanner

→ Next state logic

```
process (current state, decoder 1, decoder 2)
    when(S1)
        next state = S2
    when (S2)
        next state = decoder 1
        :
        :
    when (others)        to ensure no sequential circuits. (tutorial)
        next state = S1
```

→ Output Logic

```
process (                    )
    when (S1)
        //Specify ALL control signals.
    :
    :
```

→ 3rd process :- To instantiate flip flops.

15/10

* Quiz

- Average CPI = Weighted mean of CPIs for different type of instructions

  where weight = frequency of occurrence of each type of instructions

* Say, memory R/W takes 3 ns
ALU operation 2.5 ns
RF R/W 2 ns

Then, clock cycle chosen should be at least 3 ns.

→ Single cycle implementation
- Wasn't possible in CISC (complex & variable-length instructions)
- Possible in RISC, because state transitions are simple and similar.

$S \rightarrow T \rightarrow T' \rightarrow T'' \rightarrow S' \Rightarrow S \rightarrow S'$

• Single cycle implementation :- CPI = 1

eg Arithmetic/Logical instructions

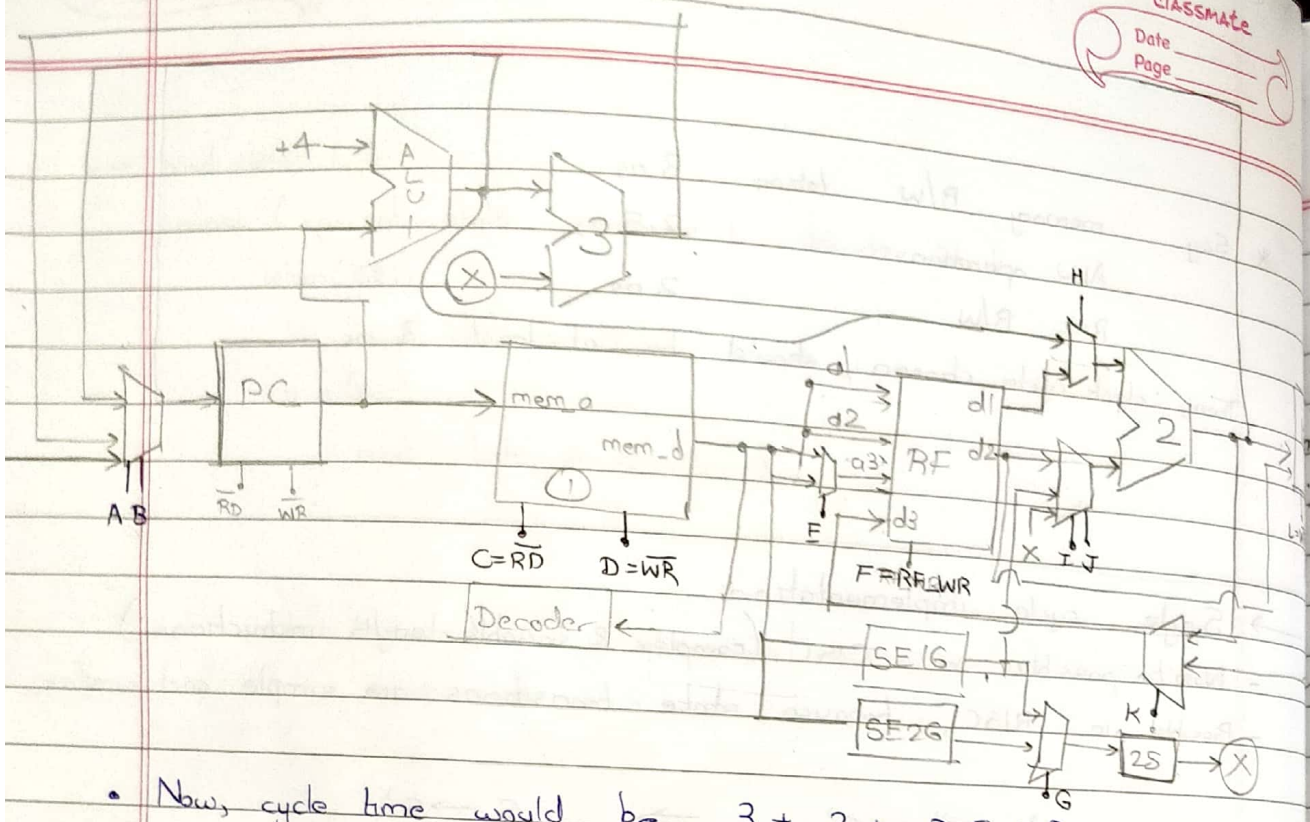
When we do entire instruction in single cycle, there cannot be storage in between because writing happens at clock edge

pc → alu1_a, mem_a
+4 → alu1_b
alu1_out → pc
mem_d_{21-25} → rf_a1
mem_d_{16-20} → rf_a2.
rf_d1 → alu2_a.
rf_d2 → alu2_b
alu2_out → rf_d3
mem_d_{11-15} → rf_a3

• Controller is no longer an FSM. It is a combinational logic that generates appropriate control signals based on value of opcode.

Scanned by CamScanner

a3 inputs :- 11-15 OR 16-20



- Now, cycle time would be $3 + 2 + 2.5 + 2 = 9.5$ ns

  Mem read + alu1 | RF read | alu_2 | RF_write

- All signals will stabilize by 9.5 ~~nsec~~ ns.
  - PC and RF will be updated at clock edge at the end of 9.5 ns.

eg LOAD

pc → alu1_a, mem1_a

+4 → alu1_b

alu1_out → pc

$mem1_d_{21-25}$ → rf_a1

$mem1_d_{0-15}$ → SE16 → alu2_b

rf_d1 → alu2_a

alu2_out → mem2_a

mem2_d → rf_d3

$mem1_d_{16-20}$ → rf_a3

Mem1 for code

Mem2 for data

, ALU3 is adder only

eg STORE

pc → alu1_a, mem1_a

+4 → alu1_b

alu1_out → pc

mem1_d_{21-25} → rf_a1

mem1_d_{0-15} → SE16 → alu2_b

rf_d1 → alu2_a

alu2_out → mem2_a

mem1_d_{16-20} → rf_a2

rf_d2 → mem2_d

* ~~Control~~

eg BEQ R1, R2, Imm16

pc → mem1_a, alu2_a

+4 → alu1_b

mem1_d_{20-25} → rf_a1

mem1_d_{16-20} → rf_a2

rf_d1 → alu2_a

rf_d2 → alu2_b

alu1_out → alu3_~~out~~ a

mem1_d_{0-15} → SE16 → ~~alu3_b~~ 2S → alu3_b

if (alu2_z) { alu3_out → pc }

else { alu1_out → pc }

P.T.O

Scanned by CamScanner

eg JUMP

pc → mem1_a, alu2_a

+4 → alu1_b

alu1_out → alu2_a.

mem1_d_{0-25} → SE 26 → 2S → alu2_b

alu2_out → pc.

