CW

29 /August 2023.

X86 Processor

→ 8085, 8086 : initial registers.

386, 786, 5086, 7086 → X 86 Generation (x86 Assembly).

X86 Processor → 32 bits (4 bytes can be processed in parallel)

→ 64 bits (8 bytes " " ")

→ x86 (32/64 bits) Assembly.

If source data is 256 bits → 64 bits will take 4 cycles to complete
→ 32 bits will take 8 cycles to complete.

FDE cycle.

1. CPU fetch instruction from memory (instruction queue).
2. CPU decodes instruction by looking at binary bit pattern
3. If operands are involved – operands are fetched from memory + registers
4. Instruction is Executed.
5. If O/P operand was part of instruction, the CPU stores the result of its execution in the operand

Modes of Operation

↳ Protected : all instructions are available (memory protection + multitasking)

↳ Real-address : implements programming environment of Intel 8086 processor.

↳ System-management : provides OS (for power) management, system security, diagnostics etc.

↳ Virtual 8086 - hybrid of Protected - each program has its own 8086 computer

Registers (high Speed)

1. General Purpose Register - hold data for operation. [arthematic, logical and could store address]
2. Address Registers - holds address of data / instructions
3. Status Registers - keeps current status of processor

* EIP : points to next instruction.

AX - 16 bits (AH: 8 bits, AL: 8 bits) EAX – Extented AX (32 bits) RAX – (64 bits)

Note : *downward compatibility : 64bits can be used process 32 bits not the other way around

General Purpose Register (available for data manipulation).

1. EAX (Extended Accumulator Register) – used in arithmatic, logical & control
2. EBX (Extended Base Register) – serve as address register
3. ECX (Extended Counter Register) – serves as loop counter
4. EDX (Extended Data Register) – used for I/O & used with AX for * & ÷

GAMA

LOL

29 / August 2023.

Segments. → increase RAM capacity.
→ done to implement backward compatibility
→ efficient execution

1. Code
2. Data
3. Stack segment - holds local variables + function parameters.

Von-Neuman: data & address are in same memory divided

how to access →
- using memory addresses which are stored in registers
- stores addresses of next instructions

memory segment - block of consecutive memory bytes. Each segment is identified by segment number.

offset: where next address is (distance from beginning to next instruction).

To keep track of the various program segments, segment registers are used.

EES - Extended Extra Segment
used for accessing second data segment

A memory location may be specified by providing a segment number & offset value
Segment:Offset

The program's code, data & segment are loaded into different memory segment ie code segment data segment, stack segment

Pointer and Index Registers

ESP, EBP, ESI, EDI - point to memory locations (has offset addresses).

1. ESP: used in conjuction with ESS for accessing the stack segment.
2. EBP: access data on stack and other segments.
3. ESI: used to point to memory locations in the data segments addressed by EDS
4. EDI: used to access memory location addressed by EES.

Flag Registers

- Indicates status of microprocessor - sets individual bits called FLAGS
- Status Registers: stores info whether an instruction should be executed or not.
  - ↳ Control and System Flags: control CPU operation
  - ↳ Status Flags: Status of arithmatic + logical operations.

0 = reset/clear
1 = set.

- 8 general purpose
- 6 segment registers
- EIP

→ Basic Language Elements (Assembly Language)

```
main PROC
        move eax, 5      ;   move 5 to EAX register
        add  eax, 6      ;   add 6 to EAX Register.
        INVOKE ExitProcess, 0;   end program
main ENDP
```

*Operands: a value that is

→ Mnemonic	Description
MOV	assign (move) one value to another
ADD	add 2 values
SUB	Subtract 2 values.
MUL	multiply 2 values
JMP	jump to new location

GA LOL

;Add Two.asm - adds two 32 bit integers

.386 // represents 32 bit architecture.

.model flat, stdcall // telling how much memory is reserved for data & code

.stack 4096 // memory assigned to stack.

ExitProcess PROTO, dwExitCode:DWORD // acts as return 0; stops all threads of processes.

.code // code segment

main PROC

mov eax, 5 ;

add eax, 6 ;

INVOKE ExitProcess, 0

main ENDP

END main

*Mov eax, [5]

↓

here it represents memory address (because of [])

→ Little Endian Order

- refers to order in which Intel stores integers in memory.
- multi-byte integers are stored in reverse order when most significant byte is stored at MSB and smallest is stored at LSB.

→ Big Endian Order

- most significant byte is stored at lowest address.

