228.Summary ranges

12 June 2023 09:58 PM

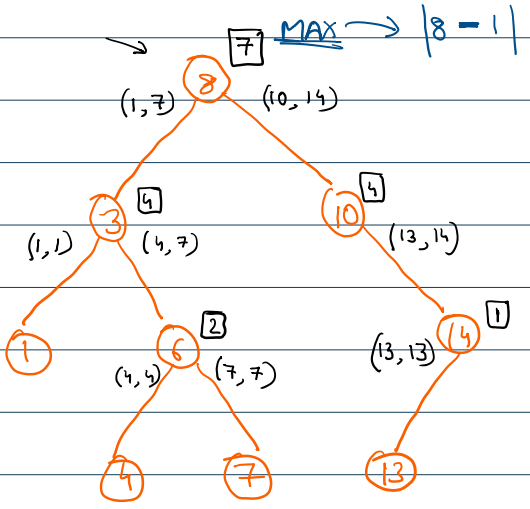
1146.Snapshot array

12 June 2023 09:59 PM

1026

14 January 2024 06:14 PM

Maximum diff. b/w node & ancestor :



∀ node,

we'll find (min, max) from both its LT & RT.

Then compute V :

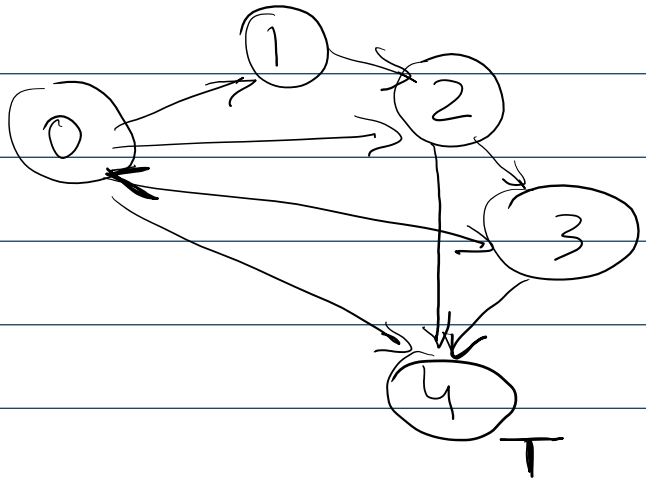
$$V = \left.\begin{array}{l} |\,min - curr\,|, \\ |\,max - curr\,| \end{array}\right\} \text{Max.}$$

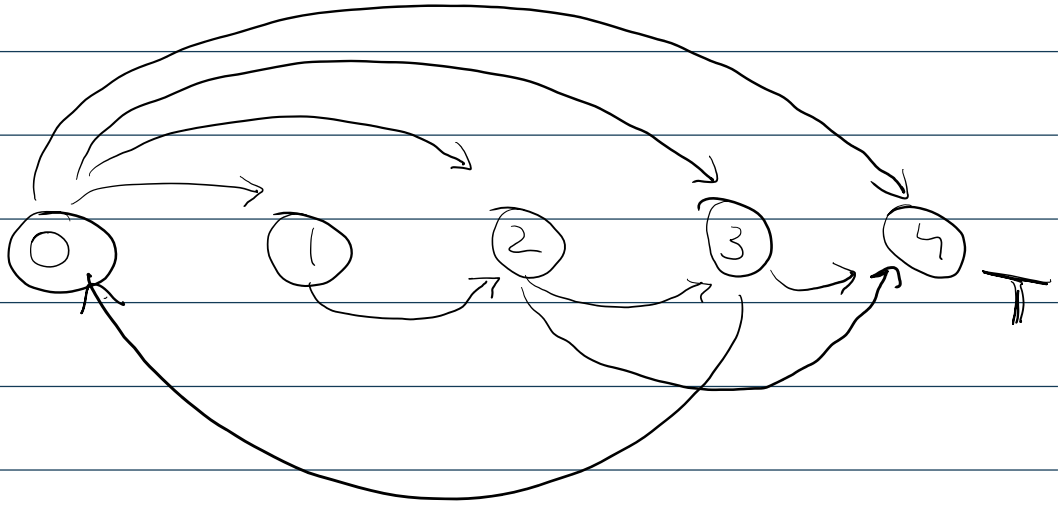
&,

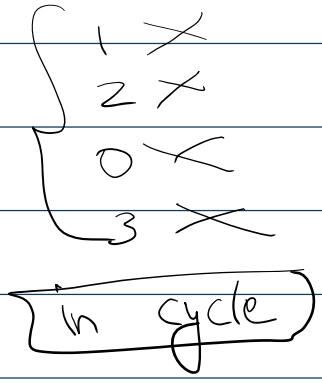
iterate till we find Max. (V)

