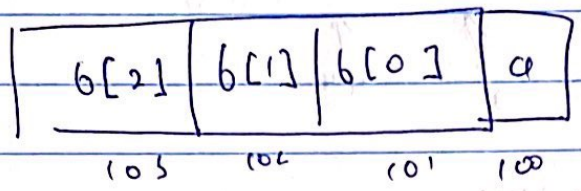
Note: we are refering to byte oriented architecture with 32 bits. in this course.

Operand width alignment

```
char a;
char b[3];
```



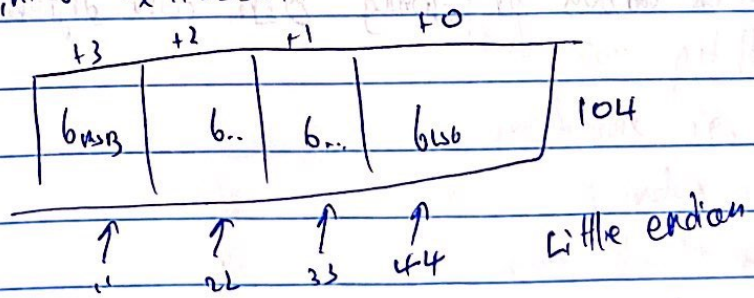
→ packing operands to save space

There is always space/time trade off.
↳ it is okay to waste space in order to save time according to compiler

Endianness → the way bytes are stored in memory.

Little Endian ⇒ LSB @ word address

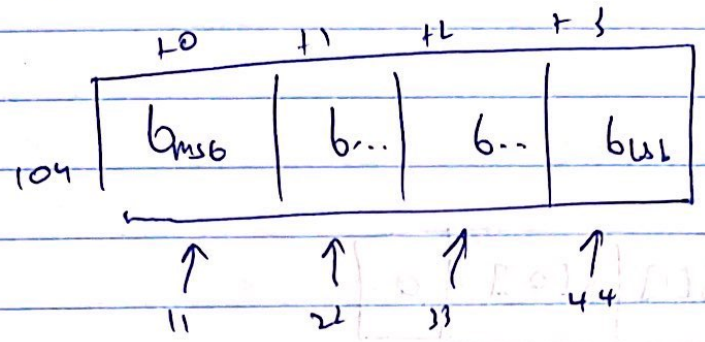
```
int b = 0x11223344
```



Little endian

Big Endian

Big endian → MSB @ word address



int b = 0x 11 22 33 44

So whats the diff?

ldb r1, mem[104]

Big endian => get 11 into r1

Little endian → get 44 into r1

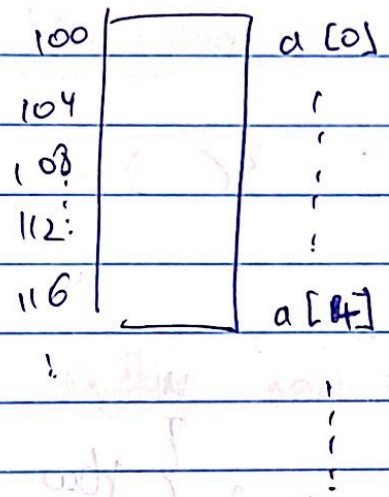
~~Read~~

Endian ex matters for storing data with different precisions

Scanned with CamScanner

Accessing array operands

int a [100]



one approach

$r_b = 100$

to load a [8]

ld r_s, 32 (r_b)

base + offset addressing mode

~~Hello~~

loop

c[i] = a[i] ⇒ introduce new addressing base + index

i = i+1

end loop

↳ we can put this in the architecture ie.

1 index = 4 memory space.

~~thus, this happens by the~~

Conditional Statement

if <predicate> {

} else {

}

To do:

- eval predicate
- Break sequential flow of code
- rejoin

} How do we do this

eval predicate → op code for ALU

Break sequential flow ?

- need to know where we are → PC
- new instruction ? → Branching

BEQ r_1, r_2, offset

⇓

if $r_1 = r_2$

PC ← PC + offset

else do nothing

PC-relative addressing mode ⟸

rejoin ⇒ unconditional ~~jmp~~ jump

Do we really need a new instr for unconditional jumps??
No, we can ~~s~~ simulate it with BEQ

Loop statement

for ( ){

}

Anything new needed in ISA?

JAL – jump and Link

JAL r_{link}, r_{target}

$r_{link} \leftarrow PC$
$PC \leftarrow r_{target}$

} remembering return address.

Scanned with CamScanner

Stack → acts as a communication area caller & callee

implemented as a data structure in memory.

Scanned with CamScanner

Stack

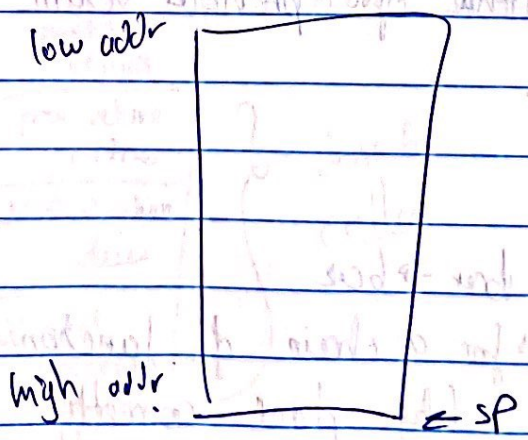
Stack as common area between caller & callee

↳ we can use registers for returning result but we still need to use stack as registers are limited.

Software convention for registers

s0 - s2 :- caller's registers → values in s registers are expected to be preserved.
t0 - t2 :- temporary registers
a0 - a2 :- parameter passing registers
v0 :- return ~~value~~ value address
rlink → ra : return address
rtarget → at : target address
sp : stack pointer

stack pointer



convention

- decrement + push
- increment + pop
- top of stack is last item placed in it

Stack Procedure → done by Compiler

Step 1

Caller saves any registers t0-t3 on the stack (if it needs the values in them)
↳ temporary registers aren't protected

Step 2

caller places the parameters in a0-a2
↳ use stack for additional parameters if needed
↳ this is called spilling to the stack

Step 3

caller associates space for any additional return values on stack

Step 4

Caller saves contents of previous return address currently in ra.

why?
foo → bar → baz
↳ for a chain of function calls to take place correctly.

Scanned with CamScanner

Step 5

Caller executes JAL at ,ra

(no effect on stack)

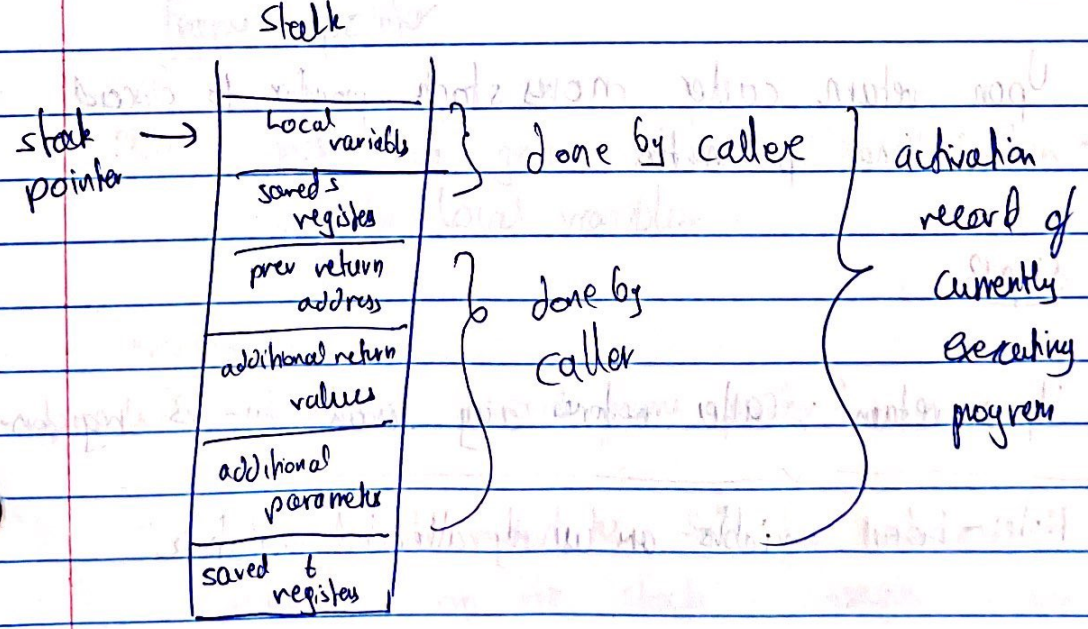
Step 6

Callee saves any of registers s0 → s3 that it plans to use during its execution on the stack.

Step 7

Callee allocates space for any local variables on stack.

~~Step 8~~ The stack ~~now~~ till now looks like this.



Step 8

Prior to return, callee restores any saved registers s0-s2 registers from the stack.

Step 9

Upon return, caller restores previous return address to ra

Step 10

Caller ~~restores~~ stores additional return values as desired. Callee would have to put the additional return values in the assigned space on the stack.

Step 11

Upon return, caller moves stack pointer to discard additional parameters.

Step 12

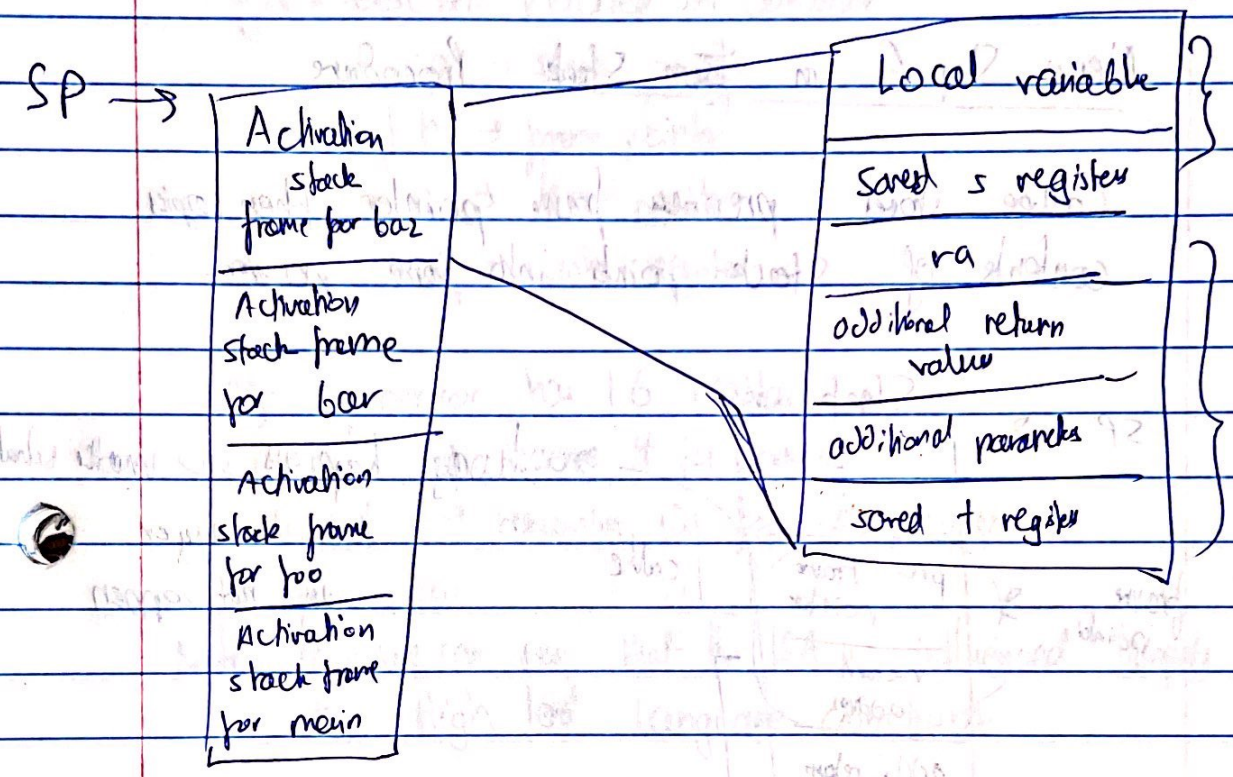
Upon return, caller restores any saved t0-t3 registers

Note:- Local variables are usually allocated on stack.

Scanned with CamScanner

if we have a chain of function calls,

main → foo → bar → baz



Frame pointer

Note: stack can grow while inside a function due to local variables.

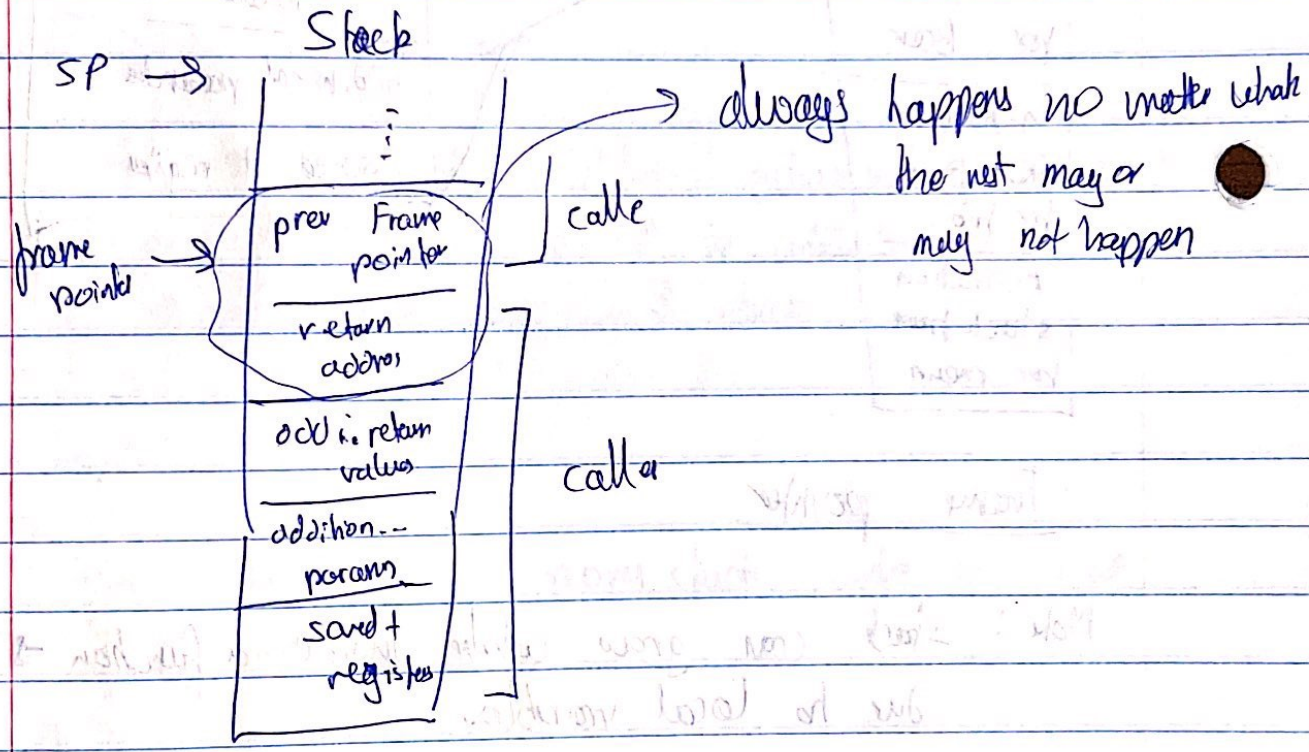
Why do we need frame pointer?

- Offset with respect to SP for referencing variables on the stack changes ⇒ pain for compiler writer

solution? Have a fixed harness on the stack for referencing variables on stack ⇒ frame pointer.

New Step 6 in ~~Fom~~ Stack Procedure

Callee stores previous frame pointer then copies contents of stack pointer into frame pointer.



This simplifies address generation for compiler.

Recap

registers ~~reserved~~ reserved for compiler
- sp → stack pointer
- fp → frame pointer
- at → target register
- ra → link register

eg. processor has 16 registers
→ reserve 4 for compiler
→ remaining 12 for user program

Note: Thus we can see that the ISA is influenced largely by High level language constructs.

Architecture Styles

- stack oriented architecture → all variables are on stack, all operands are implicit (usually used by calculators)
- memory oriented → everything stored in memory, no registers
- ~~accu~~ ACC oriented → one register called accumulator, all operations done with respect to that

Scanned with CamScanner

- register oriented :-
- Hybrid :- memory oriented and register oriented.