→ Directives vs Instructions

↳ tells assembler what to do. — to tell CPU what to do.

→ Procedure

[NAME] PROC ... [NAME] ENDP

LABEL (optional), Mnemonic, Operands → Instruction Format.

Integer Literals:

{+|-} digits (radix) binary = b/y octal = q/o hexadecimal = h... decimal = t/d

Identifier → identifies variable.

MOV destination, source.

↓

Can't IP, EIP, CS

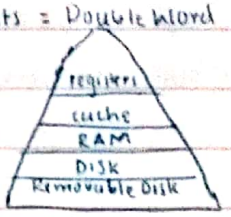
not immediate to segments

destination = source (must)

Size Size

← otherwise assembler gives error

1, 0 = bit.
4 bits = Nibble
8 bits = Byte
16 bits = Word
32 bits = Double Word



Memory hierarchy

possible
register, register
register, value
register, memory.

```
.data
    bVal BYTE 100
    bVal2 BYTE ?
    wVal WORD 2
    dVal DWORD 5
.code
    mov ds, 45
    mov esi, wVal
    mov eip, dval
    mov 25, bval
    move bval2, bval
```

as memory to memory transfer is ≠ possile ∴ transfer to register → register to memory.

Direct Memory Operations

- DMO is a

Little Endian order VS Big Endian order eg (12345678h)

0000	78	0000	12
0001	56	0001	34
0002	34	0002	56
0003	12	0003	78

MOV Instruction

MOV destination, source

no more than one memory operand is allowed.
↳ mostly CPU has control.
CS, EIP, IP cannot be destination

SWORD - Signed WORD (16-bits).

SDWORD - Singned double

DB - define byte. (1 Byte)

DW - define word (2 Byte)

DD - define doubleword (4 byte)

DQ - define quad. (8 byte)

DT - define 10 bytes

Operand	Description
reg8	8 bit general purpose
reg16	16-bit general purpose.
reg32	32-bit general purpose.
reg	any general-register
sreg	16-bit segment CS, DS, SS.
imm	8, 16, 32 bit immediate value

Multiple Initializers.

label refers to the offset of the first initialiser

```
list    BYTE   10,20,30,40.
list    BYTE   10,32,41h,00100010b
liste   BYTE   0Ah, 20h, 'A', 22h
```

to ensure that assembler doesn't think this is a label

```
.data
Array1  BYTE   10h, 20h, ?, 40h
        BYTE   50h, 60h, 70h, 80h
        WORD   81h
        ~~BYTE~~  ~~81h~~, 82h, 83h
list3   BYTE   ?, 32, 41h

.code
        MOV al, array1 + 7        ; al = 80h.
```

→ equivalent to a[7]

offset
0000 10h G
0001 20h O
0004 40h D
50h
?

Defining strings

String ends with null byte (containing 0) called null-terminated string.

```
greeting1   BYTE   "Good afternoon", 0   → 15 bytes
greeting2   BYTE   'Good night', 0
```

→ shows its null terminated.

each character uses a byte of storage.

rule that byte must be separated by commas doesn't apply on strings

values to be repeated. → uninitialized

DUP operator.

↳ allocates storage for multiple data items, using integer expression as a counter

```
x   BYTE  20 DUP(0)              ; 20 bytes, all equal to zero.
y   BYTE  20 DUP(?)              ;20 bytes, uninitialised.
s   BYTE  4  DUP ("STACK").      ; 20 byts  STACKSTACKSTACKSTACK
z   BYTE  7  DUP (2 DUP(17,19))  ; 17 19 17 18
var4 BYTE 10,3 DUP(0) ,20        ; 10 000 20.
```

array

↳ DUP directive is used to duplicate or replicate data values in an array or a block of memory

17 19 17 19 17 19 17 19 17 19. 17 19 17 19

17 19 17 19

↳ signed word.

Defining WORD and SWORD.

assigns 16 bit.

The legacy DW allows us to store signed num without explicit definition

Declaring Uninitialized Data.

.data ? – used to declare an uninitialized data segment

– executable memory is used less – assigned at compile time.

Symbolic Constants.

EQU – integer values

TEXTEQU – string

Current Location Counter: symbol $ is used which returns offset of current location

```
selfPtr DWORD $
```

declares selfPtr and initializes it with variable's offset value.

Calculating sizes of Array and String.

```
List BYTE 10,20,30,40.
listsize = $ - list        → this is equivalent to list[0].
```

listsize must follow immediately after list

offset_a3 : ($ - arr3)/4

so we get in word.

