CS 6354: Tomasulo

21 September 2016

To read more...

This day's paper:

Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units"

Supplementary readings:

Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, section 3.4-5

Shin and Lipatsi, *Modern Processor Design*, section 5.2

Intel Skylake

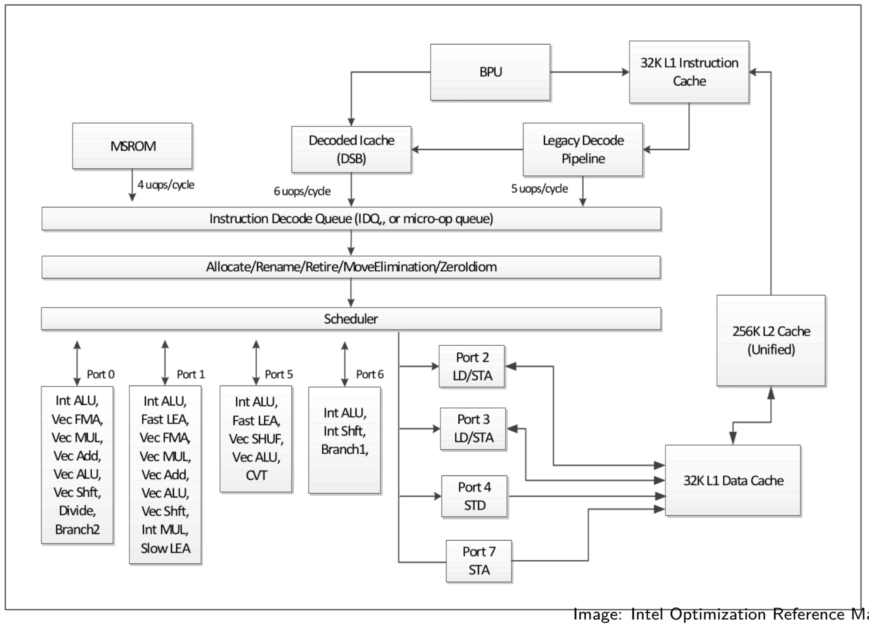

BPU
32K L1 Instruction Cache
MSROM
Decoded Icache (DSB)
Legacy Decode Pipeline
4 uops/cycle
6 uops/cycle
5 uops/cycle
Instruction Decode Queue (IDQ,, or micro-op queue)
Allocate/Rename/Retire/MoveElimination/ZeroIdiom
Scheduler
256K L2 Cache (Unified)
Port 0
Port 1
Port 5
Port 6
Int ALU, Vec FMA, Vec MUL, Vec Add, Vec ALU, Vec Shft, Divide, Branch2
Int ALU, Fast LEA, Vec FMA, Vec MUL, Vec Add, Vec ALU, Vec Shft, Int MUL, Slow LEA
Int ALU, Fast LEA, Vec SHUF, Vec ALU, CVT
Int ALU, Int Shft, Branch1,
Port 2 LD/STA
Port 3 LD/STA
Port 4 STD
Port 7 STA
32K L1 Data Cache

Image: Intel Optimization Reference Manual

Scheduling

How can we reorder instructions?

Without changing the answer

Recall: Data hazards

Instructions had wrong data

... because they weren't executed one-at-a-time

Example: reading old value of register

Recall: Read-after-Write

```
r1 <— r2 + r3
r5 <— r1 — r5
```

	r1 ← r2 + r3		r4 ← r1 - r5	
1	*IF*			
2	*ID*:	read r2, r3	*IF*	
3	*EX*:	temp1 ← r2 + r3	*ID*:	read r1, r5
4	*MEM*		*EX*:	temp2 ← r1 - r5
5	*WB*:	r1 ← temp	*MEM*	
6			*WB*:	r4 ← temp2

Write-after-Write

```
r1 ← r2 + r3 ; (1)
...
r1 ← r6 + r7 ; (2)
r4 ← r2 + r1 ; (3)
```

time	r1 ← r2 + r3	r1 ← r6 + r7	r4 ← r2 + r1
1		read r6, r7	
2	read r2, r3	compute	
3	compute	write r1	
4	write r1		
5			desired value
6	value read		read r1, r2
7			compute

Write-after-Read

```
r1 ← r2 + r3 ; (1)
r3 ← r4 + r5 ; (2)
```

time	r1 ← r2 + r3	r3 ← r4 + r5
1		read r4, r5
2		compute
3		write r3
4	read r2, r3	
5	compute	
6	write r1	

