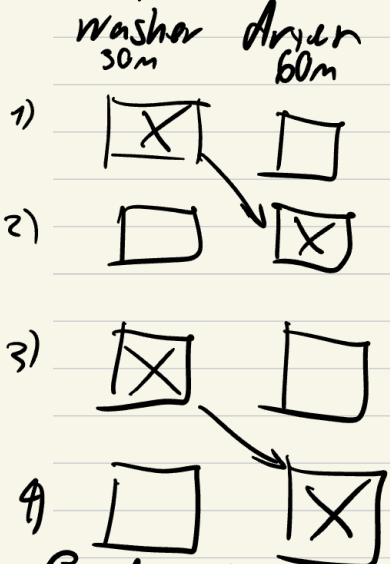
AI Hardware

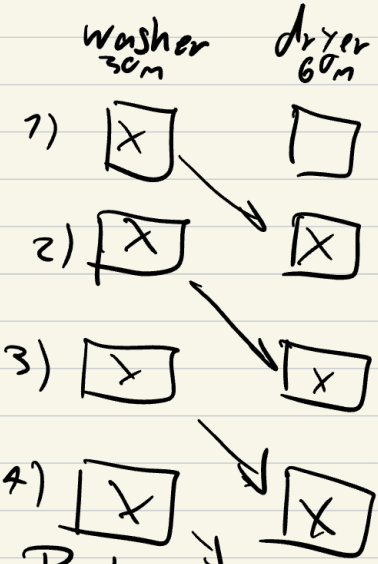
Seeraj Chetput

Review Computer Structures 6.004 MIT OCW

7 - Performance Measures
Latency & Throughput

Laundry ex:

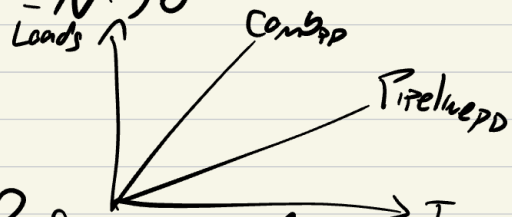




Combinational
$T_{PD} = N(washer_{PD} + dryer_{PD})$
$= N \cdot 90$

Pipeline
$T_{PD} = N \max(washer_{PD}, dryer_{PD})$
$= N \cdot 60$

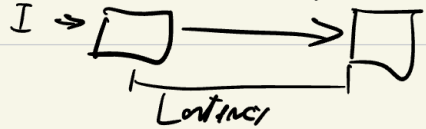
→ don't account for set up time



Performance Measures:

Latency:

Delay from when an input is established until associated output w/ that input becomes valid

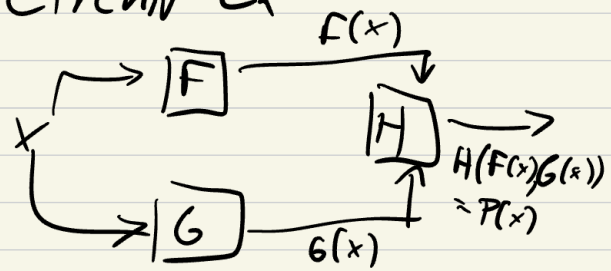


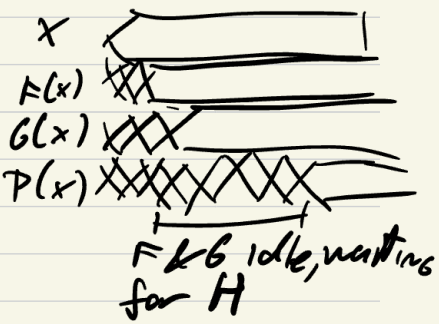
Comb. latency = 90 min
Pipeline latency = 120 min

Throughput:
Rate at which I/O's are processed
Comb throughput = 1/90 min
Pipeline throughput = 1/60 min

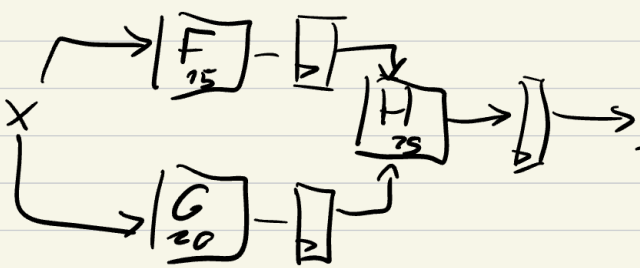
Comb does 1 job in less time
Pipeline produces > output / time

Circuit ex:

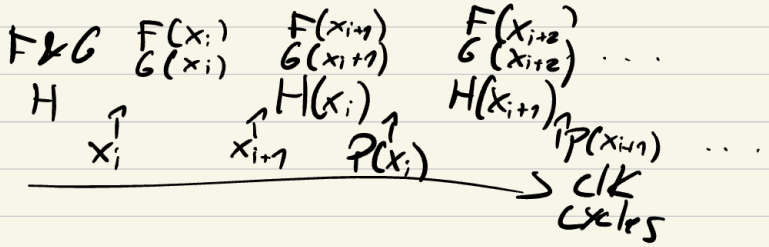




Bad throughput!



	Lat	Throughput
UnPIP	45	1/45
2-stage	50	1/25
	worse	better!



Pipeline Conventions

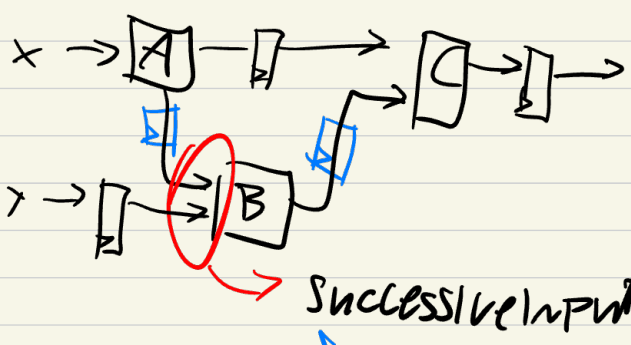
Def:
Well-formed K-stage Pipeline ("K-Pipeline")
acyclic circuit w/ exactly K regs $\forall$ path I->O
Comb. circuit = 0-stage Pipeline

Composition Convention
($\forall$ Pipeline stage $\Rightarrow \forall$ K-stage Pipeline) has a reg on I's O

Always:
CLK common to all regs must have T sufficient to cover $t_{comb} + t_{PD} + t_{setup}$ ($= T_{CLK}$)

Latency of K-Pipeline $= K \cdot T_{CLK}$
Throughput of K-Pipeline $= f_{CLK}$

Ill-formed Pipelines



Pipelining methodology

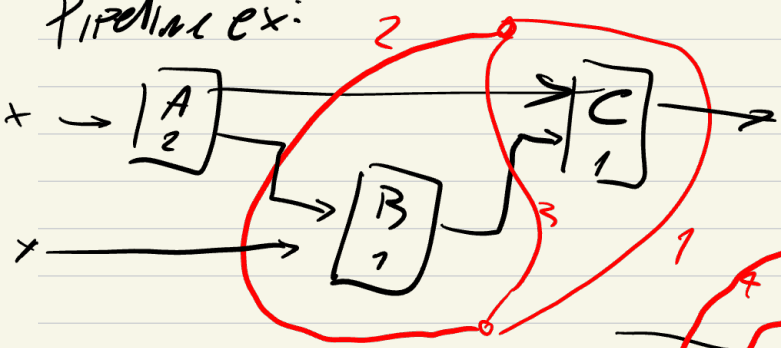
1) Draw line that crosses $\forall$O & mark endpoints as terminal points
2) Continue drawing lines b/w terminal points across various comb. ckts ensuring $\forall$ connec. crosses $\forall$ line in same direction

Lines demarcate pipeline stages
Adding pipeline reg $\forall$ point where separating line crosses a connec.