$ current offset address.

= used for assignment/updation of variables

equ integer can't be updated but string can be updated.

↳ constant symbol

eg exp equ 2*5

exp .equ 9 // can't redefine.

exp=20

eg txt equ <"ABC"> //can redefine

txt equ <"abc"> as in angular brackets

txt2 equ <10*10> ;text

eg symb equ exp

mv (symbol) equ mov // instead of mov mv can be used.

axmov equ <mv al,v1>

axmove equ <mov,al,v1>

list db exp, txt, txt2, symb.

axmov equ <mv,bl,v1>

GAMA

NOTES

Directives tells compiler what to do while Instructions tell CPU what to do

Procedure [Name] PROC
" "
[Name] ENDP

Character literal 'A', 'N' String literal 'ABC'

myVar DWORD 26 ← Directive
Instruction ← mov eax, myVar

MOV destination, source

EIP, CS, IP cannot be destination

- 386 directive that identifies the program as 32 bit program (can access 32 bit address register)
- model flat, stdcall selects the program memory model and identifies the calling convention

Std call keyword tells the assembler how to manage the runtime stack when procedures are called.

calling convention: how subroutines receive parameters from their caller and how they return a result.

- Stack 4096 sets aside 4096 bytes of storage for runtime stack.

STACK
- holds passed parameters
- holds address of the code that called function. The CPU uses this address to return when the function call finishes, back to the spot where the function was called.
- holds local variables.

- code marks beginning of the code area of program

Multiple Initializers → list BYTE 10h, 32, 00101111b, 'A', ?, 38

Defining Strings: ends with null character (byte) ex g1 BYTE "Good", 0
each letter uses a byte so total 5 bytes used.

.Dup – duplicates values.

Symbolic constants (symbol definition) created by associating an identifier
name = expression [cannot be redefined] ; name is symbolic constant

1. Carry Flag - Subtract smaller value from larger.
when addition results in

2. Sign : MSB = 1 Sign flag on

3. Overflow : Overflow XOR of ~~last bit~~ carry in & last carry out

4 Auxiliary : 3rd bit to 4th bit value shift.

5. Parity : even number of 1s in lower byte.

NOTES

Overflow

→ Unsigned Overflow

when there is carry out of the MSB

correct answer larger than biggest unsigned number

On subtraction unsigned overflow occurs when there is borrow into MSB. correct answer is smaller than 0

→ Signed Overflow

- addition of numbers with same sign, Signed overflow occurs when sum has different sign.
- Subtraction of numbers with different signs is like adding numbers of same sign. $A-(-B) = A+B$ $-A-(+B) = -A+(-B)$

Signed overflow occurs when result has different signs than expected.

- OVERFLOW IS IMPOSSIBLE WITH DIFFERENT SIGNS

SHIFT INSTRUCTIONS.

for shift instructions bits shifted out are lost.

for rotate instruction bits shifted out are moved in from other end.

opcode destination, 1 [shift/rotate by 1]

opcode destination, N [shifts/rotates by N]

→ SHL (multiplies by 2)

MSB shifted to CF ; 0 is moved into rightmost [LSB]

Effects on flags

SF, PF, ZF reflect the result.

AF : undefined

CF : last bit out of MSB

OF, 1 if result changes sign on last shift

eg 10001010, 3

1. CF [1] 00010100 OV:1 SF:0 ZF:0 AF=NULL PF:1
2. CF [0] 00101000 OV:0 SF:0 ZF:0 " PF:1
3. CF [0] 01010000 OV:0 SF:0 ZF:0 " "

SAL Instructions

Used for signed (emphasizes on arithmetic nature)

1000 0000₂

CF=1 0000 0000

CF=0 0000 0000 CF=OV=0 SF=0 ZF=0

→ SHR Instruction. (divides by 2).

0 shifted into MSB and LSB moved to CF

SHR destination, N

→ SAR Instruction.

MSB retains original value (rest works like SHR).

SAR destination, N.

for odd numbers halves + rounds down

→ Signed and Unsigned Division

for unsigned use SHR

for signed use SAR

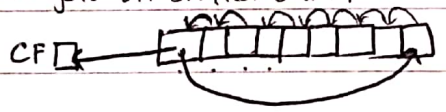
eg To divide by 4 use 2 shifts as $2^2 = 4$.

