Dynamic ILP with O-O-O Execution

Soumyajit Dey, Associate Professor,
CSE, IIT Kharagpur

February 11, 2021

ILP

Overlap execution of instructions -> instruction level parallelism.

- Branch pred removes pipeline problems w.r.t. control hazards
- But data dependency hurts IPC
- Can we have IPC >1 with ILP ?

ILP

Try many instructions executing together

instrs	1	2	3	4	5
r1=r2+r3	F	D			
r4=r1-r5	F	D			
r6=r7 $\oplus$ r8	F	D			
r5=r8 $\times$ r9	F	D			
r4=r8 + r9	F	D			

[1]

- Can have parallel fetch
- Can try parallel decode in same cycle ?

[1] we are not using actual asm here

instrs	1	2	3	4	5
r1=r2+r3	F	D/R	exec stage : r2+r3		WB: r1
r4=r1-r5	F	D/R	exec stage : r1-r5		
$r6=r7 \oplus r8$	F	D			
$r5=r8 \times r9$	F	D			
r4=r8 + r9	F	D			

[2]

- Decode is also a read register stage
- 2nd instr uses old r1
- cannot execute in same cycle

[2] we are not using actual asm here

The execute stage

Can fwd-ing help ?

- fwd values to next instr before writeback
- fwd can help in computing a value and providing for execution in next cycle to next instr
- In this case, both instr are executing in same cycle, cannot happen -> there will be one stall
- one stall, other instrs can go ahead

instrs	1	2	3	4	5
r1=r2+r3	F	D/R	exec stage : r2+r3		WB: r1
r4=r1-r5	F	D/R	exec stage : r1-r5		
$r6=r7 \oplus r8$	F	D			
$r5=r8 \times r9$	F	D			
r4=r8 + r9	F	D			

- Hence an ideal processor with any no of executions units also cannot have ∞ IPC / 0 CPI (Otherwise, I can have any no. of instructions completing in 5 pipeline stages giving 0 CPI)
- has to obey RAW dependency, fwd-ing cannot go back in time

RAW dependency example

$I1 \rightarrow I2$, $I3 \rightarrow I4 \rightarrow I5$: CPI ? (Ignore the the delays in other stages except EX stage)

- I1, I3 can execute in 1st exec cycle
- I2, I4 can execute in 2nd exec cycle
- I5 in 3rd cycle

3 cycles for 5 instrs. CPI = 0.6
$I1 \rightarrow I2 \rightarrow I3 \rightarrow I4 \rightarrow I5$: CPI = 1

WAW deps : considering inf resoures and no delays

r1=r2+r3	$\cdots$	EX	MEM	WB: r1	
r4=r1-r5	$\cdots$		EX	MEM	WB:r4
$r6=r7 \oplus r8$	$\cdots$	EX	MEM	WB: r6	
$r5=r8 \times r9$	$\cdots$	EX	MEM	WB: r5	
r4=r8 + r9	$\cdots$	EX	MEM	WB: r4	

- program semantics not maintained : due to OOO exec
- need to track the dependency and write in correct order

r1=r2+r3	$\cdots$	EX	MEM	WB: r1		
r4=r1-r5	$\cdots$		EX	MEM	WB:r4	
$r6=r7 \oplus r8$	$\cdots$	EX	MEM	WB: r6		
$r5=r8 \times r9$	$\cdots$	EX	MEM	WB: r5		
r4=r8 + r9	$\cdots$	EX	MEM			WB: r4

Removing dependencies

- RAW : true deps –> need to delay instructions to preserve semantics
- WAW, WAR : – false / name deps [nothing related to actual semantics, introduced artificially by constraining number of registers] –> remove by register duplication / renaming

Duplicating register values

I1: r1=r2+r3	···	EX	MEM	WB: r1		
I2: r4=r1-r5	···		EX	MEM	WB:r4	
I3: r3=r4+1	···			EX		
I4: r4=r8 -r9	···	EX	MEM	WB: r5		
I5 : read r4 n work	···					

- I5 receives r4 from I2 as last update - wrong semantics
- solution : duplicate r4 : store multiple values in some memory
- I5 finds among all versions which is latest (w.r.t. instr order)
- I3 also has two possibilities to choose for r4 : since it succeeds both I2, I4 in EX : should choose I2 following instr order

Renaming

separates concept of arch regs (used by prog/compiler)