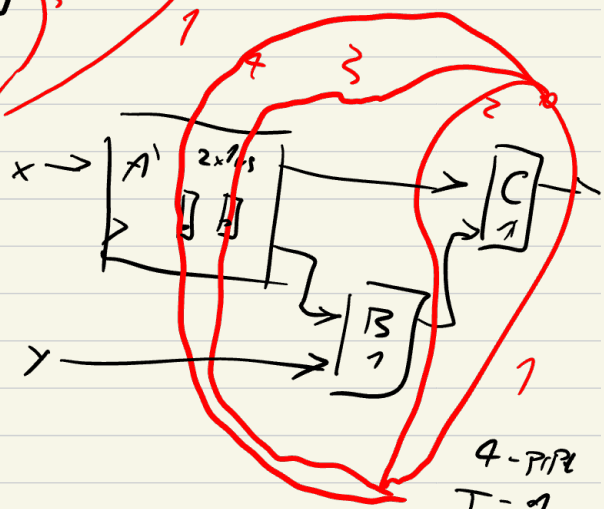
will always Generate a valid Pipeline

Strategy: focus attention on Placing regs around bottlenecks

Pipeline ex:



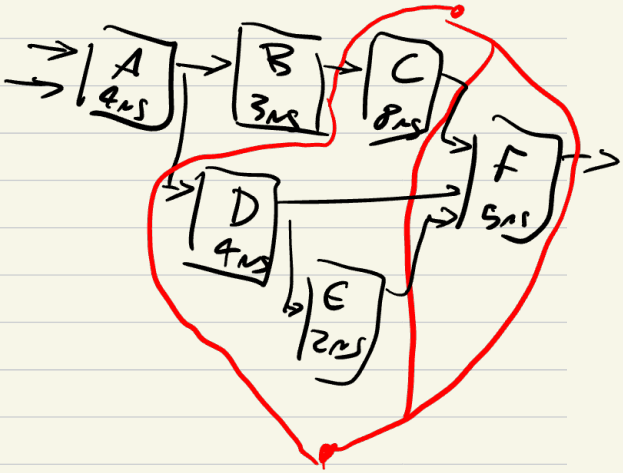
	L	T
0	4	1/4
1	4	1/4
2	4	1/2
3	6	1/2



Pipelined Comp:
Pipelined sys's can be hyerarchical
Replacing a slow comp comp. w/ K-Pipe version may ↓ T_{CLK}
Must account for new Pipeline stages in new plan

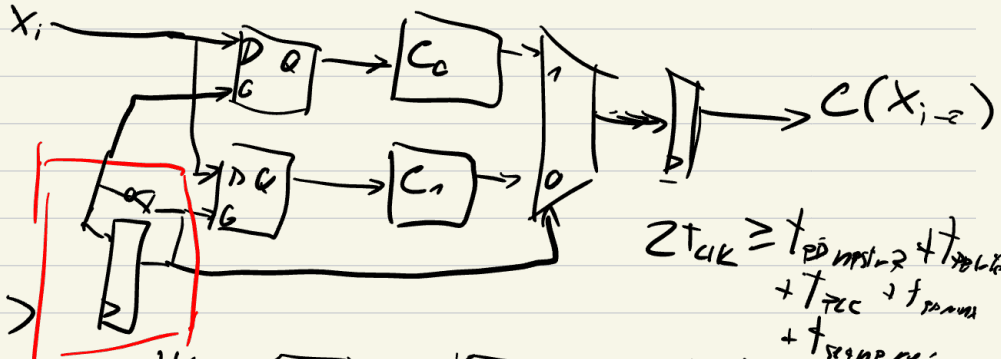
Ex: C slowest comp.
limits $T_{CLK} = 8ns$
$\Rightarrow T = 1/8ns$

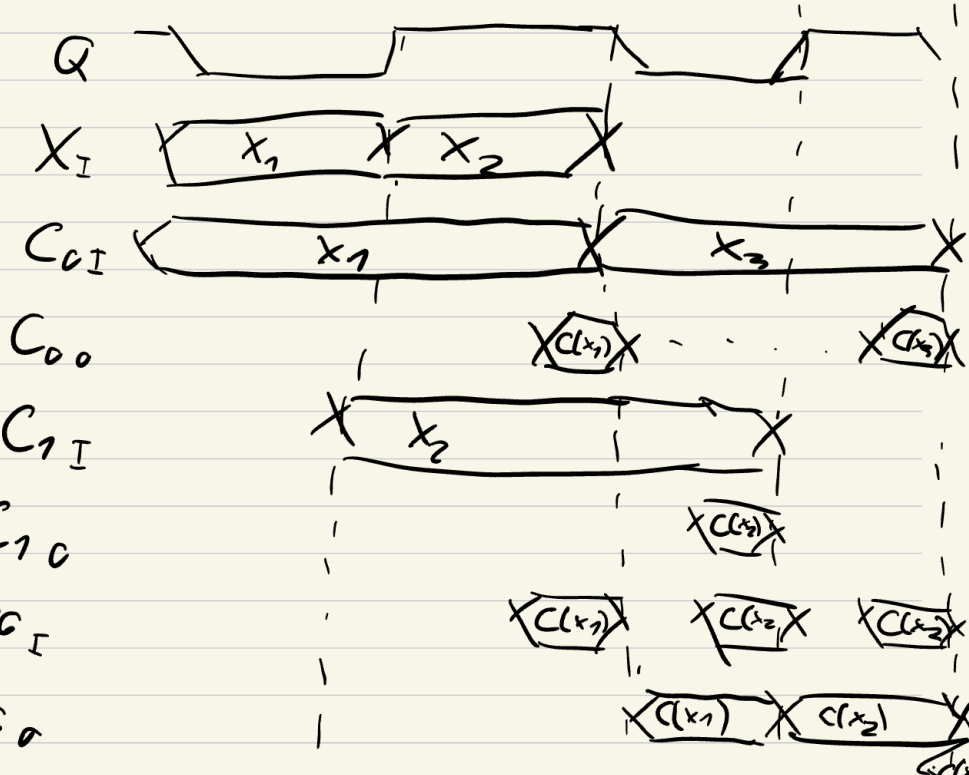
Improve by:
a) Pipelined C
b) Interleaving $\Rightarrow$?
2 copies of C



D-latch

Circuit interleaving



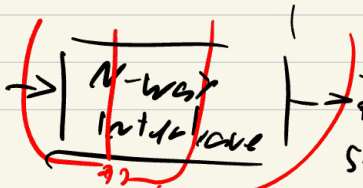


2-clock maximizing

"In by t_i, out by t_{i+2}

$T = 1/clk$

$L = 2 clks$



Combine Techniques

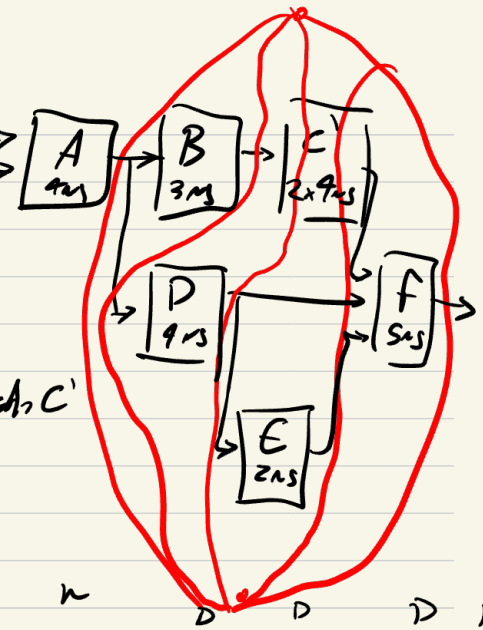
Interleaving + Pipelining

C' interleaves 2×C

Effective T_{clk}=4ns, L=8ns

C' behaves as 2-stage pipeline

2 pipelining contours must pass through C'



Bottleneck: C → F

Parallelism

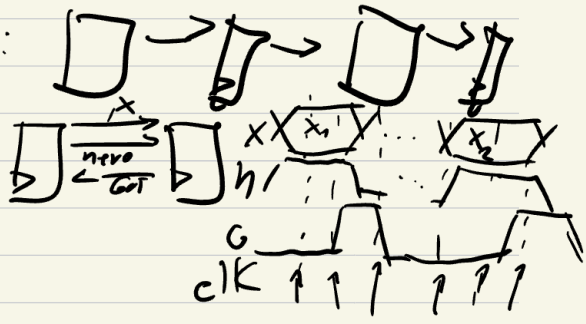
Interleaving

+ Pipelining

+ Parallelism

Control Structure Alternatives

Sync, globally-timed:

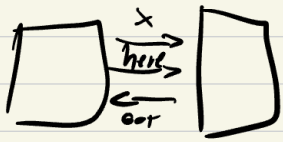


Sync, locally-timed:

