Defining Comp Arch.

ISA :→ the programmable instruction set of a processor → the lang to communicate with a μp.

Can be of diff types →
reg-mem ISA → x86
Load-store ISA → RISC (arms, mips).

Mem addressing
↳ typically byte aligned.
a mem object of size s bytes residing at addres A will imply →
A mod s = 0

Addressing modes
Ways to specify mem objects.

MIPS modes → Register, immediate, displacement / offset based

ARM modes → MIPS modes + PC relative addr. sum of 2 regs,

Type & size of operands → 8, 16, 32, 64, → bits.
↳ double.
char | int/float
micode

Operations
data transfer, Arithmetic, Logical, Control,

Encoding
R format: opcode rs rt rd
I format ~~data transfer~~ opcode rs rt immediate.
J format jump. → opcode address

Energy & Power within µP

Dynamic Power consumption in clocked circuits

$$P_{dynamic} \propto \frac{1}{2} \cdot C \cdot V_{dd}^2 \cdot f$$

clk rate ~ 3.3 GHz 2003 onwards saturated.

Techniques

① clock gating → turn off unactive units.

② DVFS.

③ On chip temp sensor
↳ turn off units based on thermal scenario.

DRAM freq, spin disk at low rate.

Cost trend

$$\text{IC cost} = \frac{\text{die cost + testing \& packaging cost}}{\text{Final test yield}}$$

Dependability

$$\text{Availability} = \frac{MTTF}{MTTF + MTTR}$$

MTBF,

$$MTTF_{system} = \frac{1}{\text{failure rate}}$$

System with multiple subsystem
→ failure rates add up.

$$MTTF_{\text{power supply pair}} = \frac{MTTF_{\text{supply}} / 2}{MTTR_{\text{supply}}}$$
$$MTTF_{\text{supply}}$$

Performance index

(1) response time / exec time

(2) total work done inside some T → Throughput

$$\text{execution} \propto \frac{1}{\text{performance}}$$

Benchmarks.

set of representative workloads in a domain.

Desktop → SPEC — standard Perf Eval Corp

SPECINT
SPECFP

Server → SPECFS — file server
SPECWeb — web server
SPECvirt — virtualized services
TP (Transaction processing benchmarks)

Emb

EEMBC
electronic design News Embedded Microprocessor Benchmark Consortium.

Quantitative principles of Computer Design.

(1) Take advantage of parallelism
Moore's law does not hold any more.

(2) Principle of Locality.

(3) Optimize the common case first.

Amdahl's Law

$$\text{Speedup} = \frac{\text{Performance of entire task using the enhancement when possible}}{\text{Performance of entire task w/o any enhancement.}}$$

$$Ex_{new} = Ex_{old} \times \left[\left(1 - \text{fraction}_{\text{enhanced}}\right) + \frac{\text{fraction}_{\text{enhanced}}}{\text{speedup}_{\text{enhanced}}} \right]$$

$$\therefore \text{speedup}_{\text{overall}} = \frac{Ex_{new}}{Ex_{old}} = \frac{Ex_{old}}{Ex_{new}}$$

$$= \frac{1}{\left(1 - \text{fraction}_{\text{enhanced}}\right) + \frac{\text{fraction}_{\text{enhanced}}}{\text{speedup}_{\text{enhanced}}}}$$

$$CPI = \frac{\text{CPU cycles for program}}{\text{Instruction count}}$$

$$\text{CPU Time} = \text{Instr count} \times CPI \times \text{clk cycle time}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{clock cycle}}$$

$$\text{CPU clock cycles} = \sum_i IC_i \times CPI_i$$

$i \rightarrow$ some specific instr.
instr i'

$$CPI = \frac{\sum_{i=1}^{n} IC_i \times CPI_i}{\text{Instr count}} = \sum_{i=1}^{n} \frac{IC_i}{\text{Instr count}} \times CPI_i$$

ILP,

hazard discussion — directly from book.
Dynamic scheduling with — directly from book.
Tomasulo algorithm — directly from book.
discussion

↳ notes

Dynamic Scheduler

(1) → can handle binary execution, even if the compiler has not performed instr scheduling.

(2) can exploit dynamic dependencies unknown at compile time.

(3) tolerate execution delay due to events like mem access, cache miss, - - - .

key idea → allow instr occurring later to go forward if hazards are resolved (structural / data hazards).

separate the issue process to two stages
- check structural hazards,
- wait for ~~stalls~~ operand availability.

issue in in-order, execution & completion in end-of-order.