Instruction format

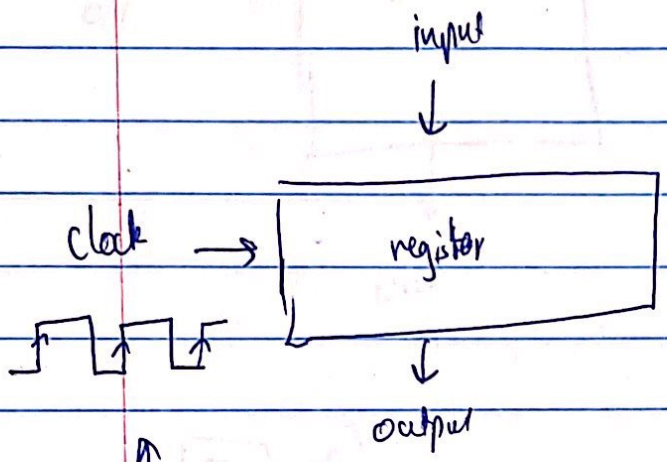
0 - operand → (only opcode)
1 - operand
2 - operand
3 - operand
same length instruction (LC2200, MIPS)
variable length instruction
Ex. stack oriented architecture will have 0 operands.

Scanned with CamScanner

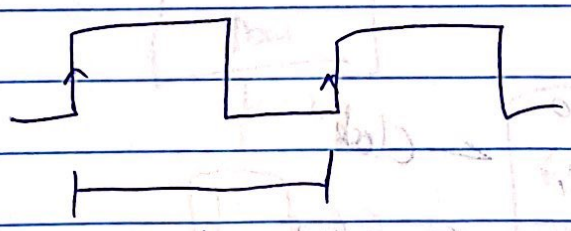
Chapter 3

25/1/22

Logic Design Review



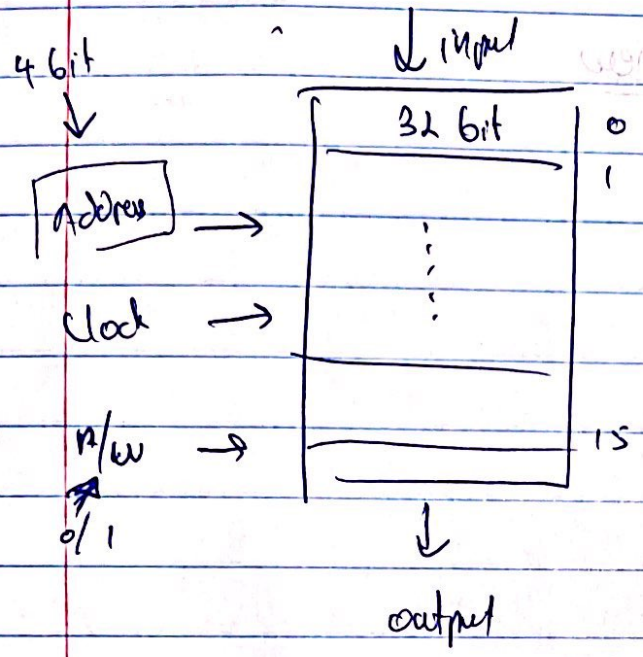
usually happens on rising edge.



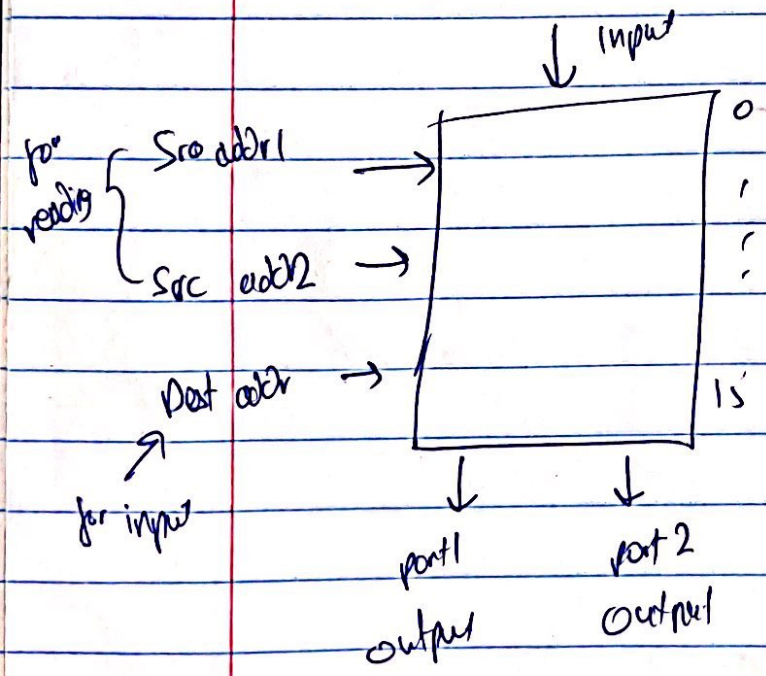
input stabilises

Scanned with CamScanner

Register file?

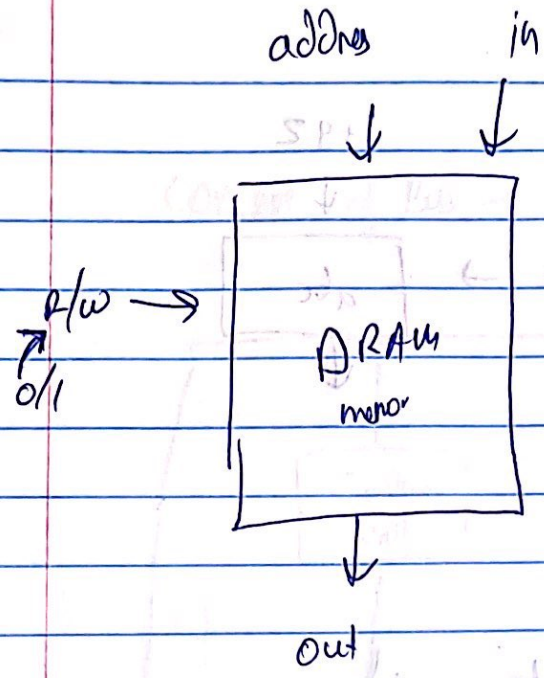


Dual Port register file



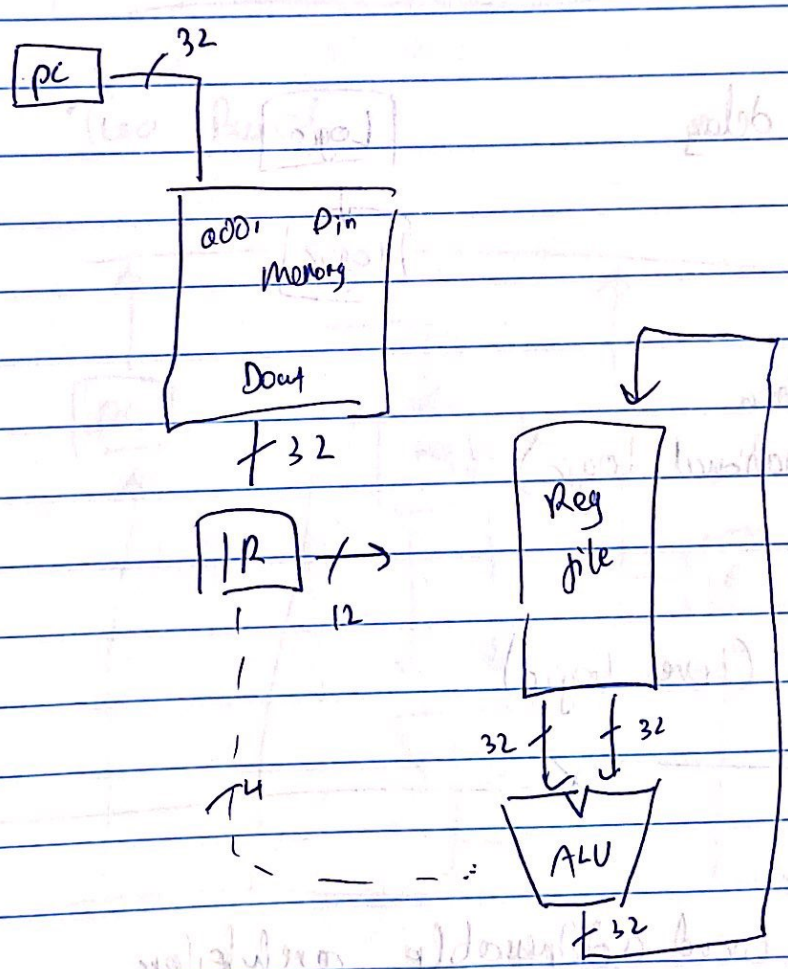
* Allows to read from 2 addresses in one clock cycle.

Scanned with CamScanner



Note: No clock for DRAM.

Level Logic.



level logic +

PC
↓
Mem
↓
IR
↓
Reg
↓
ALU
↓
Reg

combination logic → instant

1 clock

reading register, no clock

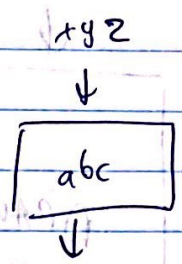
1 clock to write.

~~what done PC → MEM → IR →~~

Delays

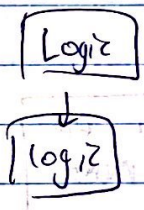
input { setup (before) / Hold (after)

clock →



output stable
(before it can be safely read)

propogation delay
(wire)



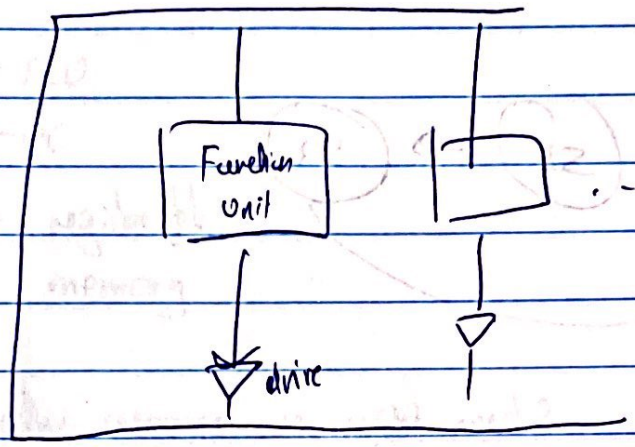
ALU - operation
(Combinational logic)

access time
(level logic)

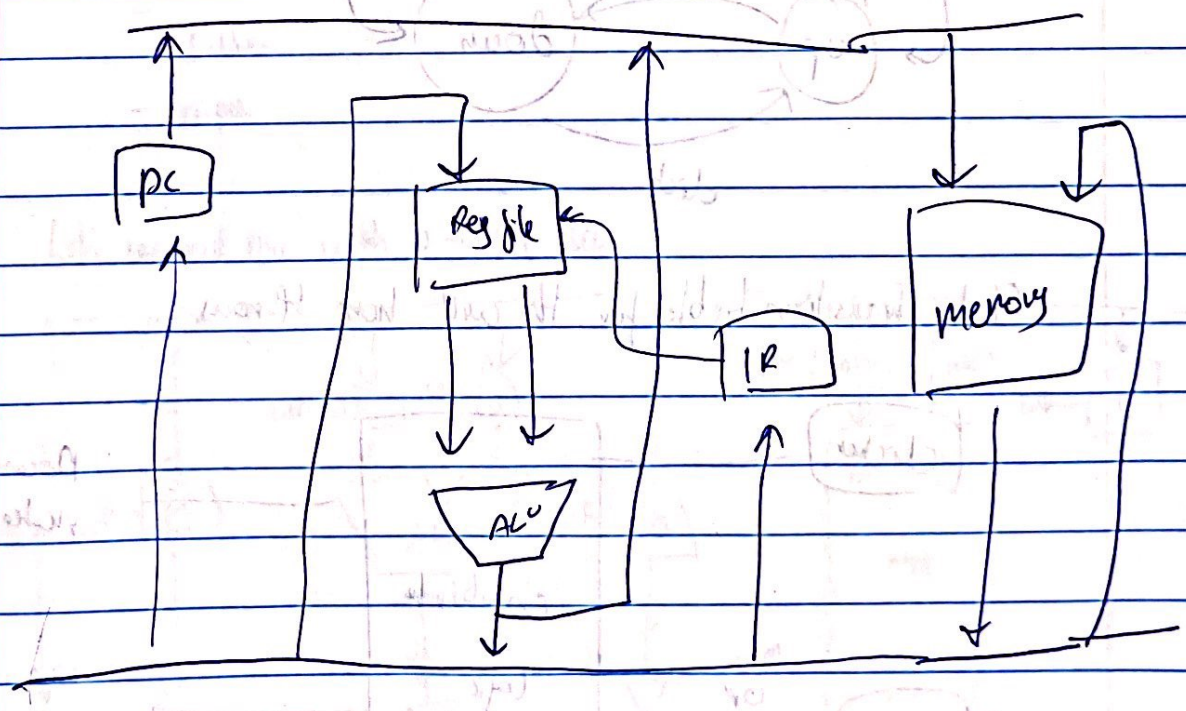
LC - 2200
32 - bit word addressable architecture

(Little Endian)

Concept of Bus → shared wire



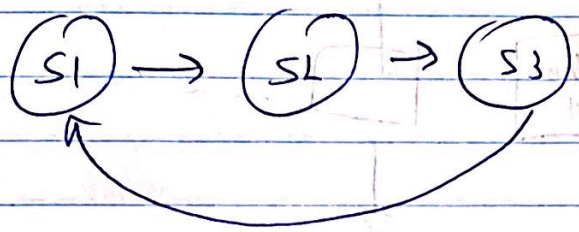
Two Buses?



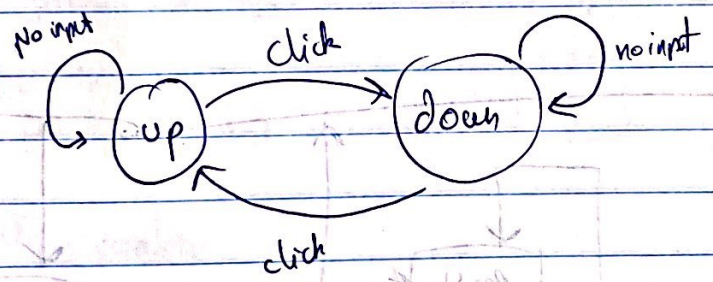
PC → mem → IR → reg file → ALU → reg file
takes 2 clock cycles with 2 bus.

Scanned with CamScanner

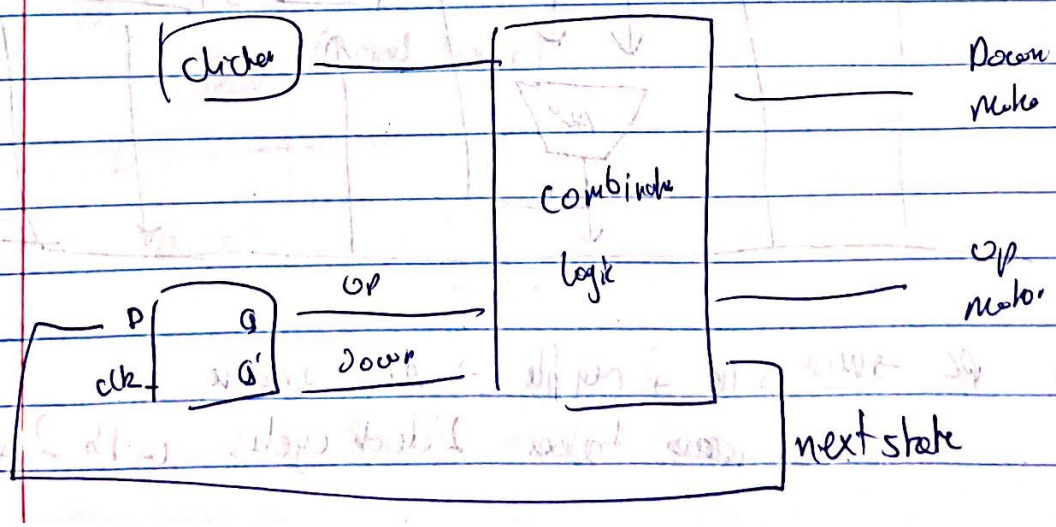
Finite State Machine (FSM)



example. garage door with ~~clicker~~ clicker



State Transition table for this will have 4 rows.