ROTATE INSTRUCTIONS.

→ Rotate left. (ROL)

MSB is shifted to LSB (rightmost)

CF also gets bit shifted out of MSB.



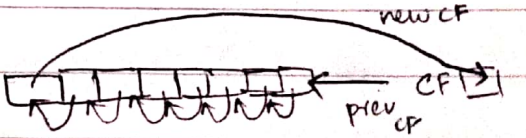
→ Rotate Right (ROR)

LSB shifted to MSB and LSB also goes in CF

→ Rotate Carry Left

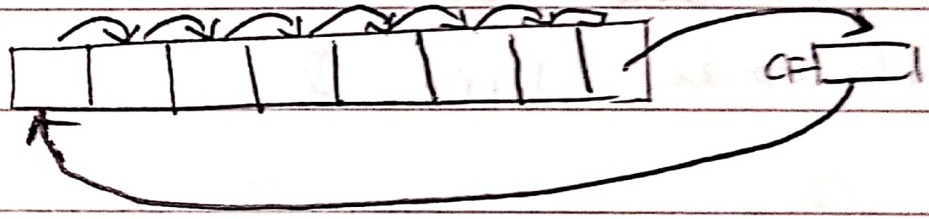
shifts bits to left. MSB shifted to CF

previous CF value is moved to LSB



→ Rotate carry Right

bits rotated to Right



Effects ON Flags .

SF, PF, ZF reflect result .

AF undefined

OF = 1 if result ~~on~~ changes of sign on last rotation

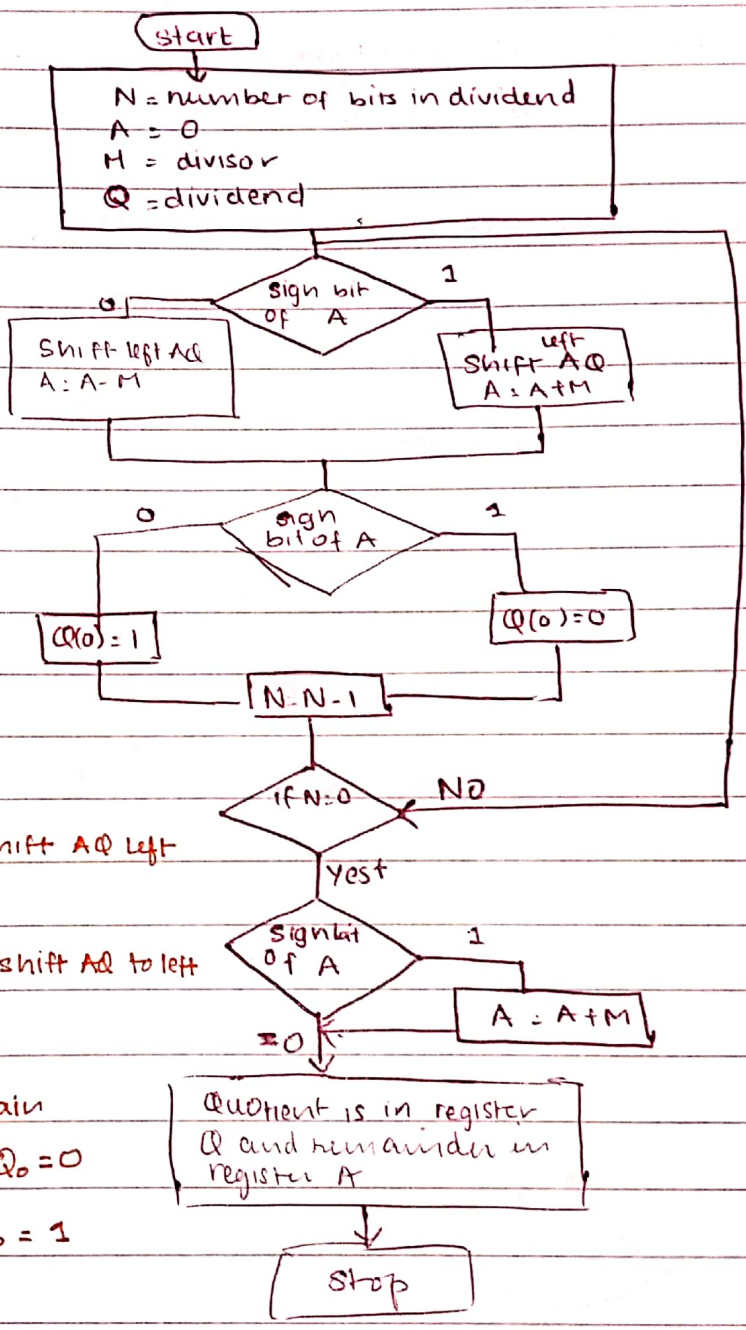
CF = last bit shifted out

24 / Oct / 2023.