802

4 September 2024 03:44 AM

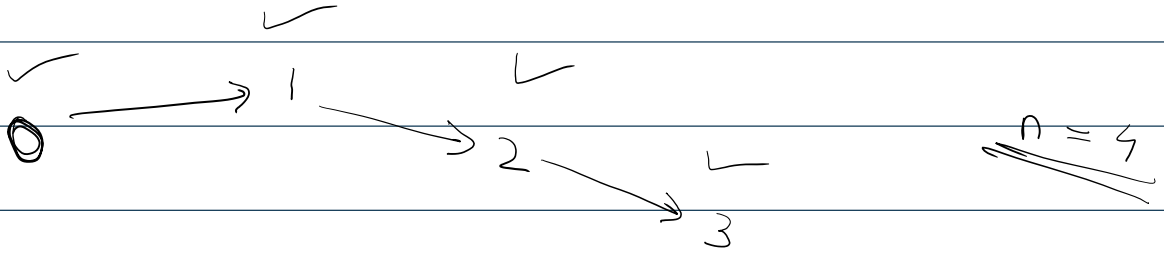


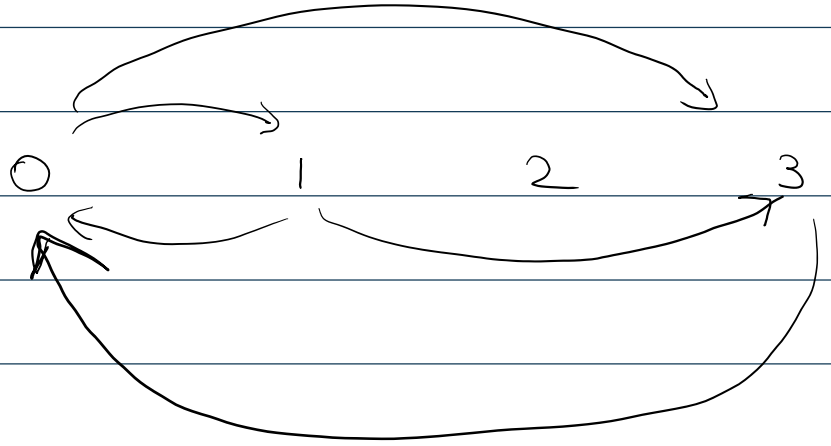


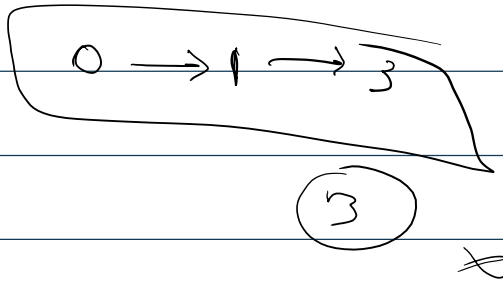


841

4 September 2024 04:57 AM



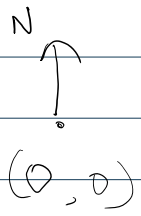




BFS ✓

DFS —

use ctx var

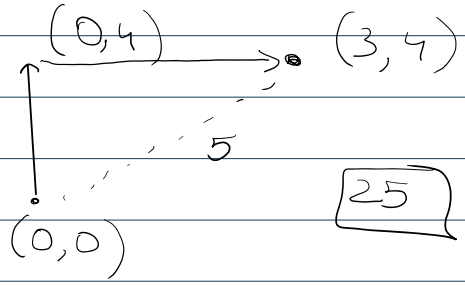


-2 : L , -1 : R

$[1,9] \in \mathbb{R}$

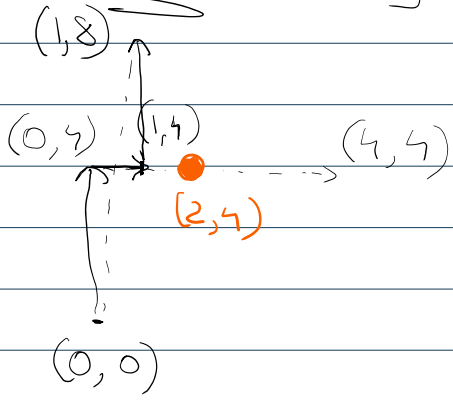
x^2+y^2

obs: []



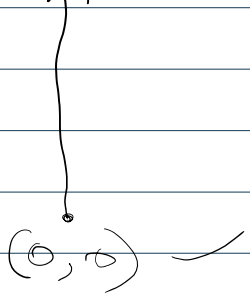
25

obs: [2,4]



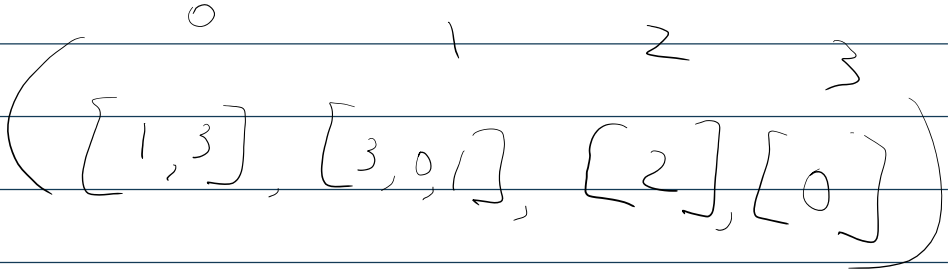
$\sqrt{1^2+8^2}^2 \rightarrow 65$

(0,0) → 36



841

4 September 2024 06:34 AM



n = 4

cnt = ~~0~~ 1

2

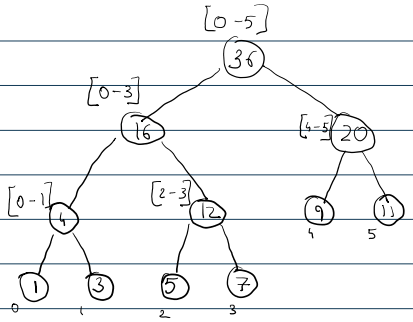
vis

T	T	F	F
0	1	2	3

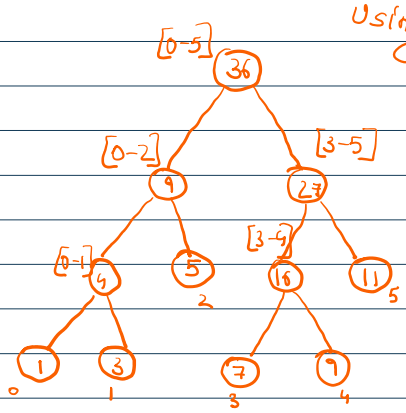
Segment tree

0	1	2	3	4	5
1	3	5	7	9	11
1	4	9	16	25	36

n = 6



OR



Using Mid-computation

$L_T \Rightarrow [L, mid]$

$R_T \Rightarrow [mid+1, R]$

→ leaf nodes are arr. elements

$L_c = 2i + 1$
$R_c = 2i + 2$
} for Root@ i index

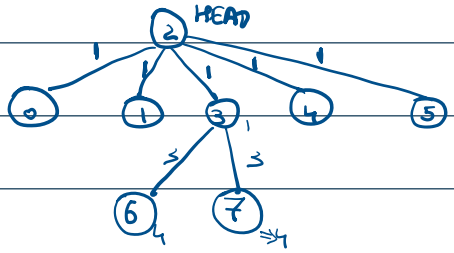
nodes
$2n - 1$

seg. arr :

