Data and Control Hazards

CS 3410, Spring 2014

Computer Science
Cornell University

See P&H Chapter: 4.6-4.8

Announcements

- Prelim next week
  - Tuesday at 7:30
  - Upson B17 [a-e]*, Olin 255[f-m]*, Philips 101 [n-z]*
  - Go based on netid
- Prelim reviews
  - Friday and Sunday evening. 7:30 again.
  - Location: TBA on piazza
- Prelim conflicts
  - Contact KB , Prof. Weatherspoon, Andrew Hirsch
- Survey
  - Constructive feedback is very welcome

Administrivia

Prelim1:

- Time: We will start at 7:30pm sharp, so come early
- Loc: Upson B17 [a-e]*, Olin 255[f-m]*, Philips 101 [n-z]*
- Closed Book
  - Cannot use electronic device or outside material
- Practice prelims are online in CMS
- Material covered everything up to end of this week
  - Everything up to and including data hazards
  - Appendix B (logic, gates, FSMs, memory, ALUs)
  - Chapter 4 (pipelined [and non] MIPS processor with hazards)
  - Chapters 2 (Numbers / Arithmetic, simple MIPS instructions)
  - Chapter 1 (Performance)
  - HW1, Lab0, Lab1, Lab2

Hazards

3 kinds

- Structural hazards
  - Multiple instructions want to use same unit

- Data hazards
  - Results of instruction needed before

- Control hazards
  - Don’t know which side of branch to take

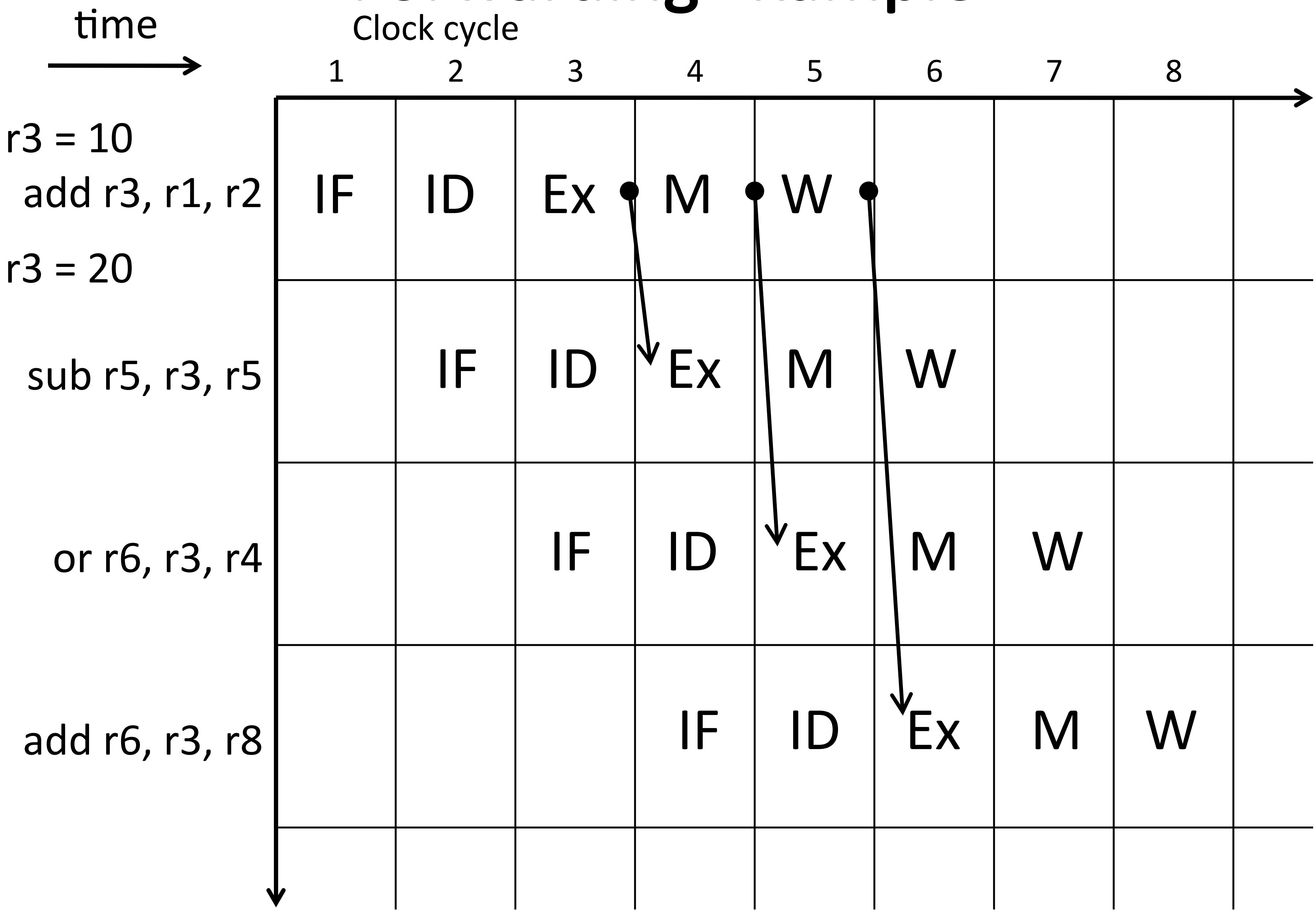
How to handle data hazards

- What to do if data hazard detected?
- Options
  - Nothing
    - Change the ISA to match implementation
  - Stall
    - Pause current and subsequent instructions till safe
      - Slow down the pipeline (add bubbles to pipeline)
  - Forward/bypass
    - Forward data value to where it is needed

Forwarding

Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register)