→ Control Signals.

	A	B	C	D	E	F	G	H	I	J	K	L	M
000000 (R type)	0	0	0	1	1		0	1					
⋮													

Simplify ~~to find~~

eg D is always 1

→ Cycle Time

- LOAD takes most time (costliest path)

$$\underset{\text{mem1}}{3} + \underset{\text{rf}}{2} + \underset{\text{alu}}{2.5} + \underset{\text{mem2}}{3} + \underset{\text{rf}}{2} = 12.5\text{ ns}$$

→ Performance = # of instructions × $\underbrace{\text{CPI}}_{=1}$ × Cycle time

- In single cycle implementation, performance does not depend on fractions of different type of instructions.

Scanned by CamScanner

- Performance :- Arithmetic / Logical :- 50%
  Load :- 20%
  Store :- 10%
  ~~Branch~~ BEq :- 15
  Jump :- 5%

For multicycle, Performance = N × ~~[illegible] 12.45 ×~~ 4.15 × 3ns
= 12.45

single cycle = 12.5

- Depending on fractions of instruction types, either of single cycle or multicycle implementation.
- In spite of fewer registers, single-cycle is more expensive overall in terms of hardware.
- Single cycle :- Lowest possible CPI (1)
  Multi cycle cycle time.
- Possible to strike a balance :- eg- 2 cycle implementation.

→ Pipelining.

- Begin next instruction before previous is over, using the idle resources.
- In RISC, subtasks in every instruction are very similar - fetch instruction, fetch operands, alu operation, etc.

- It is possible to get least possible cycle time (like multicycle)
  - If we assume all instructions to be independent, then we can do pipelining and then we will get cycle time equal to max{sub-task time}

Scanned by CamScanner

→ Subtasks

1. R-type :- Instruction fetch, Instruction Decode / Operand Read, Execute, Write Back
2. LOAD :- IF, ID/OR, EX, Memory Read, WB
3. Store :- IF, ID/OR, EX, Mem
4. BEQ :- IF, ID/OR, EX.
5. JUMP :- IF, ID

exhaustive set of sub-tasks

- First 5 cycles, no instruction will be completed. (or 4)
  After that, one instruction will be completed every cycle.
  (for independent instructions and if instructions are consecutive from top to bottom (no control flow changes in between))

→ Performance $= N \times \underbrace{1}_{CPI} \times \underbrace{3ns}_{\text{Cycle time}}$

Scanned by CamScanner

- Performance :- Arithmetic / Logical :- 50%
  Load :- 20%
  Store :- 10%
  ~~Branch~~ BEq :- 15
  Jump :- 5%

For multicycle, Performance = N × ~~[illegible]~~ 4.15 × 3ns
= 12.45

single cycle = 12.5

- Depending on fractions of instruction types, either of single cycle or multicycle implementation.
- In spite of fewer registers, single-cycle is more expensive overall in terms of hardware.
- Single cycle :- Lowest possible CPI (1)
  Multi cycle cycle time
- Possible to strike a balance :- eg - 2 cycle implementation.

→ Pipelining.

- Begin next instruction before previous is over, using the idle resources.
- In RISC, subtasks in every instruction are very similar - fetch instruction, fetch operands, alu operation, etc.

- It is possible to get least possible cycle time (like multicycle)
  - If we assume all instructions to be independent, then we can do pipelining and then we will get cycle time equal to max{sub-task time}

Scanned by CamScanner

Mealy :- Output Logic & Next-state logic depend on both input and current state

Moore :- NSL is same as before.
OL depends only on current state.

→ Subtasks

1 R-type :- Instruction fetch (IF), Instruction Decode / Operand Read (ID/OR), Execute (EX), Write Back (WB)

2 LOAD :- IF, ID/OR, EX, Memory Read (Mem), WB

3 Store :- IF, ID/OR, EX, Mem

4 BEQ :- IF, ID/OR, EX.

5 JUMP :- IF, ID, EX

exhaustive set of sub-tasks

- First 4 cycles, no instruction will be completed.
  After that, one instruction will be completed every cycle.
  (for independent instructions and if instructions are consecutive from top to bottom (no control flow changes in between))

→ Performance = $N \times \underbrace{1}_{CPI} \times \underbrace{3ns}_{\text{Cycle time}}$

CPI ≈ 1 Cycle time = max(subtask time)

- Every instruction will go through all 5 stages
  IF → ID/OR → EX → Mem → WB
  eg - JUMP will waste time in Mem & WB stages

  - Wasting time :- Do the job in each state, but don't write to any register (using control signals on registers)

  - Making 5 stages common for all instructions converts each stage from a ~~Mealy~~ Moore machine (which does not depend on instruction) to a ~~Moore~~ Mealy machine (which does ~~not~~)

Moore :- Control signals generated depended only on current state.
Mealy and op-code

- To use pipelining, all instructions must use similar type of stages, and in same order too.

Cycle 1
2
3
4
5

	IF	ID/OR	EX	Mem	WB
Cycle 1	I_1				
2	I_2	I_1			
3	I_3	I_2	I_1		
4	I_4	I_3	I_2	I_1	
5	I_5	I_4	I_3	I_2	I_1

Latency

→ How do you design a pipeline (determine the required subtasks)?

∴ - Start from multicycle :- We will have minimal resources used.
Could be an issue if resource is shared by subtasks

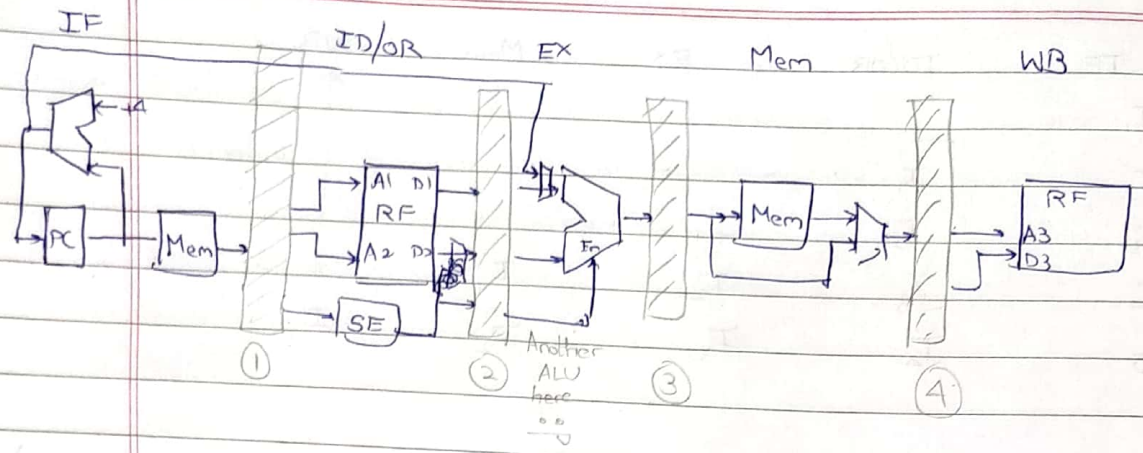
eg - In multicycle, same memory is used by IF & Mem.