- Physical regs: all places a cpu can store a value
- Renaming : rewriting prog to use physical regs
- Register Allocation Table (RAT) : mapping of physical reg which has value for arch reg

RAT

i-th location in RAT : location which has physical register name that stores latest value for r-i

```
add r1,r2,r3;
```

//fetch, decode, rewrite instr

```
add p17,p2,p3;
```

//registers to be read are replaced based on RAT
//write location given new physical reg

RAT

entry	RAT
0	p0
1	p17
2	p2
3	p3
4	p4
5	p5
6	p6
7	p7
8	p8

RAT

i-th location in RAT : location which has physical register name that stores latest value for r-i

```
add r1,r2,r3;
sub r4,r1,r5;
```

```
add p17,p2,p3;
sub p18,p17,p5;
```

RAT

entry	RAT
0	p0
1	p17
2	p2
3	p3
4	p18
5	p5
6	p6
7	p7
8	p8

RAT

```
add r1,r2,r3;
sub r4,r1,r5;
```

```
add p17,p2,p3;
sub p18,p17,p5;
```

RAT

entry	RAT
0	p0
1	p17
2	p2
3	p3
4	p18
5	p5
6	p6
7	p7
8	p8

RAT

```
add r1,r2,r3;
sub r4,r1,r5;
xor r6,r7,r8;
```

```
add p17, p2,p3;
sub p18,p17,p5;
xor p19, p7,p8;
```

//update RAT
everytime you write

RAT

entry	RAT
0	p0
1	p17
2	p2
3	p3
4	p18
5	p5
6	p19
7	p7
8	p8

RAT

```
add r1,r2,r3;
sub r4,r1,r5;
xor r6,r7,r8;
mul r5,r8,r9;
```

```
add p17, p2,p3;
sub p18,p17,p5;
xor p19, p7,p8;
xor p20, p8,p9;
```

//update RAT everytime you write

RAT

entry	RAT
0	p0
1	p17
2	p2
3	p3
4	p18
5	p20
6	p19
7	p7
8	p8

RAT

```
add r1,r2,r3;
sub r4,r1,r5;
xor r6,r7,r8;
mul r5,r8,r9;
add r4,r8,r9;
```

```
add p17, p2,p3;
sub p18,p17,p5;
xor p19, p7,p8;
xor p20, p8,p9;
add p21, p8,p9;
```

//any read of r4
after I2 is from p18
//any read of r4
after I5 is from p21

RAT

entry	RAT
0	p0
1	p17
2	p2
3	p3
4	p21
5	p20
6	p19
7	p7
8	p8

False deps

	Fetched	Renamed
I1	mul r2,r2,r2	mul p7,p2, p2
I2	add r1,r1,r2	add p8,p1, p7
I3	mul r2,r4,r4	mul p9,p4, p4
I4	add r3,r3,r2	add p10,p3, p9
I5	mul r2,r6,r6	mul p11,p6, p6
I6	add r5,r5,r2	add p12,p5,p11

true dep (RAW) in both : I1->I2 : r2 ; I3->I4 : r2 ; I5->I6 : r2 ; output dep (WAW) in fetched: I1-> I3 -> I5 : r2 anti-dependence (WAR) in fetched: I2 -> I3, I4 -> I5 : r2

Renamed program does not have any false (WAW, WAR) deps. CPI = 2/6 unlike original CPI = 1

What ILP is actually

ILP =IPC when

- proc execs an instruction in 1 cycle, and can execute any no of instr in 1 cycle : the only constraint is causality (in form of RAW)
- ILP is more linked to program; assumes ideal proc, not property of real proc
- program with control deps : ILP calculation assumes branches are predicted and instructions at predicted branch address start immediately if there is no dependency (they do not even wait till the branch instr to execute)
- IPC of a program is w.r.t. a given proc

ILP $\neq$ IPC

I1	add r1,r2,r3
I2	sub r4,r1,r5
I3	xor r6,r7,r8
I4	mul r5,r8,r9
I5	add r4,r8,r9

Processor spec:

- 2-issue
- O-O-O
- 2 MUL,
- 2 ADD/SUB/XOR

ILP :only one RAW , ILP = 5/2
IPC:

- cycle 1 : I1, I3, I4
- cycle 2 : I2, I5
- IPC = 5/2
- same!!