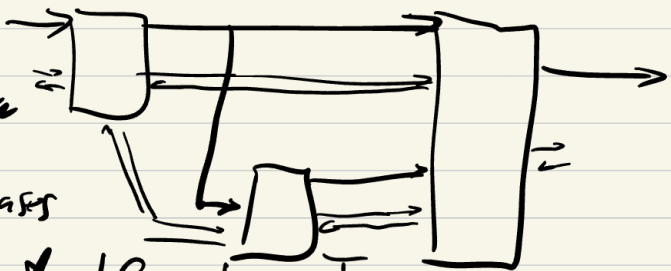
local FSMs control flow

ct data w/ handshake

Async, locally-timed:
using transition signalling



Self-timed ex:
each comp. spec. its own time constraints
local adapt to spec. cases
async or sync



Central Structure Taxonomy

easy to design
fixed-sized interval
can be wasteful

	Sync	Async
Glob. timed	centralized clk FSM Generates $\forall$ ctrl signals	Central ctrl unit tailors ctrl time slice to ctrl tasks
locally timed	Start/Finish signals Gen. by $\forall$ major subsyst. sync w/ global clk	each subsyst. take async Start, Gen async finish (perhaps w/ local clk)

large sys's ⇒ very complx time Gen. Ne!

best way to build large sys's w/ indep. timed comps

let's design work tools extensively is worth it in special cases

Summary
Latency: t for $I \to O$
Throughput: #O's/t
comb: $L = t_{PD}$, $T = 1/L$

K-Pipelines
$\forall$ reg on output
K regs $\forall$ path $I \to O$
I avail. after clk;
O avail after K clk
$t_{CLK} = t_{PD,reg} + t_{PD\,bw} + t_{setup}$

$T = 1/t_{CLK}$

↑T ⇒ split slow. PIP. stages repl. interlocking if no further split

$L = K t_{CLK} = K/T$
$L_{PIP} \gg L_{comb}$

Caches & Memory Hierarchy

Single-cycle CPU
Data & Instr. Memory the slowest piece of HW in design
↑ # MOSFETs (also ↑ space)
↑ Propagation delay

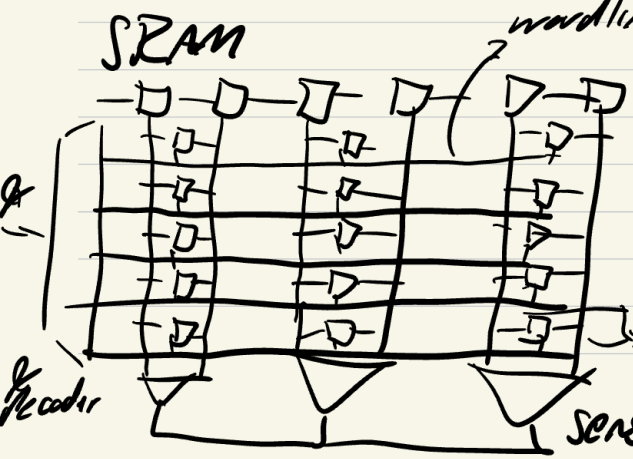
Execution starts by fetching instr. from main mem
All data processed by CPU loaded/stored from main mem
Most prog. manipulate much more data that can be accom. in reg file
Most CPUs limited by b/w (B/s) CPU ↔ main mem (memory bottleneck)

Memory Technologies

	cap	Latency	Cost / GB	
reg	1Kb	20ps	$$$$	processor datapath
SRAM	16KB-16MB	1-16ns	~$1K	mem hierarchy
DRAM	~16GB	80ns	~$10	
Flash*	~100GB	100ms	~$1	I/O subsyst
HD *	~1TB	10ms	~$0.20	

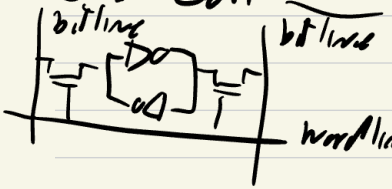
*non volatile

SRAM



Drivers
↓ bc. holds 3b
Periphery circuit to decode
↓ r/w ops.
Access SRAM → provide unique A to specify bc.

SRAM cell



6-Mosfet (6T) cell
2 CMOS inverters (4T) bistable elt
+ 2T

During r, drivers recharge & bitlines to VDD & disconnect
bitlines floating @ 1
& decoder sets 1 of wordlines high, connecting rows of bit cells
to their bitline
& cell in active word pulls 1 of the bitline to 0 Gnd
⇒ slow, since bitline ≤ ↑ C_{total} & T's of Mosfets small to
keep cell as small as possible
Sense amps sense change in bitline voltages, produce O data

During w, drivers set & hold bitline to desired vals
& decoders activates 1 wordline
Each cell in word is overpowered by drivers, stores val

Multiported SRAMs

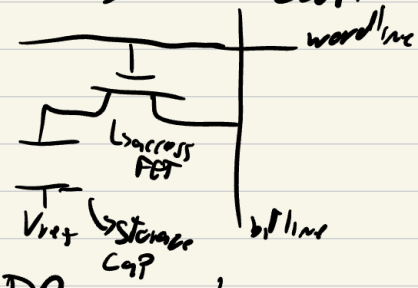
so far can do either 1 r/1w
Multi r/w by + 1 set of wordlines & bitlines per port
Cost/bit ? N ports:
N wordlines
2N bitlines
2N accessFETs
wires often dominate area → $O(N^2)$ area!

Summary:

Array of K×b cells (K words, b cells/word)
cell bistable elt + access T
Analog circuit to allow r/w
read: precharge bitlines, activate wordline, sense
write: drive bitlines, activate wordline, overpower cells

DRAM

1T DRAM cell



20× smaller area
denser, cheaper!
C leaks q. must be refreshed periodically (~ms)

DRAM w/r

w: drive bitline, activate wordline, discharge C
r: precharge bitline to $V_{DD}/2$
activate wordline
C & bitline share q
If C discharged, $V_{bitline}$ ↓
If C charged, $V_{bitline}$ ↑
Sense bitline to determine 0/1

reads are destructive!
data must be rewritten to cell after r.

Summary DRAM

1T DRAM cell: T+C
↓size vs SRAM vs destructive reads & C leakage
DRAM arrays include circ. to
- r word after r
- refresh every word periodically

vs SRAM:
- 20× denser
- ~2-10× slower

Non-Volatile Storage: Flash

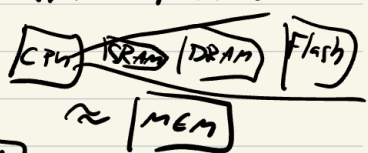
e^- @ float gate ↓ strength from ctrl gate
⇒ no inversion
⇒ NFET off even when word is high

Very dense: multi b/T, r/w in blocks
Slow: 10-100ms
Limited #w: ↓q floating gate (w) ↑V damage T

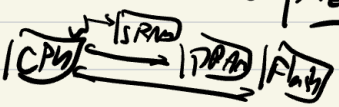
Non-Volatile Storage: HD
Extremely slow (~10ms)
Cheap

Memory Hierarchy
Want large, fast & cheap memory, but
Large = slow
Fast = expensive

Idea: Hierarchical syst of memories w/ different tradeoffs to emulate large, fast, cheap memory?

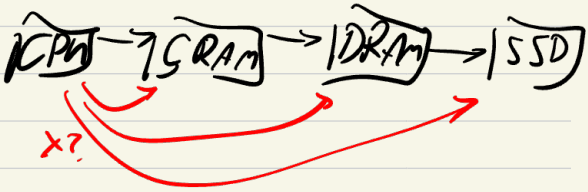


Interface
Approach 1: Expose hierarchy



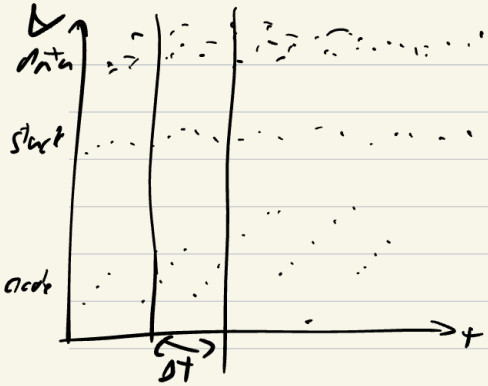
regs, SRAM, DRAM, flash, HD
each avail as storage alloc.
To all programmers "use cleverly"

Approach 2: Hide hierarchy
Program. model: single mem, single space
machine transp. stores data in fast/slow mem depend. on usage & patterns