0	1	2	3	4	5	6	7	8	9	10	11
36	9	27	4	5	16	11	1	3	-	7	9

1376

6 September 2024 11:29 PM



{4}

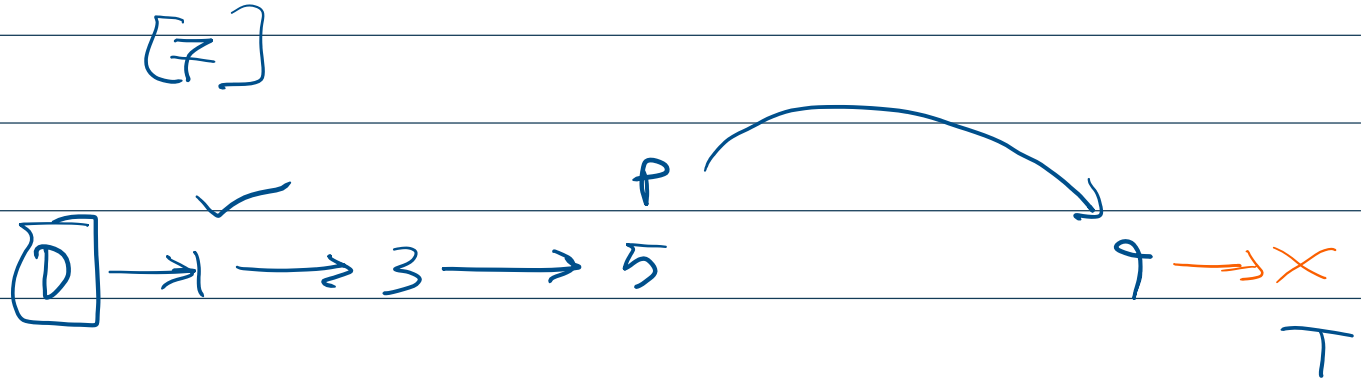
u	v	wt
mgr[i]	i	informtime[u]
2	0	1
2	1	1
-1	2	

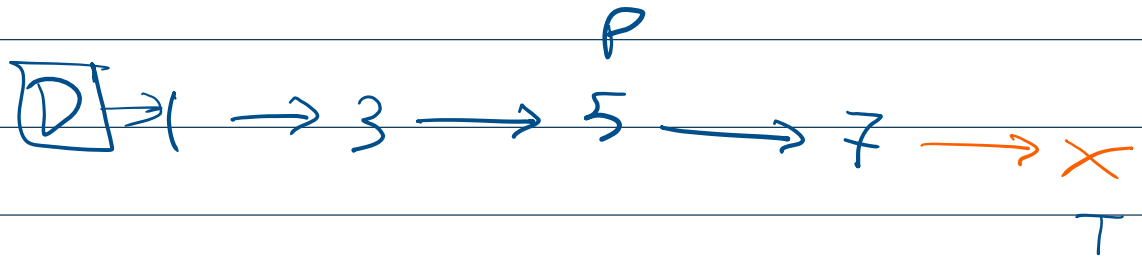
2

informtime[curr_mgr]

3217

7 September 2024 12:27 AM



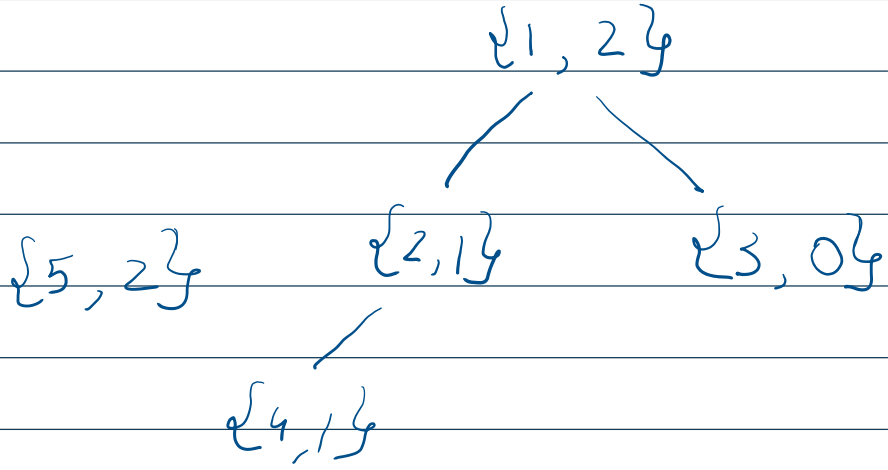


$k = 4$, $n = 3$

[1 2 3 4 5]

diff. w.r.t n' 2 1 0 1 2

Max-heap (k)



arr. → 1, 2, 3, 4, 5 , x = 3

K = 4

MIN HEAP

	1	2	3	4	5
diff.	2	1	0	1	2

{2, 5}

{2, 1}

K = 4

res : 3, 2, 4, 1

	0	1	2	3	4
arr. →	1	2	3	4	5
diff.	2	1	0	1	2

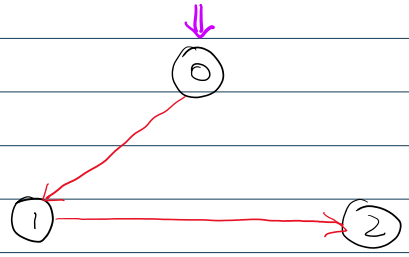
L, R, R

K = 4

x = 3

1129

10 September 2024 02:05 AM

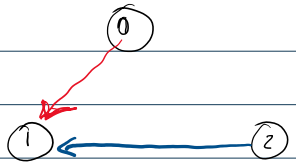


directed

n = 3

answer : [0, 1, -1]

0 1 2



n = 3

answer : [0, 1, -1]

nodes 0 1 2

```
n == 0 → ret m
else
    ret (n, m%n)
```

gcd(18, 6)

(6, 0) 6

(10, 3)

(3, 1)

(1, 0) 1

(6, 10)

(10, 6)

(6, 4)

(4, 2)

(2, 0) 2

Moore's law

Transistors / chip will double roughly 1-2 yr.