ILP $\neq$ IPC

I1	add r1,r2,r3
I2	sub r4,r1,r5
I3	xor r6,r7,r8
I4	mul r5,r8,r9
I5	add r4,r8,r9

Processor spec:

- 2-issue
- O-O-O
- 2 MUL,
- 1 ADD/SUB/XOR

ILP :only one RAW , ILP $= 5/2$
IPC:

- cycle 1 : I1, I4
- cycle 2 : I2
- cycle 3 : I3
- cycle 4 : I5
- IPC $= 5/4$
- ILP $\geq$ IPC in real proc

Improving IPC

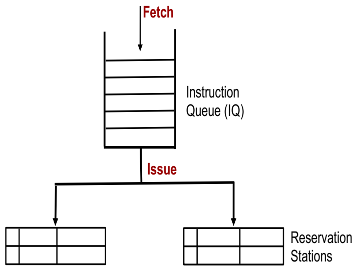
- control dep -> BP
- WAR / WAW -> renaming
- RAW -> OOO
- structural deps -> wider issue HW design, add more resources

Tomasulo′s Algorithm

- IBM 360
- Used for floating point O-O-O based on which instruction is ready
- Register renaming

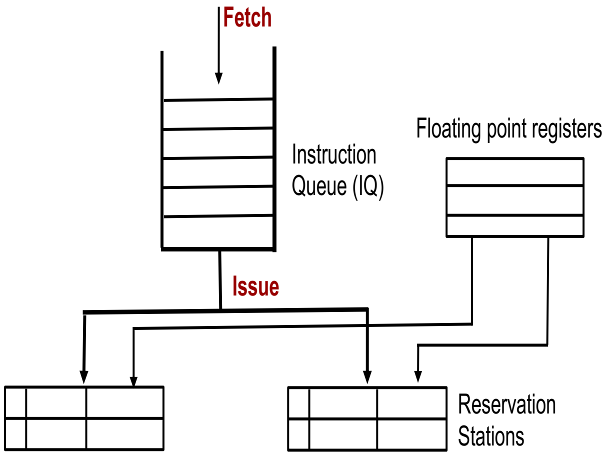
Modern processors do this 1) for all instrs, 2) add exception handling support

Tomasulo's Algorithm



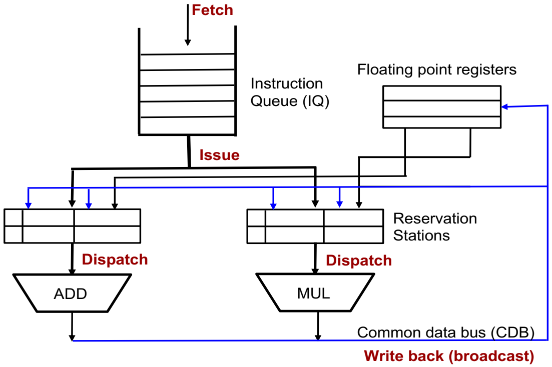
- Fetched instructions are lined up in-order in IQ
- moved to Reservation stations (RS)
- Instrs seat in RS awaiting parameters to be ready

Tomasulo's Algorithm



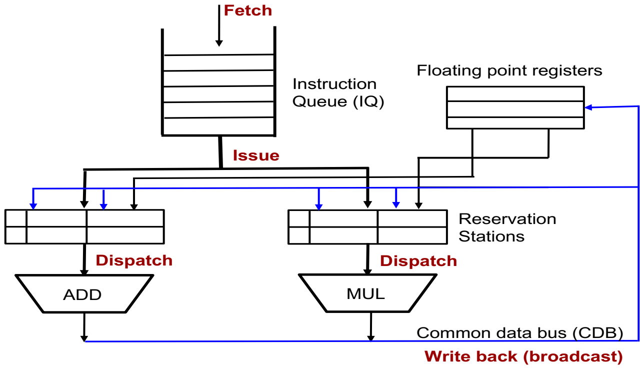
- Available operands get inserted to RS from register file

Tomasulo's Algorithm



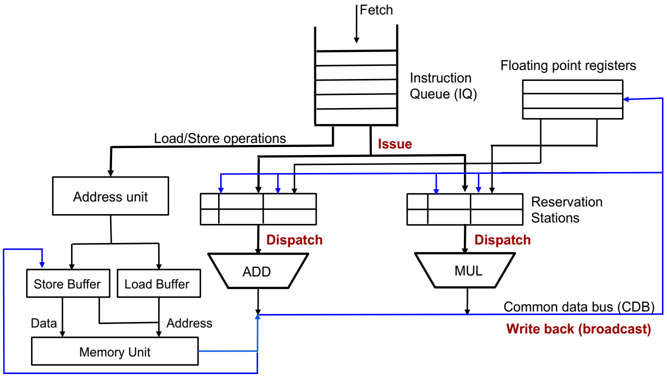
- Each execution unit has its own RS
- Exec unit o/p-s are broadcast in a bus