Scanned with CamScanner

27/1/22

Chapter 3

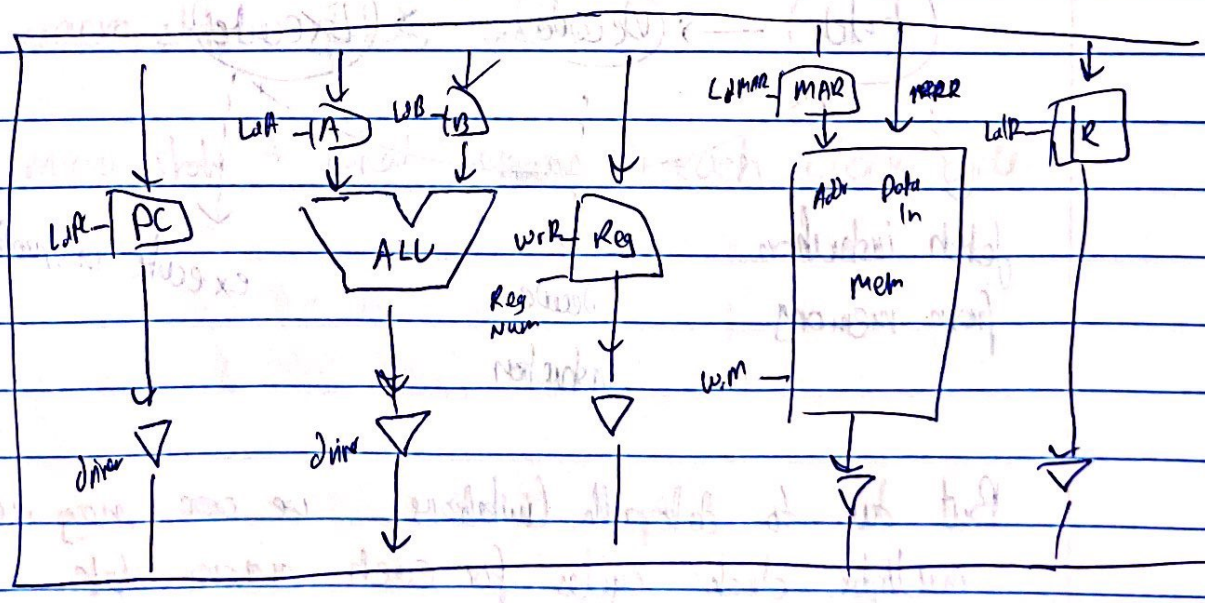
ISA dictated resources

- ALU
- PC
- register file
- memory

other resources we might need?

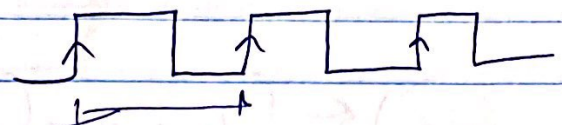
- IR
- MAR
- clock
- Buses
- drivers

Let's connect this with a single Bus



Scanned with CamScanner

All load signals are conditioned by clock

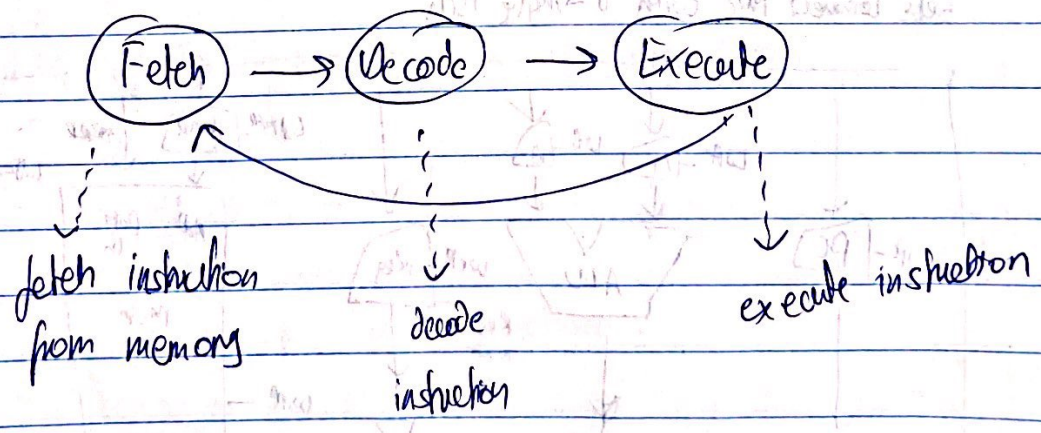


edge (positive) triggered
→ combinational logic here

width of clock pulse → read 3.4.2

~~Control Unit Design~~

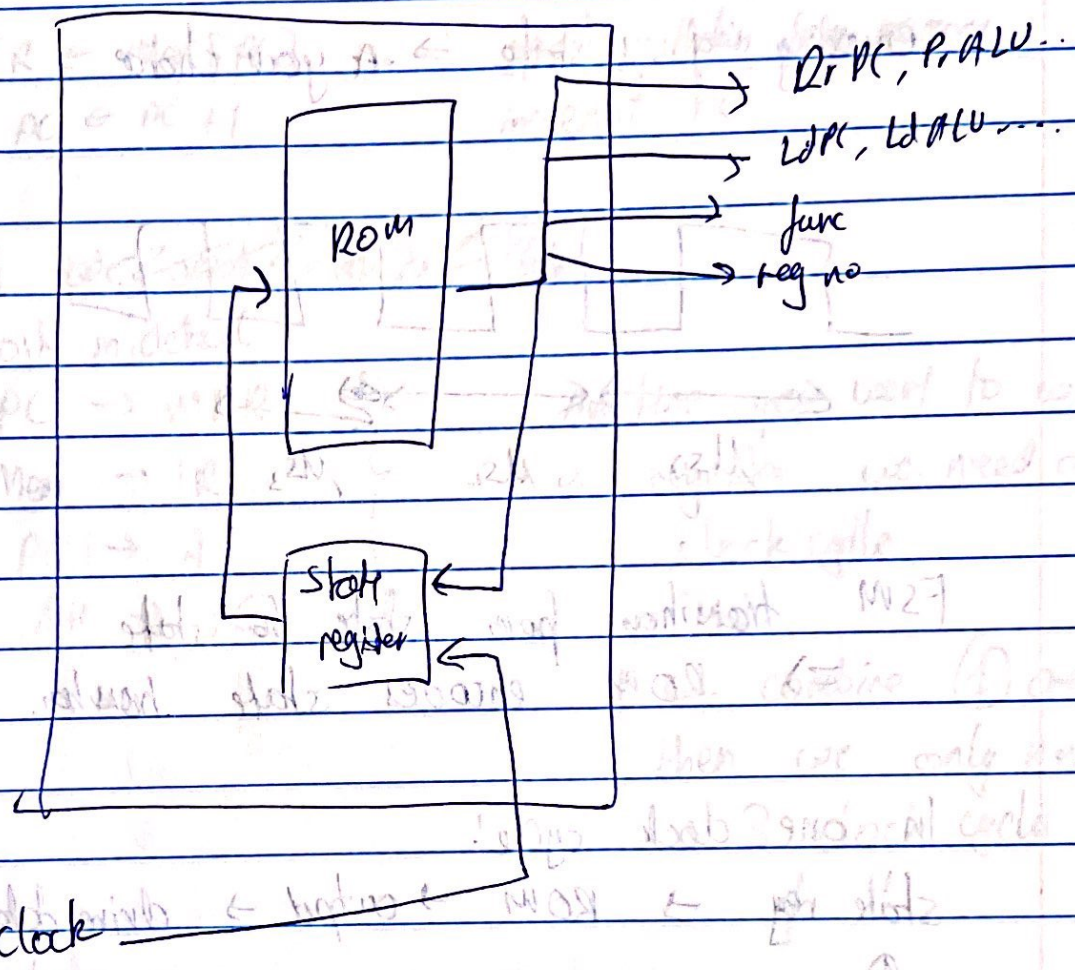
At the Macro level control unit does



But due to datapath limitations, we ~~can~~ may need multiple clock cycles for each macro state.

Scanned with CamScanner

Control unit = State reg + ROM



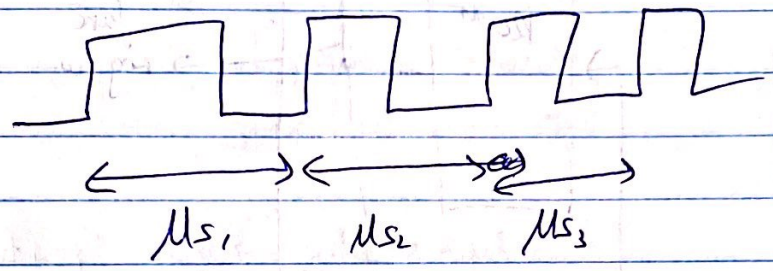
macro state → fetch, decode, ...

micro state → what happens in each clock cycle.

Scanned with CamScanner

state register ⇒ current microstate

encoding of μ state ⇒ ~~A~~ your choice



FSM transition from state to state
⇒ ROM encodes state transition.

In one clock cycle:
state reg → ROM → output → drive datapath

next state ← do the work
(combinational logic)

What is control unit design?
- simply an algorithmic exercise
- programming at a different level of abstraction

Fetch Macrostate

① IR ← Mem[PC] : read instruction from memory
② PC ← PC + 1 : increment PC

& ~~to need multiple clocks~~
work in detail

① PC → MAR
② Mem → IR
③ PC → A
④ AH → PC

Anytime you want to write to a register we need one clock cycle
↓
If we combine ① and ②, then we only need 3 clock cycles instead of 4.

Microstate.

fetch 1

PC → MAR
PC → A
control signals needed
OrPC
Ld MAR
Ld A

Scanned with CamScanner

fetch 2

Mem[MAR] → IR

~~Control st~~

Control signals

D. Mem

Ld IR

fetch 3

A+1 → PC

Control signals

~~Fetch~~ Fetc

:

We will come back to decode instruction later.

ADD instruction

$R_x \leftarrow R_y + R_z$

IR	Opcode	R_x	R_y	~~X~~	R_z

This works for any R-type instruction, only ALU op changes.

add 1

$R_y \rightarrow A$

RegSel = 01
Dr REG
LdA

~~Fct~~ 1 microstate = 1 clock cycle

add 2

$R_z \rightarrow B$

RegSel = 10
Dr REG
LdB

add 3

$A+B \rightarrow R_x$

Func = 00
Dr ALU
RegSel = 00
LdREG

Scanned with CamScanner

BEQ

→ requires more thought

→ if ($R_x == R_y$) then PC ← PC+1 + signed address offset
else nothing.

idea → subtract and check for equality, we can add special circuitry to check for equality to 0.

beq 1

Rx → A
RegSel = 00
Dr REG
Ld A

beq 2

Ry → B
RegSel = 01
Dr Reg
Ld B

beq 3

A - B
Load reg 2 with result of zero detector
func = 10
Dr ALU
Ld 2

Scanned with CamScanner

1-2-22

M.C.U sequences.

2 ROM → ifetch, beqt } 2 entries

Chapter 2 Recap

Chapter 2: Partnership between H/W & S/W

S/W — compiler

H/W — FSA

Chapter 3: H/W

Scanned with CamScanner

Chapter 4: Partnership between H/W & S/W

S/W — OS

H/W — ISA, enhancements to datapath & control.

- external interrupts
- traps
- exceptions

} Program discontinues.

	Sync/Async	Internal/External	intentional	Ex.
Exception				
Trap				
Interrupt	Async	External	Yes	I/O device completion

New info

Interrupt handler:

save $k0 → special registers available to the OS, save return address.

save process registers

execute device code

restore process registers

return to original program.

restore $k0

If we get an interrupt during another interrupt, we need to save $k0 on stack so that we need to know when to return.

problem.

saving $k0 is 2 steps.

I_1: decrement SP.

I_2: save $k0 on stack.

what if error occurs in the middle of these 2? we will loose k0.

Solution? We need to make ($I_1 + I_2$) atomic

How?

↳ Turn interrupts off before executing $I_1 + I_2$.

↳ We do this in the interrupt macro state.

Scanned with CamScanner

But we ~~don't~~ still want higher priority interrupts.

↳ 2 new instructions.
- enable interrupts
- disable interrupts

Handler:

save $k0
enable interrupt.
save process registers.
execute device code
restore process registers
disable interrupts
restore $k0
~~return to original program.~~ enable interrupts
jump $k0

↑ This needs to be atomic.

~~Definition: Important~~

atomic { new instruction:
Load PC from $k0
enable interrupts.

Chapter 5: Performance and pipelining.

Execution time of program = # of instructions * CPI (cycles per instruction) * cycle time

$$= \left(\sum_{i=1}^{n} N_i \times CPI_i \right) * \text{cycle time}$$

$$\text{or} = N * CPI_{Avg} * \text{cycle time}$$

Branch Prediction

Assume branch not taken hardware.

↳ easier to predict this, as we don't need to calculate offset

↳ If we need to take the branch, we squash instructions in ID/RR + IF

Specination: $\frac{\text{execution time on ref. machine}}{\text{execution time on target machine}}$

How to improve processor performance?

$\Rightarrow$ reduce execution time

How?

# of executed instructions ↓
CPI ↓
Cycle time ↓

Amdahl's Law

$$\text{Time}_{\text{after}} = \frac{\text{Time affected}}{\text{Amount of improvement}} + \text{Time unaffected}$$

Latence vs Thruput

↓ CPI

↓ IPC → pathway to pipelined processor design

Scanned with CamScanner

Chapter 5: Pipelined Processor Design.

Exec time = N * (CPIavg) * cycletime

↓

pipelining

Structural hazard in pipeline occurs due to hardware limitations in datapath.

Data Hazard

Read-after-write (RAW)

True dependency

I_1 R1 ← R2 + R3

I_2 R4 ← R1 + R5

~~Write~~ write-after-read (WAR)

anti-dependency

I_1 : R4 ← R1 + R5

I_2 : R1 ← R2 + R3

Write-after-write (WAW)

Output dependency

I_1 : R1 ← R4 + R5

I_2 : R1 ← R2 + R3

Scanned with CamScanner

Control Hazard

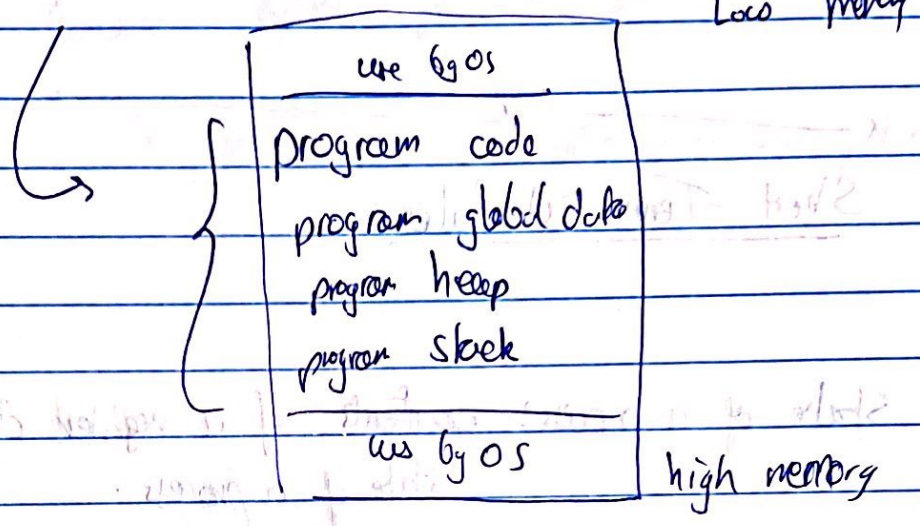
Occurs with Branch, you cannot read in next instruction until we know the outcome of the branch comparison.

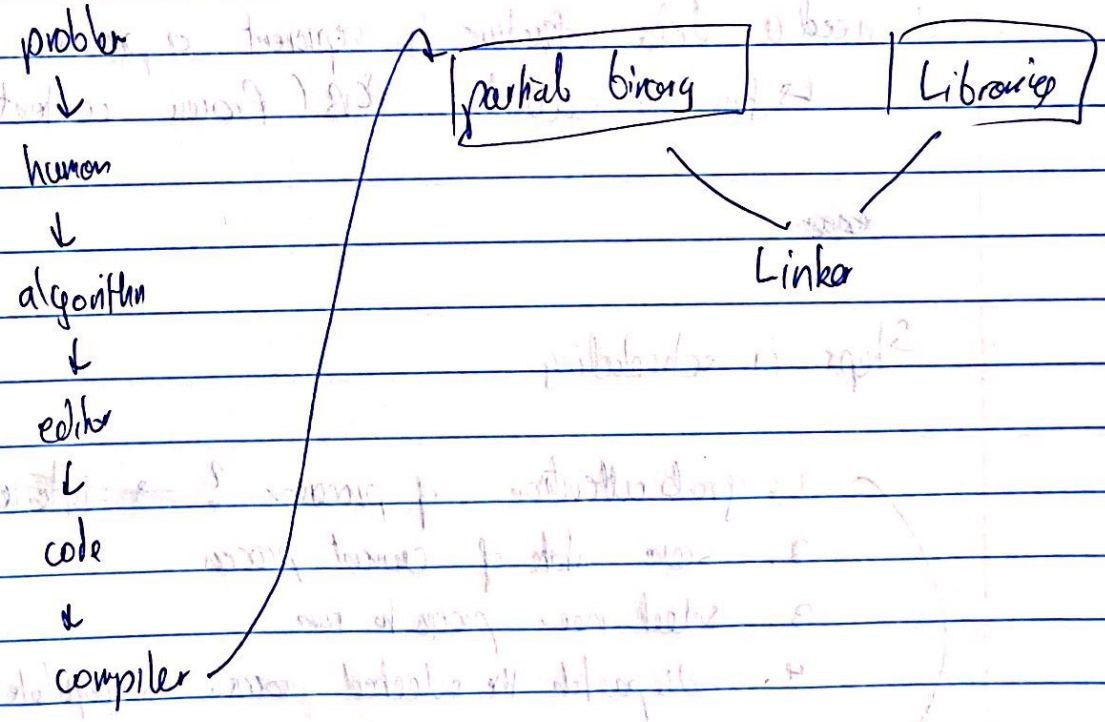
conservative ~~handle~~ handling of branches there will always be atleast 1 bubble
atleast 2 bubbles

Scanned with CamScanner

Processor Scheduling - Chapter 6

Memory footprint of program





Scanned with CamScanner

program vs process?