Dennard Scaling → (power dissipated)

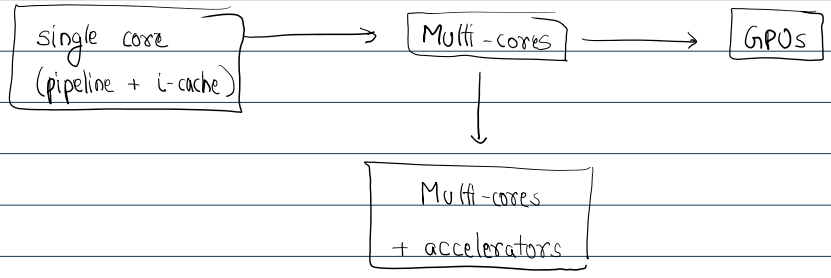
→ Power density remains const.

→ Performance per watt ↑ exponentially.

SPEC → std. Performance Eval.ⁿ Cooperative

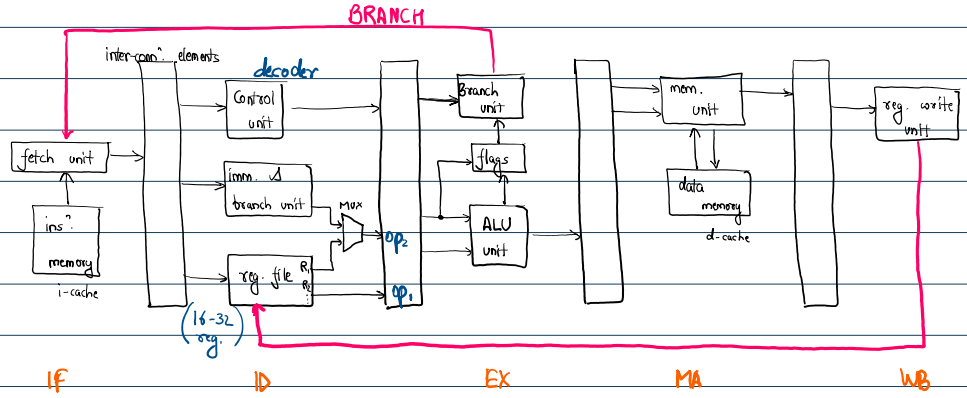
↑ in Performance:

gen^. IN-ORDER pipeline < n-issue IN-ORDER pipeline < n-issue OOO pipeline



Out-of-order Pipeline:

Processor with 5-stages:



pipeline eliminates idleness in processor.

{ except for time to fill & drain it. }

	ins^n. load	exec^n.	commit
In-order pipeline :	in-order	in-order	in-order
Out-of-order pipeline :	ooo	ooo	in-order

→ Pipelines use -ve edge triggered latches / buffer blw ✝ 2 stages, to store the imm^t. results ↓ synchronize the clock.

* Hazards :
  i) Structural hazards → same resource / h/w access
  ii) Data hazards → data dependency (RAW, WAR, WAW)
  iii) Control hazards → Branch statements

Data interlocks : Stalls
identifies dependencies on ins^n. ↓ inserts NOP ins^n. / bubbles whenever it finds suitable.
→ stalls the pipeline, Performance ↓

→ Optimization for RAW dependency :
Forwarding : from MA to EX stage, resulting in NO stall.
{WB to reg. file, will happen in background}
forwarding unit uses MUX to determine, which i/p needs to be taken for ALU exec^n.

Types of forwarding paths :

WB → MA	RAW	load / store dependency
WB → EX	RAW	load / store dependency
WB → ID	RAW	load / store dependency
MA → EX		ALU dependency

* forwarding cannot help in case of LOAD-use hazard (RAW dependency)
⇓
requires 1 stall cycle.

* dis-adv. of in-order pipelines :
→ load-use hazard : causes 1 stall each
→ Usually we have multicycle MA stage

→ load-use hazard : causes 1 stall each
→ usually we have multicycle MA stage
→ Convey effect (Head of line) HOL blocking
→ Taken branches have 2 stalls each
→ IPC < 1

* Performance eq^n.

$$\frac{\#\text{Programs}}{\#\text{sec.}} = \frac{IPC \times freq.}{\#ins^n.}, \text{ for 1 program}$$

or

$$\text{Performance} \propto \frac{IPC * freq.}{\#ins^n.} \qquad IPC = \frac{1}{CPI}$$

→ #ins^n. depends on compiler.

→ freq. depends on transistor technology & arch. (#Pipeline stages)

(clock Period) $t_{clk} = \frac{t}{k} + \Delta$, where, k-stage pipeline, with Δ as buffer delay.

$$f_{max} = \frac{1}{\frac{t}{k} + \Delta} \Rightarrow \boxed{f_{max.} \propto k}, \text{ give } \frac{t}{k} \gg \Delta$$

→ IPC again depends on arch. & compiler. { lower stalls, IPC ↑ }
i.e, forwarding,
rearranging code, keeping in mind data-dependency.

* Methods to ↑ IPC :
→ forwarding
→ having more not taken branches
→ faster ins^n. & data memories.