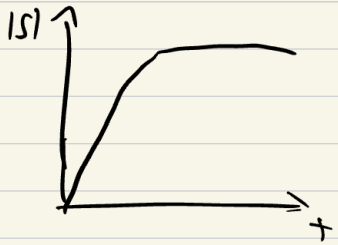


Locality Principle
Keep most often-used data in small, fast SRAMs (often local to CPU)
Refer to mem roughly for rem. data
Why it works: LOCALITY
Locality of Reference: access to X at t ⇒ access to X+Δx at t+Δt more probable as Δx, Δt → 0

Memory Reference Patterns



S set of locations accessed during Δt
Working set S changes slowly wrt access time Δt



Caches

Small, interim component that transp. retains (caches) data from recently accessed locations
- ?? fast access if data is cached otherw accesses slower, larger cache/mem
- Exploits locality principle

Comp. systs often use multiple levels of cache
Widely applied beyond HW (e.g. web caches)

Typical mem Hierarchy

		Access time	Capacity	Managed by
on data path	regs			
on chip	lvl 1 cache	Gets slower	Gets bigger	HW
	lvl 2 cache			
	lvl 3 cache			
other chips	MM			
	Flash drive			SW/OS
mech. dev.	HD			

processor must deal w/ variable mem access

Cache access

CPU ↔ Cache
1) Cache hit
2) Cache miss

Cache Metrics

Hit Ratio: $HR = \frac{hits}{hits + misses} = 1 - MR$

Miss Ratio: $MR = \frac{misses}{hits + misses} = 1 - HR$

Avg. Mem. Access Time (AMAT) = $Hit_{Time} + MR \cdot Miss_{Penalty}$

Goal of caching: ↓ AMAT

Formula can be applied recursively w multi-level hierarchies

Ex: How high of a HR?

HR we need to break even?

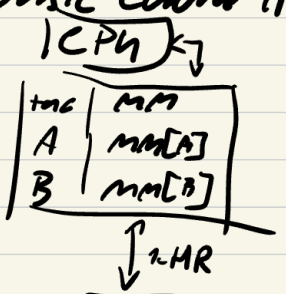
(MM only: AMAT = 100)

$100 = 4 + (1 - HR)100$ HR = 96%

HR to achieve AMAT = 5?

$5 = 4 + (1 - HR)100$ HR = 99%

Basic Cache Algorithm



on ref to mm[X] look for X among tags

HIT: X = TAG(i)

Read: return DATA(i)

write: change DATA(i), start w to mm[X]

MISS: X ∉ TAG

Repl. sel: sel some k to hold mm[X] (Calloc)

Read: Read mm[X], set TAG(k) = X, DATA(k) = mm[X]

write: start w to mm[X]
set TAG(k) = X, DATA(k) = new mm[X]

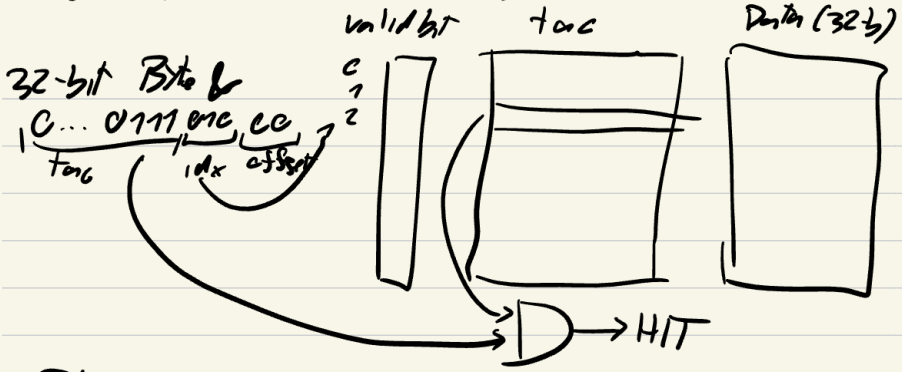
Q: How to "search" cache?

Direct Mapped Caches

Each word w memory maps into a single cache line

Access (for cache w 2^w lines) Index into cache w/ W bits (idx bits)
read out valid bit, tag & data
if (valid bit) & tag matches upper bits, HIT

Ex: 8-location DM cache (W=3)

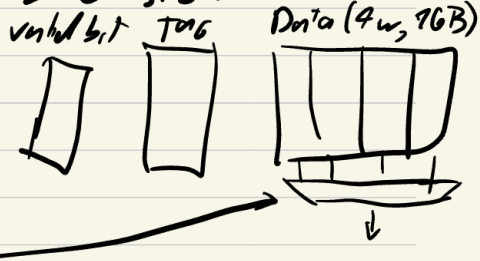


Block Size

Take adv. of locality: increase block size

✓ Reduces size of tag mem

✗ Fewer blocks in cache

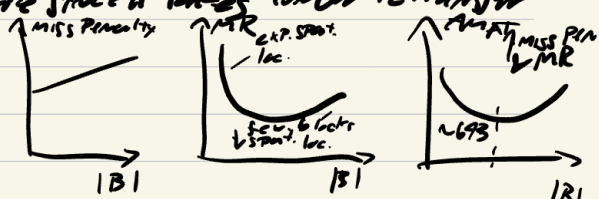


32b Byte Addr

Block size tradeoffs

Larger Block Size

- Advantage of spatial locality
- Incur larger miss penalty since it takes longer to transfer blocks into cache
- Can increase avg HT & MR
- AMAT = HT + MissPenalty · MR



Direct-Mapped Cache Problem: Conflict Misses

Assume 128-line DM cache, |B| = 1w

Consider looping code, steady state. Word A (not B!)

Inflexible mapping (A can be in only 1 cache loc.) ⇒ conflict misses!

Fully-Associative Cache

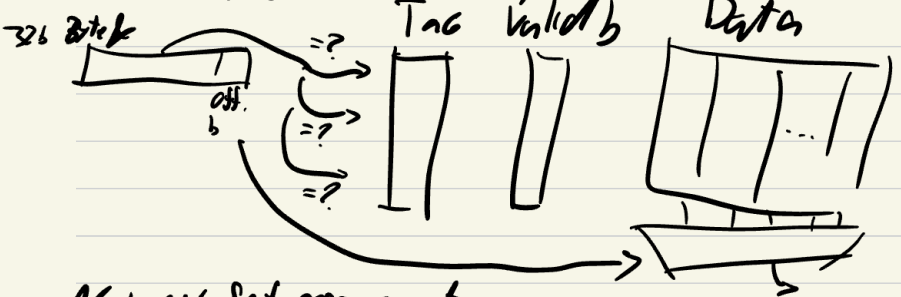
Opposite extreme: b can be $\forall$ loc.

No cache idx

Flexible (no conflict misses)

Expensive: Must comp. tags $\forall$ entry in parallel to find match

(called CAM in HW)



N-way set associative

Compromise b/w direct-mapped & fully-associative

rows = # sets

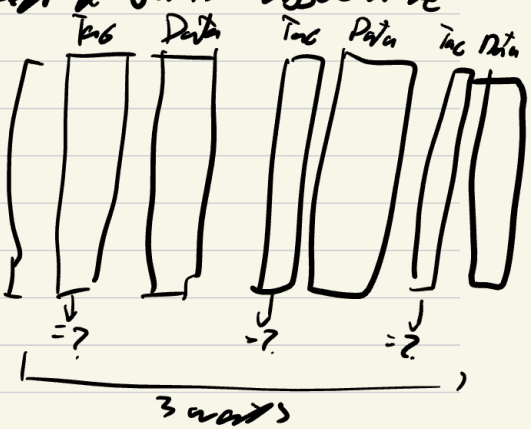
cols = # ways

Set size = "set associativity"

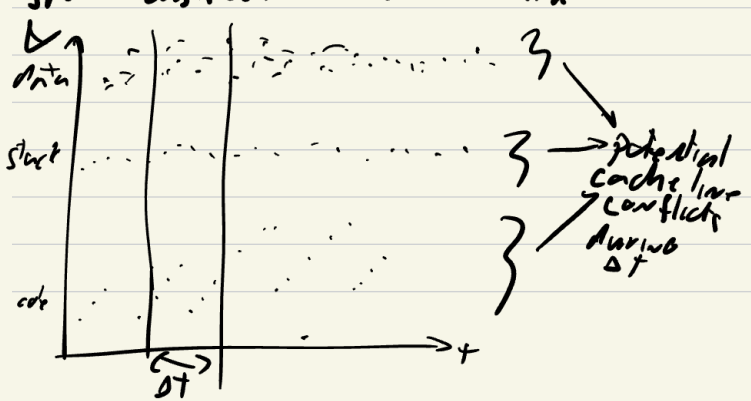
(e.g. 4-way → 4 entries/set)