Tomasulo's Algorithm



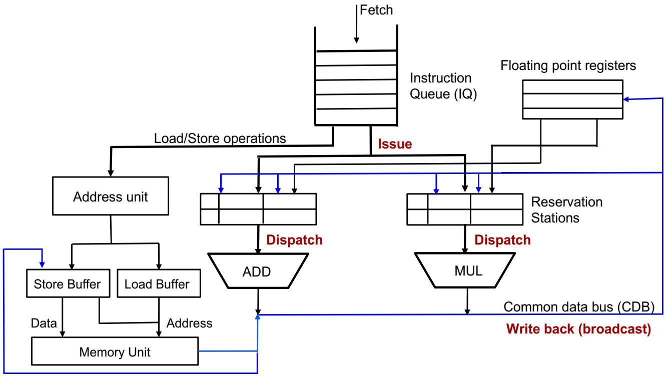
- Exec unit o/p-s are broadcast in a bus to update - reg file, RS stations with dependent input operands
- Note that CDB has two lines per RS, (same register can be both operands!)

Tomasulo's Algorithm



- ld/st instr goes to address generation unit instead of RS
- next queued to ld/st buffer
- ld buffer provides address, st buffer provides data+adress to memory system

Tomasulo's Algorithm



- ld instr result from memory is updated in reg file by CDB
- CDB also broadcasts to store buffer any update in a matching register
- Original scheme does not reorder ld/st sequences

Tomasulo's Algorithm

- Sending instruction from IQ to RS/ld-st units : ISSUE
- Sending instr from RS to exec unit : DISPATCH
- WRITE RESULT / BROADCAST

ISSUE

- ISSUE : in-order
- Determine operand status
- get free RS of matching type (ADD/MUL)
- put instr in RS
- tag destination register

ISSUE Example

IQ

4	f1=f2+f3
3	f4=f1-f2
2	f1=f2/f3
1	f2=f4+f1

RS1	+	0.7	3.14
RS2			
RS3			

ADD

f1	3.14
f2	-1.0
f3	2.7
f4	0.7

Register file

RAT

f1	
f2	RS1
f3	
f4	

RS4			
RS5			

MUL

check if i/p ready

ISSUE Example

IQ

4	f1=f2+f3
3	f4=f1-f2
2	f1=f2/f3
1	f2=f4+f1

RAT

f1	RS4
f2	RS1
f3	
f4	

check if i/p ready

RS1	+	0.7	3.14
RS2			
RS3			

ADD

RS4	/	RS1	2.7
RS5			

MUL

f1	3.14
f2	-1.0
f3	2.7
f4	0.7

Register file

ISSUE Example

IQ

4	f1=f2+f3
3	f4=f1-f2
2	f1=f2/f3
1	f2=f4+f1

RS1	+	0.7	3.14
RS2	-	RS4	RS1
RS3			

ADD

RAT

f1	RS4
f2	RS1
f3	
f4	RS2

RS4	/	RS1	2.7
RS5			

MUL

w/o a free RS, no instr issue

f1	3.14
f2	-1.0
f3	2.7
f4	0.7

Register file

check if i/p ready

ISSUE Example

IQ

4	f1=f2+f3
3	f4=f1-f2
2	f1=f2/f3
1	f2=f4+f1

RS1	+	0.7	3.14
RS2	-	RS4	RS1
RS3	+	RS1	2.7

ADD

f1	~~RS4~~ RS3
f2	RS1
f3	
f4	RS2

update RS by renaming

RS4	/	RS1	2.7
RS5			

MUL

f1	3.14
f2	-1.0
f3	2.7
f4	0.7

Register file

DISPATCH

- When an RS's execution o/p is broad cast, all RS operands waiting for it gets updated
- Say RS1 o/p 0.3 is in CDB

RS1	+		
RS2	-	RS4	RS1
RS3	+	RS1	2.7

ADD

RS4	/	RS1	2.7
RS5			

MUL

DISPATCH

- When an RS's execution o/p is broad cast, all RS operands waiting for it gets updated
- Say RS1 o/p 0.3 is in CDB

two instr dispatched from RS3, RS4

RS1			
RS2	-	RS4	0.3
RS3	+	0.3	2.7

ADD

RS4	/	0.3	2.7
RS5			

MUL

DISPATCH : more than one instr ready

- When an RS's execution o/p is broad cast, all RS operands waiting for it gets updated
- Say RS1 o/p 0.3 is in CDB