- ∴ Start with single cycle to get maximal resources.

→ We generate control signals through combinational decoder of single cycle implementation (like always), and "remember" them in a register
The control signals for each instruction must be propagated consecutively to each task along with the instruction itself.
The respective subtask uses whichever control signals it needs.
- So every subtask will have a register (to store control signals) of the size equal to number of control signals flowing from previous state to that state
~~Register in subtask 1 will have size equal to total number of control signals.~~
- ~~PC is also propagated through the stages, as it will be required for returning control flow changes~~

Scanned by CamScanner

Instructions move in lock step manner (= FIFO, no jumping)



- Signals crossing interface after IF :
  Instruction (32 bit), PC+4 (32 bit), etc

- Interface 2 :-
  PC+4 (32-bit), Operand 1 (32-bit), Operand 2 (32), Sign extended Imm (32), ALU control signals, destination register ~~reg~~ address, etc

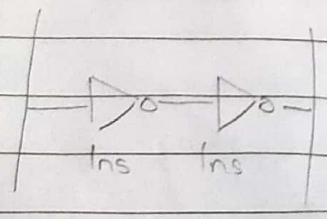
- Interface 3 :-
  ALU output (32), A3 address etc

- Interface 4 :-
  Destination register address, data to be written, etc

→ How many stages to divide the instruction into?

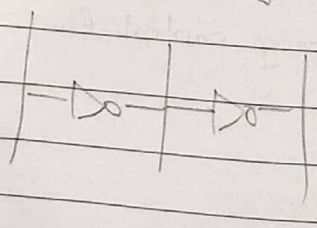
- More no. of (smaller) stages will give lower cycle time ☺
  - Becomes costlier, as you need more registers for interfaces ☹
  - Performance ↑, CPI still ≈ 1

One stage



Cycle time = 2ns

Two stage



Cycle time = 1ns

- Choose the value of no. of stages (n) that maximize $\frac{\text{Performance}}{\text{Cost}}$

$$\text{Cost} = \underbrace{G}_{\text{cost without pipelining}} + (n-1)\underbrace{k}_{\text{avg cost of interface registers}}$$

$$\text{Performance} = \frac{1}{\underbrace{\frac{T}{n}}_{\text{time without pipelining}} + \underbrace{q}_{\text{Some small delay (inherent)}}}$$

$$* \quad \frac{\partial}{\partial n} \frac{1}{\left(\frac{T}{n} + q\right)\left(G + (n-1)k\right)} = 0$$

22/10

→ Our assumptions :- (Hazards)

eg Dependent instructions :- I1 : $r_1 = r_2 + r_3$

I2 : $r_4 = r_1 + r_5$

In ID/OR stage, I2 will read incorrect r_1

- Detecting violation of independence of consecutive instructions :-

Source registers in ID/OR stage instruction = Destination register of Ex stage instruction.

i.e → $(RS1^{ID/OR} == RD^{Ex}) \;||\; (RS2^{ID/OR} == RD^{Ex})$

- If this violation occurs, 'freeze' I2 in the ID/OR stage for 3 cycles (until I1 completes). Then resume.
  - On every violation, we are losing 3 cycles
  - eg - If 10% instructions have this 'immediate' dependency.

New CPI = $1 + 3 \times 10\% = 1.3$

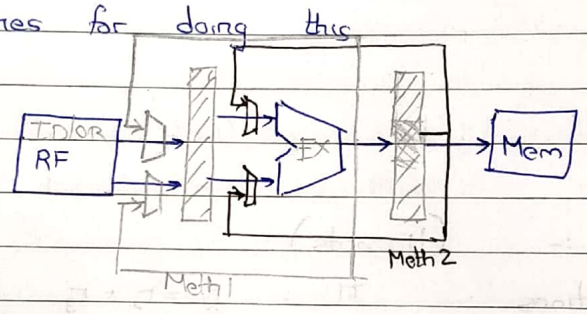
I1	—	—	—	—	
I2	I1	—	—	—	freeze
I3	I2	I1	—	—	
I3	I2	—	I1	—	
I3	I2	—	—	I1	
I3	I2	—	—	—	resume

Scanned by CamScanner

- The result of I_1 is getting written back in stage 5. (WB).
But it is being produced in stage 3 (EX) itself.
This result will be stored in the pipelining interface register at the boundary of EX and Mem. Let I2 fetch the result from there.

I_1 — — — —
I_2 I_1 — — —
I_3 I_2 I_1 — —
I_4 I_3 I_2 I_1 —

Two possibilities for doing this



→ Non-consecutive dependency. 23/10

I_1 :- $r_1 = r_2 + r_3$
I_2 :- $r_4 = r_5 + r_6$
I_3 :- $r_7 = r_1 + r_8$

1 $(RS1 \text{ or } RS2)^{ID} == RD^{Mem}$:- Where to fetch operand from?

2 $(RS1 \text{ or } RS2)^{ID} == RD^{WB}$:- Obtain result from rf-d3 of WB stage.

→ Multiple ~~write~~ conflicts.

$r_1 = r_2 + r_3$
$r_1 = r_4 + r_5$
$r_6 = r_1 + r_7$

Scanned by CamScanner

- Priority order :- For conflicting dependencies, fetching result :-

  EX > Mem > WB

  newest instruction → EX; oldest instruction → WB

→ Load instruction produces result in Mem stage.

$I1 :- r_1 \leftarrow (R2 + 50)$

$I2 :- r_3 = r_1 + r_2$

When I2 is in ID/OR, the required result has not been produced yet. Forwarding will not work. We have to wait (freeze) for one cycle at least in such cases.

1	–	–	–	–	
2	1	–	–	–	
3	2	1	–	–	~~freeze~~
4	3	2	1	–	freeze .... 'Bubble' or 'Stall'
4	3	2	–	1	resume
5	4	3	2	–	

23/10 – We are losing one cycle for every instruction of this type.

eg – α% AL instructions, β% Load (of which λ% have such immediate hazard), γ% Store, δ% Branch.