NON-RESTORING ALGORITM (Division).

Convert all algorithms to assembly

→ get description.



1. Initialized values
2. Check sign bit of A
3. If sign bit of A = 1 then shift AQ left and perform A+M = A

   If sign bit of A=0 then shift AQ to left and perform A=A-M
4. Check sign bit of A again
5. if sign bit is 1 the $Q_0 = 0$

   if sign bit is 0 then $Q_0 = 1$

   [Q_0 is LSB of Q]
6. N is decremented
7. If N≠0 loop from step again

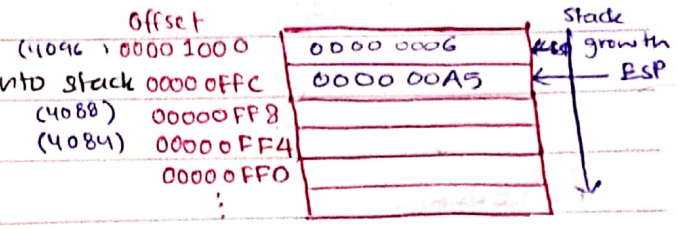
   else

   Check sign bit of A if 1 then A : A + M
8. else A contains remainder

   Q contains Quotient

Stack Operations

- LIFO (Last In First Out).
- ESP (Extended Stack Pointer) is used to manage memory
- ESP holds 32 bit offset into some location on stack
- Use instructions such as CALL, RET, PUSH, POP to manipulate ESP



- ESP always points to last value pushed onto stack

PUSH Operation

- (32 bit) decrements Stack pointer (ESP) by 4 (32 bits) and copies value into the location in stack pointed by ESP.
  - ↳ if 16 bit then decrements by 2
- Stack grows downward in memory.
- Each time new value is added location of ESP points to newly added value

POP Operation

removes value from stack

The value pointed by ESP is removed and ESP increase by 4 (32 bits)
- if 16 bits then increments by 2

* when calling a subroutine, you pass input values and arguments by pushing them on stack
* provides temporary storage for local variables inside subroutines

PUSH/POP Instructions

- pushes reg/mem value onto stack

```
PUSH reg/mem16      // decrements ESP by 2
PUSH reg/mem32      // decrements ESP by 4
PUSH imm32          // decrements ESP by 4
POP  reg/mem16      // increments ESP by 2
POP  reg/mem32      // increments ESP by 4
```

↳ copies popped value in reg/mem

PUSHFD / POPFD Instructions

- PUSHFD is used to push 32-bit EFLAGS register onto stack
- POPFD is used to pop/reload EFLAGS from stack
- MOV instruction cannot be used to copy the flags to a variable, so PUSHFD is the way to save value of flags
- This allows us to restore value ~~from~~ of flags from stack.

```
pushfd          ; save current value of flags
// code
popfd           ; restores value of flags
```

* Ensure if pushfd is used then your program executes popfd also and not skip it
* This is however ERROR-PRONE, so there is another less error-prone way

PUSH EFLAGS on stack and pop immediately and store in a variable.

```
• data
  saveflag DWORD ?
• code
  pushfd
  pop saveflags      ; this stores in save flags variable
  push saveflags
  popfd              ; copies value of saveflags to EFLAGS.
```

PUSHAD, PUSHA, POPAD and POPA

1. PUSHAD: pushes all 32 bit general purpose registers onto stack. [EAX, ECx, EDx, EBX, ESP (value of ESP before execution of PUSHAD), EBP, ESI, EDI]
   The stack stores in the above mentioned order ↑
2. POPAD: pops registers off the stack in reverse order EDI; ESI ..... EAX
3. PUSHA: pushes 16-bit general purpose registers (AX, CX, DX, BX, SP, BP, SI, DI)
4. POPA: pops 16-bit general purpose registers in reverse order

* for 16-bit programming mode use PUSHA; POPA
* for 32-bit programming mode use PUSHAD; POPAD

- If you create a procedure which uses general purpose register you may use pusha/pushad and popa/popad.

```
SUB proc
  pushad
  :
  popad
ret
BNDP
```

* procedures returning results in one/more registers should not use

GAMA

Stack Frame is created by following steps:

1. Passed arguments are pushed on stack
2. The subroutine is called, causing the subroutine return address to be pushed on the stack. (push وال address الى).
3. As the subroutine begins to execute, EBP is pushed on the stack
4. EBP is set equal to ESP

   From this point on, ESP is used as reference for all of subroutine

Passing by Value is when value of register is sent

Passing by Refrence is when offset is passed.