Forwarding Example



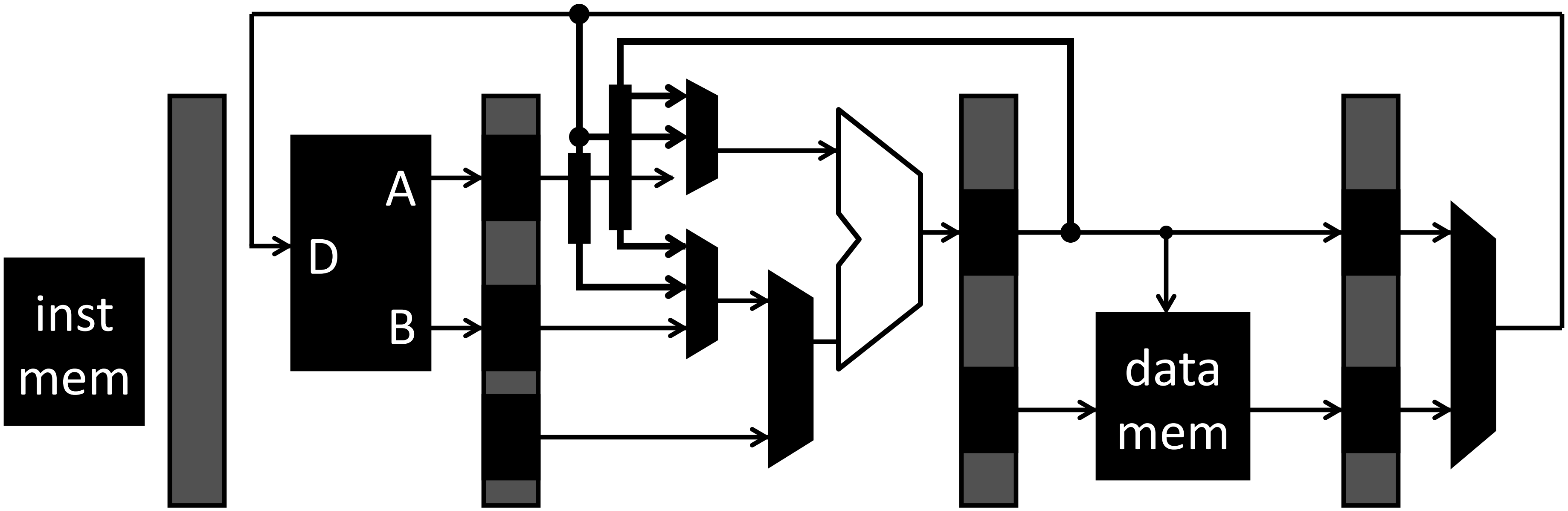
Forwarding

Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register)

Three types of forwarding/bypass

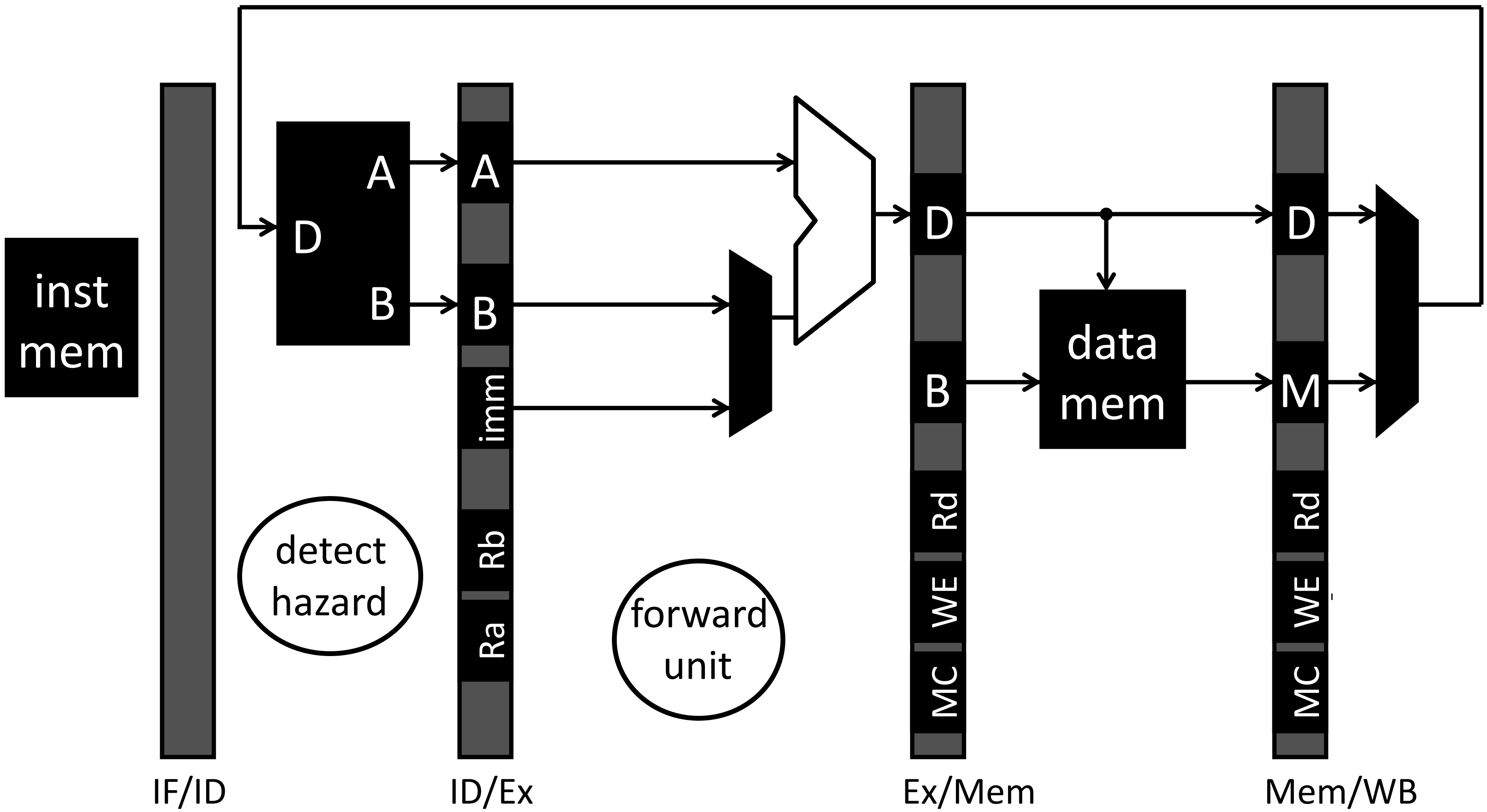
- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W→Ex)
- RegisterFile Bypass