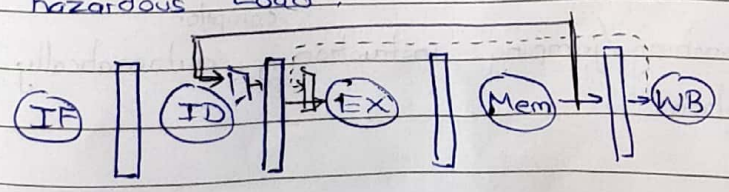
New CPI $= 1 + \beta\lambda(1)$

- Bubble or Stall :-

All interface registers should have an enable signal, which is made zero to stop updation

Make enable of PC zero, too.

- For hazardous Load.



Scanned by CamScanner

→ Minimizing the damage of Load hazard :-

eg $A = B + C + D$ → load r_1, B

$\left(\begin{array}{l} T1 = B + C \\ A = T1 + D \end{array}\right)$

A, B, C, D, T1 are in memory

load r_2, C
add r_3, r_2, r_1
store r_3, t_1
load r_4, D
add r_5, r_4, r_3
store r_5, A

$\xrightarrow[\text{compiler}]{\text{Smart}}$

load r_1, B
load r_2, C
load r_4, D
add r_3, r_2, r_1
store r_3, t_1
add r_4, D
store r_5, A

'Compiler optimization'

→ Change in Control Flow

eg I_1 contains JUMP.

∴ I_2 is not supposed to be executed.

	IF	ID/OR			
a)	1	—	—	—	—
b)	2	1	—	—	—

- Every interface register should have an extra bit, which will decide if an instruction is to be disabled.

If I_1 is JUMP, then the disabled I_2 will go through all stages, but will not write to any registers, flags

Meth 2 Delayed Branch Execution

✗ After every branching/jumping instruction, compiler automatically adds a NOP instruction.

P.T.O.

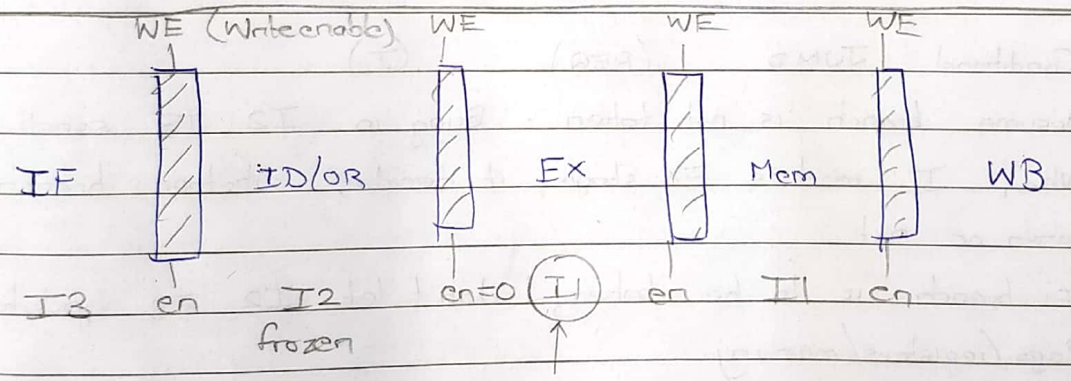
If I2 is JUMP,

- If I2 does not depend on I1 (unconditional jump), move I1 to just after I2.
- If I2 depends on I1 (conditional jump), add a NOP instruction to just after I2.

→ Conditional branching

eg I1 is BEQ

1	_	_	_	_
2	1	_	_	_
3	2	1	_	_



Comes here because interface register 2 was not updated in previous cycle.
This extra instruction needs to be killed.

- Instead of using $(R_{S1}^{ID} == R_D^{EX}) \,||\, (R_{S1}^{ID} == R_D^{Mem}) \,||\, (R_{S1}^{ID} == R_D^{WB})$, we now need more checks because I1 is duplicated

Use :- $(R_{S1}^{ID} == R_D^{EX} \;\&\&\; R_{WE}^{EX} == 1)) \,||\, (R_{S1}^{ID} == R_D^{Mem} \;\&\&\; R_{WE}^{Mem})) \,||\, (\ldots\ldots)\ldots$

&& it should be LOAD instruction

&& MEM_RD == 0

Only LOAD instruction reads from data memory.

Stall Requirement detection

If this condition is met, $IF_{EN} = ID/OR_{EN} = PC_{EN} = RF_{WR} = 0 = Flag_{WR}$

$Mem_WR = 1$

→ Unconditional JUMP

Let all interface register have a bit 'valid'
If I1 is found to be jump, make 'valid' of I2 (in IF) '0'.
If valid = 0, don't let that instruction write to flags, registers, memory.

☆ Assume that for JUMP instruction, we can update PC as soon as ID stage (using separate ALU)

→ Conditional JUMP (BEQ) (I_1)

- Assume branch is not taken. Bring in I2, I3 serially. When I_1 reaches EX stage, it decides whether branch is taken or not.
- If branch is to be taken, don't let I2, I3 update flags/registers/memory
  - Penalty of 2 cycles :(
  - * Most of the times, branches are taken
- Assume that if branch is to be taken, PC is updated at the end of EX stage.

• Our initial assumption cannot be to take the branch, as we don't know yet where to branch to.

• $\alpha \rightarrow$ AL, $\beta \rightarrow$ Load, $\gamma \rightarrow$ Store, $\delta \rightarrow$ Branch (Conditional), $\epsilon \rightarrow$ Jump
λ% independency, 70% branches are taken

$CPI = 1 + \beta\lambda(1) + \epsilon(1) + \delta(0.7)(2)$
☆ Why

→ Unconditional JUMP

Let IF interface register have a bit 'valid'
If I_1 is found to be jump, make 'valid' of I2 (in IF) '0'.
If valid = 0, don't let that instruction write to flags, registers, memory. (make WE of subsequent interface registers 0)

☆ Assume that for JUMP instruction, we can update PC as soon as ID stage (using separate ALU)

→ Conditional JUMP (BEQ) (I_1)

- Assume branch is not taken. Bring in I2, I3 serially. When I_1 reaches EX stage, it decides whether branch is taken or not.
- If branch is to be taken, don't let I2, I3 update flags/registers/memory (assumption is invalidated)
  - Penalty of 2 cycles :(
  - * Most of the times, branches are taken
- Assume that if branch is to be taken, PC is updated at the end of EX stage.

• Our initial assumption cannot be to take the branch, as we don't know yet where to branch to.

• $\alpha \rightarrow$ AL, $\beta \rightarrow$ Load, $\gamma \rightarrow$ Store, $\delta \rightarrow$ Branch (Conditional), $\epsilon \rightarrow$ Jump
λ% independency, 70% branches are taken

$CPI = 1 + \beta\lambda(1) + \epsilon(1) + \delta(0.7)(2)$
☆ Why

25/10 If branch is taken,

I1	—	—	—	—
I2	I1	—	—	—
~~3~~	~~2~~	1	—	—
20	—	—	1	—
21	20	—	—	1

- Optimization for when branch is taken :-

For first time of execution, ~~compute~~ proceed as before.
Compute destination of branch and store it in Look-up table.