Ex:
```
DIV.D   F0  F2  F4
ADD.D   F6  F0  F8      S.D   F6  0(R1)
SUB.D   F8  F10 F14
MUL.D   F6  F10 F8
```

name dep [anti-dependence (WAR) → (ADD, SUB), (S.D, MUL)
           o/p - dependence WAW → ADD, MUL.

data dependence (DIV → ADD,
                  ADD → S.D,
                  SUB → MUL)

Eliminate all name deps by renaming.

```
DIV.D   F0  F2  F4
ADD.D   S   F0  F8
S.D     S   0(R1)
SUB.D   T   F10 F14     // also, need to replace future F8 reads by T
MUL.D   F6  F10 T
```

Tomasulo

Register renaming performed by reservation stations. (Rs)

Rs → buffer instr operands waiting issue.

a Rs will fetch the operand as soon as it is computed & available in FU o/p.

pending instr → will designate Rs for their i/ps.

More Rs than registers.
eliminates hazards (WAR, WAW)

CDB → common data bus.
update all units waiting for o/p in parallel → FU op → Reg file
→ Rs.

Load/store buffer
↳ hold data/addr coming from/going to Rs.

Tomasulo scheme → uses Rs as extended virtual registers.
other approaches may use additional registers.
ROB etc.

1 cycle latency → data computed in FU & in reg file.

Load/store instr → step 1 → compute effective addr when base reg is available & it is placed in load/store buffer.
Loads exec with availability of memory.
Stores exec ... value being stored in rs.
→ in program order.

FU o/p → CDB → Reg file
→ RS

Register status tag.

The tag for any register says which RS contains the instr that will produce result for this register.

Each RS has seven fields.

Op → operation to perform on source operands S_1 & S_2

Q_j, Q_k → which RS is due to produce results.
= 0 ⇒ result already available.

V_j, V_k → operand values when available

A → hold info for mem addr calculation ld/st.

Busy → status field,

Regfile Q_i → which RS o/p comes here.
for each reg

Ex:→

1.
L.D	F_6,	32(R_2)
L.D	F_2,	44(R_3)
MUL.D	F_0,	F_2, F_4
SUB.D	F_8,	F_2, F_6
DIV.D	F_{10},	F_0, F_6
ADD.D	F_6,	F_8, F_2

from book (p-177)

all instrs issued.
first load completed. result in CDB.
LD_2 → waiting on memory.

can avoid WAW, WAR hazard,
ex:→ DIV, ADD both issued
2 possibilities → (i) instr to update F_6 before DIV is complete.
LD → then V_k has result → DIV execute
→ ADD exec in ||.

(ii) LD not complete, Q_k → Load, RS
Let ADD exec & update F_6.

Problems on Throughput

You want to build a transaction system with capacity of ...
If you use 120 processors to achieve this capacity, where each processor transaction would take 10 ms on a single processor, how many ms each extra processor must work!!

Single processor throughput = $\frac{1}{10 \text{ ms}}$ = 100 trans/sec

desired capacity gain = $\frac{1000}{100} = 10$.

$$10 = \frac{120}{1 + (120-1) \times x}$$

$$100 + (120-1) \times x = 1000$$

$$\Rightarrow x = \frac{900}{119} \text{ trans/sec.}$$

$\therefore$ $\frac{900}{119}$ trans — 1 sec.

$\Rightarrow$ 1 transaction = $\frac{119 \times 1000}{900}$ ms.

computing average throughput

→ n workloads, each requiring m operations, consider that throughput for a processor for these workloads are

$\frac{m}{t_1}, \frac{m}{t_2}, \cdots \frac{m}{t_n}$. $\therefore$ average throughput = $\frac{m \cdot n}{\sum t_i}$

diff no. of operations = $\frac{m_1 + \cdots + m_n}{t_1 + t_2 + \cdots + t_n}$ = $\frac{mn}{\frac{m}{t_1} + \cdots + \frac{m}{t_n}}$

→ harmonic mean.

Compute average speedup →

		proc A	proc B
(100 ops)	workload 1	10 op/sec	20 op/sec
	— — — 2	20 op/sec	10 op/sec.

HM of speedups B/A

$$= \frac{1}{\frac{1}{2}\left(\frac{10}{20} + \frac{20}{10}\right)} = 0.8.$$

HM of speedups A/B

$$= \frac{1}{\frac{1}{2}\left(\frac{10}{10} + \frac{10/20}{10/10}\right)}$$

$$= \frac{1}{\frac{1}{2}\left(\frac{20}{10} + \frac{10}{20}\right)}$$

$$= 0.8 \text{ ms.}$$

confusing, both can not be true.