Forwarding Datapath 1



add r3, r1, r2	IF	ID	Ex	M	W			
sub r5, r3, r1		IF	ID	Ex	M	W		

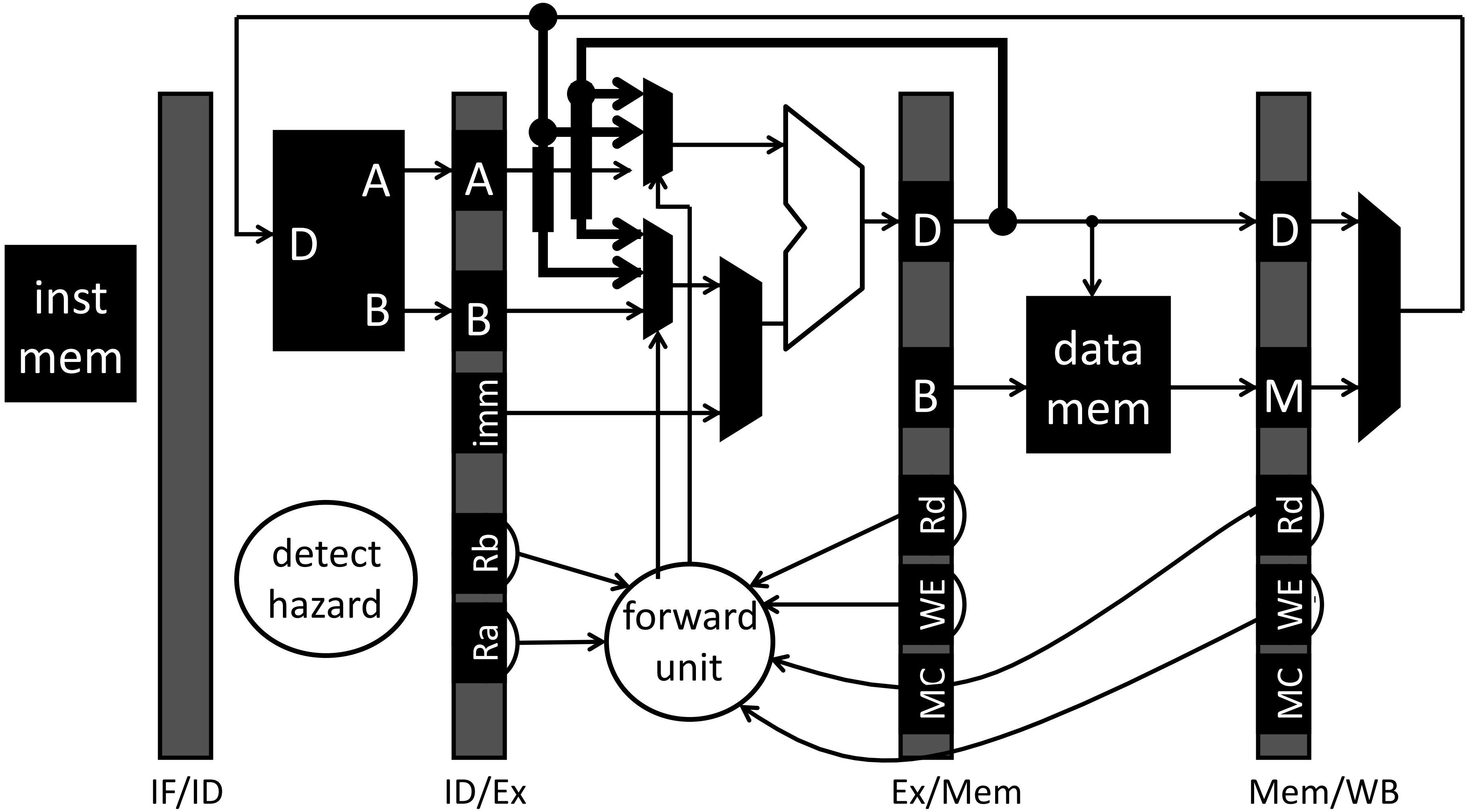
Forwarding Datapath



Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass

Forwarding Datapath



Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass

Forwarding Datapath 1

Ex/MEM to EX Bypass

- EX needs ALU result that is still in MEM stage
- Resolve:

  Add a bypass from EX/MEM.D to start of EX

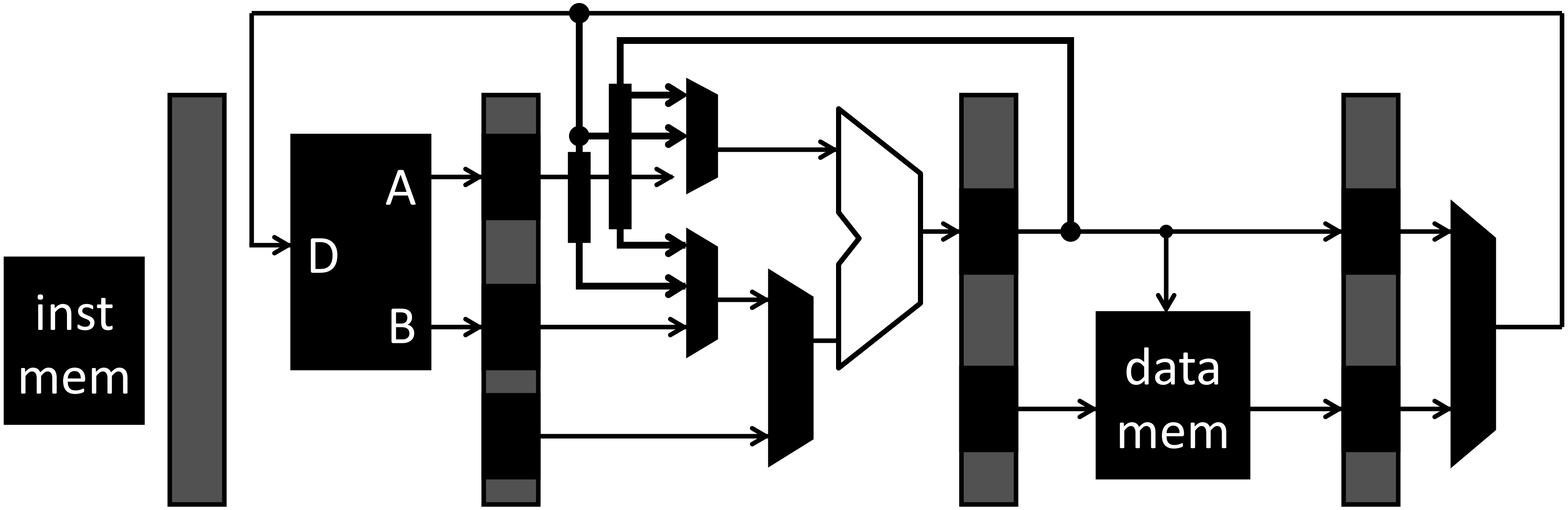
How to detect? Logic in Ex Stage:

forward = (Ex/M.WE && EX/M.Rd != 0 &&

ID/Ex.Ra == Ex/M.Rd)

|| (same for Rb)

Forwarding Datapath 2



add r3, r1, r2	IF	ID	Ex	M	W			
sub r5, r3, r1		IF	ID	Ex	M	W		
or r6, r3, r4			IF	ID	Ex	M	W	

Forwarding Datapath 2

Mem/WB to EX Bypass

- EX needs value being written by WB
- Resolve:

  Add bypass from WB final value to start of EX

How to detect? Logic in Ex Stage:

forward = (M/WB.WE && M/WB.Rd != 0 &&
ID/Ex.Ra == M/WB.Rd &&
|| (same for Rb)

Is this it?

Not quite!