- process → program in execution

Short-Term scheduler

State of a process :- contents of a register define the state of a process.

∴ need a data structure to represent a process
↳ this is called PCB (Process context Block)

Steps in scheduling

1. grab attention of processor ? ~~→ interrupt~~
2. save state of current process
3. select new process to run
4. dispatch the selected process. → populate registers.

pre emptive vs non preemptive ?
↳ forcing — need interrupt
↳ let the process finish

x1d

Context Switch

Processor going from P1 to P2

↳ (can happen preemptively or non-preemptively)

Scanned with CamScanner

Chapter 5: Hazards

Hazards:- They reduce efficiency of pipeline / pipeline will execute less than one instruction every clock cycle

Structural Hazard

- due to limitations in hardware

Example: If we only had one ALU in EX stage, everytime we encounted a BEQ instruction, ~~we~~ we have to spend 2 clock cycles in Ex stage, one for comparison, other for address computation.
2 solutions.
1. Feedback line
   - Ex should tell IF and ID/RR to pause for one clock cycle
   - EX should also send a NOP opcode to the next stage. → this is called <u>stalling</u> the pipeline
   - ~~This is~~

Note: NOP is a bubble → output buffer remains ~~unchang~~ unchanged from prev clock cycle.

2. Add more hardware
   - adding an extra ALU in EX stage will solve the issue too.

Scanned with CamScanner

Data Hazard

3 types:

1. RAW (Read After Write)

I_1 : $\boxed{R1} \leftarrow R2 + R3$
I_2 : $R4 \leftarrow \boxed{R1} + R5$

2. WAR (Write after Read)

I_1 : $R4 \leftarrow \boxed{R1} + R5$
I_2 : $\boxed{R1} \leftarrow R2 + R3$

3. WAW (Write after Write)

I_1 : $\boxed{R1} \leftarrow R4 + R5$
I_2 : $\boxed{R1} \leftarrow R2 + R3$

RAW Hazard

Why is it a problem?

$$\boxed{IF} \rightarrow \underset{I_2}{\boxed{ID/RR}} \rightarrow \underset{I_1}{\boxed{EX}} \rightarrow \boxed{MEM} \rightarrow \boxed{WB}$$

I_1 is still being calculated when I_2 is in the ID/RR

Scanned with CamScanner

stage where its reading the register file. I_1 will only be written to the reg file in the WB stage which means, I_2 is reading incorrect data from the reg file.

<u>Solution 1</u>: Stall the pipeline until I_1 has been retired (stall for 3 clock cycles)

	IF	ID/RR	EX	MEM	WB	
		I_2	NOP	NOP	NOP	I_1

To do this, ID/RR need hardware to know when to stall.

<u>Register File</u> Add a busy bit to register file.

	B
	B
	B
	B
	B

↳ ID/RR stage determines destination register for an instruction and sets the appropriate B bit for that register (I_1 would set B for register R1)

↳ WB stage will clear B bit when it writes to that register.

↳ When I_2 gets to ID/RR, it will stall till Busy condition goes away (3 clock cycles later)

Scanned with CamScanner

Solution 2: Forwarding

- This will try to reduce the number of stalls.

How does it work?

↳ a stage that generates a new value for a register looks around to see if any other stage is awaiting this new value. If so, they will forward the value to the stage that needed it.

Register File

	B	RP
	B	RP
	B	RP
	B	RP
	B	RP

- We add a bit called RP (Read Pending) to each register.
- When an instruction hits ID/RR stage, if B is set for a register and it wants to read, then it sets RP for the same register
- If EX, MEM, WB sees that the RP is set for a register for which it is generating a new value, then it supplies the generated value to ID/RR. This requires running wires from these stages to ID/RR.

- In our example, we go from 3 bubbles to 0 bubbles as the EX forwards the value to ID/RR.

Scanned with CamScanner

RAW Hazard by Load instructions.

I_1 : Lw $\boxed{R1}$, 0 (R2)
I_2 : Add R4, $\boxed{R1}$, R4

- Value for R1 is not available until MEM stage

$\boxed{IF} \rightarrow \boxed{ID/RR} \rightarrow \boxed{EX} \rightarrow \boxed{MEM} \rightarrow \boxed{WB} \rightarrow I_1$

I_2 NOP NOP NOP

3 bubbles without forwarding.

Since R1 is available in Mem stage, we can forward from Here.

$\boxed{IF} \rightarrow \boxed{ID/RR} \rightarrow \boxed{EX} \rightarrow \boxed{Mem} \rightarrow \boxed{WB}$

I_2 NOP I_1 . . .

forward

- Note: we still need 1 bubble.

WAR Hazard

- Not a problem as the value to read has already been copied into the

WAW Hazard

- simple solution: stall the ~~pipeline~~ instruction that needs to write to a register in the ID/RR stage if it finds the B bit set for the register. This instruction will proceed once the preceeding instruction that set the B bit is cleared by WB stage.

Control Hazards

- refers to breaks in sequential execution of program due to branches.

Solution 1: Stall pipeline until Branch is resolved.

If branch not taken → 1 bubble

IF → ID/RR → EX → MEM → WB →

	BEQ			
	ADD	BEQ		
Bubble.	ADD	NOP	BEQ	
	LW	ADD	NOP	BEQ

However, if branch is taken, we need to fetch from new address which is clocked into PC at the end of

EX stage for BEQ. ∴ There is a 2 cycle stall. if branch taken
else 1 cycle

Dealing with Branches

1. Delayed Branch.

- ignore problem in hardware, let compiler take care of it.
- Assume NOP after BEQ always executes.
- Compiler will replace this NOP with another instruction

2. Branch prediction.
   - assume outcome of branch and start letting instructions into the pipeline
   - If we predict wrong, we flush the pipeline (add another feed back line for flushing from EX to IF and ID/REG)

3. Branch prediction with Branch Target Buffer (BTB)

BTB is a table →

Addr of branch inst	Taken/NT	Addr of target

This table records a history of branches. If we encounter the same branch again, we use the table to predict. If we miss predict, we flush the pipeline.

Scanned with CamScanner

Dealing with interrupts in pipelined processor

- Interrupts can occur at any time

When do we take the interrupt?

① Stop, Drain, go to int
↳ NO
↳ very slow

② ~~Let~~ ~~Board~~ take interrupt immediately?
↳ NO
↳ ~~can~~ issues with leaving the state ~~to~~ changed.

Solution:-

- Interrupts caught be a specific stage of the pipe
- program execution will restart at that execution stage
- stages ahead are allowed to complete
- preceeding stages are flushed.

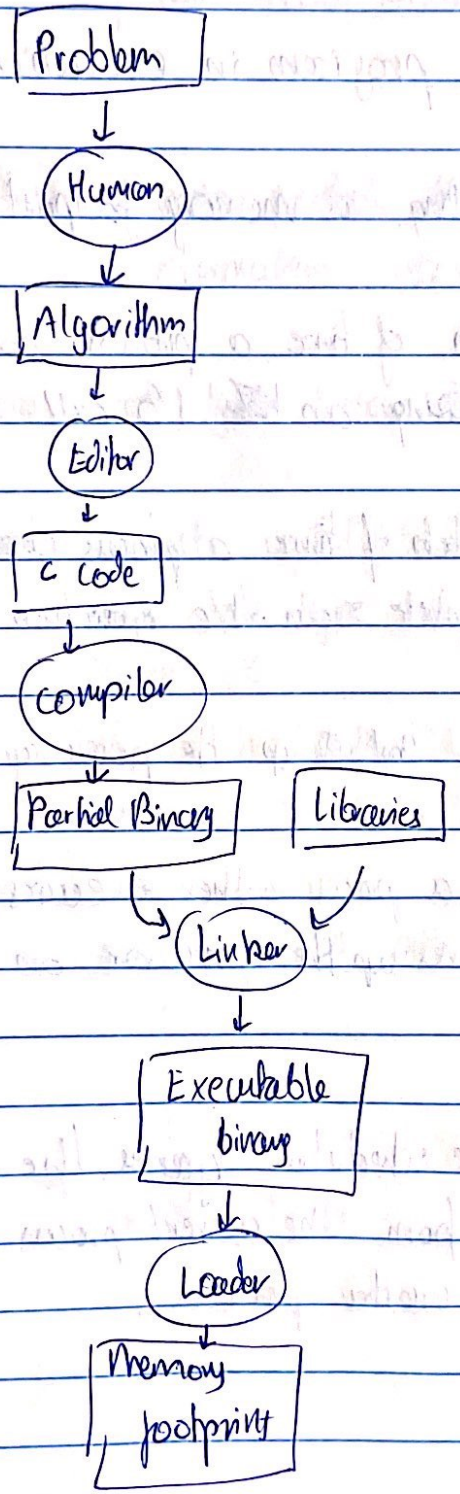
Advanced pipelining

Scanned with CamScanner

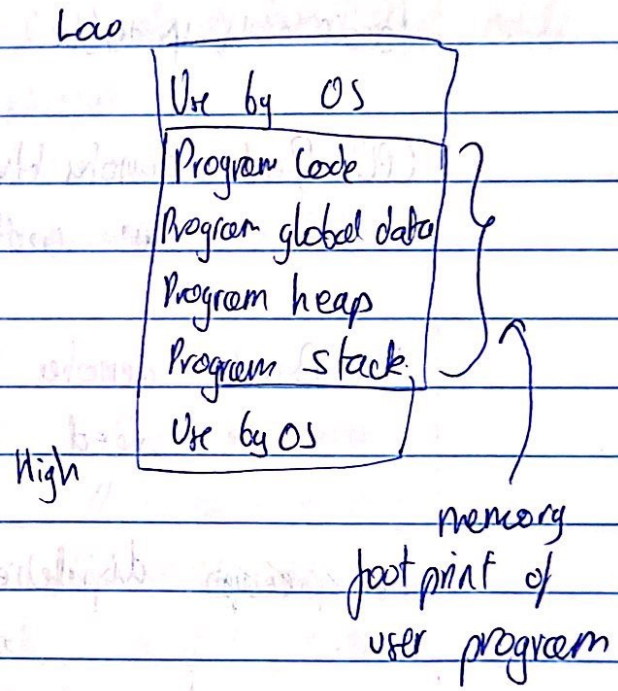
Chapter 6: Processor Scheduling

- How to manage processor, a scarce resource.

Life cycle of program creation



Memory footprint



Scanned with CamScanner

scheduler:- • a functionality of the O.S. that deals with allocating processor resource to programs
• it is a set of routines which like all programs requires the processor to do its work.

process:- a process is a program in execution

loader:- responsible for creating a memory footprint.

CPU Burst:- denotes the sketch of time a process would run without making an ~~I/O~~ I/O call.

I/O Burst:- denotes the sketch of time a process would need to complete an I/O operation

~~non-preemptive~~ dispatcher:- entity that sets up the process registers

non-preemptive scheduler:- a process either executes to completion or gives up the processor on its own volition

preemptive scheduler:- the scheduler yanks the processor away from the current process to give it to another process.

Scanned with CamScanner

steps for scheduling

1. Grab attention of processor
2. save state of currently running process
3. select new process to run
4. dispatch newly selected process to run on the processor

PCB:- Process control block, it is a data structure that holds information about a program.

```
enum state_type {new, ready, running, waiting, halted};

typedef struct control_block_type {
    enum state_type state;          // current state
    address PC;                     // where to resume
    int reg_file[NUM REGS];         // contents of registers
    struct control_block * next_pcb; // list pointer
    int priority;
    address address_space;          // where in memory
    :
    :
    :
} control_block;
```

Scanned with CamScanner

- one PCB for each process
- scheduler maintains all PCBs in a linked-list, the ~~ready~~ ready queue

ready queue

→ [PCB$_1$] → [PCB$_2$] → - - - - → [PCB$_n$]

- role of scheduler is to quickly make its decision and get out of the way.
- PCB does not keep track of internal registers.
- OS also maintains I/O queue

I/O queue

→ [PCB$_1$] → [PCB$_2$] → - - - - → [PCB$_n$]

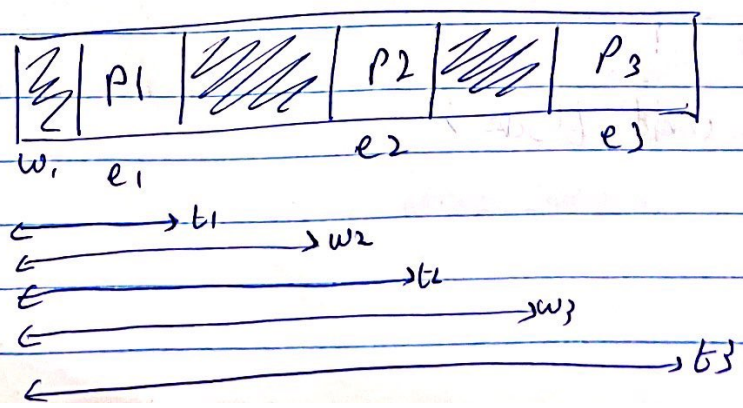
Performance metrics

system centric

- Throughput :- number of jobs executed per unit time
- Average turnaround time :- average elapsed time for jobs entering and leaving the system

user centric

- Response time :- measures elapsed time for a given job.



Throughput = $3/t_3$ jobs/sec

Average turnaround time = $\frac{t_1 + t_2 + t_3}{3}$ secs

Average wait time = $\frac{(t_1 - e_1) + (t_2 - e_2) + (t_3 - e_3)}{3}$ secs.

Starvation :- scheduler has to ensure that all jobs make forward progress towards completion. ~~If any job~~ Starvation is when a job does not make any forward progress. ~~Quantitative~~ Quantitatively, this is shown by an unbounded response time.

<u>Non-Preemptive scheduling Algorithms</u>

- FCFS
- SJF
- Priority

<u>FCFS</u> (First Come first Served)

- prioritise ~~jobs~~ jobs based on its <u>arrival time</u>

- no starvation for any processes.
- convoy effect can be seen : short jobs get stuck behind long ones if the long one arrived first

Scanned with CamScanner

~~Short~~

SJF (Shortest Job First)

- SJF scheduler looks at CPU burst times needed by processes that are ready to run and selects the one that needs the shortest CPU burst.

- better response time for short jobs
- Better average waiting time
- potential for starvation for long jobs.
  - ↳ to overcome this we can use "aging"
  - ↳ aging gives preference to jobs waiting on ~~a~~ ready queue for a long time.