+ why index starts at 0 from C++

- code

```
Main PROC
    PUSH 5
    PUSH 6
    CALL AddTwo
    ADD ESP, 8    [write this statement]
    MAIN ENDP
AddTwo PROC
    PUSH ECP
    MOV EBP, ESP
    MOV EAX, [EBP+8]
    ADD EAX, [EBP+12]
    POP EBP
    RET 8    [or write this statement]
    AddTwo ENDP
```

When a local variable is passed through procedure always execute

push EBP

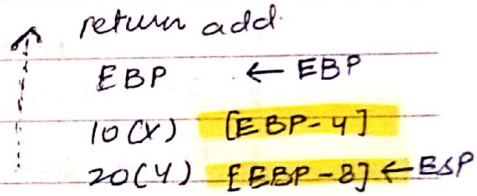
mov EBP, ESP

Two ways to clear stack.

- POP elements
- or just decrement ESP
- RET8 means release 8 bytes after function return.

Creating Local Variables.

```
MySub PROC
    Push ebp
    mov ebp, esp
    MOV DWORD PTR [ebp-4], 10
    MOV DWORD PTR [ebp-8], 20.
    mov esp, ebp
    pop ebp
    ret
MySub ENDP
```



- Parameters can be passed on stack as well as in registers

Stack Frames

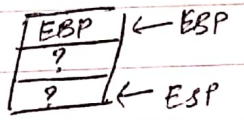
- passed arguments are pushed on stack
- The subroutine called causing the return address to be pushed on stack.
- As the subroutine begins to execute, EBP is pushed on the stack
- EBP is set equal to ESP
- If any local variable were created ESP is decremented to reserve space for variables on the stack
- if any registers need to be saved they are pushed on stack.

Enter and leave Instructions

- ENTER perform 3 ops:
  1. Push EBP on stack
  2. Set EBP = ESP
  3. reserve space for local variables ie sub esp, numbytes

ENTER numbytes, nestinglevel
(numbytes, nestinglevel: immediate values)

eg mySub PROC
ENTER 8,0



Leave terminates the stack frame procedure.
written before ret instruction

LOCAL Directive

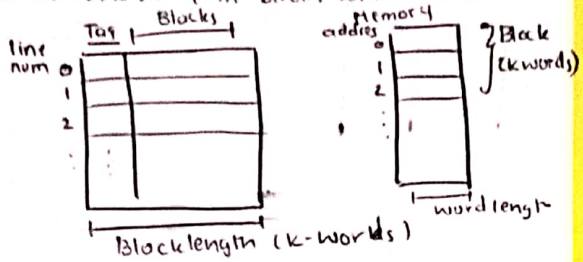
- Saves space for local variable on stack
declaring must happen after MySUB Proc instruction.

LOCAL var[10] : BYTE

(Saves space for one/more variables)

Cache

There are 3 cache levels L1, L2 and L3
L2 is slower and larger than L1
L3 is slower and larger than L2
Cache consists of m blocks called lines.



Mapping Function

- algorithm for mapping main memory into cache lines
- A means of knowing which main memory occupies a cache line

o direct o Associative o Set Associative

EXAMPLE

- Cache can hold 64 kBytes
- data is transferred b/w M.M and cache in blocks of 4 bytes each
- Main memory consists of 16 MBytes with each byte directly addressable by 24 bit address.
  Thus for mapping purpose we can consider main memory to consist of 4 M blocks of 4bytes each.

Direct Mapping

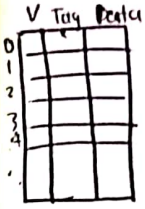
maps each block of main memory into only possible cache line.

$$i = j \bmod m$$

i = cache line number
j = main memory block number
m = number of lines in the cache.