Types of Data Hazards

Read-after-Write (RAW)

also called: true dependence

Write-after-Write (WAW)

also called: output dependence

Write-after-Read (WAR)

also called: anti-dependence

a problem with names

write-after-write

```
r1  ← r2 + r3  ; (1)
r1× ← r6 + r7  ; (2)
r4  ← r2 + r1× ; (3)
```

write-after-read

```
r1  ← r2 + r3  ; (1)
r3× ← r4 + r5  ; (2)
```

no problem if we used a different name each write

register renaming

```
original code    with renaming
r1 ← r2 + r3     new1 ← r2   + r3    ;(1)
r7 ← r1 + r3     new2 ← new1 + r3    ;(2)
r1 ← r6 + r7     new3 ← r6   + r7    ;(3)
r4 ← r2 + r1     new4 ← r2   + new3  ;(4)
r2 ← r4 + r5     new5 ← r4   + r5    ;(5)
```

new name	old name	from	up to
new1	r1	(1)	(2)
new2	r7	(2)	—
new3	r1	(3)	—
new4	r4	(4)	—
new5	r2	(5)	—

scheduling with renaming

different architectual (external) and internal register names

new internal name on each write

register renaming state

original code	with renaming
r1 ← r2 + r3	x09 ← x02 + x03
r7 ← r1 + r3	x10 ← x09 + x03
r1 ← r6 + r7	x11 ← x06 + x10
r4 ← r2 + r1	x12 ← x02 + x11
r2 ← r4 + r5	x13 ← x12 + x05

external name	internal name
r1	~~x01~~ x09 x11
r2	~~x02~~ x13
r3	x03
r4	~~x04~~ x12
r5	x05
r6	x06
r7	~~x07~~ x10
r8	x08

Diversion: SSA

compiler technique: static single-assignment (SSA) form

eewrite code as code with immutable variables only

makes optimization easier

if you know it — this will seem familiar

scheduling with renaming

#	(renamed) instructions	run on	done?
(1)	x05 ← Mem[x03]	Load	yes
(2)	x06 ← x01 + x02	Add1	yes
(3)	x07 ← x01 × x02	Mult	yes
(4)	x08 ← x05 × x04	Mult	yes
(5)	x09 ← x05 + x04	Add2	yes
(6)	x10 ← x07 + x06	Add1	yes

time				

Might have second adder, but x5 is not ready.

int. name	ready?
x01	yes
x02	yes
x03	yes
x04	yes
x05	yes
x06	yes
x08	yes
x09	yes
x10	yes

handling variable times

scheduling is reactive

Load took longer? Doesn't matter.

Don't try to start things until ready.

Running out of register names?

recycle names with no operations, external name

still out of names? don't issue more instructions

reservation stations vs registers

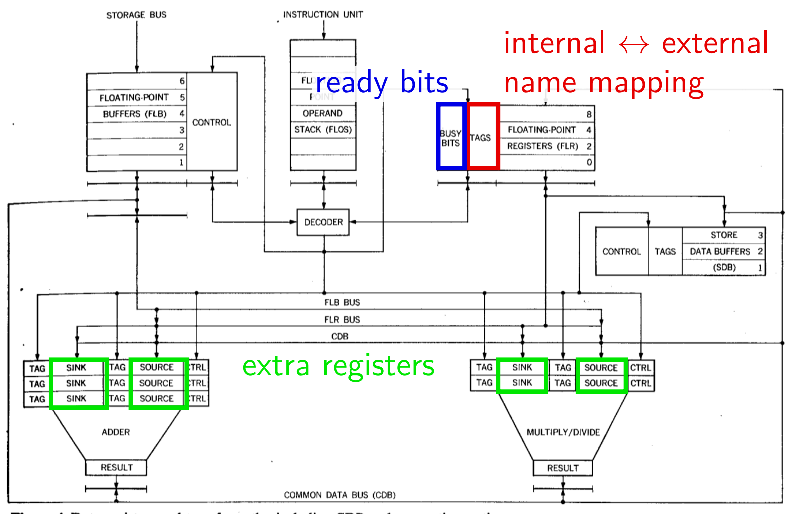
Tomasulo paper doesn't seem to have extra registers