Geometric mean makes sense = $\sqrt{\frac{100}{20} \times \frac{10}{10}} = 1$
A/B

$\frac{100}{10} \quad \frac{100}{20}$

Consider latencies →

LD — 1 cycle
ADD — 2 cycle
MUL — 6 cycle
DIV — 12 cycle

CLK			
1	LD F6 ... issue		
2	LD, ex	, LD$_2$ issue	
3	LD, com	, LD$_2$ ex	, MUL issue
4.	LD$_2$ com	, MUL exec	, SUB issue
5.	MUL ex,	SUB ex,	DIV issue
6.	n	n ,	n ex, Add issue
7.	n	, SUB commit,	n , Add ex
8.	n	,	, wait , n
9.	n	,	, n , Add comm
10.	MUL commit,		, DIV ex ,

MUL can execute w/o LD doing WB to F$_2$ using RS

ILP ≠ IPC

= in perfect proc with perfect BP.

Ex → 2 imul, 0-0-0, 1 MUL FU
2 intr /cycle 2 ADD/SUB/XOR

					cycle
ALU1 →	ADD	R1	R2	R3	1
	SUB	R4	R1	R5	2
use ALU2 →	XOR	R6	R7	R8	1
	MUL	A5	R8	R9	1
	ADD	R4	R8	R9	1

$\frac{5}{2} = 2.5$ ILP

IPC

ILP in program properties → no info of processor resource.

IPC

	cycle
ADD	1
SUB	2
XOR	1
MUL	1
ADD	2

FU
ADD
SUB
XOR
MUL
ADD

5/2

if only 1 ADD/SUB/XOR → IPC = $\frac{5}{4}$

IPC ≤ ILP

Tomasulo back

Instr state

Issue	Wait until	Action

FP op	station r empty	if (Regstat[rs].Qi ≠ 0)
		{ Rs[r].Qj ← Regstat[rs].Qi
		Rs[r].Qj ← 0 }
		else
		{ Rs[r].Vj ← Reg[rs];
		Rs[r].Qj ← 0; }
		if (Regstat[rt].Qi ≠ 0)
		{ Rs[r].Qk ← Regstat[rt].Qi }
		else
		{ Rs[r].Vk ← Regs[rt];
		Rs[r].Qk ← 0; }
		Rs[r].Busy ← yes;
		Regstat[rd].Q ← r;

Load /store	Buffer r empty	if (Regstat[rs].Qi ≠ 0)
		{ Rs[r].Qj ← Regstat[rs].Qi }
		else
		{ Rs[r].Vj ← Regs[rs];
		Rs[r].Qj ← 0; }
		Rs[r].A ← imm
		Rs[r].Busy ← yes
		Regstat[rt].Qi ← r;

Load only	→	
store only	→	if (Regstat[rt].Qi ≠ 0) { Rs[r].Qk ← Regstat[rt].Qi } else, { Rs[r].Vk ← Regs[rt].Qi Rs[r].Qk ← 0; }

Book example 181 prog

Doing ld & st o-o-o:,

If ld st access same addrs:,

:) ld before st in program order,
interchange will result in WAR

:) st is before ld, interchange will lead to RAW

Similarly, interchanging two stores → WAW;

∴ for executing a ld, check if ∃ incomplete st with same addr.
similarly, check for stores.

one option is to calculate effective addresses in program order.

If ld addrs, match the addr of any active st, then ld instruction is not sent to load buffer until the conflicting store completes.

stores operate similarly, only that you need to check for same addresses in both ld & st buffers.

Drawbacks of Tomasulo
↳ HW support, RS & CDB.

Renaming architectural registers to larger set of physical registers using RAT (Reg allocation table)

& buffering of ~~oper~~ operands from reg file.

RAT → ..... translate architectural registers to physical registers.

HW based speculation

A wide issue processor may need to execute a branch every cycle to maintain max perf.

HW based speculation

(1) dynamic branch prediction → chose which instrs to execute;

(2) speculation to allow instr exec before control dep are resolved.

(3) dynamic scheduling (to deal with diff combination of basic blocks.)

w/o speculation → dynamic sched works only inside a BB.

Extend Tomasulo to support speculation.

separate o/p the two parts

↳ bypassing of results among instrs for executing them speculatively & the actual completion of instrs.

introduce an 'instr commit' phase, to allow instrs to exec o-o-o but commit in-order.

adding a phase means introducing additional data buffers. → **ROB**

commit instr.

ROB → fields → op, destination, value, ready field.

op: Br, st, ld, ALU

destination: <MEM, Reg>

value: ↓ result before commit

ROB subsumes store buffers.

Ex.

Every Instr has a position in ROB before commit.
Hence, we tag update results with ROB entry no, rather than RS no.