Priority

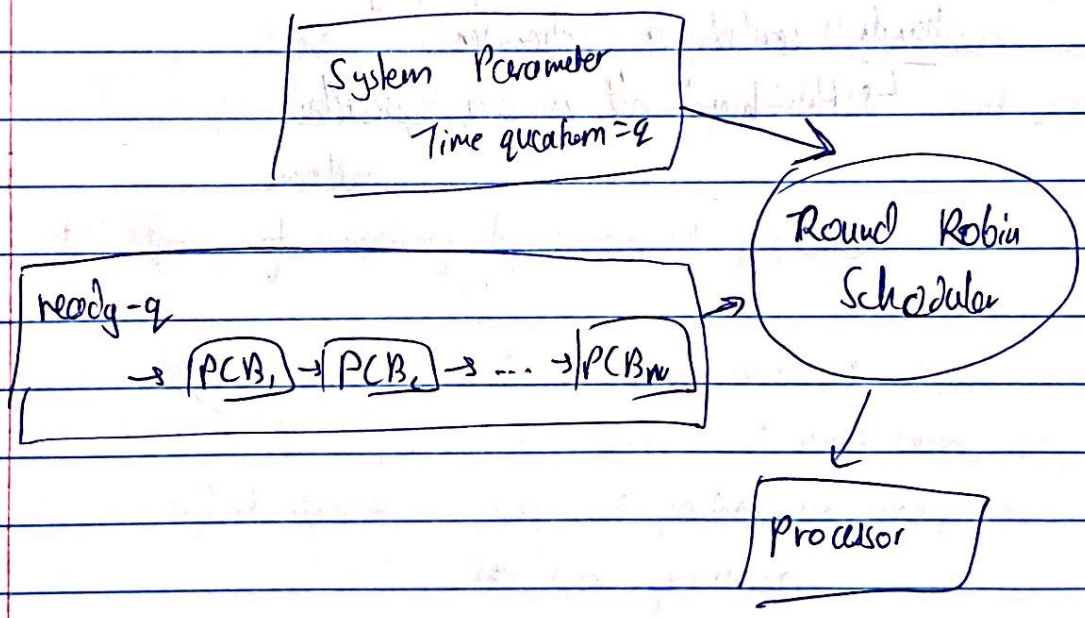
- ready-queue consists of several sub-queues, one level for each priority.
- Each subqueue is FCFS

Scanned with CamScanner

Preemptive Scheduling Algorithms

- scheduler is able to assume control of processor at any time.
- scheduler is able to save the state of the currently running process for proper resumption from point of preemption.
- any previous algorithms can be made preemptive

Round Robin



- Scheduler assigns the processor in time quantum units q also known as time slice
- each process in the ready-queue gets it share of the processor for timeslice q.
- Once the time slice is up, the currently scheduled process is placed at the end of the ready queue, and the next process is scheduled on the processor.

Scanned with CamScanner

- Each process gets the illusion of running on a dedicated processor with a speed 1/n.

- timer:- a hardware device called timer interrupts the processor when time quantum q expires.
- interrupt handler for timer interrupt is an integral part of round robin.
- ↳ upon an interrupt, timer interrupt takes control of processor
  ↳ handler saves context of program in its PCB and hands control to scheduler
    ↳ this hand off is an upcall

Scanned with CamScanner

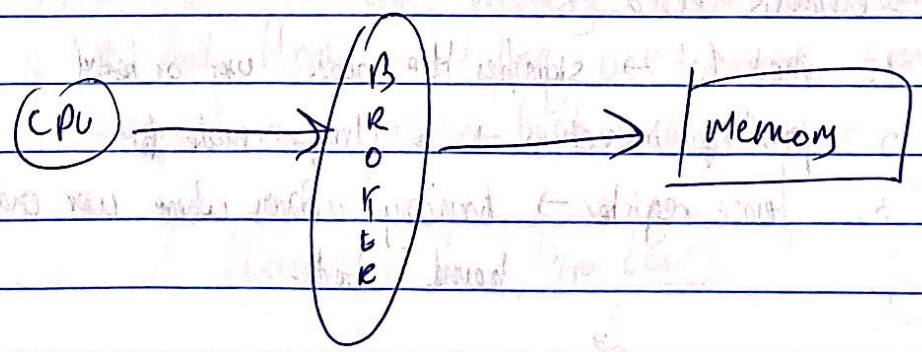
Chapter 7: Memory Management Techniques

Functionalities by memory manager in OS

1. Improved resource utilisation
   Note: daemon processes :- background processes by OS.
2. Independence and Protection
   - independence for each process
   - protection from one another (virus, etc)
3. Liberations from resource limitations.
   - memory manager and architecture work together to devise mechanisms that gives the programmer an illusion of a large memory, while the actual memory may be smaller.
4. Sharing of memory by concurrent processes.

implicit sharing:- multiple browser windows of an app with diff websites but share same code of application

explicit sharing:- copy and pasting an image from ppt to word processor.

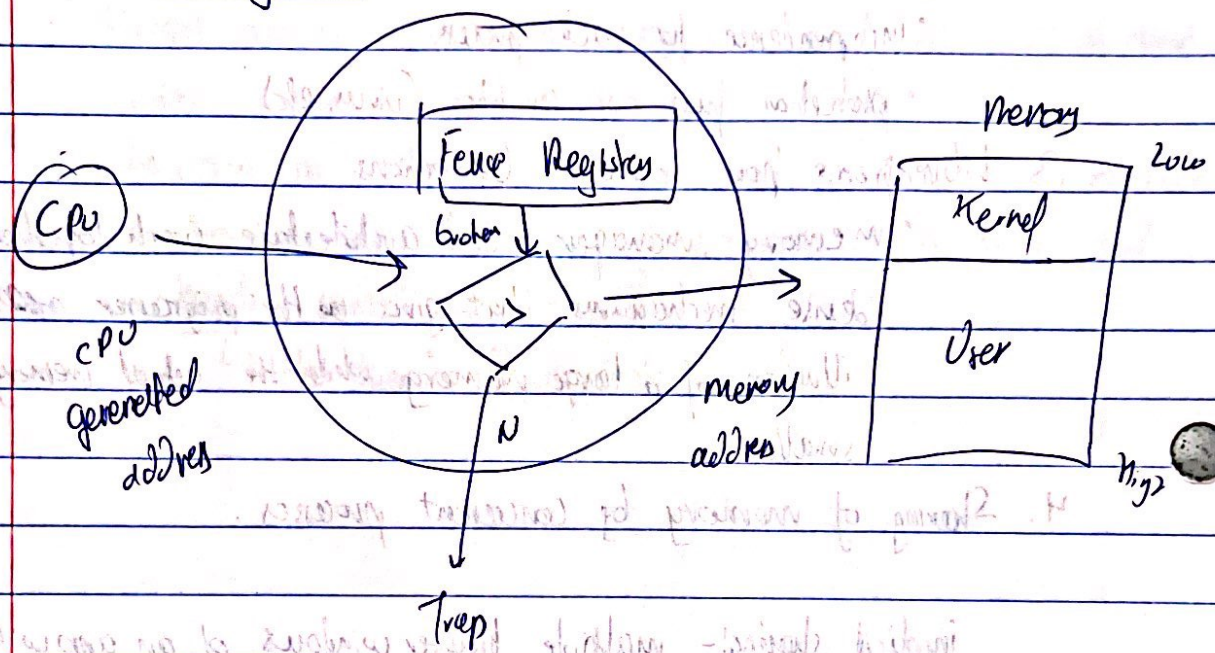


Broker :- piece of hardware that provides mechanism needed for memory management policies. maps a CPU

memory address to a real address in memory.

Simple Schemes for Memory Management

Fence Registers



- create boundary between kernel space and user space in memory.

3 elements needed

1. mode bit → signifies the mode user or kernel
2. privilegen instruction → to change mode bit
3. fence register → boundary address where user ends and kernel starts.

Bound registers

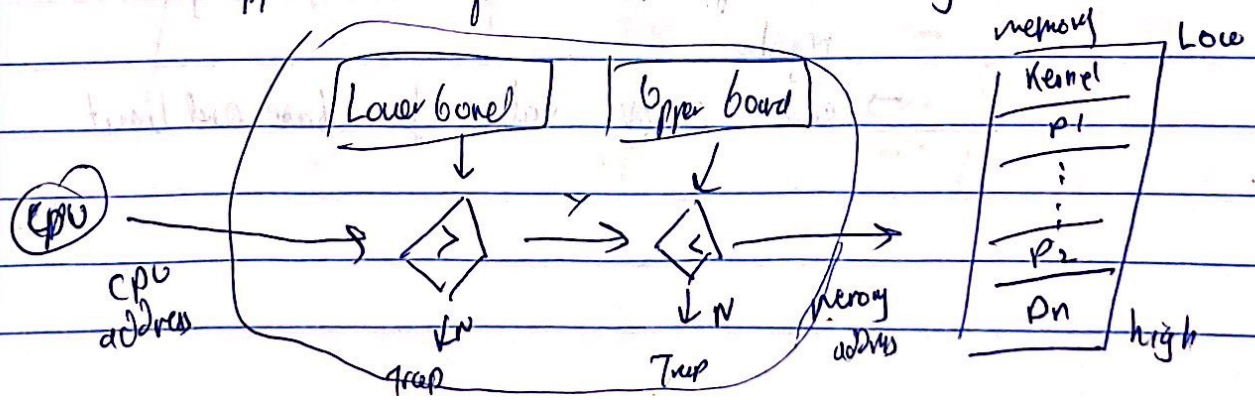
Static relocation :- refers to memory bounds for a process being set at time of linking the program and creating an executable file.

upper bound and lower bound registers :- part of the PCB
↳ linker allocates particular region in memory at link time
↳ loader fixes the upper and lower registers at load time
↳ this never changes for the life of the program

"Swapping" :- act of moving inactive processes from memory to disk to make good use of memory.

⚠ Problem with bound registers :-

If we swap out a process, we can only swap it back into the exact memory location it was assigned. If that block isn't free, we cannot bring the swapped out process back into memory.



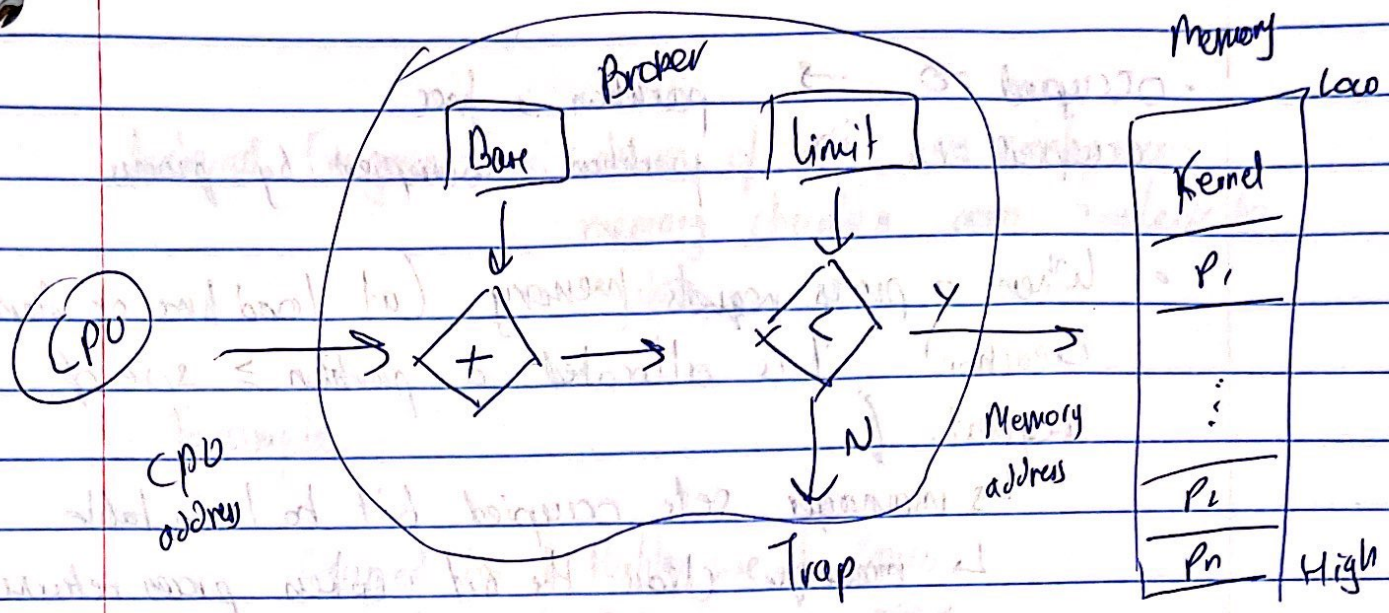
Dynamic relocation

- Static memory → poor memory utilisation.
- Dynamic relocation : refers to the ability to place an executable into any region of memory that can accomodate the the memory needs of the process
- different from static relocation :-
  - ↳ in dynamic, the memory address generated by program can be changed during execution of program, in static they are fixed
  - ↳ when swapped out, it need not come back to the same spot in memory.

architecture support

- Base and limit registers. (Part of PCB)
  - → CPU generated address is shifted by value in base register
  - → loader assigns values for base and limit

Scanned with CamScanner



Memory Allocation schemes

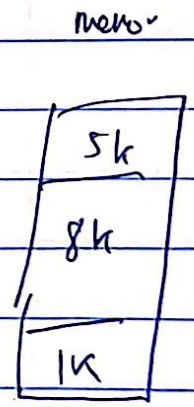
1. Fixed Size Partitions
2. Variable Size Partitions

} hardware :- base + limit registers.

Fixed Size Partitions

- memory manager divides memory in fixed size partitions.
- the following table is kept in kernel space.

Occupied bit	Partition Size	Process
0	5K	xxx
0	8K	xxx
0	1K	xxx



Scanned with CamScanner

- occupied = 0 → partition is free
  occupied = 1 → partition is occupied by process

- When a process requests memory (at load time or during execution), it is allocated a partition ≥ size of request.
  ↳ manager sets occupied bit to 1 in table
  ↳ manager clears the bit when process returns the chunk of memory.

~~Exam~~

Internal Fragmentation :- wasted space internal to fixed size partitions that leads to poor memory utilization.

Internal fragmentation = Size of partition − Actual memory request

example :- partitions free = 1KB, 5KB, 8KB
request = 6KB
memory allocated = 8KB

∴ internal fragmentation = 8 − 6 = 2KB

2KB of memory is unused / wasted.

Scanned with CamScanner

External Fragmentation:- sum of all non-contiguous memory chunks ~~area~~ available to system.

Example:-

Occupied bit	Partition size	Process
0	5K	xxx
1	8K	P1
0	1K	xxx

If another process request 6KB of memory while P1 is is 8KB.

↳ It is NOT possible to satisfy this request eventhough 5+1 = 6KB

why?

↳ they are not contiguous. whereas a process's request needs to be contiguous.

~~Fan~~

Scanned with CamScanner

Variable Size Partitions

- To overcome internal fragmentation, lets allocate variable sized partitions upon request.

- memory manager will build allocation table on the fly.

Starting table. →

Start address	Size	Process
0	13KB	Free

After a few allocations.

Start address	Size	Process
0	2K	P1
2K	6K	P2
8K	3K	P3
11K	2K	FREE

If P1 completes the job, and P4 requests 4KB

→ we still cannot solve the issue of external fragmentation.

Scanned with CamScanner

As processes complete there will be holes in memory
↳ if adjacent blocks of memory are freed, memory manager can combine them into one big chunk.

How to fill holes?

1. Best fit
   → find the space that will waste the least amount of storage.

2. First fit
   → fill a first hole that is free (inefficient)

<u>Compaction</u>

- resorts to compaction when level of external fragmentation goes beyond tolerable limits.

- very expensive, ~~increase~~ cost increases with number of processes.
- essentially its reordering the processes in memory

- it is rarely done

Scanned with CamScanner

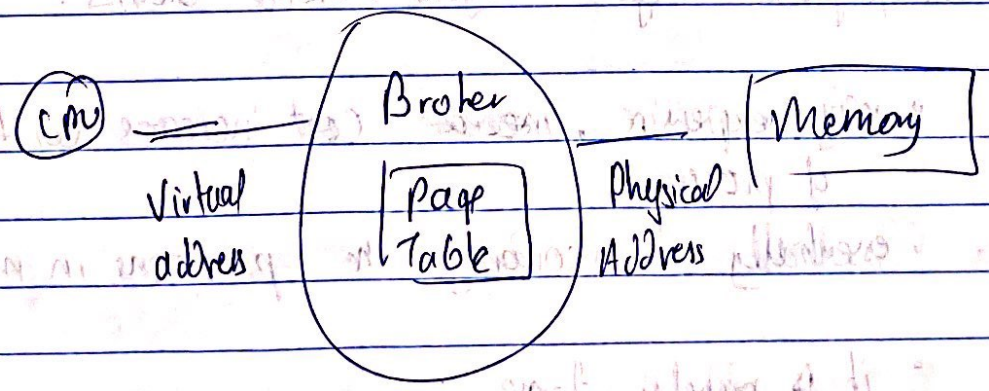
Paged Virtual Memory

- as size of memory grows, external fragmentation increases. Solution?

- till now we have assumed memory to be contiguous.
  ↳ concept of virtual memory helps us get around this problem using paging.

- Idea: preserve the user's view of contiguous memory.