* ↑ freq. is LIMITED by
↓

i) Power, $P \propto f^3$

ii) Change in temp, $\Delta T \propto P$

iii) ↑ in pipeline stages ≈ More hazards, more forwarding paths, design complexity

iv) Effect of latch delay

* $CPI_{actual} = CPI_{ideal} + (stall\_rate * stall\_penality)$

* **Super-scalar Processor:** helps ↑ IPC

→ can execute multiple ins^n per cycle.
{ by issuing more ins^n per cycle }

i.e,

Intel Pentium : had 2 in-order pipelines (U & V)
ins^n : i i_{+1}

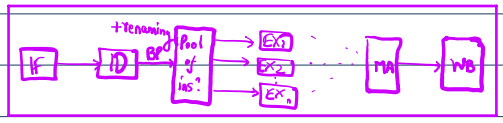
dis-adv.:

→ dependency b/w ins^n.

→ $O(n^2)$ forwarding paths for n-issue processor.

→ complicated logic for detecting hazards, dependencies, & forwarding.

* **Out-of-order Pipelines:**



→ don't follow Program order for exec^n
{ it is as per data dependence order }

extracts more //ism. from code segment
{ ins^n lvl parallelism }

Steps: i) Creates a pool of ins^n { fetches: Program order }
ii) find mutually independent ins^n, having operands ready
iii) Executes them ooo

→ larger the pool ⇒ more ILP can be exploited,
ins^n window given ins^n are on correct path.
(typical size: 64-128)

Branch Prediction:

Consider,

Consider,

#ins". → n

#branches → n/5

P(E) predicting any given branch INCORRECTLY → p

Then,

P(E) predicting ALL branches CORRECTLY → $(1-p)^{n/5}$

⇒ P(E) of making atleast a single mistake in a pool of 'n' ins". → $1 - (1-p)^{n/5}$

→ We require Branch Prediction to be highly accurate.
i.e, it does have a very high substantial penality.
⇓
even with 97.5 % acc. in BP ⇒ we have 40% chances of having incorrect ins". in window.

→ Acc. of BP limits the size of ins". window.

* Types of data - dependencies:

RAW → true dependency {causes problem in-order pipeline}

(Op-dependency) WAW
(Anti-dependency) WAR
} false dependency, caused due to finite no. of reg.
issue {in OOO pipeline} Sol": Register renaming

(done by HW, internally)
Register renaming:
In the assembly eq" code,
we map V architectural register to a physical reg.,
in-order to eliminate false dependencies.

↘ Create new avatar of reg. whenever we have a W op".
i.e,

W R R | W R | W R R . . . .
Avatar 1 | Avatar 2 | . . . .

(write once, read many times)

→ Only RAW dependencies exists in the code, (if any)
after renaming

→ Only RAW dependencies exists in the code, (if any) after renaming.

⇓

higher ILP

Precise exceptions :

→ we need precise behaviour of ooo pipeline, to handle stop & resume effectively (CSw.), in case of an exception/interrupt

Consider, intr. after k^{th} ins.,

Then,

ins.: till k should has executed completely & written their results to MEM./Reg. file.

&, $(k+1)^{th}$ & later, should not appear to have started their EX atall.

* IF :

To explore ILP (in ooo), we need to ↑ fetch B.W.

→ But it incurs other overhead, → we only know PC value

→ no time to look at ins.n (prior decode in IF stage)

such as for B.P → it becomes difficult to predict correct set of ins.n, in a multi-way branch.

{nested branch structure}

for an optimal compiler, if we have a fetch BW = 4, we'll expect it to place Branch statements as far as possible.

Branch Predictor :

Concerns : i) Predict if an ins.n is BR or not.

ii) if BR, predict its dir (forward / backward)

iii) predict its target (EA, if taken)

PC → BP → Prediction

① given an ins^n. with same PC value,
→ it remains as BR or non-BR, throughout program runtime

⇓

$\forall$ PC, we remember if its a BR & its type:
- → Unconditional BR — TAKEN — B
- → Conditional BR — (need to Predict) — BEQ, BNE
  (depending on result of a prev. CMP ins^n., some flags are SET)
- → fn. call — TAKEN — call
- → return — TAKEN — iret
- → Non-BR

Using a str. in H/w: **Ins^n. Status table (IST)**

⇓

uses LSB 10-bits of PC, to mark the ins^n. as to which category it belongs.

provides better RANDOMNESS in case of Spatial & temporal locality

→ dis-adv.:

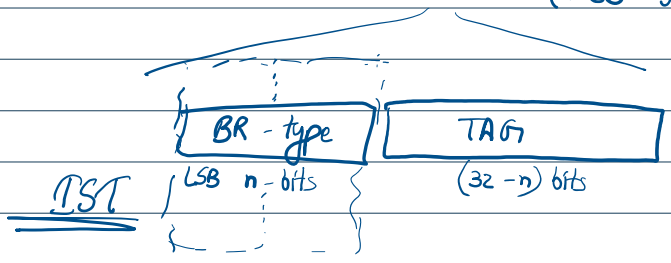
Destructive interference:
many addresses maps to same entry & keep updating the entries in IST.

i.e.
- 1 PC being BR as lsb 10-bits,
- another being non-BR with same IST entry

} **BRANCH ALIASING** (due to small-sized IST)

⇓

Sol^n.: To distinguish, AUGMENT each entry of IST with the TAG value (MSB 32−n bits)



→ If for a PC, we don't have entry in IST,
→ we'll have to predict if its a BR or not.

→ IST works because of temporal locality exhibited by codes (i.e, LOOPs)

② Prediction :

→ Branches in loops have roughly similar behaviour most of the time. Predominantly NOT TAKEN.

I] Simple BIMODAL Predictor : (1-bit Predictor)

∀ PC in IST, it remembers the last outcome of conditional BR.

↳ ∴, uses it for current Prediction.