But has reservation stations

… with tags

these are extra registers and their names

pieces in Tomasulo



scheduling with reservation buffers

#	(renamed) instructions	run on	done?
(1)	`x05 ← Mem[x03]`	Load	yes
(2)	`x06 ← x01 + x02`	Add1	yes
(3)	`x07 ← x01 × x02`	Mult	yes
(4)	`x08 ← x05 × x04`	Mult	
(5)	`x09 ← x05 + x04`	Add2	
(6)	`x10 ← x07 + x06`	Add1	yes

dispatching transmits register values

	Add1	Add2	Mult	Load
source 1 tag	x01x07	x05	x01x05	x03
source 1 ready?	yesnoyes	noyes	yesnoyes	yes
source 2 tag	x02x06	x04	x02x04	
source 2 ready?	yesyes	yes	yes	
sink tag	x06x10	x09	x07x08	x05

common data bus

results are broadcast here

tag ≈ internal register name

reservation stations listen for operands

register file listens for register values

keeps register file from being bottleneck

fancy buses: mutliple value+tags per clock cycle

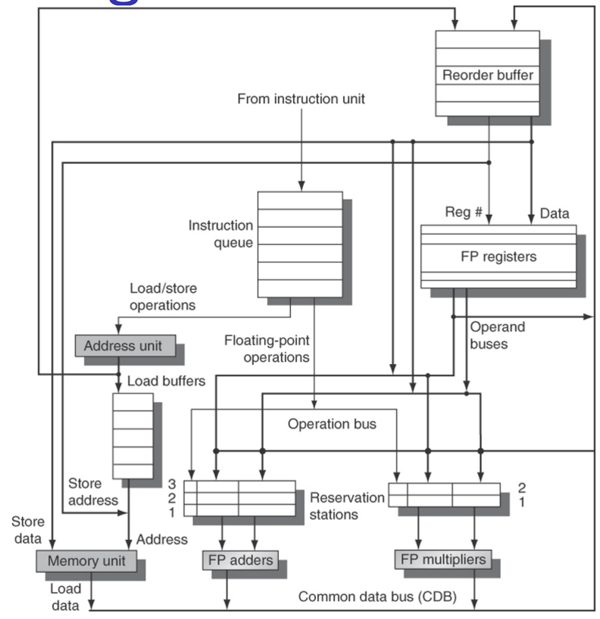
issuing instructions

assign tags for operands

instruction will execute when operands are ready

handles variable length operations (e.g. loads)

integrating with reorder buffer



Hennessy & Patterson Figure 3.11

integrating with reorder buffer (2)

reorder buffer just another thing listening on bus

multiple entries in reservation stations

instead of dispathcing one instruction, issue a list

reservation station starts whichever one gets operands first

variations on reservation stations

Intel P6: shared reservation station for all types of operations

MIPS R10000 (next Monday's paper): read from shared register file (with renaming)

Intel P6 execution unit datapaths

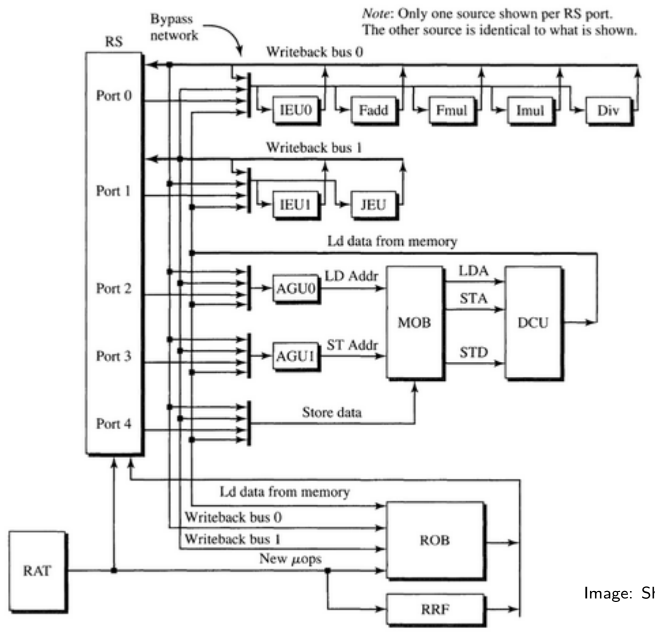


Image: Shen and Lipatsi, Figure 7.14

summary

register renaming to avoid data hazards

- otherwise even write-after-write, write-after-read a problem

shared bus to communicate results

register file, reservation buffers listen on bus

can dispatch to buffer before value ready