Forwarding Datapath 2

add r3, r1, r2	add r3, r1, r2
sub r5, r3, r5	sub r3, r3, r5
or r6, r3, r4	or r6, r3, r4
add r6, r3, r8	add r6, r3, r8

How to detect? Logic in Ex Stage:

M/WB (WE on, Rd != 0) and (M/WB.Rd == ID/Ex.Ra)

also NOT(Ex/M.Rd == ID/Ex.Ra) and (WE, Rd!= 0))

Rb same as Ra

Register File Bypass

Register File Bypass

- Reading a value that is currently being written

Detect:

((Ra == MEM/WB.Rd) or (Rb == MEM/WB.Rd))
and (WB is writing a register)

Resolve:

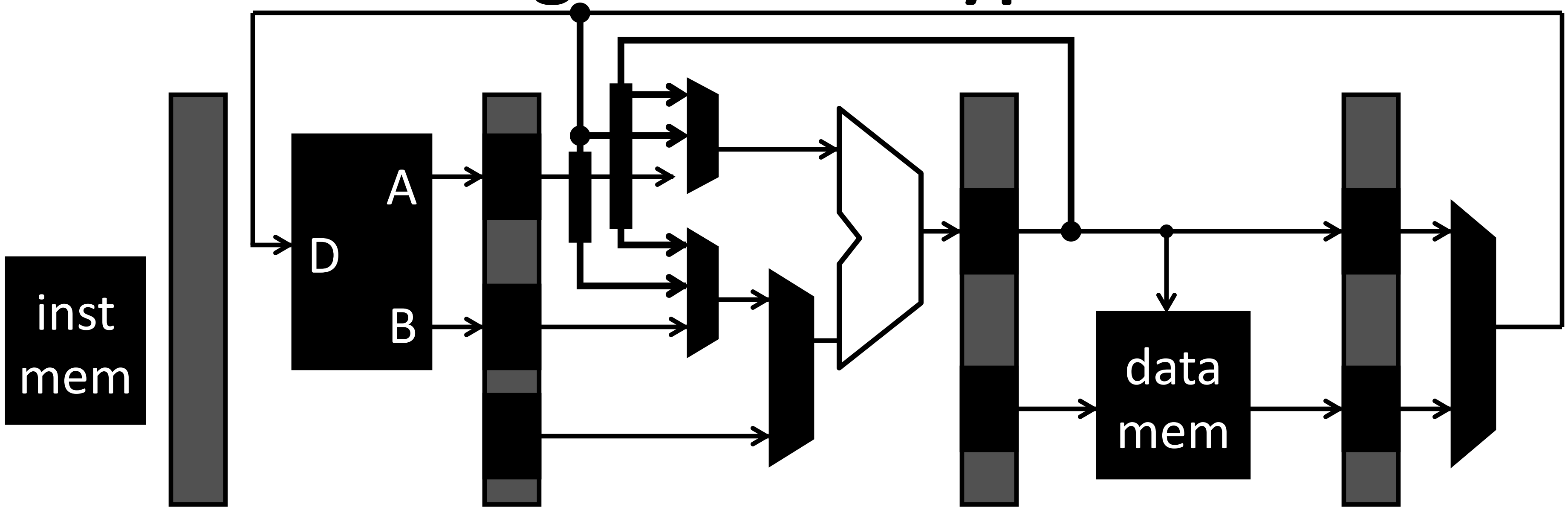
Add a bypass around register file (WB to ID)

Better: just negate register file clock

– writes happen at end of first half of each clock cycle

– reads happen during second half of each clock cycle

Register File Bypass



add r3, r1, r2	IF	ID	Ex	M	W			
sub r5, r3, r1		IF	ID	Ex	M	W		
or r6, r3, r4			IF	ID	Ex	M	W	
add r6, r3, r8				IF	ID	Ex	M	W

Are we done yet?

```
add r3, r1, r2
 lw r4, 20(r8)
  or r6, r3, r4
add r6, r3, r8
```