Address length = (s+w) bits
Addressable units no = 2^{s+w} words/bytes
Block size = Line size = 2^w words/bytes.
No. blocks in main memory = $\frac{2^{s+w}}{2^w} = 2^s$
Number of lines in cache = $m = 2^r$
Size of cache = 2^{r+w}
Size of tag = (s-r) bits

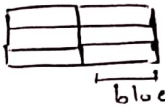
- Cache is divided into fixed-size blocks containing multiple words of data
- Principles of temporal and spatial locality tell us that recently accessed data and data close to it, are likely reused in future
- when accessing data if it is already present in cache we call it cache hit, else cache miss



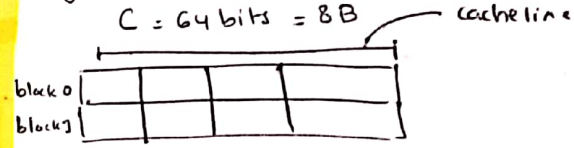
- data is stored in blocks
- blocks contain multiple words of data.
- each block is selected by index
- valid bit indicates if data is valid or not
- Tag is remaining part of address

Terminologies.

Capacity(C): Total number of bytes that can be stored in cache.

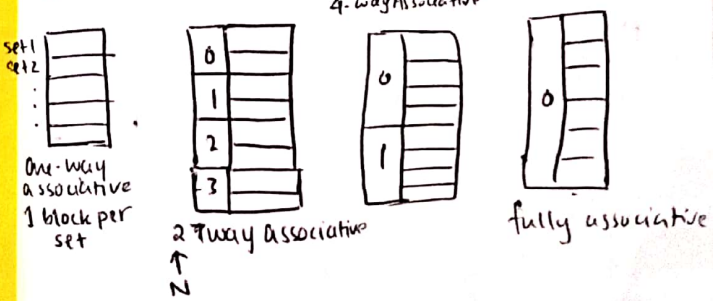


eg b = 32 bits = 4B
C = 64 bits = 8B



offset bits $\log_2 b = \log_2 4 = 2$

- Block size(b): bytes of blocks in cache
  $B = C/b$ in this eg $8/4 = 2$
- Degree of associativity (N): number of blocks in a set
- Number of sets: $S = B/N$



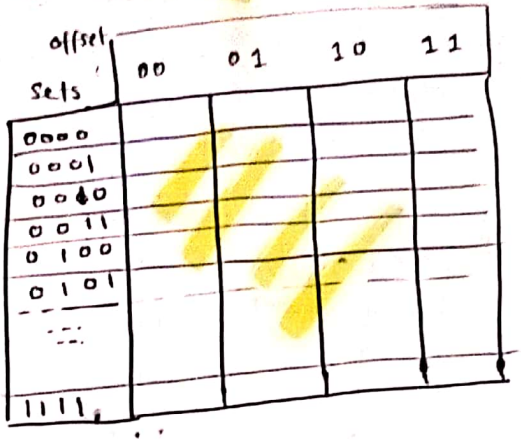
eg b = 4B
C = 64 B
N = 1 (Direct)

$B = C/b = 64/4 = 16$

$S = B/N = 16/1 = 16$

$\log_2(S)$ = Set bits = 4

$\log_2(b)$ = offset bits = 2



highlighted part will all add up to capacity ie 64.

eg b = 32B
C = 128B
N = 1 (Direct).

$B = 128/32 = 4$

$S = B/N = 4/1 = 4$

$\log_2 S$ = Set bits = $\log_2(4) = 2$

$\log_2 b$ = offset bits = $\log_2(32) = 5$

adds up to 128

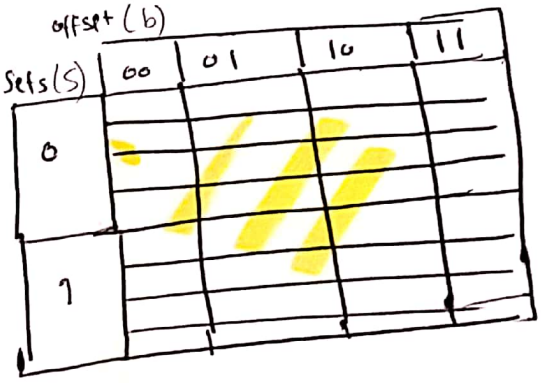
eg b = 4B
C = 32B
N = 4 (4-way associative)