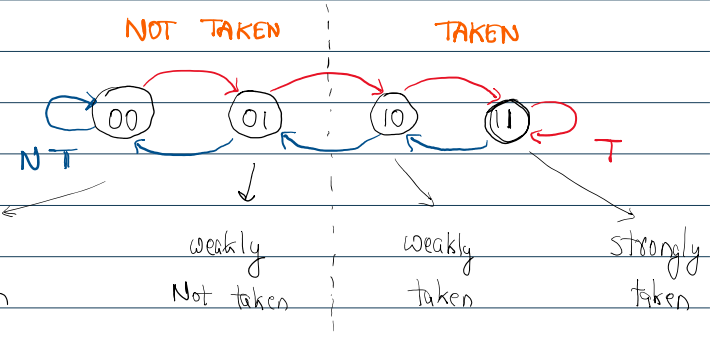
Same augmented IST table, along with 1-bit for BR as TAKEN / NOT TAKEN for last recorded outcome of BR.

II] Saturating counter (2-bits predictor)

→ Instead of having 1-bit history, uses 2-bits for hysteresis.

Algo. for updation:
→ if a BR is TAKEN, ↑ saturating ctr.
else, ↓

for Prediction: 00, 01 → TAKEN
10, 11 → NOT TAKEN



→ works well when BR is biased towards 1 dir. {either mostly TAKEN or NT}

* Global History Register (GHR) :

Consider, a shift-register, that records history of last n- branches encountered by processor {regardless of PC value}

⇓

1 bit for each BR (0 → NT, 1 → Taken)

Ily,

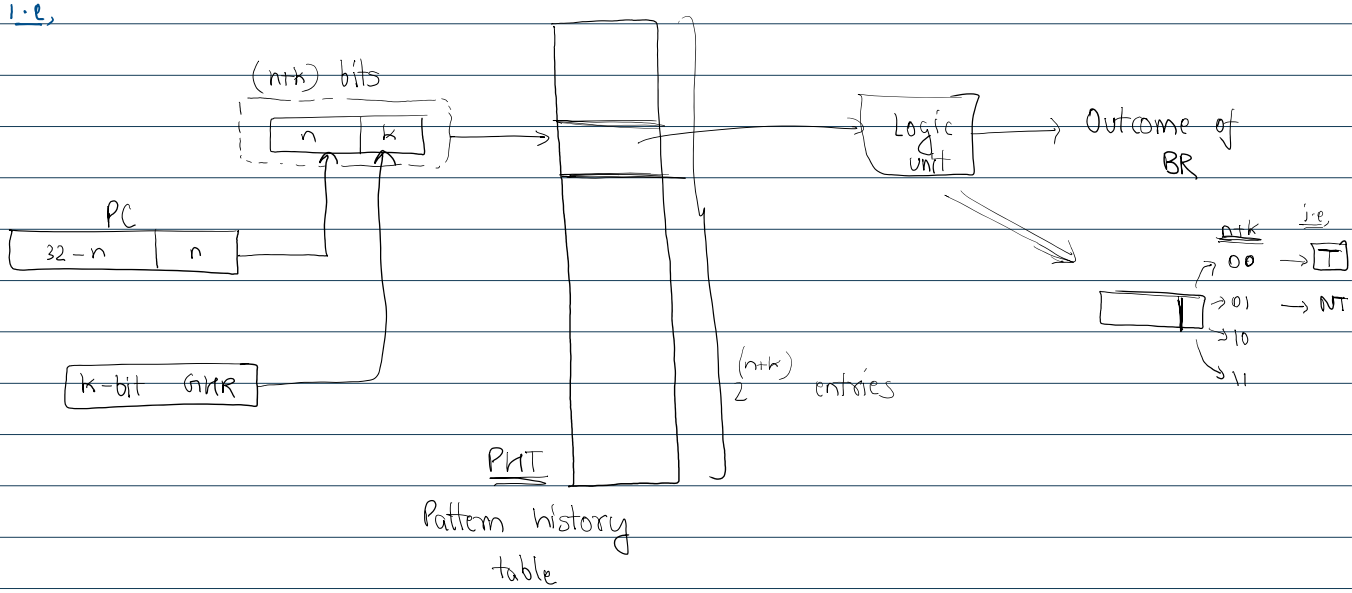
we have 2-bit shift-register (GHR)

↳ uses the history of last 2 BR ins".

III] GAp Predictor :

→ use k-bit GHR along with Pattern History Table (PHT) to decide the outcome of a BR ins".

i.e,



GHR : captures the behaviour of related branches.
i.e, {within a fn. scope}

IV] PAp Predictor :

→ uses multiple GHRs ∪ n-bits of PC, to decide outcome of the BR. → Most Accurate.

V] Tournament Predictor :

- very biased BR → bimodal predictor with saturating ctr., works well.
- Alternating pattern → PAp predictor, works well
- for a general Piece of code, we'll use 2 Predictors, ∪ choose 1 of them {based on behaviours}

→ To identify the target of BR ins^n.,

we augment the IST table ⟹ Branch Target Buffer (BTB)

i.e,

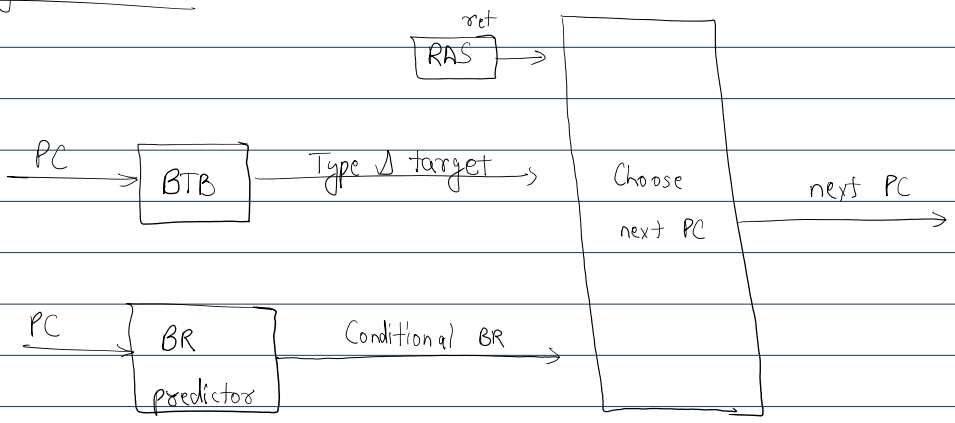
it now hold EA aswell.