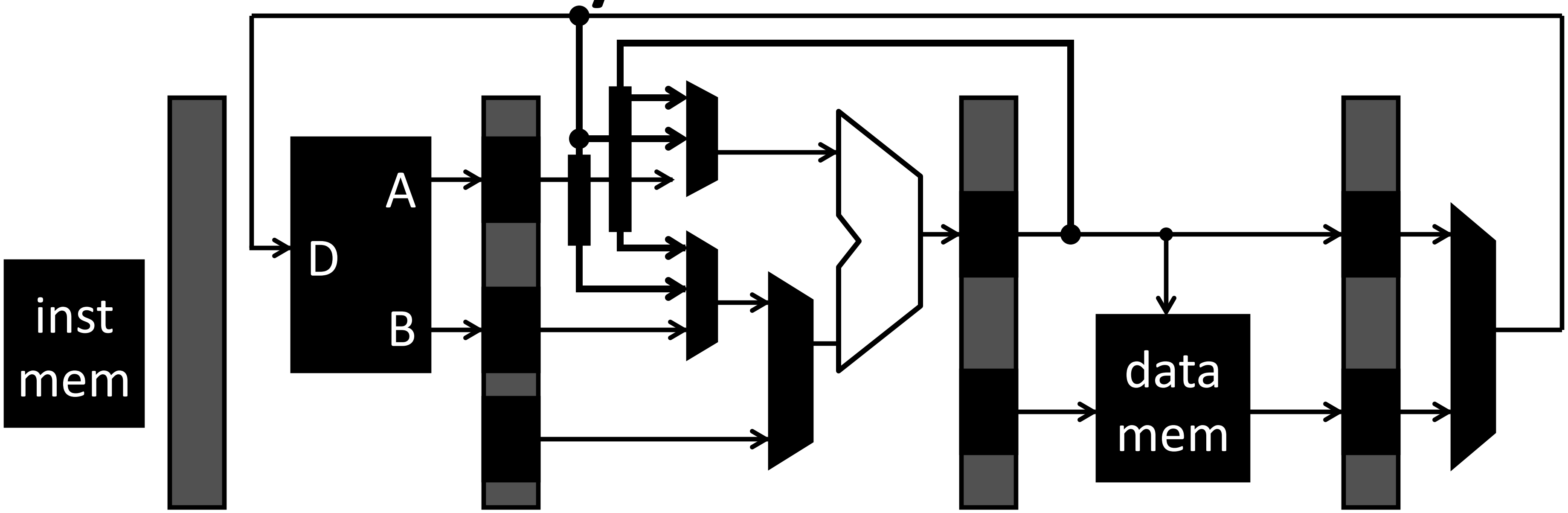
Memory Load Data Hazard

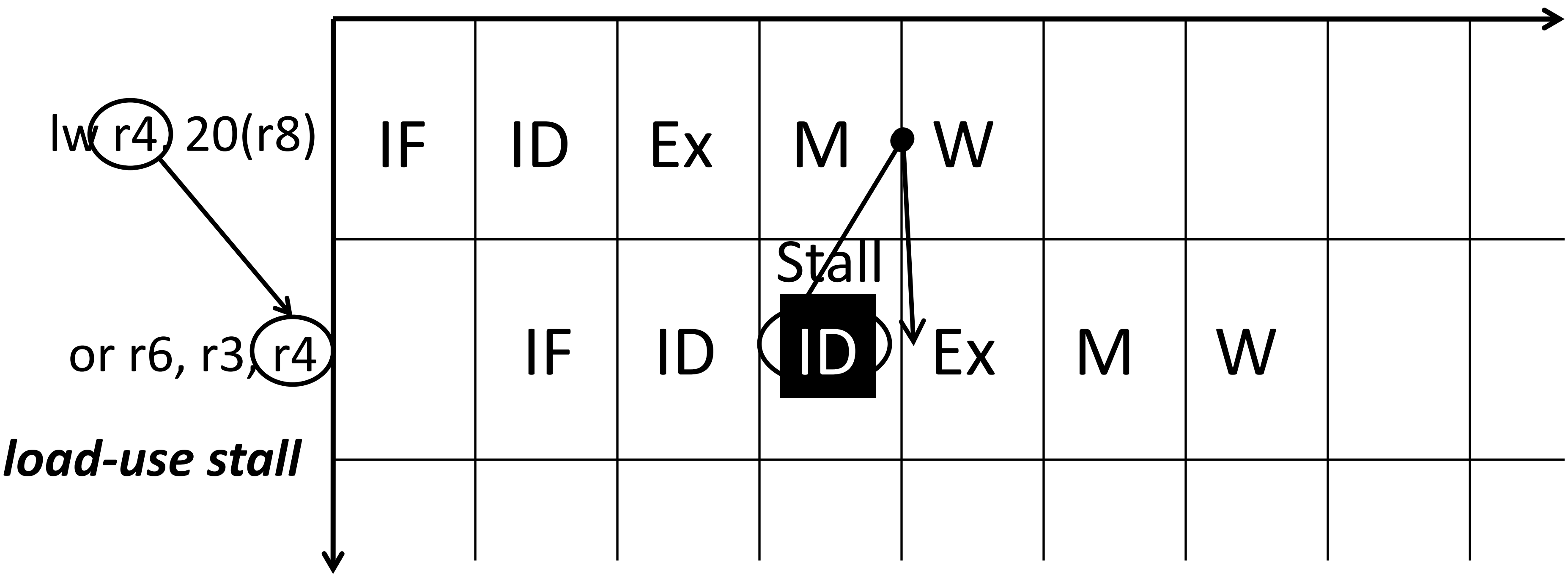
What happens if data dependency after a load word instruction?

Memory Load Data Hazard

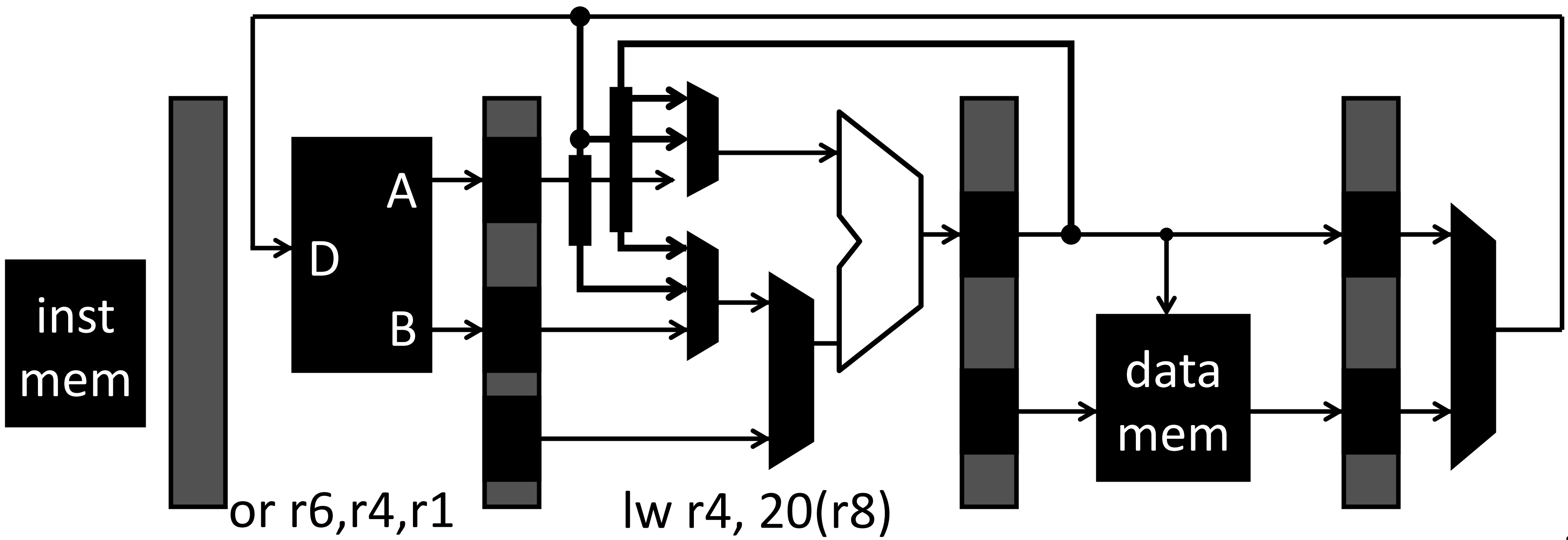
- Value not available until after the M stage
- So: next instruction can't proceed if hazard detected