$B = C/b = 32/4 = 8$

$S = B/N = 8/4 = 2$

offset bits = $\log_2(4) = 2$

Set bits = $\log_2(2) = 1$



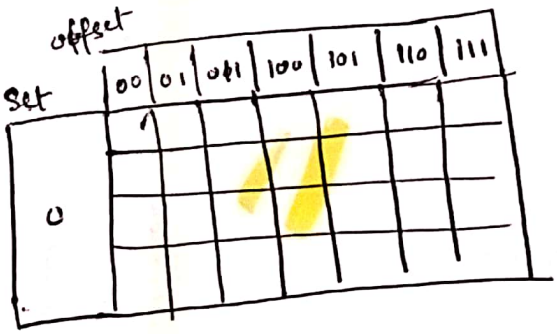
each block adds up to C ie 32

eg b = 8B
Full Associative cache.
Single set with four ways

$C = 8B * 4 = 32B$

$B = C/b = 32/8 = 4$

$S = B/N = 4/4 = 1$



adds up to 32.

- **Tag bits:** We add tags to the cache which lets us supply the address bits to ~~supply~~ let us distinguish b/w different memory locations that map to same cache block

- **Valid bit:** indicates whether data is valid (1) or not (0).
  1. at start all V bits are 0
  2. when data is loaded into a particular cache block, the corresponding valid bit is set to 1.

Example 1.

b = 4B
C = 32B
N > 1.

B = C/b = 32/4 = 8B
S = B/N = 8
Set bits $\log_2(8) = 3$
offset bits $\log_2(4) = 2$

Tag bits = address − set bits − offset bits.

sets	V	Tag	00	01	10	11
000						
001						
010						
011						
100						
101	1	0000000	00	00	00	14
110						
111						

32 block capacity.

0x014

0000 0001 0100
Tag | set | offset

data at 014 in M.M is
00000014

LRU (Least Recently Used) replacement

- In 2-way associative cache, 1-bit is required to keep check of recently used data block.
  1. When block is accessed LRU is set to 0 and increment all other values accordingly
  2. When data is evicted from a block, replace the block with max value of LRU bit

Example 2.

b = 2B
C = 16B
N = 2 way

B = C/b = 16/2 = 8B
S = B/N = 8/2 = 4B
set bits = 2
offset bits = 1.
address bits 8
Tag bits = 8 − 2 − 1 = 5 bits.

sets	V	LRU	Tags	0	1
00					
01					
10					
11	0	0	11111	FE	FF

eg 0xFF
1111 1111
tag | set | offset

Example 3

b = 1B
Full Associate cache.
Single set with 4 ways.
Address bits 8 bits.

C = 1 × 4 = 4
B = C/b = 4/1 = 4
Set = 4/4 = 1
set bits = 0
offset bits = 0
Tag = 8 − 0 − 0

	V	LRU	Tag	Data
0				

Type of Cache Miss

- Compulsory Miss: The first request to cache block is called compulsory miss, because the block must be read from memory regardless of cache design.
  ↳ at first cache is empty so =
- Capacity Miss: occurs when the cache is too small to hold all concurrently used data
- Conflict Miss: caused when several addresses map to the same set and evict blocks that are still needed

Cache Parameters.

1. Increasing cache capacity reduces capacity + conflict miss but doesnt effect compulsory miss.
2. Increasing block size could reduce compulsory miss (due to spatial locality) But as block size increases, the number of sets in fixed-size cache decreases increases probability of conflicts.
3. N-way associative cache reduces conflicts by providing N blocks in each set where data mapping to that set might be found Each memory address still maps to a specific set but it can map to any one of N-blocks in the set.

*Associative caches are usually slower and expensive

Locality

1. Temporal Locality: if data is used recently and is likely to be used again
   Such data is kept in higher levels of memory
2. Spatial Locality: if data used nearby data may be used eg array values.
   bring nearby data of the data currently accessed in higher levels of memory

$$\text{Miss Rate} = \frac{\text{Misses}}{\text{Total memory accesses}} = 1 - \text{Hit Rate}.$$

$$\text{Hit Rate} = \frac{\text{Hits}}{\text{Total memory accesses}} = 1 - \text{Miss Rate}$$

Cache Write Policies.

- Write Through
  - when cache memory is updated simultaneously main memory is also update.
  - at any time data in cache and main memory is same.
- Write Back
  - during write operation only cache location is updated
  - when update occurs in cache this is marked by a flag also known as modified/dirty bit
  - when word is replaced from cache based on value of dirty bit, the data is written to main memory.
- Write Around (aka write no allocate).
  - the write operation directly goes to main memory without affect the cache
- Write Allocate.
  - load newly written data to cache. and if that data is needed soon it will be available in cache.