PC	Branch target Address
100	700
150	755
⋮	⋮

Such instructions are mostly executed many times (in a loop)
Second time onwards, change assumption to branch taken

1	—	—	—	—
2	1 (instruction decoded)	—	—	—
20 (branch taken)	~~2~~	1	—	—
21	20	2	1	—

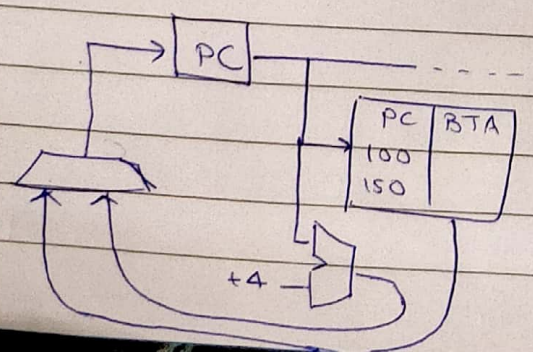
One cycle penalty still

Now assumption is correct 70% of the times.

☆ Verify – New CPI = $1 + \beta\lambda(1) + \epsilon(1) + \delta(0.3)(1)$

- Better still (for no penalty if branch is taken) detection in IF stage of I1

Match the Lookup table with new value of PC.
If current PC is found to be present in table, fetch I20 in next consecutive cycle.



1	—	—	—	—
20	1	—	—	—
⋮				

☆ CPI = $1 + \beta\lambda(1) + \epsilon(1)$

Scanned by CamScanner

→ If branch is not taken (2 cycle penalty)

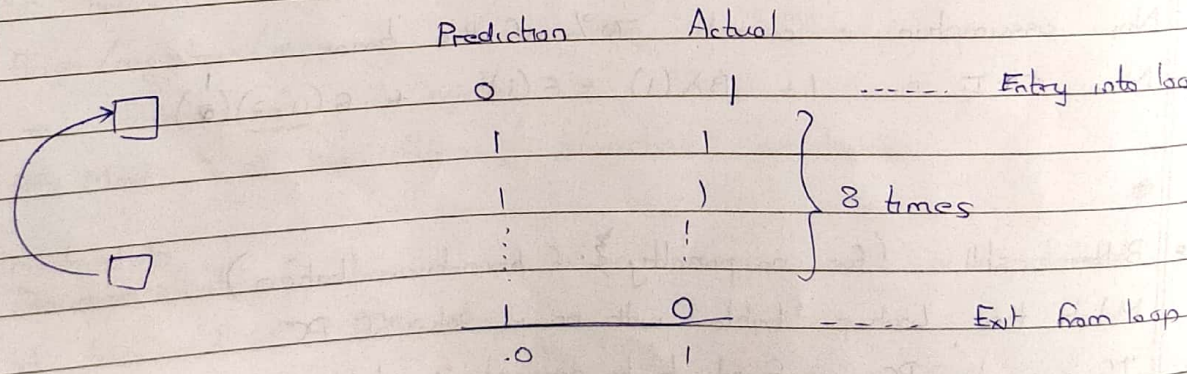
- We have lost original PC. Where to go ~~to~~?
  - We need to carry forward original PC+4 (~~of I1~~) forward
    Anyways we were carrying it forward till EX to compute PC + 4 + Imm

→ Further optimization:-

- Branch instructions tend to show similar behaviour as the last time they were executed.
  - In the back-up table, add a bit to say if last time branch was taken or not.
    Change your assumption accordingly.

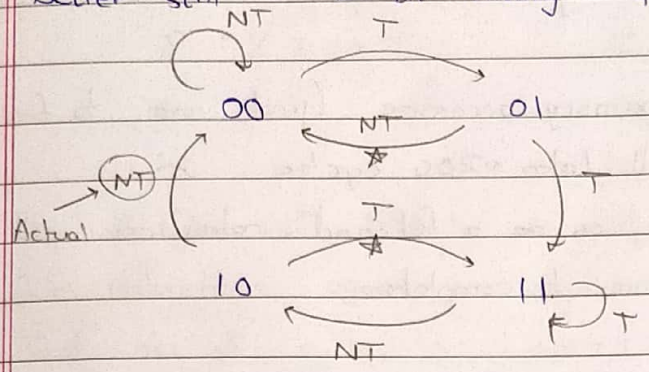
PC	BTA	History (0 → not taken)

In execution stage of I1, update history bit every time.



We were right 80% of time. (no penalty)

- Better still :- 2 bit history (past 2 decisions)



$H_{t-1}H_{t-2}$

Prediction

00 → Strongly NT

01 → Weakly NT

10 → Weakly T

11 → Strongly T.

Now you will not make mistake for entry into loop (90% correct)

Scanned by CamScanner

29/10

MEMORY SYSTEM

- For modern memory (DRAM), one memory access takes 150 – 200 cycles (not 1 as before)
  - Not counting multiple memory accesses (just one, to fetch ir), each instruction will take ~200 cycles
  - Pipelining makes no sense, as a fetched instruction will take only #stages cycles more to complete.

- One possible solution :- 'Cache'

Between processor and memory, add a very small, but fast block of memory. This has, say, only 8 bytes (worth 2 instructions) and is made from more expensive SRAM.
SRAM memory is accessible in single cycle (must fetch consecutively written instructions)

- Fetch 8 bytes (2 instructions) from memory into SRAM (~200 cycles)

Processor fetches I_1 (200st cycle)
I_2 (201nd cycle) ---- pipelining
I_3 (402rd cycle)
I_4 (403th cycle) --- pipelining

- Performance almost doubles by introducing SRAM, as at least 2 instructions are being pipelined.

eg Eg Program

```
for i in range(n):
    I1
    I2
    I3
    I4
```

Assume instruction do not access memory

If SRAM is 16 bytes, 4 instructions will be fetched in 200 cycles and stored in SRAM
For every subsequent iteration, we don't need to fetch again and instructions can be pipelined as before
CPI → 1 :)

Scanned by CamScanner

eg for i in range (100):
if i % 2 == 0 : I_1 I_2 I_3 I_4
else I_5 I_6 I_7 I_8

- With 16 bytes of SRAM, we will have to fetch and replace the four instructions again and again.
Total no. of cycles = $(200 + 1 + 1 + 1) \times 100$

- With 32 bytes of SRAM, we can get CPI → 1, but only if all 8 instructions are consecutively written. Otherwise same as before
$(200 + 1 + 1 + 1) + (1 + 1 + 1 + 1) \times 49$

- With 2 SRAM chunks of 16 bytes each, we can get CPI → 1
The instructions in each individual chunk must be consecutive.
$(200 + 1 + 1 + 1 + 200 + 1 + 1 + 1) + (1 + 1 + 1 + 1) \times 48$

- The SRAM chunks are hidden from programmer and known as 'Cache'.