Memory Load Data Hazard



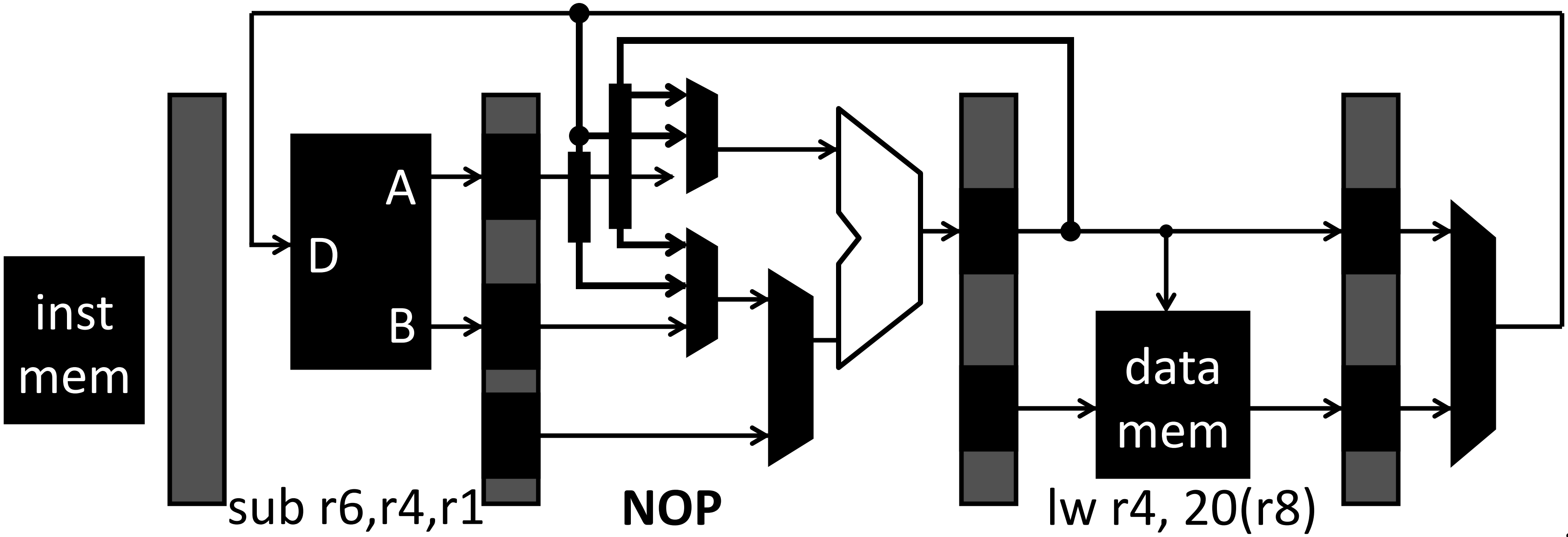


Memory Load Data Hazard



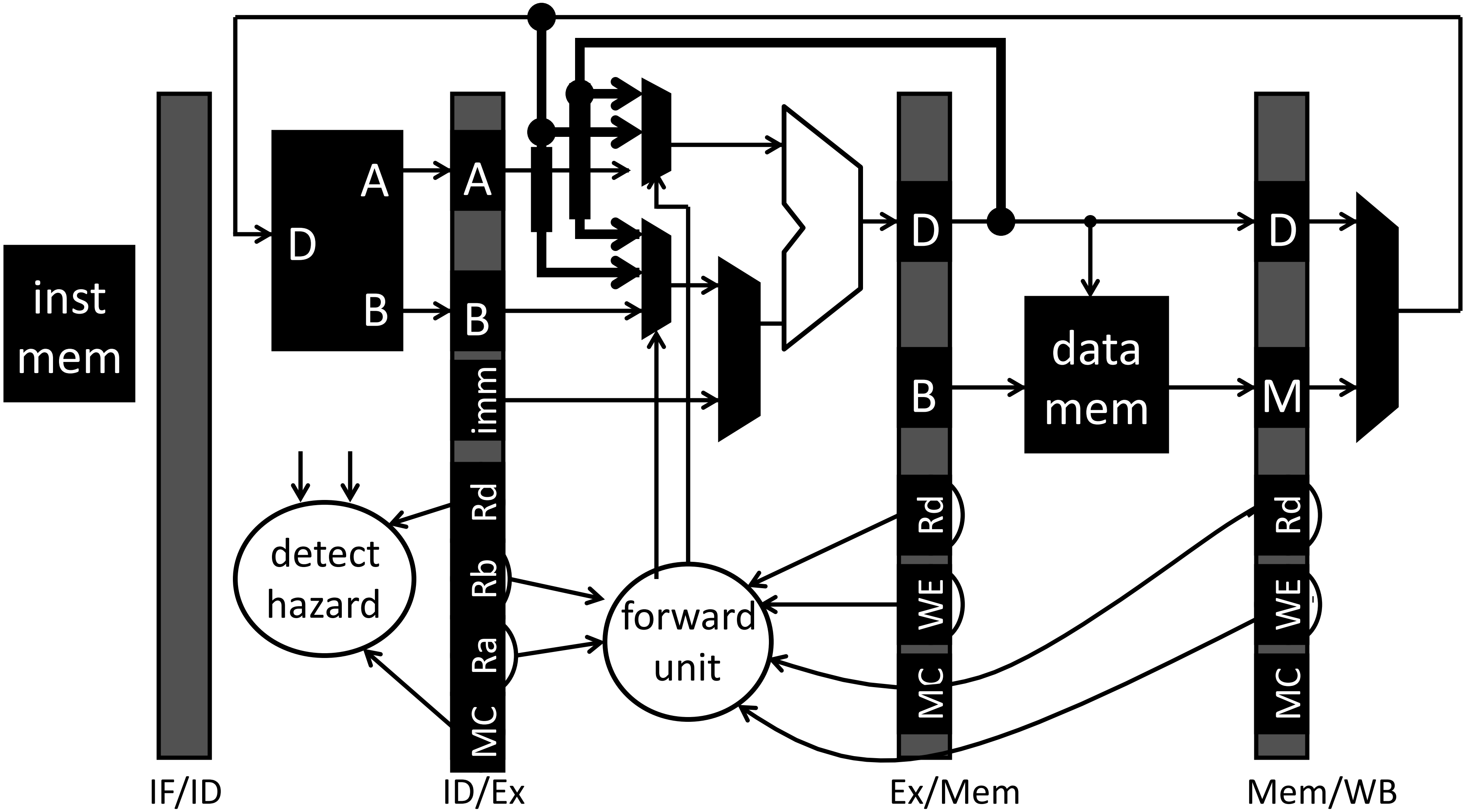
lw r4, 20(r8)	IF	ID	Ex	M	W			
or r6, r3, r4		IF	ID	Stall ID	Ex	M	W	
load-use stall								

Memory Load Data Hazard



lw r4, 20(r8)	IF	ID	Ex	M	W		
or r6, r3, r4		IF	ID	Stall ID	Ex	M	W
load-use stall							

Memory Load Data Hazard



Stall =
If(ID/Ex.MemRead &&
IF/ID.Ra == ID/Ex.Rd

Memory Load Data Hazard

Load Data Hazard

- Value not available until WB stage
- So: next instruction can't proceed if hazard detected

Resolution:

- MIPS 2000/3000: one delay slot
  - ISA says results of loads are not available until one cycle later
  - Assembler inserts nop, or reorders to fill delay slot
- MIPS 4000 onwards: stall
  - But really, programmer/compiler reorders to avoid stalling in the load delay slot

For stall, how to detect? Logic in ID Stage

- Stall = ID/Ex.MemRead &&

  (IF/ID.Ra == ID/Ex.Rd || IF/ID.Rb == ID/Ex.Rd)

Quiz

```
add  r3, r1, r2
nand r5, r3, r4
add  r2, r6, r3
lw  r6, 24(r3)
sw  r6, 12(r2)
```

Quiz

```
add  r3, r1, r2
nand r5, r3, r4
add  r2, r6, r3
lw  r6, 24(r3)
sw  r6, 12(r2)
```

5 Hazards

Quiz

`add r3, r1, r2`	
`nand r5, r3, r4`	Forwarding from Ex/M→ID/Ex (M→Ex)
`add r2, r6, r3`	Forwarding from M/W→ID/Ex (W→Ex)
`lw r6, 24(r3)`	RegisterFile (RF) Bypass
`sw r6, 12(r2)`	Forwarding from M/W→ID/Ex (W→Ex)

Stall

\+ Forwarding from M/W→ID/Ex (W→Ex)

5 Hazards

Data Hazard Recap

Delay Slot(s)

- Modify ISA to match implementation

Stall

- Pause current and all subsequent instructions

Forward/Bypass

- Try to steal correct value from elsewhere in pipeline
- Otherwise, fall back to stalling or require a delay slot

Why are we learning about this?

Logic and gates

Numbers & arithmetic

States & FSMs

Memory

A simple CPU

Performance

Pipelining

Hazards: Data and Control

Control Hazards

What about branches?