Compare $\forall$ tag $\forall$ ways in parallel

N-way cache = N direct-mapped caches in parallel



Direct-mapped & fully-associative special cases of N-way set-associative



Associativity Tradeoffs

More ways
↓ conflict misses
↑ Hit Time

Associativity Implies Choices

Direct-Mapped	N-way set-assoc.	Fully assoc
$k \stackrel{?}{=} 1$ Tag	$k \stackrel{?}{=} N$ Tags	$k \stackrel{?}{=} \forall$ Tag
Loc A can be stored in exact. 1 cache line	Loc A can be stored in exact 1 set, but in any of the N cache lines belonging to that set	Loc A can be stored in any cache line

Replacement Policies

Optimal Policy (Belady's MIN): replace the block that is accessed the furthest in the future → req. Knowing the future

Idea: If a block has not been used recently, it's often less likely to be accessed in the near future

Least Recently Used (LRU): replace block that was accessed furthest in the past
- Works well in practice
- Needs to keep ordered list of N items → N! orderings
  → $O(\log_2 N!) = O(N \log N)$ "LRU" bits + cmplx logic

Caches often implement cheaper approx of LRU

Other Policies: FIFO

Random → not very good, but doesn't have adversarial access patterns

Write Policy

Write-through: CPU writes cached, but also written to MM (stall CPU until write completed). MM always holds contents
Simple, slow, wastes b/w

Write-behind: CPU writes cached, w to MM may be buffered. CPU keeps exec. while w comp. in background
Faster, still uses a lot of b/w

Write-back: CPU writes cached, not written to MM until block replaced. MM contents can be "stale"
Fastest, low b/w, more CPX
Commonly implemented in curr sys

Write-back: on ref to MM[X] look for X among tags

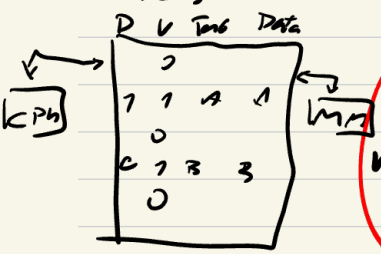
HIT: X = TAG(i)
Read: return DATA(i)
write: change DATA(i), ~~start w to MM[X]~~

MISS: X ∉ TAG
Repl. sel: sel some k to hold MM[X] (Calloc)
Read: Read MM[X], set TAG(k)=X, DATA(k)=MM[X]
write: ~~start w to MM[X]~~
set TAG(k)=X, DATA(k)=new MM[X]

write Data[k] to MM[& TAG[k]]

Write-back:
w/ "dirty" bits

add 1 bit to record whether or not block has been written to. Only write back dirty blocks



on ref to MM[X] look for X among tags

HIT: X = TAG(i)

Read: return DATA(i)

write: change DATA(i), ~~start wr to MM[X]~~ D[i] = 1

MISS: X ∉ TAG

Repl. sel: sel some k to hold MM[X] (Calloc)

Read: Read MM[X], set TAG(k) = X, DATA(k) = MM[X] D[k] = 0

write: ~~start wr to MM[X]~~ D[k] = 1

set TAG(k) = X, DATA(k) = new MM[X]

→ IS(D[K]) w Data[K] to MM[& Tag[K]]

Summary: Cache Tradeoffs

AMAT = Hit Time + MR · Miss Penalty

- ↑ Cache size: ↓ MR ↑ Hit Time
- ↑ |B|: To start w Temp. locality. ↑ Miss Penalty
- ↑ assoc. (ways): ↓ MR, ↑ Hit Time
- ↑ intell. replacement: ↓ MR ↑ Cost
- Wr to Policy: ↓ b/w, ↑ cmplxty

How to navigate all these dimensions? Simulate different Cache organizations on real programs.

Pipelining CPU

32-bit, single-cycle CPU
1 instruction / clk cycle
Circuit loads new PC → sent to MM as b addr.
→ 32-b word instr. fetched from MM
→ Opcode decoded by ctrl logic → ctrl signals
→ operands read from regfile → routed to ALU
→ mem ops: ALU → addr / Data (load)
PC+4 & ALU values can be written to regfile
T_{clk} determined by delay through ∀ comp. involved in instr. exec.
How to make it faster?

Single-Cycle Beta Performance

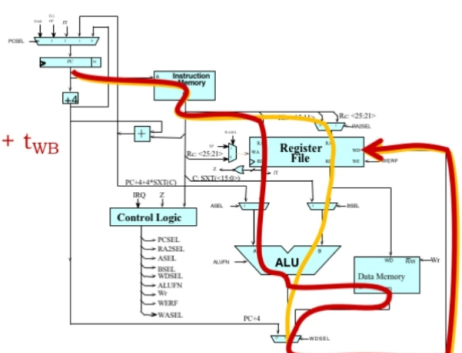
$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}$$

CPI t_{CLK}

- CPI = 1
- t_{CLK} = Longest path for any instruction

$t_{CLK} \approx t_{IFETCH} + t_{RF} + t_{ALU} + t_{MEM} + t_{WB}$

  - Slow
  - Inflexible: Instructions with smaller critical path cannot execute faster



Pipelined Implementation

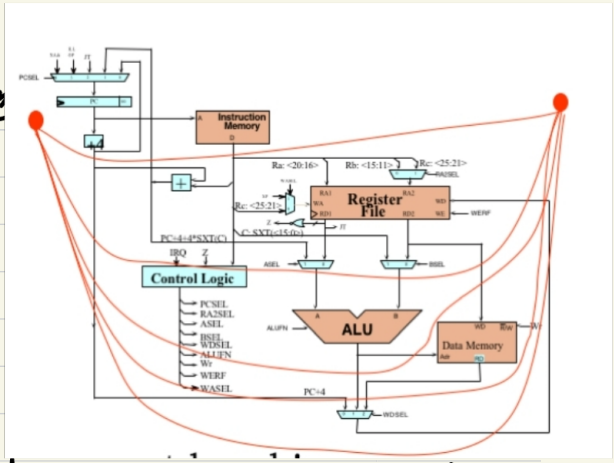
Divide datapath in pipeline stages to ↓ T_{clk}
∀ instr exec. over x cycles
Consec. instruc. overlapped
To keep CPI ≈ 1.0

I: Instr. Fetch: maint. PC, fetches instr, passes to ↓
RF: regfile: reads src from regfile, passes to ↓
ALU: Performs op in ALU, passes result to ↓
Mem: if LD, use ALU res. as addr, pass mem data, ALU res. to ↓
WB: Write-back: writes res. back to regfile

CPU has state: PC, regfile, mem

Idepend. X break

- compute next PC
- write reg into regfile



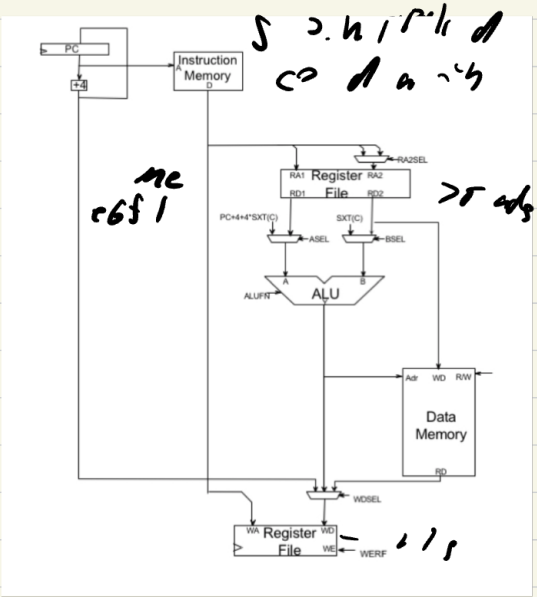
Pipeline Hazards

Pipelining: tries to overlap instrs. → may depend on res. of earlier instr.