10/10

→ Bandwidth of Memory :-

$$\frac{\text{Data}}{\text{sec}} = \frac{\text{Data}}{\text{Instruction}} \times \frac{\text{Instruction}}{\text{Cycle}} \times \frac{\text{Cycle}}{\text{Sec}}$$

$$= \left(\frac{\#\text{Bytes}}{\text{Instruction}} + \frac{\#\text{Bytes of data}}{\text{Instruction}}\right) \times \text{IPC} \times \text{freq}$$

$$= (4B + 4B) \times 1 \times 3 \times 10^9$$

max IPC = 1

≈ 24 GBps.

This is for peak demand from memory (each instruction requires 4B data to be fetched too)

→ Final CPI depends on no. of 'stalls' required by the program
- Instructions may stall, for reading from or writing to memory, for 200 cycles
- Stalling also happens to fetch instructions not present in cache

• Instructions can be 'hit' or 'miss'

present in cache
No penalty

not present
200 cycle penalty.

• No. of stalls = No. of misses × penalty

$$= \#\text{Instruction} \times \frac{\#\text{miss}}{\text{Instruction}} \times \text{penalty}$$

$$= \#\text{Inst} \times \frac{\#\text{Memory references}}{\text{Inst}} \times \frac{\#\text{miss}}{\#\text{Mem ref}} \times \text{penalty}$$

$$= \underset{(IC)}{\text{Inst Count}} \times \frac{\#\text{Mem ref}}{\text{Inst}} \times \underbrace{\text{Miss rate} \times \text{penalty}}_{\text{In the hands of memory designer}}$$

→ Miss Rate reduction

1 Increase no. of blocks in SRAM.

2 Replacing contents of cache :-
Preferentially, do not remove instructions that are consecutively likely to follow current instruction (spatial likeliness) or that are likely to be executed again at some later point in time (temporal likeliness)

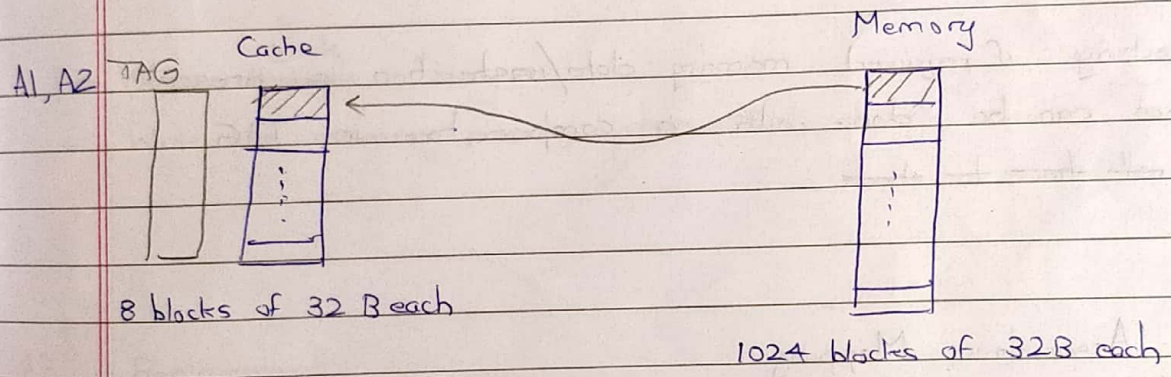
'Locality of reference'

Scanned by CamScanner

- Increasing the number of blocks in SRAM beyond a certain point will increase access time from 1 cycle to 2 cycles. :(

→ Questions to be answered

1. In which block of cache to place?
2. How to identify where required instruction is placed in cache?
3. Which ~~Whom~~ cache instruction to replace when overwriting?
4. When to write ~~to cache~~ data (obtained from processor) from cache to memory?



In memory, every byte can be addressed as

Block No.	Offset
10 bits (mem) / 3 bits (cache)	5 bits

- We will only be transferring blocks as a whole from memory to cache. After transferring, offset part of each byte remains same. Block no. will be changed accordingly
  - We maintain a mapping of which block no. of memory is ~~map~~ transferred to which block no. of cache.

  - Simplest Mapping = 'Direct Mapping'

    (Mem Block No.) % 8 = (Cache Block No.)

    eg- Mem Block 45 is transferred to Cache block 5.

P.T.O.

Scanned by CamScanner

i.e - ~~Ta~~ ~~Mem~~ ~~bloc~~

(Cache block no.) = Last 3 bits of (Mem block no.)

∴ (Cache address) = Last 8 bits of (Mem address)

- The first seven bits of Mem address (which were lost in transferring to cache) will be stored in an adjacent array to cache. (Tag)
  - TAG is an 8-element array of 7-bit elements each (one element for each block)

1/11 • Checking if required memory data/instruction is present in cache can be done with a comparator in TAG
~~- We will have to check~~

→ Fully Associative Mapping

- Any memory block number can be placed in any cache block number
  - Better utilization. :)

- Now, TAG element needs to have all 10 bits of memory block number
- Now, comparators have to check TAGs at every cache block
  - May not happen in 1 cycle :(

→ Take a middle path : 'Set Associative'
- Divide cache into multiple sets

eg - 4 sets of 2 blocks each
'2-way set associative'

Scanned by CamScanner

- Mapping of memory to each 'set' in cache is fixed.
But, within that set, any block can be used.

(Cache set no.) = Last 2 bits of (Mem block no.)
∴ Tag element size = 10 - 2 = 8 bits.

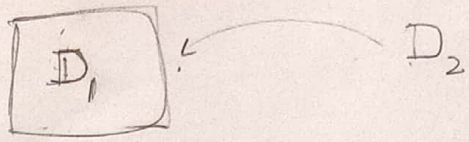
A1 Write-~~Back~~ Through Policy
Write back into memory whenever you are writing from processor to cache

2 Write-~~through~~ back Policy
Write to memory only when that block needs to be rewritten.

A3 Replacement
- In direct mapping, you don't have a choice. Need to rewrite in same block no.