Ventry → BR type | Tag_bits | Target or EA

→ for CALL / RETURN BR ins^n.,

we use Return Addr. Stack (RAS)

Summary of BR Pred.:



* Decode stage: (last part of IF unit)

Process of decoding:

→ Expand imm. val. to 32/64 bit values.

→ Extract all fields (IDs of regs.)

→ Compute BR EA (if TAKEN) → we usually have PC-relative Branches

→ Add all implicit sources (i.e, ret. ins^n.) → return addr. [RA] reg.

→ Create Ins^n. pkt. (includes Control-bits)

CISC ins^n

→ do not have fixed len.

i.e, in x86: len. varies from 1 to 15 Bytes.

⇓

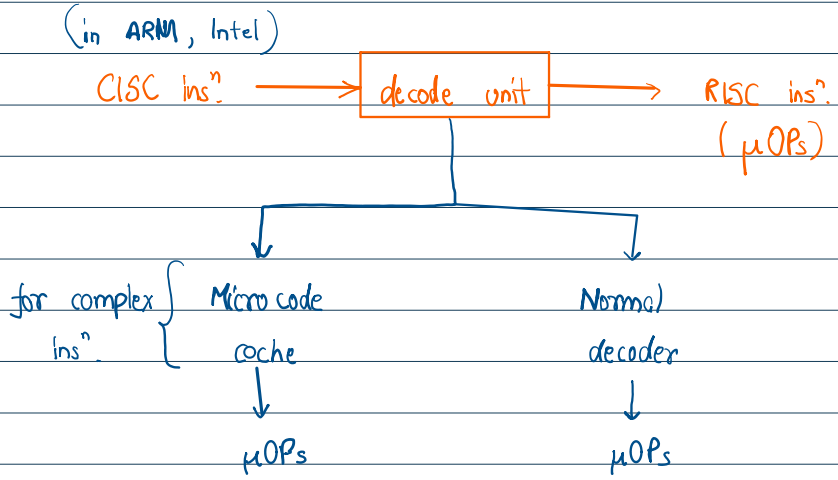
hard to fetch multiple ins^n. at once, (∵ linear scanning reqd.) SEQUENTIAL

↓

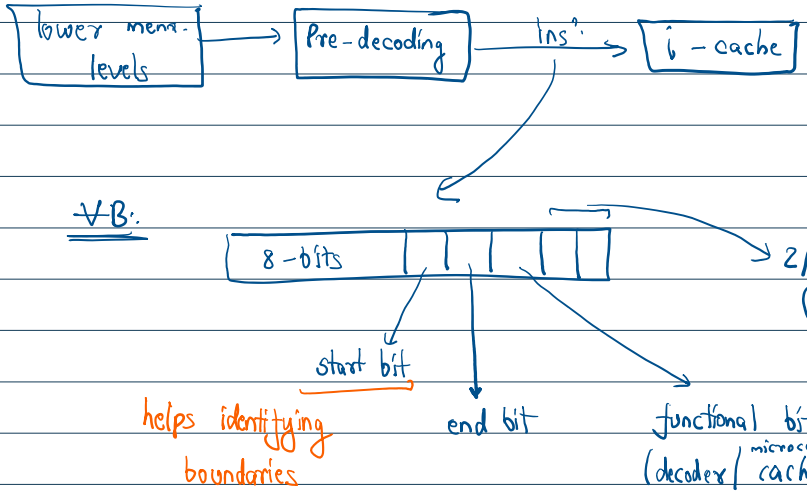
hard for OOO pipeline to process them.

↓

hard for OOO pipeline to process them.
{CISC processors internally convert it to RISC}



→ To identify START/END of ∀ CISC ins^n.,
∀ Byte is augmented with some additional info.
(done by L2 cache typically)



* ins^n. compression in i-cache:
→ using shorter width ins^n. (with smaller ISA)
→ replacing frequently executed ins^n. with codewords (using dictionary)
↓
stores decoded ins^n.

12 September 2024 09:06 AM

Issue, Execute, Commit Stages in OOO pipeline :

Reg. renaming with Phy. reg. file : (done by H/w)

		#architectural reg (visible to s/w)
(Intel, AMD)	x86	8
	x86-64	16
	ARM	16
	MIPS	32

→ used by compiler in ASSEMBLY CODE gen?

MIPS : MicroProcessor w/o Interlocked Pipeline Stages.
{ INORDER pipeline }

more the #Phy. reg. ⟶ ↑ ILP can be achieved.
(store interim values)

* Renaming uses 3 H/w structures :

i) Reg. Alias Table (RAT)
translates arch. reg. ID to phy. reg. ID.

ii) Dependency Check logic (DCL)
take cases of dependencies b/w multiple ins^n. renamed in same cycle,
{ using MUX }

iii) Free list/queue (circular Q)
Maintain a list of unmapped Phy registers.

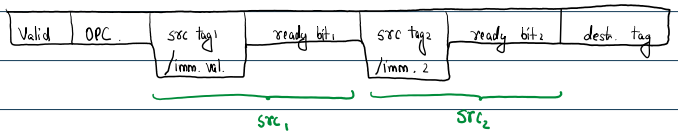
⟶ RAT also store avl. bit,
{ whether Phy. reg. value ready or not }
↳ if not, ins^n. has to WAIT in ins^n. window.

Rename table entry : Phy. reg. ID | avlbl. bit

Dispatch :
(or issue) till rename stage, ins^n. proceeds INORDER
↳ Sends the ins^n. to ins^n. window

RENAME → issue → IW
(Pool of ins^n.)

→ structure of a ins^n in Ins^n window :



tag → phy. reg. ID

When both the operands are available, the ins^n is ready to EXECUTED.

either from Reg. file / Bypass n/w.

waits in IW for its operands to be BROADCASTED onto TAG bus.