A control hazard occurs if there is a control instruction (e.g. BEQ) and the program counter (PC) following the control instruction is not known until the control instruction computes if the branch should be taken

e.g.

```
0x10:       beq r1, r2, L
0x14:       add r3, r0, r3
0x18:       sub r5, r4, r6
0x1C: L:    or   r3, r2, r4
```

Control Hazards

Control Hazards

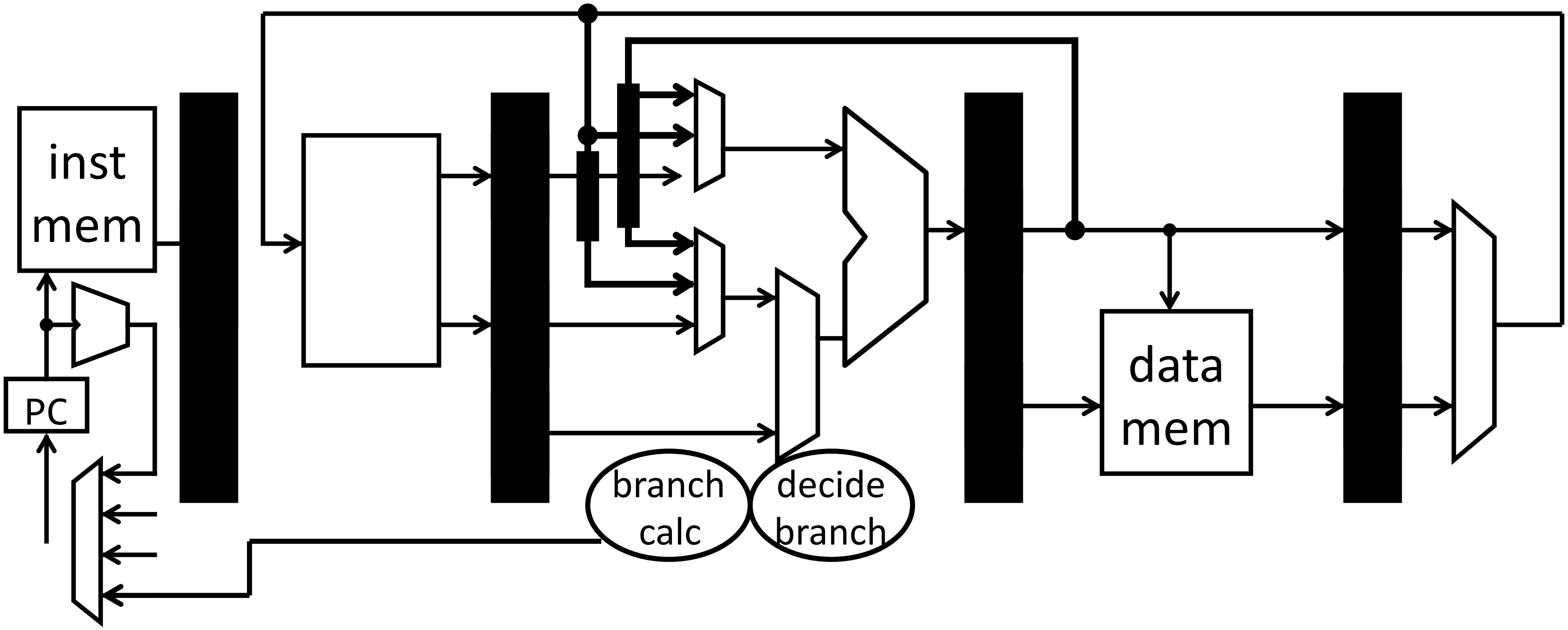
- instructions are fetched in stage 1 (IF)
- branch and jump decisions occur in stage 3 (EX)
- i.e. next PC is not known until ***2 cycles*** *after* branch/jump

What happens to instr following a branch, if branch taken?

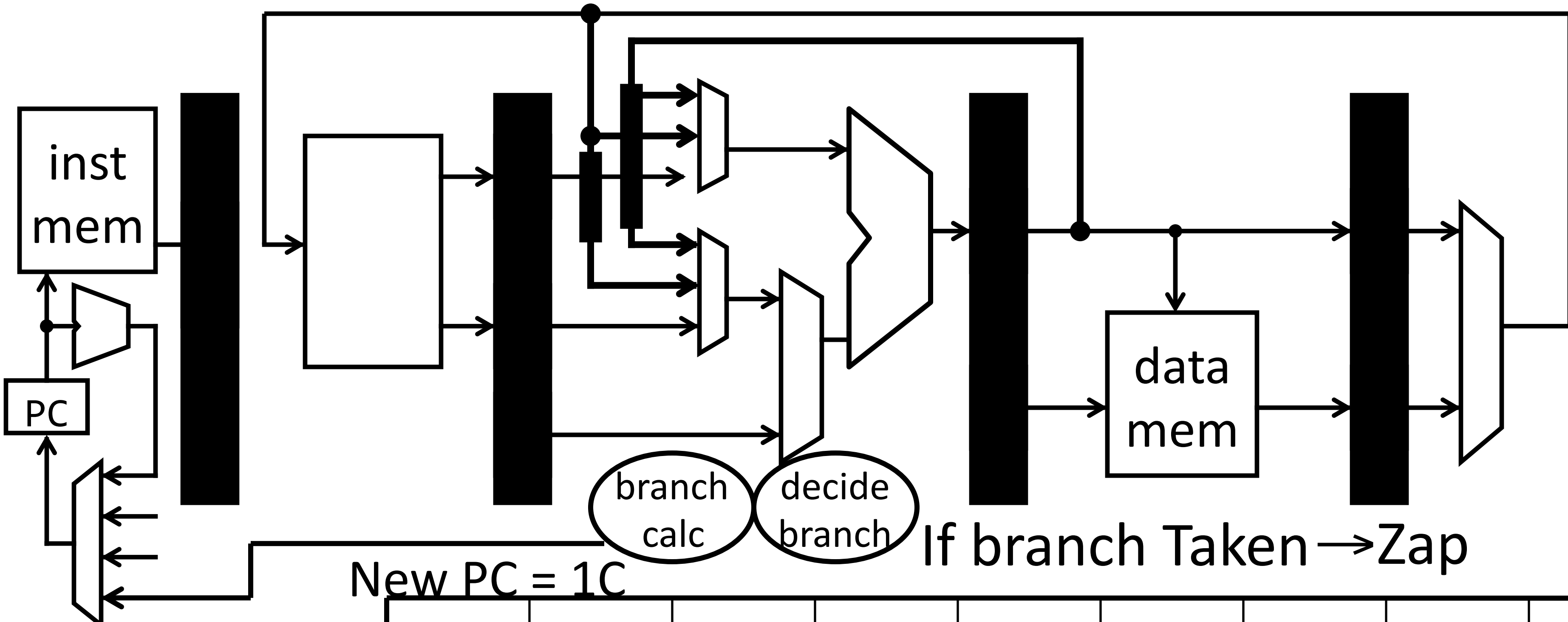
Stall (+ Zap/Flush)