  - Broker breaks up contiguous view into logical entities called pages
  - physical memory broken into page frames / physical frame
  - page size: size of logical page = size of physical frame



size of page table = $2^{(\#\text{ of VPN bits})}$

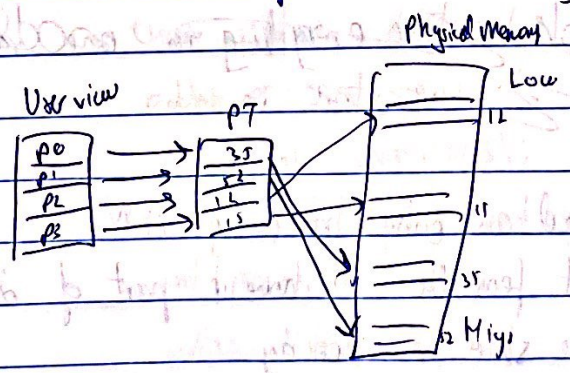
- Broker maintains a mapping between logical page and physical frame in a page table
  - ↳ decouples user's view of memory from physical organization

User view
- virtual memory
- virtual pages
- virtual addresses (generated by CPU)

⇓ Translated by Broker using page table

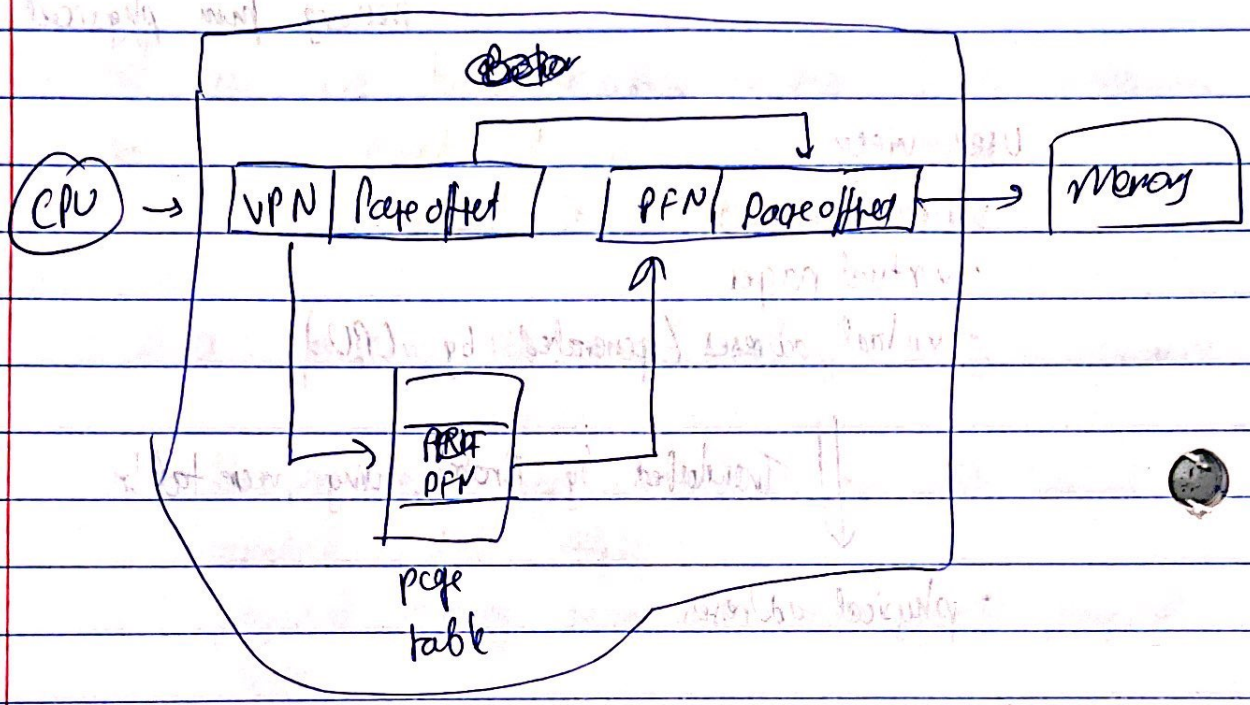
- physical addresses.

∴ size of virtual memory ≠ size of physical memory.



virtual address = 1. virtual Page Number (VPN)
2. offset

VPN translate to physical Frame Number (PFN)



~~Note: It is not feasable to store everything in one table hardware ∴ we store in~~

Note: ① We need an entry for every VPN
∴ not feasable to have it part of datapath
∴ we store in memory
② Provide memory protection for every process
∴ every process has its own page table

Scanned with CamScanner

PTBR

- Since page table is stored in memory, CPU needs to know location of page table in memory
  - ↳ new register in data path → Page Table Base Register (PTBR)
    - ↓ This contains the address of the page table for the currently running process.
- PTBR is part of the PCB
  - ↳ PTBR is loaded at context switch time.

```
typedef struct control_block_type {
    enum state_type state;
    address PC;
    int reg_file[NUMREGS];
    struct control_block *next_pcb;
    int priority;
    address PTBR;
} control_block;
```

Scanned with CamScanner

Chapter 8: Details of Page-Based Memory Management.

Demand Paging.

- programs contain the logic code and code for extreme situations
- a well behaved program only needs little memory, no need to load the entire program into memory.

Demand paging:- idea is to load parts of the program that are not in memory on demand

Hardware for demand Paging

- We know hardware fetches PFN from page table as part of address translation

- with demand paging, the translated page may not be in memory yet

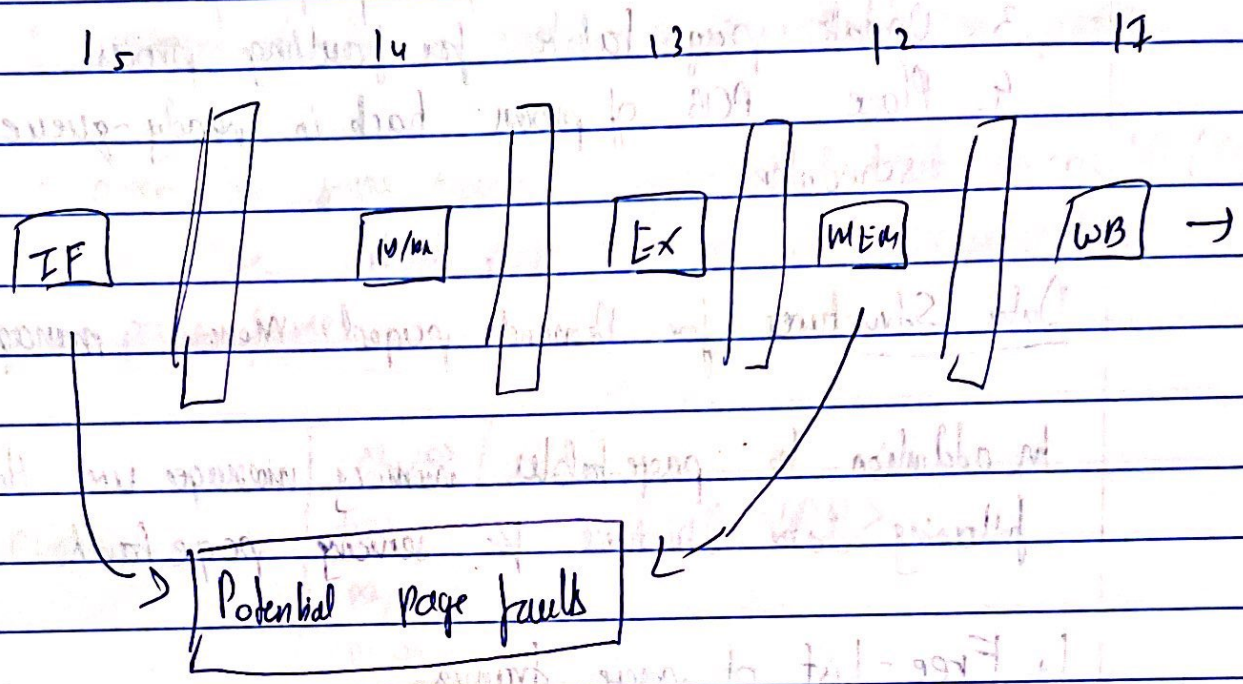
  ↳ this is called a <u>Page Fault Exception</u>

To solve this, we add a valid bit (valid=0 raises page fault)

PTE	Valid	PFN

• OS handles page fault by bringing in the missing page from memory.

But Where can page fault occure?



- Let I1 complete
- squash I5 - I3
- go to int state
- save PC of I2 for restart after servicing page fault.
- go to page fault handler

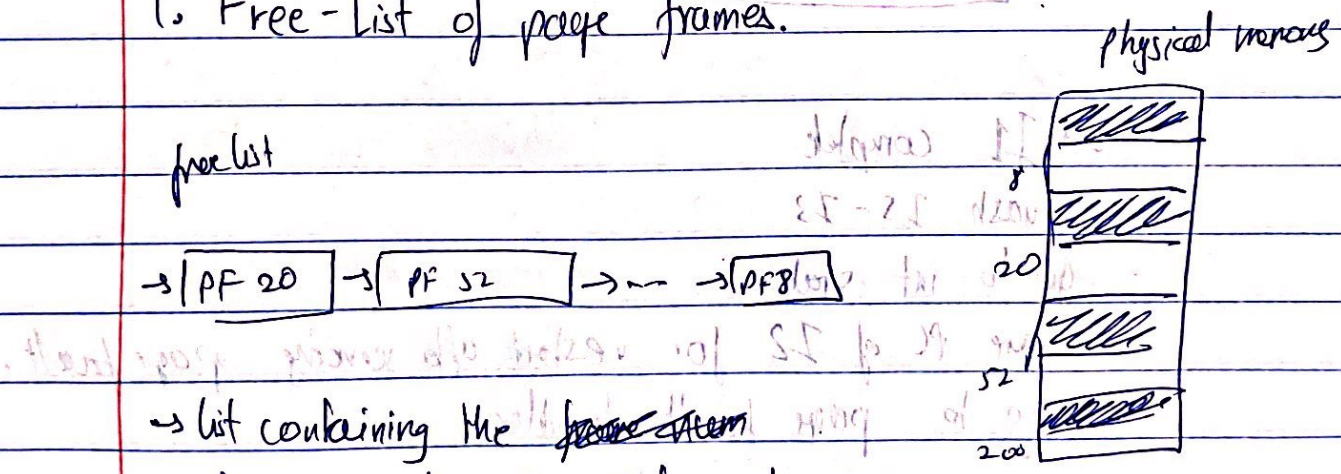
Page fault handler

1. Find a free page frame
2. Load the ~~faulting~~ faulting ~~page~~ virtual page from disk into the free page frame
3. Update page table for faulting process.
4. Place PCB of process back in ready-queue of scheduler

Data Structures for Demand-paged Memory management

In addition to page tables, memory manager uses the following data structures for servicing page faults.

1. Free-List of page frames.

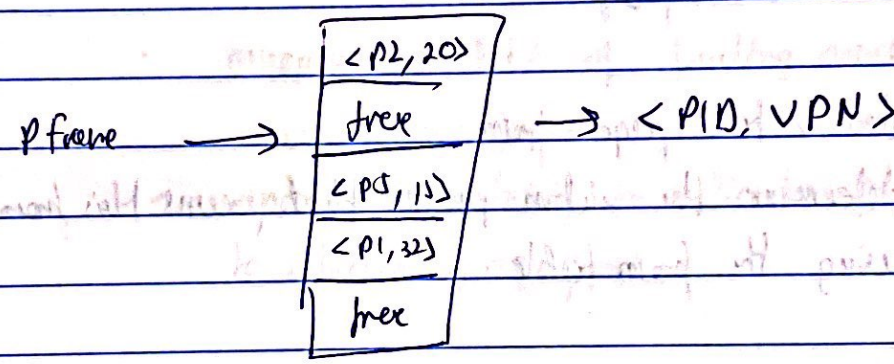


→ list containing the ~~frame num~~ frame number of all free frames

```
struct pframe {
    address PFN;
    pframe *next; }
```

2. Frame Table (FT)

- data structure that contains the reverse mapping of a page table
- number of entries = number of physical memory frames.
- given the frame number, it gives the ~~virtual~~ Process ID (PID) and the virtual page number that occupies this page frame.



3. Disk Map (DM)

- ~~each~~ DM analog to page table.
- one for each process
- maps the process virtual space to locations on disk that contain contents of the page.

Scanned with CamScanner

Anatomy of Page Fault

1. Find a free page frame:

- page fault handler looks up the free-list of page frames.
- If not empty, select one
- if empty, memory manager has to select a victim to make room for the faulting page

2. Pick the victim page

- select victim page frame
- determine the victim process that owns this frame using the frame table.

- distinction between clean page and dirty page

  - clean page:- page that has not been modified by program. which means the memory copy is identical to disk copy.
  - dirty page:- page that has been modified by program

Scanned with CamScanner

- If page is clean, manager sets PTE corresponding to page as invalid
- If page is dirty, then manager writes page back to disk determining disk location from disk map

3. Load the faulting page: using disk map for the faulting process, manager reads in the page from disk into selected page frame

4. Update page table for faulting process and the frame table.

- mapping for PTE of faulting process points to selected page frame, makes entry valid
- update frame table to indicate change in mapping to selected page frame

5. Restart faulting process

- Place PCB of faulting process on ready - queue

Note:- we cannot allow page fault handler to page fault. OS ensures certain part of OS such as page fault handler are always in memory.

Scanned with CamScanner

Page Replacement Policies

page replacement policy:- process of selecting a victim page to evict when there is a page fault and free list is empty.

Good policy

1. policy should result in least number of page faults. ∴ time spent by OS managing page faults is minimized.

2. policy should strive not to incur a page fault for a page that has just been brought into memory.

2 options for victims

1. Local victim selection
   - idea is to steal a physical frame from faulting process itself
   - simplicity → no frame table
   - poor memory utilisation
2. Global victim selection
   - idea is to steal physical ~~ph~~ frame from any process

Scanned with CamScanner

Belady's Min

- Ideally, if we know the entire string of references, we should replace the one that is not referenced for the longest time in the future
- Not feasable :- memory ~~does~~ manager does not know the future memory references.

Random Replacement

- simplest policy
- replace a page randomly

First In First Out (FIFO)

- How it works
  1. Affix a timestamp when page is brought into physical memory
  2. if page has to be ~~ex~~ replaced, choose longest resident page as victim

- no hardware assist needed
- datastructure needed :- circular queue
  - Head and tail point to 0
  - insert at tail
  - longest resident is at head ∴ remove from Head

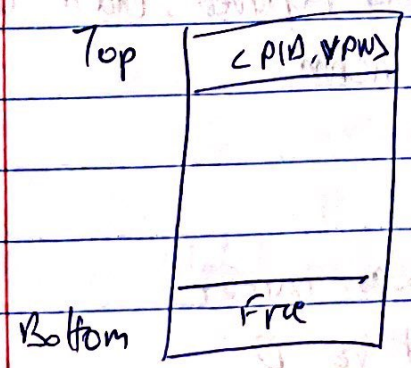
Scanned with CamScanner

- memory manager uses a "full" flag to indicate when queue is full
- queue size = # of physical frames
- if full flag is True, (head pointer - tail pointer) this calls for page replacement on next page fault. → memory manager replaces page at head
- circular queue serves the purpose of free-list and frame-table.

Least Recently Used (LRU)

- makes the assumption that if a page has not been referenced for a long time, there is a good chance it will not be referenced in the future
  ∴ victim = page that has not been used for longest time

- Push down stack



- Every memory access, CPU pushes page on top of stack, if already in stack, it is reordered to top
- Bottom of stack is least recently used
- size of stack = number of frames

Scanned with CamScanner

Approximate LRU : Small hardware stack

not viable to have LRU stack as hardware
1. high latency for such large datastructure in hardware
2. modifying stack slows down processor

Lets have a smaller stack of size N,
then stack maintains the last N references while the algorithm would pick a random page not in the stack as victim.

Approximate LRU : Reference bit per page frame

- one possibility is to track references at the page level instead of individual accesses.
  - ↳ reference bit per page frame in ~~page tab~~ page table
  - ↳ hardware sets bit when CPU accesses any location in page, software reads and resets it

Choosing victim
1. memory manager maintains bit vector per page frame called reference counter

Scanned with CamScanner

see text book. page 327

2. Periodically the memory manager reads the reference bits of all page frames and keeps them in the MSB of corresponding reference counters.

Scanned with CamScanner

(Clock algorithm)

Second chance page replacement algorithm

- extension of FIFO
- gives second chance for a page to <u>not</u> be picked as victim
- ↳ use reference bit set by hardware as an indication that page deserves a second chance to stay in memory.

algorithm

1. Initially, OS clears reference bits off all pages. hardware set reference bit for pages referenced by program

2. If a page has to be replaced, ~~victim~~ replacement candidate chosen by FIFO.

3. If candidate's reference bit is set, manager clears the bit, gives it a new arrival time and ~~repeat~~ moves it to end of FIFO queue

4. victim is first candidate in FIFO queue who's ~~&~~ reference bit is not set.