- In fully associative mapping or set associative mapping,
  - Add in empty block if present,
  - If not block is empty, we want to replace the block that is least useful,
    - Possibility - FIFO
      But first-in block need not be least useful.
    - 'Least recently used'
      eg - In 2-way set associative
      Use an LRU bit per set of 2 blocks
      LRU = 0 :- Replace 0. Make LRU = 1
      1 :- 1 . 0
      } Keep toggling LRU bit

Scanned by CamScanner

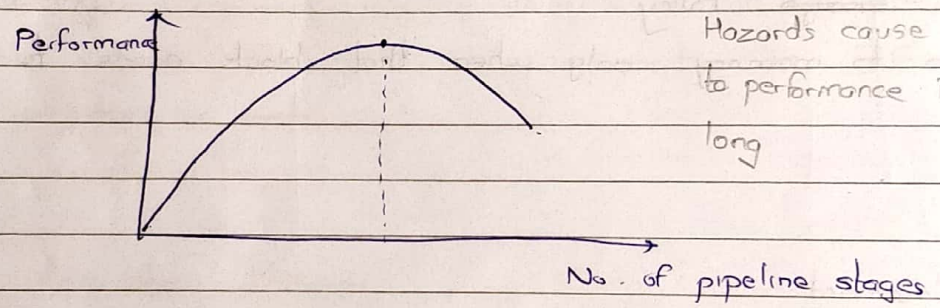


If you are using Write Back policy, maintain a 'dirty' bit for each block.
Make dirty = 1 if block was updated.
If dirty = 1, write back to memory before replacement.

SUPERSCALAR ARCHITECTURE

* IBM's experiment for pipelined design.

- Given the typical programs we run, least achievable CPI = 1.15



Hazards cause more damage to performance if pipeline is long

- What we have studied :- 'Scalar pipeline'

1) Fetch one instruction at a time.
2) All instructions pass through the same 'Unified' pipeline
3) Instructions advance in lock step.

- Lift the restriction of fetching one instruction at a time.
  Then ideal CPI < 1
- Fetch multiple instructions at a time.

- If all instructions are independent,

$a = b + c$
$d = e + f$
$g = h + i$
$j = k + l$

Then, fetch, say, 2 instructions per cycle. (Relax 1)
Duplicate the entire datapath to make two pipelines. (Relax 2)
(RF, ~~memory~~ needs to have more I/O ports)

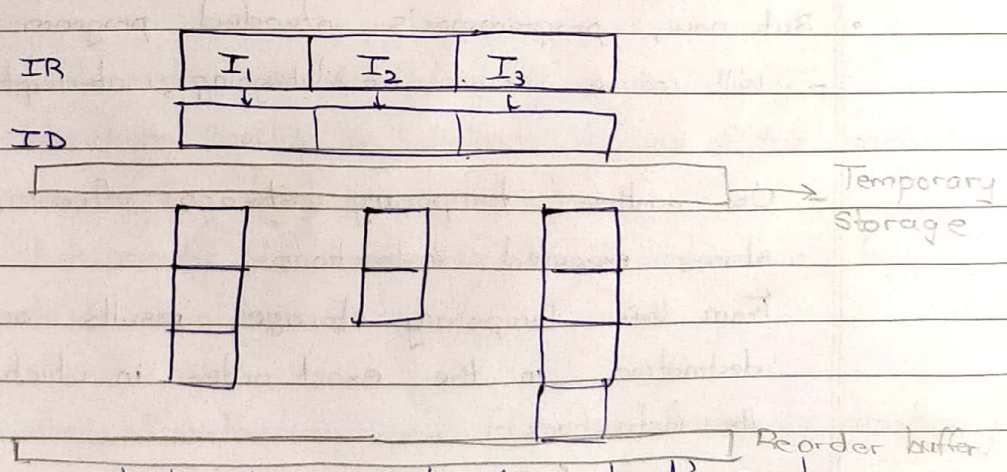
$\therefore$ IPC = 2, CPI = 0.5

- If there is dependency between instructions,

I_1 $a = b + c$
I_2 $d = a + e$
I_3 $p = q + r$
I_4 $s = p + t$
I_5 $i = j + k$

- Relax condition 3 :- Remove lock step advancement
  - By-pass I_2 and I_4 for the time being.
  - Execute I_1 and I_3 simultaneously first, I_2 and I_4 simultaneously next.

- Relax condition 2 :- Different pipelines have different number of stages.

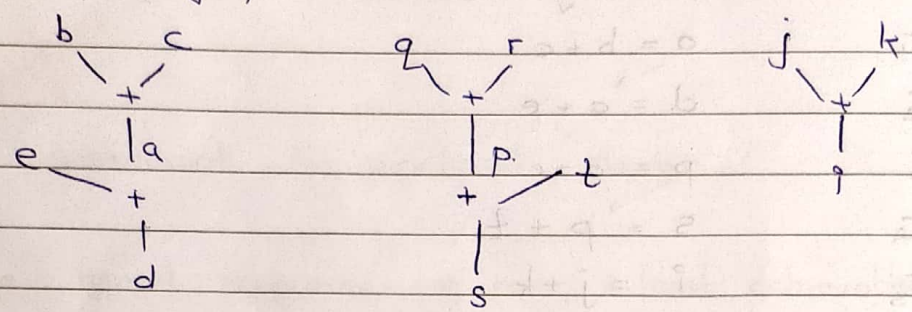


P.T.O for better diagram

Each instruction is sent to the correct 'specialized' pipeline

- Between decoder and pipelines, make a temporary storage to store, say, 100 decoded instructions.
  - In this storage, which instructions are independent is determined
    - Independent instructions are sent for execution simultaneously first.