RS1	+		
RS2	-	1.23	RS1
RS3	+	RS1	2.7

ADD

RS4	/	RS1	2.7
RS5			

MUL

DISPATCH : more than one instr ready

- Say RS1 o/p 0.3 is in CDB
- Dispatch RS2/ RS3 ??
- OLDEST first? –simple. typically used
- More deps first ? –lot of search

RS1			
RS2	-	1.23	0.3
RS3	+	0.3	2.7

ADD

RS4	/	0.3	2.7
RS5			

MUL

Broadcast / write result

f1	RS2
f2	RS1
f3	
f4	RS3

- Put tag (RS) & result in CDB
- Write reg file
- update RAT
- Free RS
- update other RS entries for dispatch (already discussed)

RS1	+	0.7	-1.0
RS2	-	RS4	RS1
RS3	+	RS1	2.7

ADD o/p RS1 : -0.3

RS4	/	RS1	2.7
RS5			

MUL

f1	3.14
f2	-1.0
f3	2.7
f4	0.7

Register file

Broadcast / write result

f1	RS2
f2	~~RS1~~
f3	
f4	RS3

RS1			
RS2	-	RS4	RS1
RS3	+	RS1	2.7

ADD o/p RS1 : -0.3

RS4	/	RS1	2.7
RS5			

MUL

f1	3.14
f2	~~-1.0~~ -0.3
f3	2.7
f4	0.7

Register file

- If multiple broadcasts ready : give priority to slower exec units

Broadcasting a stale result

- A result in CDB is stale if the RAT entry is overwritten by some other RS, how ?
- After this instr with write to a register (f1 say) got queued in some RS (RS1), let us say another instruction got queued in RS2 with write to f1
- Then RAT entry for f1 is RS2 then and no more RS1
- The broadcast from RS1 need not update RAT and reg file
- The broadcast from RS1 will just update other dependent RS fields with matching tags

Load Store instrs

Dependencies through memory

- RAW : sw to an address followed by lw from same address
- WAR : lw then sw
- WAW : sw, sw to same address

Solution ?

- lw / sw in-order –tomasulo
- identify deps and re-order - modern μp

ROB

Reorder instrs to handle exceptions

instr	IS	DIS	WR
div f10, f0, f6	1	2	42
l.d f2, 45(r3)	2	3	13
mul f0, f2, f4	3	14	19
sub f8, f2,f6	4	14	15

assume: div takes 40 cycles
div by zero detected in cycle 40
by this time o/p of instr 2,3,4 all written

- after exception handling instr 1 will re-execute with wrong arguments
- say in exception handling f6 is given a small value $\neq 0$
- same issue, if some l.d has a page-fault

–imprecise handling

Branch misprediction due to OOO

div r1, r3, r4
beq r1, r2, label
add r3, r4, r5
sub f8, f2,f6
⋮
div

assume: beq mispredicts
by this time 2nd DIV executes and generates exception-> *phantom exception*
By the time branch result is known unnecessary exception handling may be going on

assume: beq mispredicts
by this time o/p of ADD written in r3
r3 changed wrongly

Solution ?

- execute, broadcast O-O-O
- deposit values in regs in-order

ROB

	REG	VAL	DONE
1 2			
3			
4			
⋮	⋮	⋮	⋮

VAL : store instr o/p for target REG
DONE : bit to say validity : fwd to dependent instr
Two pointers:–>
COMMIT : from where to commit
ISSUE: from where to write in ROB

DIY: w/o ROB

LD R1, 0(R1)
BNE R1, R2, label
ADD R2, R1, R1
MUL R3, R3, R4
–LD dependent branch

ROB usage

IQ

⋮
r1=r2+r3

RS1	+		
RS2			
RS3			

ADD

r1			
r2			
r3			

Register file

		REG	VAL	DONE
commit	1 2			
	3			
	4			
⋮	⋮	⋮	⋮	
issue	n	r1		0

- Put instr from IQ to RS and update ROB
- update issue pointer in ROB
- RAT for r1 point to ROB entry of instr

r1	ROB:n
r2	
r3	

RAT

ROB usage

RS1	+		
RS2			
RS3			

IQ

⋮
r1=r2+r3

ADD
o/p: <ROB:n> result

Can empty RS after DISPATCH; earlier RS was kept occupied until broadcast
RS served as name tag
This is now ROB entry
All RS tags are ROB entries