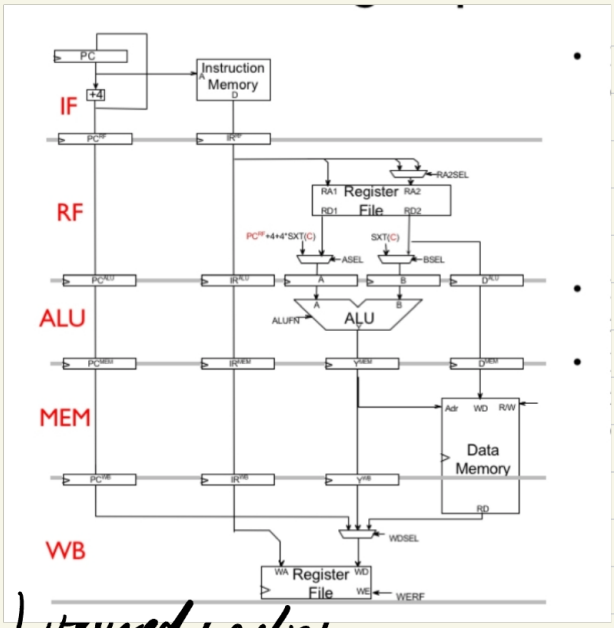
Data value → data hazard

PC → ctrl hazard (branches, jumps, excepts)

1) 5-stage Pipeline
2) Handle data hazards
3) handle ctrl hazards

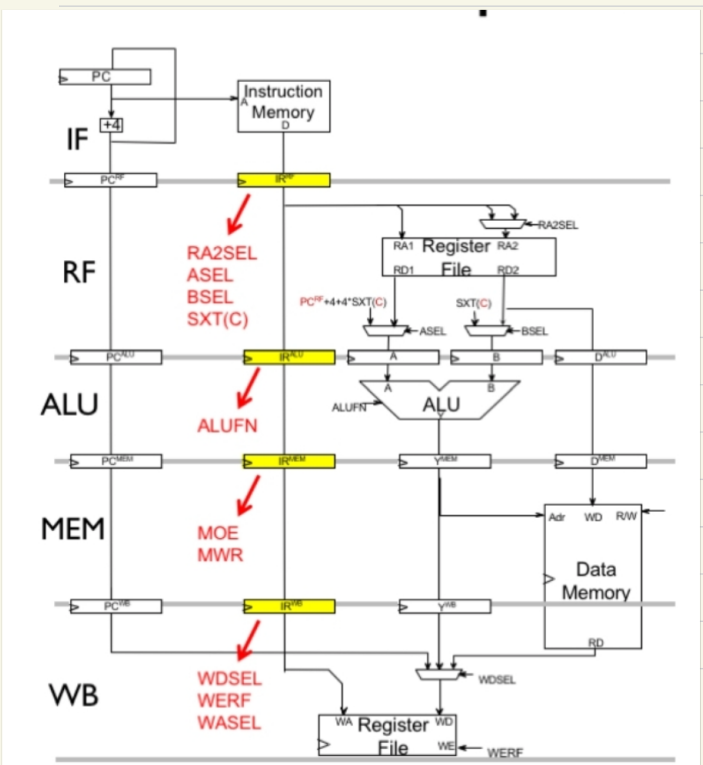


5-stage Pipeline Datapath



∀ stages services 1 instr/cycle
Data mem reads are pipelined, not combinational
Data reads appear in RD the next cycle

Pipelined control

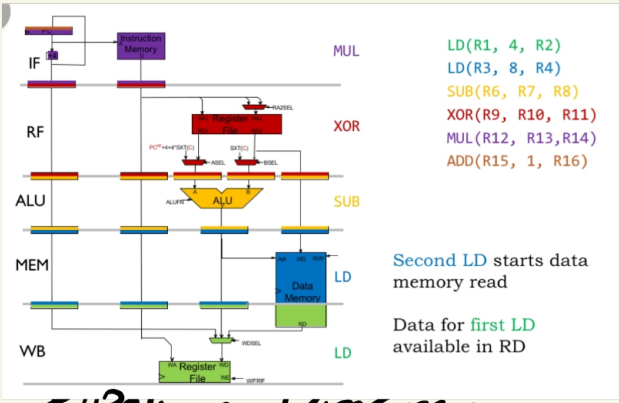


Instr. contents propagated through pipeline in Instruction Registers (IR's)

Ctrl signals ∀ stage generated from corresponding IR

Pipeline hazards will require new ctrl signals

Example of Pipelined Instructions



Pipeline Diagrams

When do r/w happen?

Cycles →

Stages	1	2	3	4	5	6
IF	LD	LD	SUB	XOR	MUL	ADD
RF		LD	LD	SUB	XOR	MUL
ALU			LD	LD	SUB	XOR
MEM				LD	LD	SUB
WB					LD	LD

Read R1 ↓ Write R2 ↑

reads @ RF

writes @ WB

Data Hazards

- Consider this instruction sequence:

```
ADDC(R1, 1, R2)
SUBC(R2, 4, R3)
MUL(R6, R7, R8)
XOR(R9, R10, R11)
...
```

	1	2	3	4	5	6
IF	ADDC	SUBC	MUL	XOR		
RF		ADDC	SUBC	MUL	XOR	
ALU			ADDC	SUBC	MUL	XOR
MEM				ADDC	SUBC	MUL
WB					ADDC	SUBC

- SUBC reads R2 on cycle 3, but ADDC does not update it until end of cycle 5 → R2 is stale!
- Pipeline must maintain correct behavior…

Resolving Stalls:

1) Stall. Wait for res to be avail. by freezing earlier stages

2) Bypass/Forward. Route data to earlier stage asap it's calculated

3) Speculate
- Guess a value & exec.
- actual val avail:
  a) ✓ → continue
  b) X → kill & restart

1) Stall

addc r1, 1, r2
subc r2, 4, r3
mul r6, r7, r8
xor x9, x10, x11

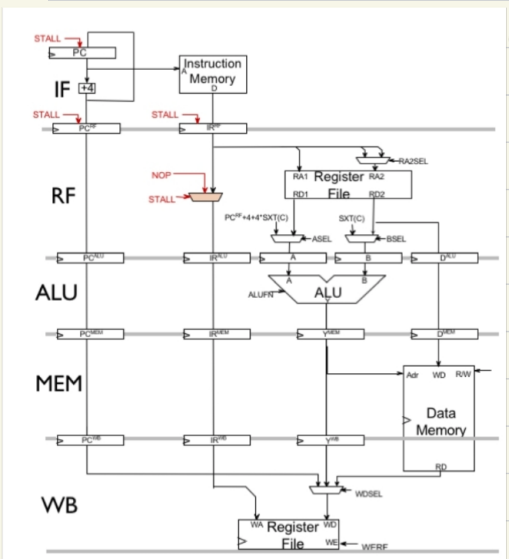
Stall ...

	1	2	3	4	5	6	7	8
IF	ADDC	SUBC	MUL	*MUL*	*MUL*	*MUL*	XOR	
RF		ADDC	SUBC	*SUBC*	*SUBC*	*SUBC*	MUL	XOR
ALU			ADDC	**NOP**	**NOP**	**NOP**	SUBC	MUL
MEM				ADDC	NOP	NOP	NOP	SUBC
WB					ADDC	NOP	NOP	NOP

↑ R2 updated

Stalls increase CPI!

Stall logic



New Stall Ctrl signal
STALL = 1
- Disables PC & RF pipe. regs
- Injects NOP into ALU

Ctrl logic sets STALL = 1 if src of instr. in RF match w/ dest of ALU, MEM, WB

addc r1, 1, r2
subc r2, 4, r3
mul r6, r7, r8
xor x9, x10, x11

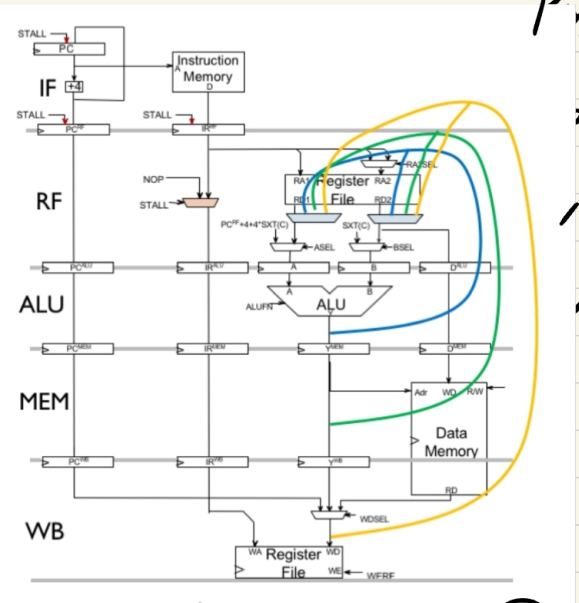
2) By Pass

addc w/ r2 by end of cycle 5
res. avail at end of ALU stage!