- Instruction flow graph



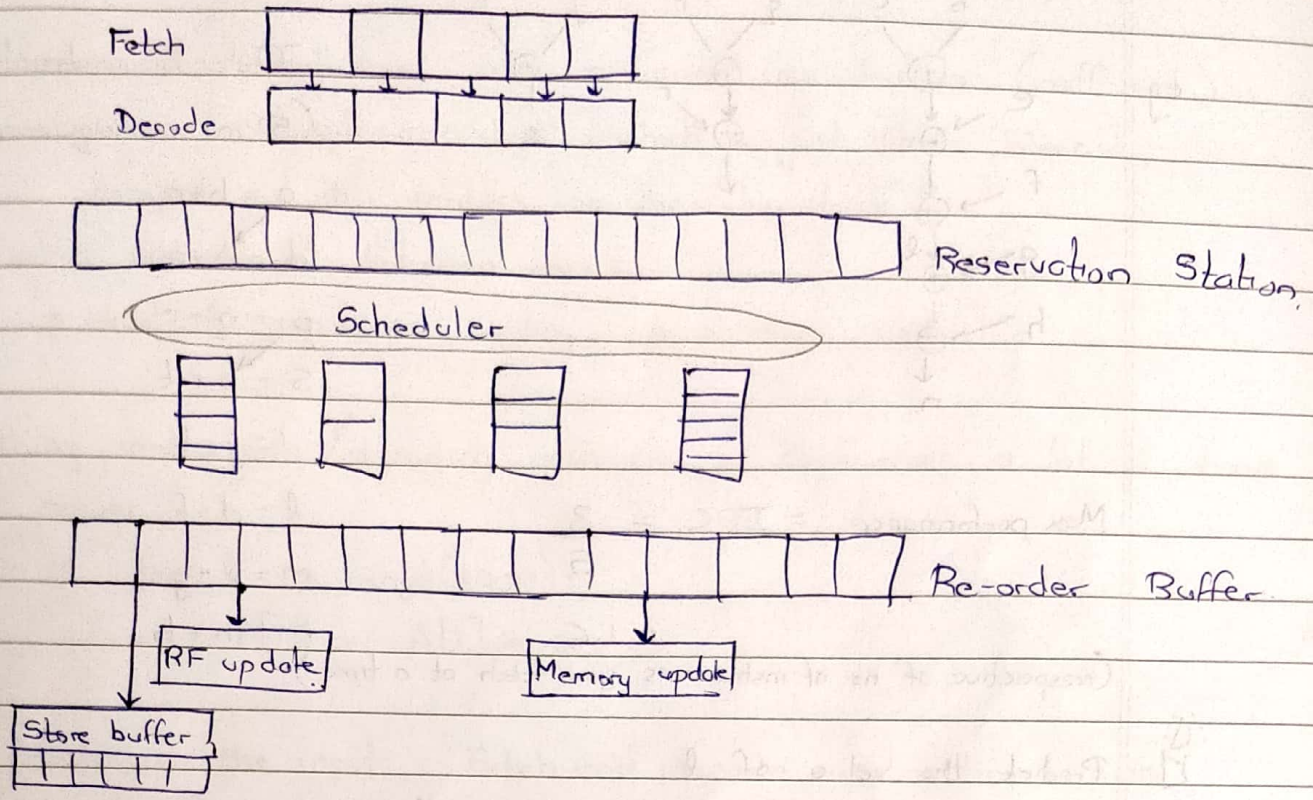
Can be done in minimum 2 cycles $\Rightarrow IPC_{max} = 2.5$, $CPI = 0.5$

- Create the following mechanism:-
  - Each instruction in temporary storage will check if its operands have been produced and are available.
    Once they are available, it is ready to be sent ~~to~~ to pipeline
    ✗ Program-guided execution, ✓ Data-guided execution
- But now, programmer's intended program order is violated.
  - Will cause issues in debugging, interrupt execution, etc.

  - Use another temporary storage after the pipelines, which stores executed instructions.
    From this temporary storage, results are written back to destination in the exact order in which programmer wrote the instructions.
    'Re-order Buffer'
  - When any instruction is added to temporary storage 1, a note is made in re-order buffer, so that instruction order is remembered

Scanned by CamScanner

6/11



- Store buffer :-

Data that needs to be stored can be temporarily stored here. This is because, memory needs to be preferentially used for Load than ~~use~~ for Store.

In most cases, Loading stage is in critical path of instruction. Writing is merely ceremonial. Any data that was to be written back and is demanded by other instructions has already been used by the other instructions

- Branch prediction :-

We are fetching several instructions simultaneously continuously. All the fetched instructions would be wasted if one of the instructions is a branching instructions. (High penalty ☹)

We need to give high priority to computing branch destination, to avoid fetching wasteful instructions.

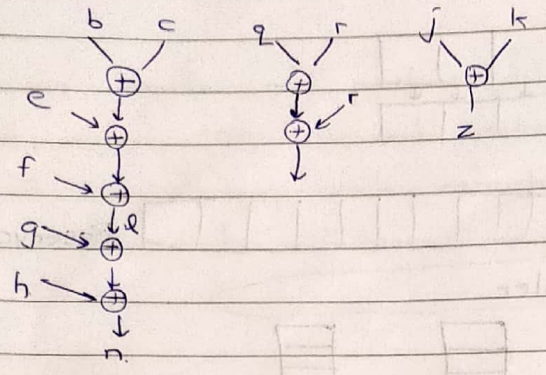
- Increasing the number of instructions you fetch simultaneously drastically increases power use. ☹

Fetching 'n' instructions simultaneously :- Performance $^{*} = n$

Power $^{*} = n^2$

Scanned by CamScanner

eg



Total 8 instructions
(5 in a sequence)

$a = b + c$
$d = a + e$
$p = q + r$
$s = p + t$
$z = j + k$
$l = d + f$
$m = l + g$
$n = m + h$

Max performance $= IPC = \frac{8}{5}$
$= 1.6$
(irrespective of no. of instructions you fetch at a time)

Predict the value of l.
If the prediction is wrong, popat.
If prediction is right, Performance $= \frac{8}{3}$:)

'Folding the flow graph'

* Power dissipation $\propto CV^2f$
Because delay of gate $\propto 1/f$, $\therefore V \propto f$
To ~~sustain~~ contain power dissipation, we had to keep frequency constant

- Before 2005 :- Faster is better
- After 2005 :- More is better
  - More number of processors (cores)
    - Can run more programs simultaneously, but no benefit for individual program/processor.
    - Better :- Run different independent parts of a program on different cores
      eg - Use each core to compute a different element in the product of two matrices
    - String of independent computation $\triangleq$ 'Thread'
    - Responsibility of programmer to identify threads in program.

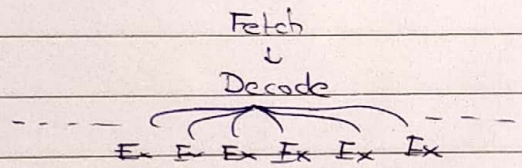
Scanned by CamScanner

- Communication between cores - Cannot use buses (inefficient)
  - Use 'Network-on-Chip', which is just like Internet, complete with routers on the chip itself.
- Caches are shared between cores
  - Need to supervise writing to or from caches.

- Fetching instructions / decoding instructions consumes a lot of time and power.

eg - for i in range(1000):
    A[i] = A[i] * c.

Δ:
- Vectorize the input. Fetch and decode instruction once, execute for different 'i' simultaneously



eg - Processing each of 1920 × 1080 pixels simultaneously as a vector.
At least process a vector of length 100 parallelly.

'GPU'
Graphics Processing Unit.

- GPU is not used 24×7.
Use it for processing normal non-display-related CPU tasks whenever free.
Send a kernel (≙ a subset of instructions of CPU program that can be vectorized) from CPU to GPU and receive executed data at the CPU.
Such GPUs are called General Purpose GPUs

P.T.O

Scanned by CamScanner

- One Fetch-Decode-Ex path in GPU is a thread.
- One Warp $\triangleq$ 32 threads
  - Minimum granularity at which GPUs are built.
- Thread block $\triangleq$ Collection of warps.

Scanned by CamScanner