IW → EX
issue

Once an ins^n has all its operands READY, it asserts its attached REQUEST line

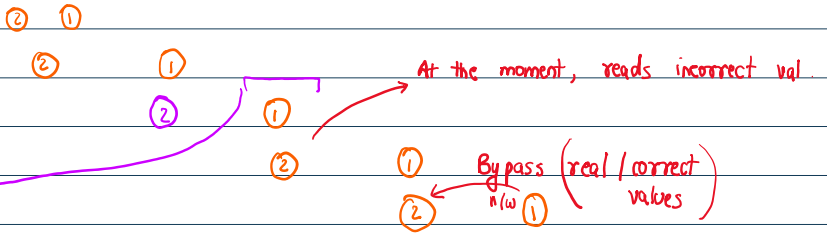
→ The Select unit chooses among these ready ins^n (GRANTs it)

→ Speculative Broadcasting :

Consider 2 ins^n with RAW dependency. (i^{th}, $i+1^{th}$)

ADD r_1, r_3, 1
SUB r_5, r_1, r_4
RAW

RENAME | DISPATCH | WAKEUP /SELECT | RF read | EXECUTE | RF write



Early Broadcast
{ to avoid STALLs }

→ Allows back-to-back EX^n of dependent ins^n

→ In-case of a 2 cycle LOAD-USE hazard :

where LOAD requires 2 cycles in EX phase, we BROADCAST TAG 1 cycle later.

{ i.e, STALL of 1 cycle persists in OOO }

i.e, for diff ins^n,
we BROADCAST TAG at diff. pt. of time.

→ During speculative BROADCAST, (Rename table)
we also send a signal to RAT table,
to update the avlbl. bit of processed reg. [Destn. reg]
{ in-order to help ins^n. that arrive after Broadcast, to not wait indefinately }

* Problem:
→ ins^n being written to the IW will miss the broadcast.
→ These ins^n would also read avlbl. bit as 0, from RAT,
↓
leading to indefinate WAITING

Sol^n: i) Split 1st half of clk cycle for DISPATCH.
& use 2nd half for BROADCAST. } NOT feasible

ii) Double BROADCAST
(↔ next cycle updation)
→ log the BROADCASTED tags in a temporary str.
→ record all ins^n that are being dispatched in a seperate str.,
called DISPATCH buffer.

In the next cycle,
→ match the tag entries in the dispatch buffer,
→ if matched, update corresponding IW entry.
→ Clear entries of last cycle from buffer.

* Reg. Read & Execute:

LSQ : Load - Store Queue
↓
Enforces mem. dependencies.

Consider,

```
ld  r1, 4[r3]
st  r2, 10[r5]   ---> if same MA.
                       ⇓
                     Problem
```

They're then sent to load - store unit:

→ loads can execute immediately.
→ stores update the processor's state (for Precise Exceptions)
{ MUST be performed in Program Order }
↳ can wait.

STORE Queue ≈ data-cache str.

↳ LOADs cannot be sent to mem. sys., until all the STOREs before it are RESOLVED, & do not write to the same addr.

* LSQ:

Op. type	Search dir.	Condition : Action
LOAD	search ← all stores before it	→ found store to same addr. : terminate & forward values → found store with unresolved addr. : terminate → Both not met : goto d-cache.
STORE	search → all LD/ST after it.	→ found store with same addr. : terminate → found store with unresolved addr. : terminate → LOAD from same addr : forward value / addr.

* Ins? Commit:

Reorder Buffer: (inorder & circular str.)

→ Contains 1 entry ∀ inflight ins^n. fetched. (in program-order)
→ After decoding the ins?, we enter it to ROB.
→ if no free entry → ooo pipeline STALLS.

Entry is cleared when an ins^n. COMMITS / Retires. → also, add reclaimed phy. reg. back to free list. (if any)
(in-order)

∀ cycle, we inspect for top [W] entries, if READY to be commit.
commit width
in ROB

↳ Stop if any entry found to be NOT READY.

{ in general, fetch Width = Commit Width }

* Unexpected events :
  - → interrupts
  - → exceptions
  - → Branch mispredictions

⇓

Some ins^n. in pipeline are on wrong path, i.e, they shouldn't be committed.

→ Mark ins^n. that has had an exception / suffered Misprediction in ROB.
In case of interrupt, mark topmost entry in ROB.

→ wait till marked ins^n. retires

→ initiate [recovery seq^n.]

→ flush ROB, LSQ, IW
& all pipeline reg.
+
Restore / Rollback arch. reg. state, PC

* VLIW Processor :
(Very long ins^n. word)

→ exploits ILP.

→ It rely on compile time analysis,
to identify & bundle together ins^n. that can be executed concurrently.

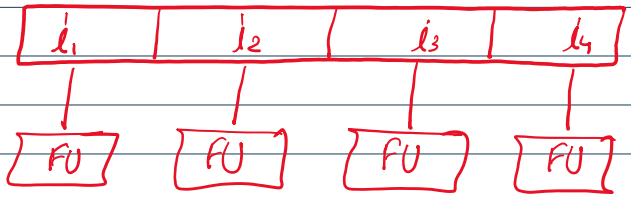
→ compiler is responsible for static scheduling of ins^n. in VLIW processors.

→ compiler finds out which operations can be executed in // in the program & groups together these op^n into single ins^n. known as ISSUE packet or Bundle, which is a very large ins^n. word.

→ compiler ensures that an op^n. is NOT issued before its operands are ready.

→ They deploy multiple independent FU.
i.e,

Bundle :



(a VLIW ins^n.)

VLIW Processors	SuperScalar Processors
→ Parallelism is performed STATICALLY at compile time, by Compiler.	→ dynamically at RUNTIME, by H/W
→ Data dependency is chked by compiler.	→ data dependency is checked & resolved by H/W
→ less //ism.	→ more //ism.
→ complex compiler.	→ complex H/W