	1	2	3	4	5
IF	ADDC	SUBC	MUL	XOR	
RF		ADDC	SUBC	MUL	XOR
ALU			ADDC	SUBC	MUL
MEM				ADDC	SUBC
WB					ADDC

ADDC result computed ↑ ↖ R2 updated

Bypass Logic



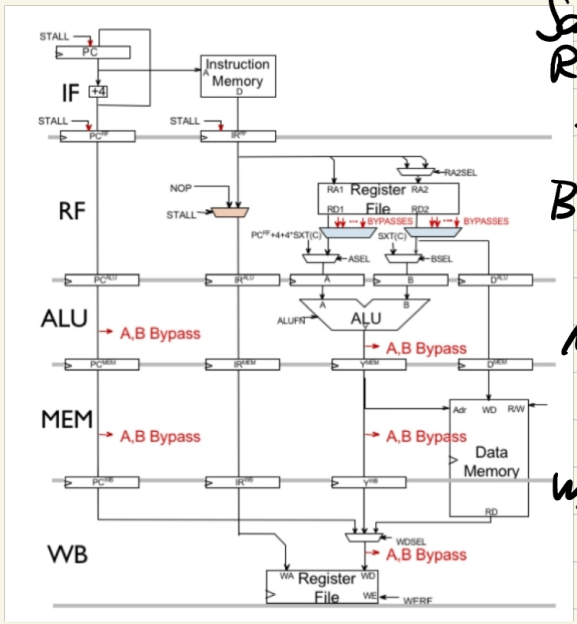
Bypass muxes to RF outputs

write ALU, Mem, WB output to mux inputs

Bypass val. if dest of instr. in ALU, Mem, WB == src instr. RF

What if multiple matches?
Select most recent instr.
ALU > Mem > WB

Fully Bypassed Pipeline



Some instr. write PC+4
Route PC ALU & PC MEM as add. bypass mux inputs

Bypasses expensive!
Wires & large muxes

Not needed! We can stall
e.g. just bypass from ALU

w/ FBP, do we need stall?

Load-to-use Stall

Bypassing cannot elim. load delays
→ data X avail. until WB!

	1	2	3	4	5	6	7
IF	LD	SUBC	MUL	MUL	MUL	XOR	
RF		LD	SUBC	SUBC	SUBC	MUL	XOR
ALU			LD	NOP	NOP	SUBC	MUL
MEM				LD	NOP	NOP	SUBC
WB					LD	NOP	NOP

LD data available ↑ ↖ R2 updated

Bypassing from WB still saves a cycle

subc r2, r, r3
mul...
xor...

Summary: Pipelining w/ Data Hazards

1) Stall
   Simple, wastes cycles, ↑CPI

2) Bypass
   ↑expensive, ↓CPI
   still needs stalls when res. produced after ALU
   - can use ↓bypass, ↑stall

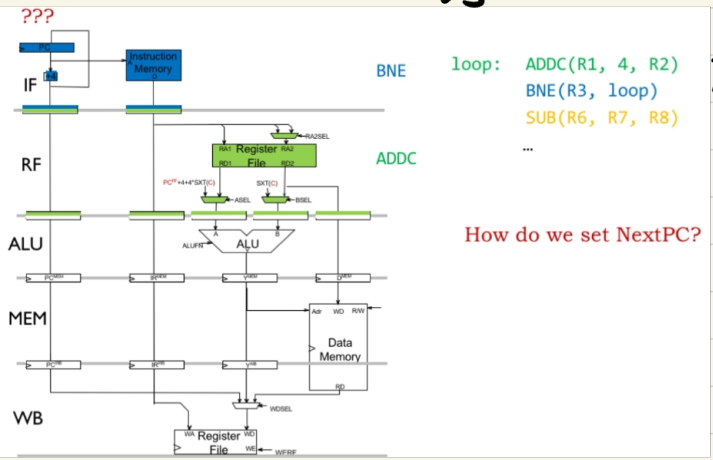
↑Pipelining ⇒ ↑Data Hazards
↓t_{clk}, ↑CPI

Compilers can help!
reorrange code to put dependent instr. further away
only works well when compiler can find indep. instrs. to move around

Change the ISA? So regs updated w/ 3 instr delay?

ISAs outlive implementations.

Control Hazards



What do we need for NextPC?
Branch: if (reg == ?) PC = 4 + offset
else PC += 4

Jump: reg = PC + 4
PC = reg

Need reg unknown until RF

Resolving Ctrl Hazards

1) Stall
2) Speculate

Stall

addc r1, -1, r3
⋮
bne r3, loop

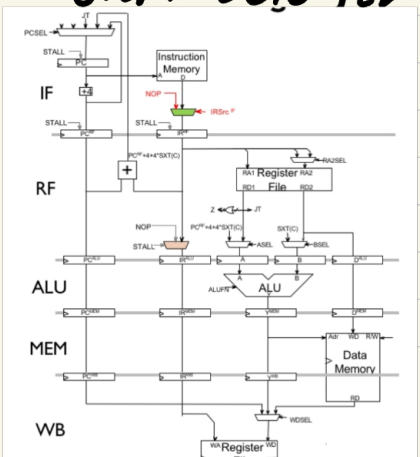
If branch/jump in IF, stall IF 1 cycle

(Assume BNE taken in ex code)

	1	2	3	4	5	6	7	8	9
IF	ADDC	MUL	BNE	NOP	ADDC	MUL	BNE	NOP	ADDC
RF		ADDC	MUL	BNE	NOP	ADDC	MUL	BNE	NOP
ALU			ADDC	MUL	BNE	NOP	ADDC	MUL	BNE
MEM				ADDC	MUL	BNE	NOP	ADDC	MUL
WB					ADDC	MUL	BNE	NOP	ADDC

↑ R3!=0 → Taken ↑ R3!=0

Stall logic for Control Hazards



IRSrc IF control signal
If opcode RF == J, B
IRSrc IF = 1, inject NOP
Set PCsel to J/B

ISA issue, Simple vs Cmpx Branches

Other ISAs ? cmpx branches resolved in ALU

	1	2	3	4	5	6	7	8
IF	ADDC	MUL	BNE	SUB	**NOP**	ADDC	MUL	BNE
RF		ADDC	MUL	BNE	**NOP**	NOP	ADDC	MUL
ALU			ADDC	MUL	BNE	NOP	NOP	ADDC
MEM				ADDC	MUL	BNE	NOP	NOP
WB					ADDC	MUL	BNE	NOP

More annulments (but sometimes fewer instructions)

2) Speculate

Good Guess for NextPC? = PC+4

Assume BNE not taken

	1	2	3	4	5	6	7	8	9
IF	ADDC	MUL	BNE	SUB	XOR				
RF		ADDC	MUL	BNE	SUB	XOR			
ALU			ADDC	MUL	BNE	SUB	XOR		
MEM				ADDC	MUL	BNE	SUB	XOR	
WB					ADDC	MUL	BNE	SUB	XOR

Start fetching at PC+4 (SUB) but BNE not resolved yet…

Guessed right, keep going

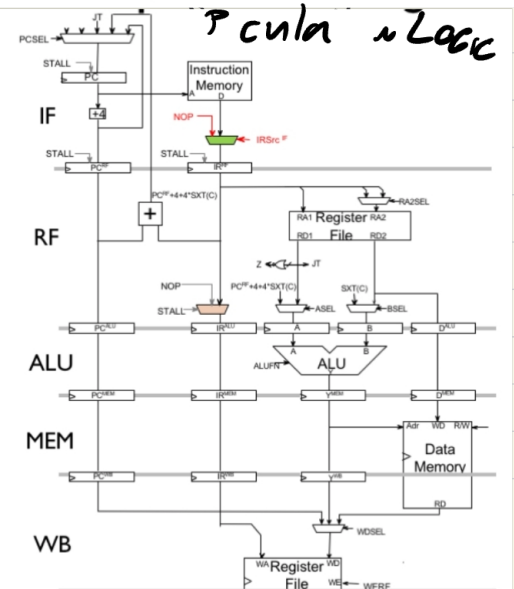
Assume BNE taken

	1	2	3	4	5	6	7	8	9
IF	ADDC	MUL	BNE	SUB	ADDC	MUL	BNE	SUB	ADDC
RF		ADDC	MUL	BNE	**NOP**	ADDC	MUL	BNE	**NOP**
ALU			ADDC	MUL	BNE	NOP	ADDC	MUL	BNE
MEM				ADDC	MUL	BNE	NOP	ADDC	MUL
WB					ADDC	MUL	BNE	NOP	ADDC

Start fetching at PC+4 (SUB) but BNE not resolved yet…

Guessed wrong, annul SUB

Pipeline Logic



Familiar: same path circuitry

Instead of setting IRSRC IF = 1 for branches, set = 1 when branch taken

If opcode RF == J/taken B

- IRSrc IF = 1, inject NOP to annul fetched instr. ("branch annulment")
- Set PCsel to J/B

Branch Prediction

Always Guessing PC+=4 wastes a cycle on taken J/B
~↑10% CPI
W/ deeper pipelines, taken B wastes many more cycles

Modern CPUs dynamically predict outcome of control-flow instr.
- Predict both B condition / target
- Works well bc B have repeated behavior
  e.g. B loops taken
  term/limit/error not taken

Branch Delay Slots

Change ISA so instr. following J/B always exec.