Scanned with CamScanner

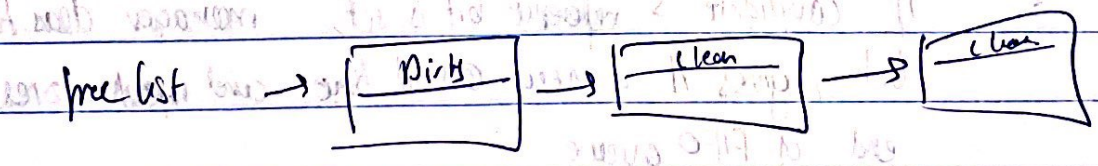
Optimisation of Memory management

1. Pool of free page frame:-
   - not a good idea to wait for page fault to select page frame
   - manager always keeps a minimum number of page frame ready for allocation
     - handled by paging daemon (periodically checks free list)

2. Overlapping I/O with processing:-
   - dirty page needs to be written to disk before getting added to free-list.

   Instead,

   we can have a write I/O queue ~~and~~ where the dirty page is added to this queue and the free list.
   - The write to disk happens concurrently with ~~the~~ a CPU activity
   - manager skips over dirty page on free list



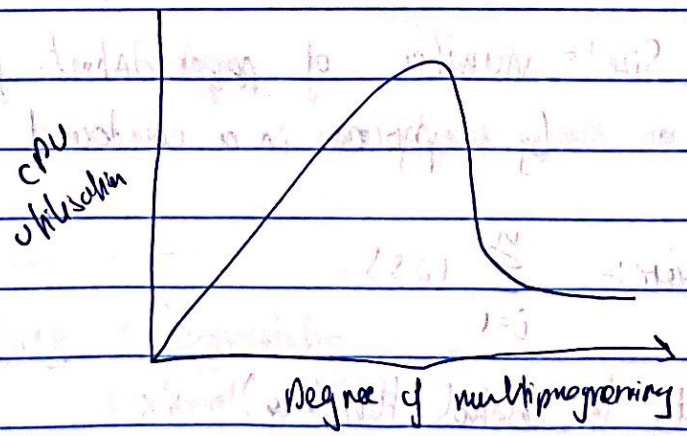
3. Reverse mapping to page tables.
   - To meet min threshold requirements, paging daemon may take away page frames from a process.
   - If the same process wants to run, it will page fault.
   - ~~We can add~~ To every node in free-list, we add

reverse mapping <PID, VPN> showing last hosted virtual page

- upon page fault, <PID, VPN> of faulting process compared to entries in free-list, If found then ~~rest~~ re-establish mapping of matched entry
- removes the need ~~of~~ for additional read/write to disk
- this along with clock sweep gives frames a 3rd chance.

Thrashing

- Thrashing:- system is not getting useful work done
  of → ↳ processes spend more time paging than computing
- Example: If there are too many processes in memory (high degree of multiprogramming) then process are continuously paging against one another to get physical memory ∴ non of the process are making forward progress
  Solution:- reduce degree of multiprogramming



- Paging is implicit I/O ∴ with excessive paging, CPU bound process

Scanned with CamScanner

can become I/O bound

∴ CPU scheduling should also take into account memory usage

How to control thrashing?

- ensure each process has sufficient number of pages frames allocated so it does not page fault frequently.
  - ↳ principle of locality to ~~reduce~~ be used
    - ↳ in a reasonably sized window of time, process only accesses small portion of its memory footprint
      - ∴ we only page fault when locus changes

Working Set :- set of pages that defines the ~~locus~~ locus of activity of a program.
(working set changes over time)

(WSS) Working Set Size :- number of ~~pages~~ distinct pages touched by a process in a window of time

memory pressure :- $\sum_{i=1}^{n} WSS_i$

How exactly to control thrashing?

1. If total memory pressure exerted on a system is greater than total available physical memory,

then ~~decrease~~ manager decreases degree of multiprogramming and vice versa.

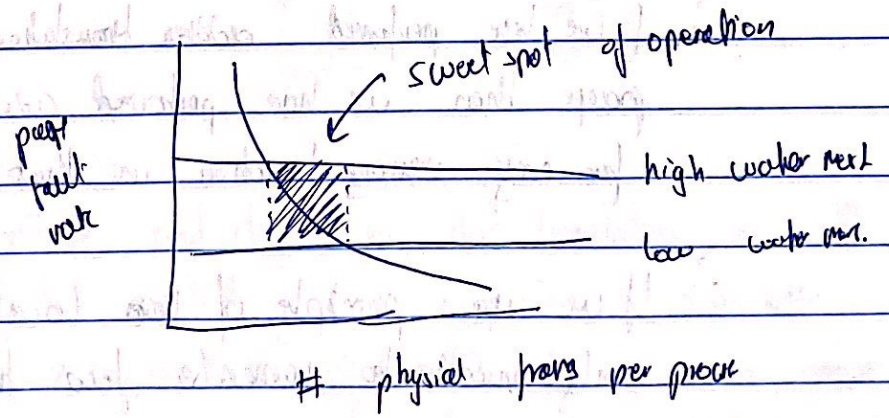
WSS is measured every Δtime units by sampling reference bits of physical frames.

↳ after sampling reference bits, they are turned off.

2. another method is to use observed page fault rate as a measure of thrashing.

low water mark :- if page fault rate reduces below this, increase ~~deg~~ degree of multiprogramming

high water mark :- if page fault rate goes above this, reduce degree of multiprogramming



Paging + Fragmentation

External → No

Internal → Yes

Scanned with CamScanner

Translation Lookaside Buffer (TLB)

With virtual memory
2 memory accesses
1). PTE
2) Actual memory content

Can we reduce it to 1 memory access? Yes.

- entire PT cannot be in ~~memory~~ hardware, but a subset can

- within a page, memory locations are ~~cont~~ contiguous
  ∴ if we have performed address translation for a ~~page~~ page then we have performed address translation for every memory location in that page

  ∴ If we use principle of ~~loc~~ locality, we only need to remember few translations at a time

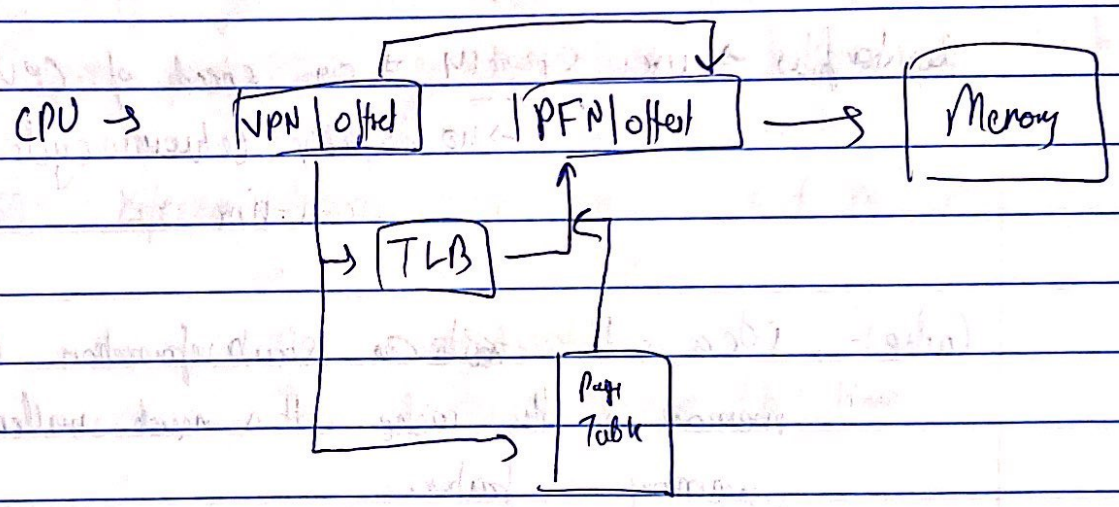
∴ TLB is a small hardware table which holds recent address translations
- ~~address~~ address translation ~~of~~ for a page needs to be done atleast once ∴ we need valid bit

Scanned with CamScanner

TLB

User/Kernel	VPN	PFN	Valid

TLB split in two parts - User and Kernel
↳ on context switch, OS invalidates all user space translations



Process

- Hardware first checks for valid translation in TLB
  - If hit in TLB, good, avoided trip to page table in memory
  - If miss in TLB, page table entry in memory supplies PFN
  - hardware enters resulting translation into TLB for future access

hardware needs to look through entire table to find a match
∴ TLB is called Content addressable memory (CAM)
↳ special case for cache

Scanned with CamScanner

Chapter 9: Memory Hierarchy

access time:- delay between submitting a request to memory and getting data.

cycle time:- time gap between two successive requests to memory

DRAM → diff cycle time and access time

transmission delays on buses affect these

Register file → uses SRAM → on speed of CPU
↳ no disparity between cycle time and access time

Cache:- idea is to ~~stash~~ stash information brought from memory in the cache. It is much smaller than main memory ∴ faster.

CPU → cache → memory

Why cache works?

Principle of Locality:- a program tends to access a relatively small region of memory in any given interval of time regardless of its actual memory footprint.

1. spatial locality:- high probability of a program accessing adjacent memory locations
↳ contiguous memory

Scanned with CamScanner

2. Temporal Locality:- high probability of memory accessing same memory location in the ~~near~~ near future → loops.

Terminology

Hit:- CPU finds content in cache

(h) Hit Rate:- probability of successful lookup in cache

Miss:- CPU fails to find content in cache

(m) Miss Rate:- probability of unsuccessful lookup in cache = $1-h$

Miss Penalty:- time penalty for servicing a miss

EMAT:- Effective Memory Access Time:- $T_c + m * T_m$

m = cache miss rate

T_m = miss penalty

T_c = cache access time / hit time

Why multilevel cache? (instead of 1 big cache).

↳ address the problem of access time and miss rate together

↳ 1st cache optimized for speed (∴ small) } on chip

↳ 2nd cache optimized for miss rate (∴ big) }

some also have L3 cache which is off-chip

$EMAT_i = T_i + m_i * EMAT_{i+1}$

Note:- Secondary storage acts as virtual memory, i.e. it is accessed upon page fault.

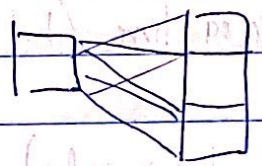
Scanned with CamScanner

Cache organization

1. placement :- where in cache to put data → mapping func
2. algo for lookup :- how to find something in cache → metadata
3. validity :- how to know if data in cache is valid. → valid bit

Direct Mapped cache

- one-to-one correspondance between memory-location and cache location
  - ↳ i.e. ~~given~~ every memory address has one designated spot in cache



index = memory size ~~%~~ mod cache size

compulsory misses :- misses caused due to cache being empty.
↳ also ~~on~~ when a location have ~~never~~ never been in memory.

$\boxed{\text{ex. mem} \to \emptyset 8}$

conflict miss :- ~~cache~~ occurs when 2 memory locations that have been in cache are fighting for same location, despite there being ~~unused~~ unused locations.

$\boxed{\text{ex} \to \text{mem} \to \emptyset 80}$

Scanned with CamScanner

Cache lookup for direct-mapped cache

- First check cache, if miss then go to memory

cache index = memory address mod cache size.

↳ essentially, we are taking n least significant bits from memory address if size of cache is 2^n entries

↳ ex. if cache size is 8 entries, then we need 3 LSB of memory address for cache index

- since multiple memory locations can exist in the same cache location, we need more information to distinguish between mappings

↳ tag:- the bits that were dropped to generate the cache index are the tag bits

↳ tag stored in cache

is this enough? No → why?

Let's assume we haven't fetched anything yet. If the CPU checks a cache location for the tag bit 0, it is going to find it, but stores nothing in memory. We therefore need a valid bit to make sure this doesn't happen.

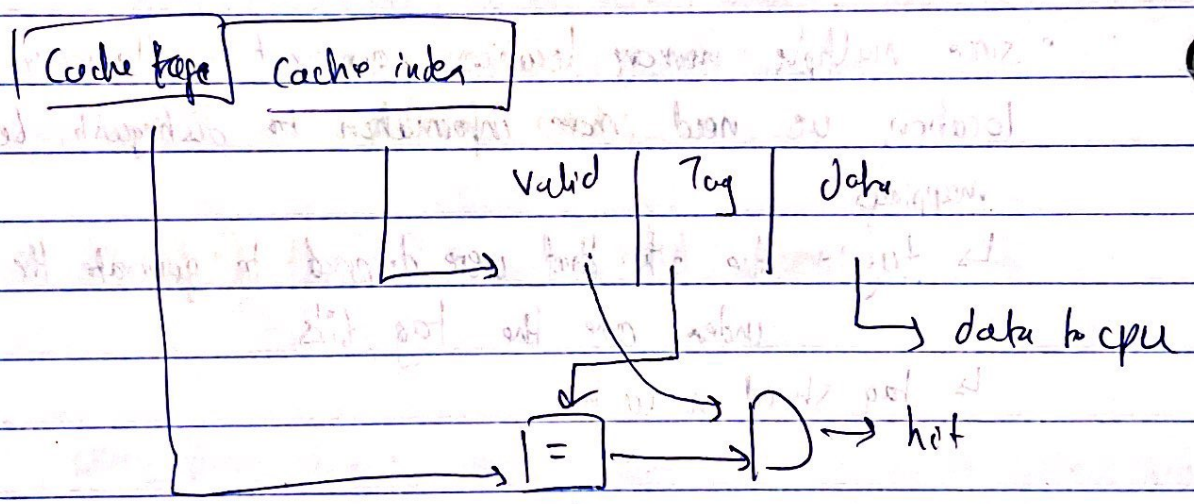
Scanned with CamScanner

Cache

	Valid	Tag	Data
cache index 0			
" " 1			
" " 2			
" " 3			
⋮			

memory address is = | Cache tag | cache index |

hardware for direct-mapped cache.



Imp cache line = cache block = cache entry = block.

valid, tag bits } metadata ⇒ space overhead = $\frac{\text{metadata}}{\text{total space}} \times 100\%$

till now we have assumed it to by word addressable.
but it depends on granularity of memory access allowed.

Scanned with CamScanner

What if it byte addressable?

- we can have one byte per cache line but having several bytes would be better by spatial locality.

Cache tag	cache index	byte offset

tip one way to reduce space ~~overhead~~ overhead would be to have multiple ~~blocks~~ words per cache line since its byte addressable.

memory stall :- processor cycles wasted due to waiting for memory operation to complete.

↳ read stall
↳ write stall

Cache read / write algorithm

IF / Mem stage accesses cache / mem

- read access from CPU
  1. Step 1 :- CPU sends index to cache, cache does lookup, if hit then cache supplies data to CPU. If miss then send address to memory. (1 clock cycle)
  2. Step 2 :- ~~CPU~~ upon sending address to memory, send NOPs down the pipeline until data received from memory. (Read stall)

Scanned with CamScanner

main memory delivers data to CPU and updates cache data and metadata (set valid bit)

- Write accesses to cache from CPU
  - ↳ Write through policy
  - ↳ write back policy.

<u>Write through policy</u> - (update cache and mem together)

1. on every write instruction, CPU writes to cache (no checks needed). cache updates data, tag field and sets valid bit. This happens during MEM stage
2. Simultaneously, CPU sends address and data to main memory. Due to speed disparity, CPU sends this to a write buffer (similar to reg file). This also happens during Mem stage
3. write buffer completes write independently of CPU. if write buffer full, then write stall until there is space in write buffer.

write allocate:- write to cache and buffer

no write allocate:- only write to buffer

Write back policy (update only cache)

1. if write hit, cache updates tag field, data, set valid bit. if write miss, do read miss steps then do write hit steps. (happens during Mem stage)
2. cache replacement policy.

This requires a dirty bit for each cache line

↳ clear bit upon processing a miss

↳ set bit upon CPU write

Note:- write buffer is also used in write back policy as servicing miss is more important than writing back dirty block that is being replaced.

optimization

① multiple writes to same location can be replaced with latest write in write buffer

② write merge :- merges independent writes to different parts of same memory block

write back is faster but dirty, and write through is cleaner but slower for multilevel memory, use write through for close level memory (L1) and write back for further memory (L2, L3)

Scanned with CamScanner

Exploit spatial locality to reduce miss rate: bring back a bigger block of data from memory upon a miss.

cache tag	cache index	block offset
t	n	b

$b = \log_2$ (Block size)

num of lines L = cache size / block size

$n = \log_2(L)$

$t = \underbrace{a}_{\text{number of bits in mem address}} - (b + n)$

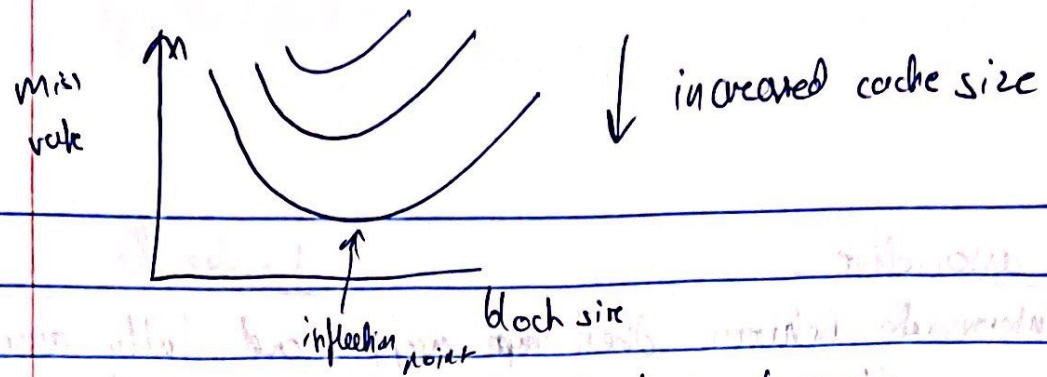
read / write now happens with the entire block.