Instr exec steps

① Issue : get instr from queue. Issue instr if there is an empty slot in a matching RS & empty slot in ROB.
Send operands to RS if available in Reg/ROB.
do book-keeping → i.e. ROB/RS entry etc.
stall if no matching RS free / ROB full.

② Execute : wait in RS for operands.
L.D → effective addr calc in int pipeline. & load from mem to ROB
S.D → only eff. addr calc.

③ Write Res :
i) FU o/p → CDB → ROB
→ RS with i/p dep.
ii) mark RS of FU as free
S.D → if i/p available → update ROB
if not → monitor CDB

④ Commit
if branch instr reach ROB head with incorrect prediction status, ROB is flushed.
If store update Mem rather than Reg file.
if others (normal commit) When instr reach ROB head, update Reg file from ROB entry then instr from ROB.

ROB ensure → ILP across BB.
o-o-o exec, in order commit.

Ex. from Book

add 2 cycles.
mul 6 cycles.
div 12 cycles.

```
L.D     F6, 32(R2)
L.D     F2, 44(R3)
MUL.D   F0, F2, F4
SUB.D   F8, F2, F6
DIV.D   F10, F0, F6
ADD.D   F6, F8, F2
```

Modifications

RS → Qj, Qk fields now have ROB entry no in place of RS no.

Register status field → ROB entry

We compute diff entries when MUL.D is under exe. SUB. has completed but cannot commit.

Fig from Book

Key observation → no. instrs after the earliest uncompleted instr is allowed to complete.

Implication of ROB → we can execute code dynamically (supporting speculation) while maintaining "precise interrupt model"

state consistency → ex: if MUL.D creates an interrupt, we can wait (before handling the interrupt) till it reaches ROB head.

↳ this yields a precise exception since instr commits are in order.

in this ex, SUB & ADD can finish, before MUL/DIV.

W/O ROB, then F8 & F6 get changed & interrupt due to MUL will lead to inconsistent state.

Common case in some arch → floating point exceptions are allowed to be imprecise → as program will terminate.

Page fault exception → need precise handling.

Fig from Book

Then a few imp points :→

exception is raised → record in ROB.

if branch mispredict → exception flushed with all instr after branch.

if instr reach ROB head & no mispredict
↳ exception taken.

general :- handle exception as soon as earlier branches get resolved.

handling stores in speculative system

store instr update mem only when ROB head reached.

∴ mem update happens only when it is now longer speculative.

store o/p → Vk field of store res station
↳ ROB

In reality, store result can arrive late, just before store ROB reach head.

directly from CDB → ROB entry.

HW tracks if any completing instr has any dependency on store with pending update.

effect → allow store to pass through with result stage w/o result available.

WAR/ WAW → in/~~out~~ order commits.

RAW → i) L.D cannot load value (2nd step if L.D exec)
if an ROB entry of S.D has matching destination field.

ii) L.D eff addrn calculation in program order w.r.t all earlier stores.

extend these ideas to → INT pipeline, multiissue proc.

More examples

Branch Misprediction Recovery.

```
LD  R1, 0(R1)        ──→ consider a cache miss here
BNE R1, R2, Label    ──→ NT
ADD R2, R1, R1       ──→ stall
MUL R3, R3, R4       ──→ WO
DIV R2, R3, R7       ──→ WO
```

Will commando make wrong updates ??

Phantom exceptions:

```
BEQ  R1  R2  Label
DIV  R0  R0  R5
```

consider BEQ predicted as NT & subsequently DIV throwing an exception.

exception result (marked as Ex) in ROB as separate entry.

quiz

```
ADD R2, R2, R1 ——— committed
LW  R1, 0(R2)  ——— executing
ADD R3, R4, R5 ——— Done
DIV R3, R2, R3 ——— executing
ADD R1, R4, R4 ——— Done
ADD R3, R2, R2 ——— Done
```

New state

committed
n
n
unexecuted. (Exception)
n
n

ROB update game

1.	R_1	$R_2 + R_3$
2.	R_3	$R_5 + R_6$
3.	R_1	$ROB_1 + R_7$
4.	R_1	$R_4 + R_8$
5.	R_2	$R_9 + ROB_2$

ROB renaming

RAT

R_1	ROB_4
R_2	ROB_5
R_3	ROB_2
R_4	

latest renames

fp exception handler can also put a new small value & let execution continue.

In original TOMasulo,

Regfile update happened based on RAT.

if a matching FU o/p with same RS Tag is ready to update.

Here, it is not the case, ROB entries get committed in sequence for correct exception handling.