	1	2	3	4	5	6	7	8
IF	ADDC	BNE	MUL	ADDC	BNE	MUL	ADDC	BNE
RF		ADDC	BNE	MUL	ADDC	BNE	MUL	ADDC
ALU			ADDC	BNE	MUL	ADDC	BNE	MUL
MEM				ADDC	BNE	MUL	ADDC	BNE
WB					ADDC	BNE	MUL	ADDC

```
loop: addc r1, -1, r3
      bne r3, loop
      mul r4, r5, r6
```
↳ executes regardless of B outcome

✓ If compiler can fill in w/ useful instr., no penalty
✗ Can't fill slot ~50% of time → NOP must, longer code
Longer pipeline → more delay slots?
B predic. works better in practice

Exceptions:

Need to: Save curr PC+4 in XP
Load PC w/ excep. vector (NMI / Xaddr)
Cause control flow hazards!
Implicit B
Want precise exceptions
- Prec. instr. must have completed
- instr. causing excep. & future instr. must not have exec
- No updates to reg / mem

Simple in single-cycle, cmplx in pipeline

When can Exceptions Happen?

Memory fault, illegal instruction, arithmetic exception...
Instr. following exceptions may already be in pipeline
→ none of them written to regs/mem!

Resolving Exceptions

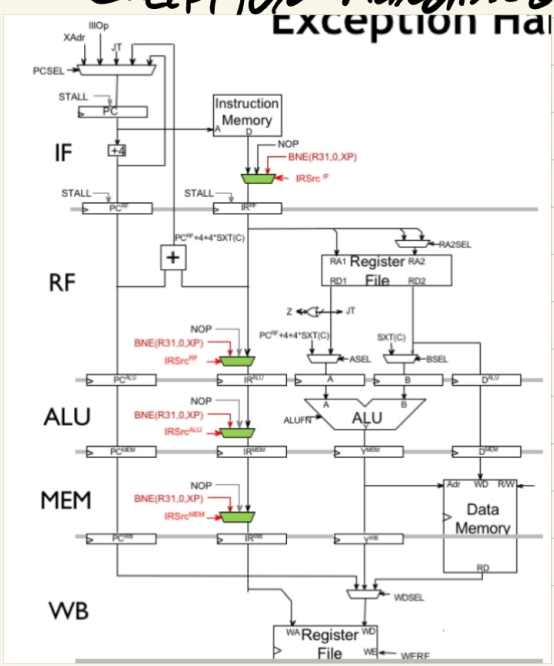
If instr. except. at stage i
- Turn instr. into BNE(r31, 0, XP) to save PC+4
- Annul instr. in stages i-1, ..., 1 (flush pipeline)
- Set PC ← IIOp / XAdr

Ex: LD w/ mem fault

	1	2	3	4	5	6
IF	LD	ST	MUL	SUB	ADDC	ST
RF		LD	ST	MUL	**NOP**	ADDC
ALU			LD	ST	**NOP**	NOP
MEM				LD	**NOP**	NOP
WB					**BNE**	NOP

Exception Handling Logic

IRSrc{IF, RF, ALU, MEM} muxes to inject NOP / BNE
- NOP if preceding instr except.
- BNE if instr. in curr stage has an exception



Multiple Exceptions?

ld → mem fault
??
mul
sub

	1	2	3	4	5	6
IF	LD	???	MUL	XORC	ADDC	ST
RF		LD	???	**NOP**	**NOP**	ADDC
ALU			LD	BNE	**NOP**	NOP
MEM				LD	**NOP**	NOP
WB					**BNE**	NOP

Invalid opcode detected

Memory fault detected

Asynchronous Interrupts

Interrupts are easier:

```
// Interrupted code:
        ...
        LD(...)
        ADD(...)
        SUB(...)   <-- Interrupt Taken HERE
        ...
// Interrupt handler:
XAdr:   OR(...)
        ...
        SUBC(xp,4,xp)
        JMP(xp)
```

	1	2	3	4
IF	ADD	**BNE**	OR	...
RF	LD	ADD	BNE	OR
ALU		LD	ADD	BNE
MEM			LD	ADD
WB				LD

Suppose interrupt req. while SUB in IF (cycle 2)

Handling:

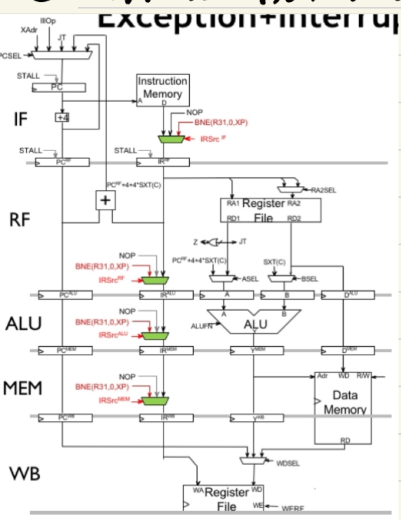
replace SUB w/ BNE(..., XP)

select Xaddr as nextPC

code handler to return SUB instr.

ADD & earlier insts. unaffected

Exception + Interrupt Handling Logic



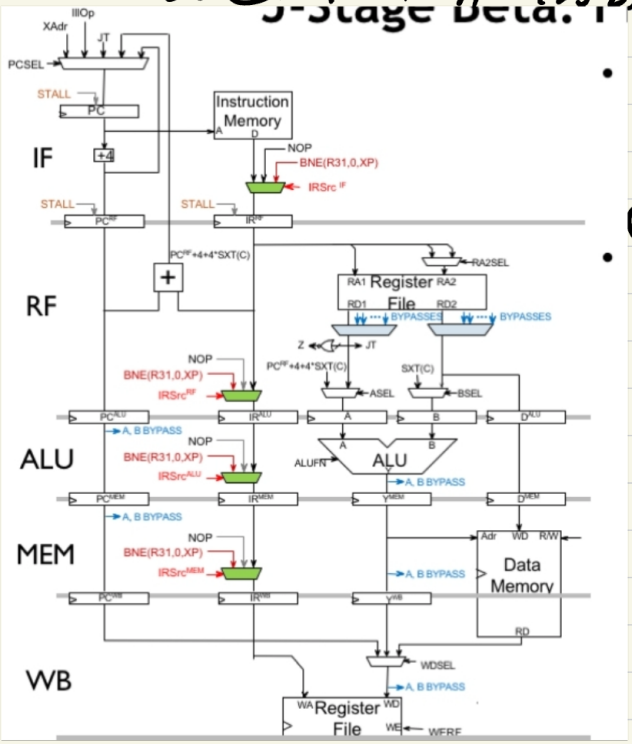
Same as before

$IR_{Src}^{IF, RF, ALU, MEM}$ muxes to inject NOP/BNE

- NOP if prec. instr. has an except.
- BNE if instruc. in curr stage has an except

Use IR_{Src}^{IF} mux to inject BNE on an interrupt (same as an except. in IF)

5-stage CPU: Final Version



- Data hazards
  - Stall IF, RF (STALL=1 + $IRSrc^{RF}$ = NOP)
- Control hazards:
  - Speculate PC+4
  - JMP or taken B in RF
    - $IRSrc^{IF}$ = NOP
    - PCSel → JT / B target
  - If except. at stage X
    - $IRSrc^{X}$ = BNE
    - Prev $IRSrc^{X}$ = NOP
    - PCSel = XAddr or IllOp
  - If interrupt
    - $IRSrc^{IF}$ = B
    - PCSel = XAddr