<u>Performance implications of increased blocksize</u>

~~does~~ will miss rate decrease forever?

No, miss rate depends on working set. Working set of program ~~may~~ changes over time ∴ large block = ~~no~~ high miss rate.

Scanned with CamScanner



~~the can get performance~~

processor performance can be affected negatively before the inflection point by increased miss penalty.

∴ to optimize we need to reduce miss rate + miss penalty

Fully Associative Cache

- cache block can host any memory block.
  - ↳ ∴ only compulsory miss and capacity miss

memory address | Cache tag | Block offset |

- cache has to search through ~~entire~~ all entries to find cache tag match.
  - ↳ one comparator for each entry so that comparison can happen in parallel.
  - ↳ too complex for any feasable sized cache
- to make optimum use of a fully associative cache, we need a good ~~rep~~ cache replacement algo.
- example of fully associative cache :- TLB

Scanned with CamScanner

Set associative

- intermediate between direct ~~map~~ mapped and fully associative.
- memory block can be associated with a set of cache blocks
- the amount of tag matching hardware required equals the degree of associativity.

cache tag	cache index	block offset

Instruction & Data cache

- combining these 2 caches will increase hit rate
- possible structural hazard:-

Reduce Miss penalty

- write buffers
- bus cycle time:- time taken for each data transfer between processor and memory. (amount of data transfer = memory bandwith). bus cycle time depends on the memory bandwith which depends on the number of data lines connecting the bus.

↳ However, miss penalty can be less than this as we can send the requested data first, then the rest later

Note: if we get a very big cache block, then we can get a second miss while still servicing the first

Chapter 9 continued

Cache Replacement Policy

- In direct-mapped cache, placement policy predetermines candidate for replacement
- In set associative / fully associative, we need to decide

- Temporal locality suggests using LRU

↳ fully associative :- apply LRU across all blocks
↳ set associative :- ~~apply~~ LRU limited to set that houses the missing memory reference.

- we need more metadata to keep track of LRU

C0			C1			LRU
V	Tag	Data	V	Tag	Data	

On each reference hardware sets LRU bit for set containing ~~cor~~ currently accessed memory block. & if C0 accessed, LRU = 0
if C1 accessed, LRU = 1.

∴ candidate for replacement is C0 if LRU = 1
candidate for replacement is C1 if LRU = 0

↳ time penalty as hardware needs to update LRU bit on every reference.

Scanned with CamScanner

While this works for 2 sets, it doesn't for more than 2.

Example, let there be 4 sets.

↳ we can have 2 LRU bits but that only tells us the ~~least~~ ~~recently~~ most recently used.

↳ instead we need an ordering vector.

↳ CPU updates ordering on each access.

C_0 C_1 C_2 C_3 LRU

$C_1 \rightarrow C_2 \rightarrow C_0 \rightarrow C_3$ replacement = C_3

$C_2 \rightarrow C_1 \rightarrow C_3 \rightarrow C_0$ replacement = C_0

Complex hardware :- 4 sets ∴ $4! = 24$ possible ordering combinations

∴ 5 bit counter to track 24 states + FSM to ~~change~~ pick candidate

if we had 8 sets ∴ $8!$ states + FSM

Types of misses

compulsory miss :- program accessing given memory location for first time during execution (cold miss)

not possible in fully associative cache by definition

conflict miss :- X used to be in the cache, cache is not full but is occupied by another memory

capacity miss :- X used to be in the cache, cache is full ∴ we have no choice but to evict.

Scanned with CamScanner

Note: we can have compulsory and capacity miss at the same time but compulsory miss dominates

$\boxed{\text{compulsory} > \text{capacity} > \text{conflict}}$

Integrating TLB and cache

TLB can be setup as Direct mapped or fully associative

Path of memory access

1. CPU generates a virtual address
   ↳ VPN broken into tag and index for TLB look up
2. TLB does virtual address to physical address translation
   If hit in TLB, no problem
   if miss in TLB, pipeline freezes
3. PA then goes to cache (I-cache or D-cache)
   if hit in cache, no problem
   if miss, freeze to pipeline to service miss

Cache controller: hardware interface to the processor for cache interrupts.

Scanned with CamScanner

Virtually indexed Physically tagged cache

- every memory access goes through TLB and then cache
  ↳ TLB in critical path but big delay due to sequential nature of path.
  Solution:- do TLB address translation and cache lookup in parallel.

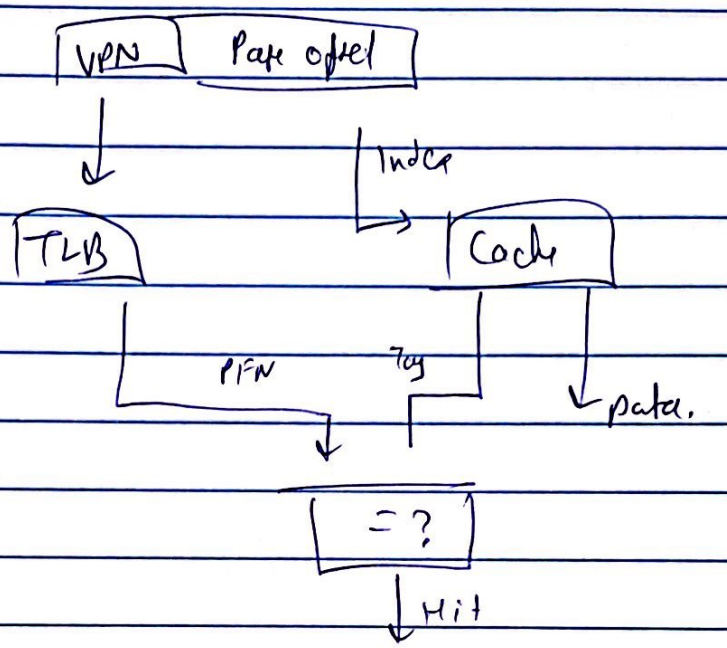
Normally,
VA ~~[illegible]~~ change to PFN and offset. and cache index comes from ~~[illegible]~~ LSB of PFN

But,
to Virtually indexed physically tagged cache:-
cache index comes from offset which doesn't change, then we can lookup cache in parallel. Tag comes from PFN

Limitation:- cache size restricted to size of offset
we could ~~[illegible]~~ increase the size of set associativity but this too is limited by ~~[illegible]~~ hardware complexity.

page coloring:- Since memory manager ~~[illegible]~~ sets up VPN to PFN mapping. Page coloring is a technique used to guarantee that more

of the virtual address bits will remain unchanged by translation process. This allows processor to have a larger cache size independent of page size (there offset)



Main memory

- one way to cut down miss penalty is increase ~~the~~ ~~pro~~ size of processor - memory bus.

Memory Interleaving

- idea is to have multiple banks of memory.
- each bank responsible for providing 1-word.
  ∴ 4 ~~bto~~ word block size can have 4 banks. Then send it back sequentially.

Scanned with CamScanner

Chapter 13: Fundamentals of Networking and Network Protocols

Basic Terminologies

host:- computer that hooks on to the network

NIC or Network Interface Card:- peripheral controller that allows to hook up our computer to the network

routers:- entities that sit in the core of the network and route messages between ISPs (using the IP address)

IP address or Internet Protocol Address :- universal addressing system for every device connected to the internet.

Internet: network of networks

Packet flow on Internet ~~exep~~ explained:-
a packet traverses through a series of queues that reside at the input of routers en route from source to destination. When there's no traffic, i.e. queues are empty, the packet sails through. However, in the presence of traffic (network congestion) a packet may experience queueing delays. If a queue associated with a router is full, then packets may be dropped leading to packet loss.

Scanned with CamScanner

Networking Software

protocol :- an agreed upon convention for two entities to interact with each other

Networking software is a crucial part of the OS, it is called the protocol stack.

↳ what is a network protocol?

↳ it is a language that defines the syntax and semantics of messages for computers to talk.

Example: Ethernet → protocol used on LAN

↳ max packet size 1518 bytes :- frame ~~[illegible]~~

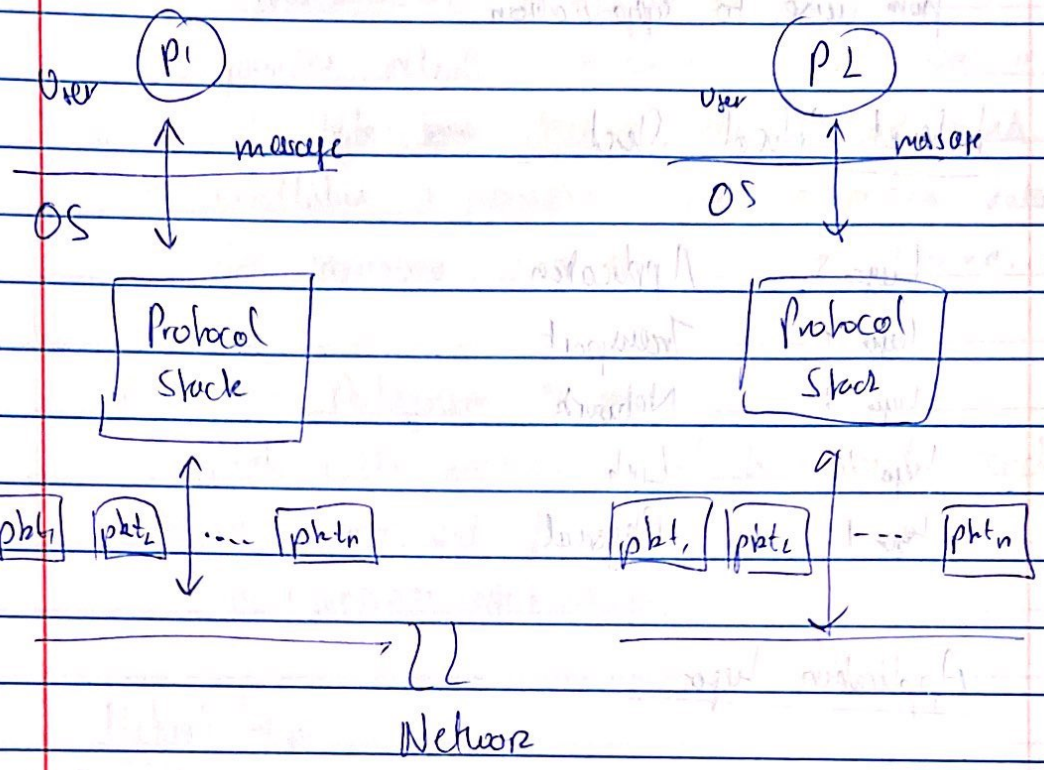
↳ frame contains data, destination address etc

Set of Problems on a network

~~1. Arbitrary me~~

1. Arbitrary message size and physical limitations of network packets
2. Out of order delivery packets
3. Packet loss in the network
4. Bit errors in transmission
5. Queuing delays en route to destination

Scanned with CamScanner



- The portion of the OS that addresses these problems is the <u>protocol stack</u>.

Protocol Stack

- One possibility:- Implement it as one big layer.
  - ↳ Problem:- if physical network changes, it affects the entire thing.
- Instead, we use Protocol Layering. (use power of abstraction)
- Why stack?
  - ↳ outgoing messages are pushed down the layers of the stack from application to physical medium
  - ↳ similarly incoming messages are popped up the stack

Scanned with CamScanner

from wire to application

Internet Protocol Stack

Layer 5	Application
Layer 4	Transport
Layer 3	Network
Layer 2	Link
Layer 1	Physical

Application Layer

- responsible for supporting network-based applications
- protocol examples
  - → HTTP → web apps
  - → SMTP → electronic mail
  - → FTP → file transfer

Transport Layer

- responsible for taking the application layer message and ferrying it between the ends of ~~communication~~ communication
- worries about breaking a message into packets / out of order delivery.
- TCP & UDP

Scanned with CamScanner

TCP - Transmission Control Protocol

↳ provides reliable, in-order delivery of data in the form of byte-~~stea~~ stream between 2 end points.

↳ establishes a connection before transmitting data

↳ after transmission, connection is closed → a.k.a. teardown

UDP - User Datagram Protocol

↳ deals with messages that have strict boundaries

↳ may arrive out of order

↳ no connection establishment

Network Layer

- Routing is the responsibility of network layer of protocol stack

  ↳ given a ~~packet from~~ packet from transport layer on sending side, it finds a way to get the packet to the intended destination address

  ↳ at receiving end, network layer passes the packet up to transport layer which will collate the packets into the message which then goes to application layer

- IP address used to determine route.

Scanned with CamScanner

Link Layer

- has role of ferrying IP packets between nodes on Internet through which a packet has to be routed.
- Example: Ethernet, Token Ring
- Network layer hand IP packet to appropriate link layer
  ↓
  Link layer delivers fragments to next hop
  ↓
  passed back up to ~~transpor~~ network layer
  ↓
  repeat until we reach destination

Physical Layer

- responsible for physically moving bits from one node to next
- electrically, optically, etc.
- intimately tied to Link layer

Scanned with CamScanner

Layer in Depth

Application Layer

- web apps use HTTP for specifying interaction between web client and servers.
- e-mail uses SMTP (simple mail transfer protocol) for specifying interactions between mail clients and servers.

→ standards independent of ~~platform~~ platform that both either the client or server for such applications.

On the other hand, operating systems provide their own unique network services. → OS provides network communication libraries.

↳ Unix provides "sockets" ~~to~~ ~~library~~ library

Transport Layer

Lets Assume the transport layer API needs to have 2 functions

1. send (destination-address, data)
2. receive (source-address, data)

expected functionality of Transport Layer

1. Support arbitrary data size at application level
2. Support in-order delivery ~~system~~ of data

Scanned with CamScanner

3. Shield application from data loss
4. Shield application from bit errors in transmission

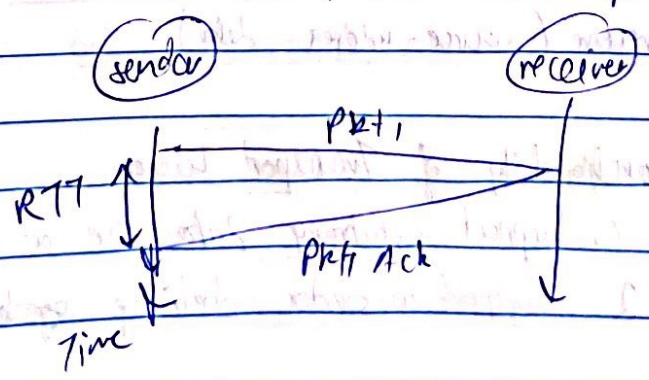
Suppose we want to transfer a message.

1. Transport layer breaks up message into packets. the peer transport layer at destination has to assemble the packets (scatter/gather)
2. Packets may arrive out of order at defination

Solution to 1 & 2, at source, assign a "sequence number" for each packet.
This will help reconstruct original message at destination
(deals with arbitrary size and out of order delivery)

3. source needs confirmation that the destination received the packets.
4. There could be packet loss and packet corruption en route.

One possibility:- Positive Acknowledgements
↳ acknowledge when packet is received.



RTT (Round Trip Time): time taken for a small message to be sent from sender to receiver and back to sender.

↳ used to estimate cumulative time delay

↳ used in selecting timeout values

RTT depends on

→ distance

→ queuing delays

→ message processing overheads.

There are a number of choices on how the transport layer deals with acknowledgements: many transport protocols

1. Stop and wait Protocols
2. Pipelined Protocols
3. Reliable Pipelined protocols

Scanned with CamScanner

Stop and wait protocol

1. Sender sends a packet and waits for positive acknowledgement (ACK).
2. When packet is received, recepient generates ACK and sends ACK containing sequence number of packet received
3. Sender waits for a period of time called timeout. If ACK not received within timeout, sender resends packet. Similarly, destination may resend ACK if same packet is received again (this shows ACK was lost en route)

Why we need sequence number if we are transmitting one packet at a time?
↳ so that we know which packet the ACK refers to, ie. eliminate ~~duplicates~~ duplicates

Simplify sequence number :- Since sequence number is mainly used to eliminate duplicates, we don't need an increasing counter as our sequence number, → a single bit is sufficient
~~↳ we keep flipping the bit~~

Alternating Bit Protocol :- using a single alternating bit as

sequence number.

Scanned with CamScanner