Regfile update :

R_1	$R_2+R_3 \longrightarrow ROB_1 + R_7 \longrightarrow R_4+R_8$
R_2	R_9+ROB_2
R_3	R_5+R_6
R_4	

ROB example (OM page 134)

Unified RS

3 RS	2 RS
ADD /sub	MUL /div

issue in in-order.

if add is free but RS for MUL/div is occupied so add instr cannot be issued if add RS is free

5 RS

ADD, MUL

Logic complicated.

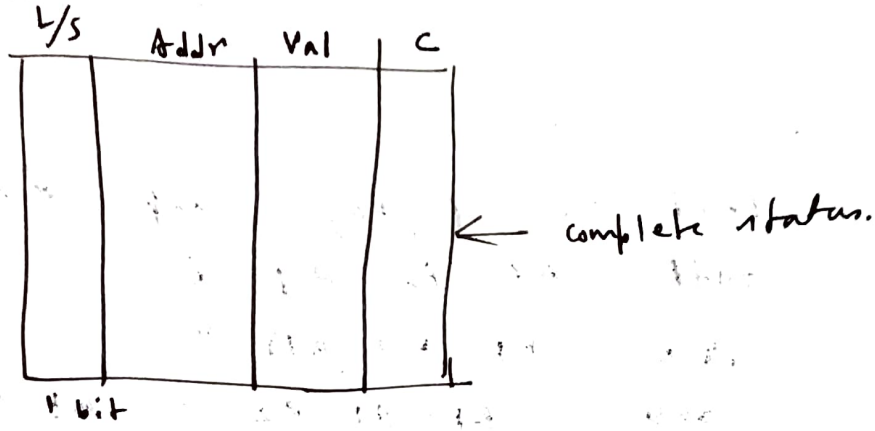
LSQ

Load → ASAP
Store → do not write till commit

LSQ supply value from store to load.

Like ROB.

only store Ld/St.



program

```
LW  R1, 0(R1)
SW  R2, 0(R3)
LW  R4, 0(R4)
SW  R5, 0(R0)
LW  R5, 0(R8)
```

L/S	Addr	Val	C
L	104		
S	204	15	✓
L	204	15	✓
S			
L	174		

→ Mem

check for match here. FW (store to ld fwd)

ambiguous ld addr computed but previous st addr not computed

options

i) Make ld/st always in order execution.

ii) wait only till all prev store add are computed & then action taken. (Waiting due to cache miss)

iii) Let the load go anyway.

3rd option is pretty aggressive.

if store pointer match add then Ld has to recover, it has loaded stale data & supplied to future instrs.

All that has cost but occurs infrequently.

000 LS execution

Load R3 = 0(R6) disp, min

Add R7 = R5 + R9 X

Store R9 → 0(R7) X

✓ Sub R1 = R1 - R2 ✓ R1 o/p

✓ Load R8 = 0(R1) ✓ ← cache hit

done.

all fetched & decoded, dispatched first load & cache miss.

what iff,

~~Mem~~[R1] = ~~Mem~~ R7 after R7 is updated by add.

the Load is to be invalidated.

options dont do o/o/o ld → st → ld

do in order for LS instrs.

Ld to st fwd

L.d → multiple stores going in, in case one matching,

S.t → ~~what to give~~ which ld to go & give value to, can be multiple lds.

Ex

L/S	PC	Address	Value	
L	0XF048	0X3290	42	→ commit
S	4C	3410	25	→ still not committed henke not in cache
S	54	3290	-17	
L	60	3418	1234	
L	F840	3290	-17	← fwd
L	F858	3300	1	
S	F85C	3290	0	
L	F870	3410	25	← not from previous. but latest; cache still not updated
L		3290	0	←
L		3300	1	
				↑ from cache.

Data cache

0X3290	42
3300	1
3410	38
3418	1234

how Ld/st commit ?!

Relation between LSQ, ROB & RS

ISSUE Ld/st
- ROB entry
- LSQ entry ---- kind of RS for Ld/st } cannot issue Ld/st unless both ROB & LSQ has free space.

Issue Non Ld/st
- ROB entry
- RS } cannot issue unless ROB & matching RS is available.

Execute Ld/st → compute addr. i> produce value i:>

for Ld → i> followed by i> as value comes from mem addr

for st → i>, i:> in any order as value comes from register.

Write-Result → only for Ld not st.

st just ~~keep~~ keeps value in LSQ. LSQ will fwd to matching successor loads & also send to memory on commit.

Ld gets results & immediately broadcasts to matching RS so that dependent instruction can now proceed.

Commit Ld → free LSQ & ROB entry.

... st → ..... & also send write to mem.