		REG	VAL	DONE
commit	1			
	2			
	3			
	4			
⋮	⋮	⋮	⋮	
issue	n	r1		0

r1	
r2	
r3	

Register file

r1	ROB:n
r2	
r3	

RAT

ROB usage

IQ

⋮
r1=r2+r3

RS1			
RS2			
RS3			

ADD

BROADCAST:
All RS tags are ROB entries
RS tags updated with result
W/O ROB : result went to REG, updated RAT
With ROB : result goes to ROB

		REG	VAL	DONE
commit	1 2			
	3			
	4			
⋮	⋮	⋮	⋮	
issue	n	r1	result	1

r1	
r2	
r3	

Register file

r1	ROB:n
r2	
r3	

RAT

ROB usage

IQ

⋮
r1=r2+r3

RS1			
RS2			
RS3			

ADD

COMMIT:
write result from ROB to REG
update RAT entry
update commit point

r1	result		
r2			
r3			

Register file

		REG	VAL	DONE
	1 2			
	3			
	4			
⋮	⋮	⋮	⋮	
issue, commit	n	r1	result	1

r1	r1
r2	
r3	

RAT

Branch misprediction handling

l.d r1 $\phi(r1)$
bne r1 r2 label
add r2 r1 r1
mul r3 r3 r4
div r2 r3 r7
mispredicted instrs
marked in red

r1	
r2	
r3	

Register file

r1	
r2	
r3	

RAT

		REG	VAL	DONE
i/c–>	1			
	2			
	3			
	4			
⋮	⋮	⋮	⋮	
	n			

ROB

Branch misprediction handling

l.d r1 $\phi(r1)$
bne r1 r2 label
add r2 r1 r1
mul r3 r3 r4
div r2 r3 r7
mispredicted instrs
marked in red

r1	
r2	
r3	

Register file

r1	ROB1
r2	ROB3
r3	ROB4

RAT

		REG	VAL	DONE
c–>	1	r1		
	2	ϕ		
	3	r2		
i–>	4	r3		
⋮	⋮	⋮	⋮	
	n			

ROB

Branch misprediction handling

The updates go to ROB instead of REG

Hence no damage

l.d r1 $\phi(r1)$
bne r1 r2 label
add r2 r1 r1
mul r3 r3 r4
div r2 r3 r7

r1	
r2	
r3	

Register file

r1	ROB1
r2	ROB5
r3	ROB4

RAT

		REG	VAL	DONE
c–>	1	r1		
	2	ϕ		
	3	r2		
	4	r3	15	1
i–>	5	r2	2	1
⋮	⋮	⋮	⋮	
	n			

ROB

Branch misprediction handling

l.d r1 $\phi(r1)$
bne r1 r2 label
add r2 r1 r1
mul r3 r3 r4
div r2 r3 r7
mispredicted instrs marked in red
Assume : l.d has a cache miss
bne, add delayed
mul : o/p = 15
div o/p = 2

r1	700
r2	
r3	

Register file

r1	~~ROB1~~
r2	ROB5
r3	ROB4

RAT

l.d returns
commit, write REG
bne still pending
add returns

		REG	VAL	DONE
c–>	1	r1	700	
	2	ϕ		
	3	r2	3	1
	4	r3	15	1
i–>	5	r2	2	1
⋮	⋮	⋮	⋮	
	n			

ROB

Branch misprediction handling

bne returns and come to commit
roll back issue
ROB invalidated
undo RAT entries
RS, ALU -> empty

l.d r1 $\phi(r1)$
bne r1 r2 label
add r2 r1 r1
mul r3 r3 r4
div r2 r3 r7
mispredicted instrs marked in red

r1	700
r2	
r3	

Register file

r1	~~ROB1~~
r2	~~ROB5~~
r3	~~ROB4~~

RAT

		REG	VAL	DONE
	1	r1	700	
i, c–>	2	ϕ		
	3	r2	3	1
	4	r3	15	1
	5	r2	2	1
⋮	⋮	⋮	⋮	
	n			

Thank You[3]

[3]Most of the material are taken from the famous book on Comp Arch by Hen/Pat, Comp Arch course by Milos Prvulovic for teaching purposes