- prevent PC update
- clear IF/ID pipeline register
  - instruction just fetched might be wrong, so convert to nop
- allow branch to continue into EX stage

Control Hazards



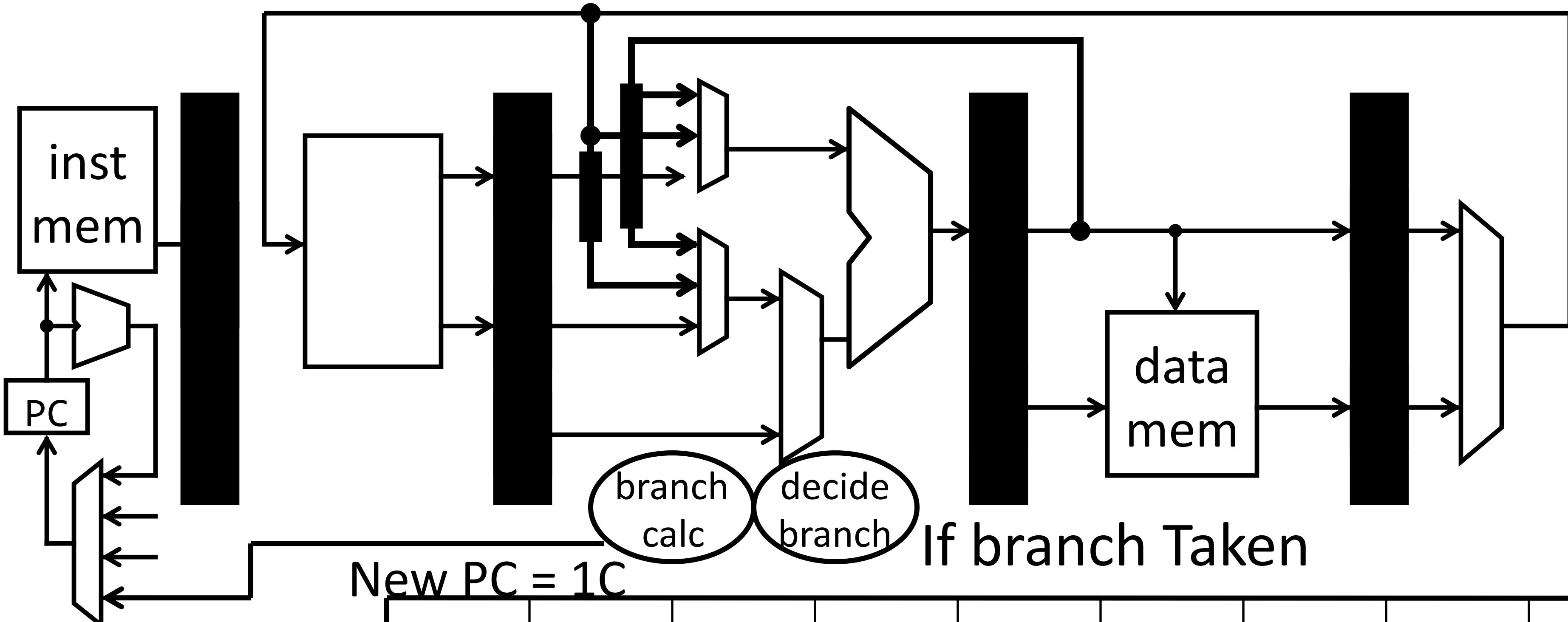
Control Hazards



If branch Taken → Zap

10: beq r1, r2, L	IF	ID	Ex	M	W				
14: add r3, r0, r3		IF	ID	NOP	NOP	NOP			
18: sub r5, r4, r6			IF	NOP	NOP	NOP	NOP		
1C: L: or r3, r2, r4				IF	ID	Ex	M	W	

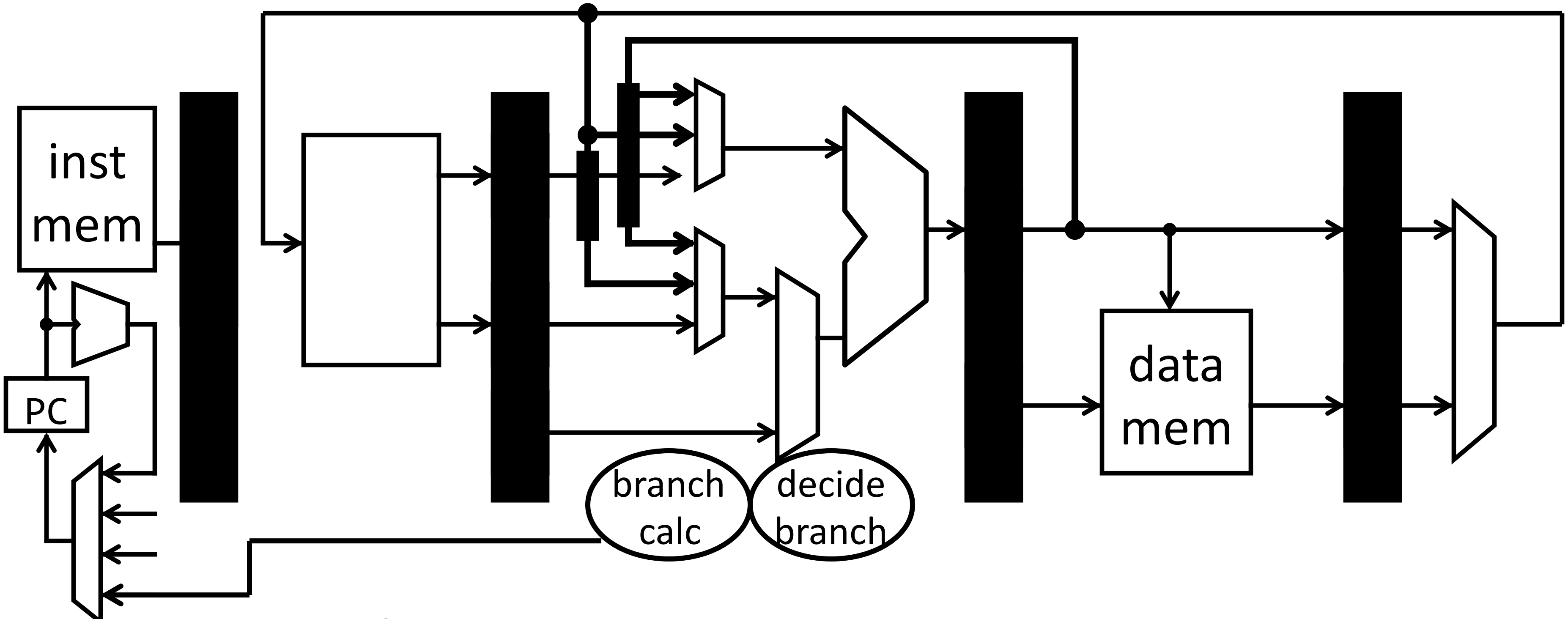
Control Hazards



If branch Taken

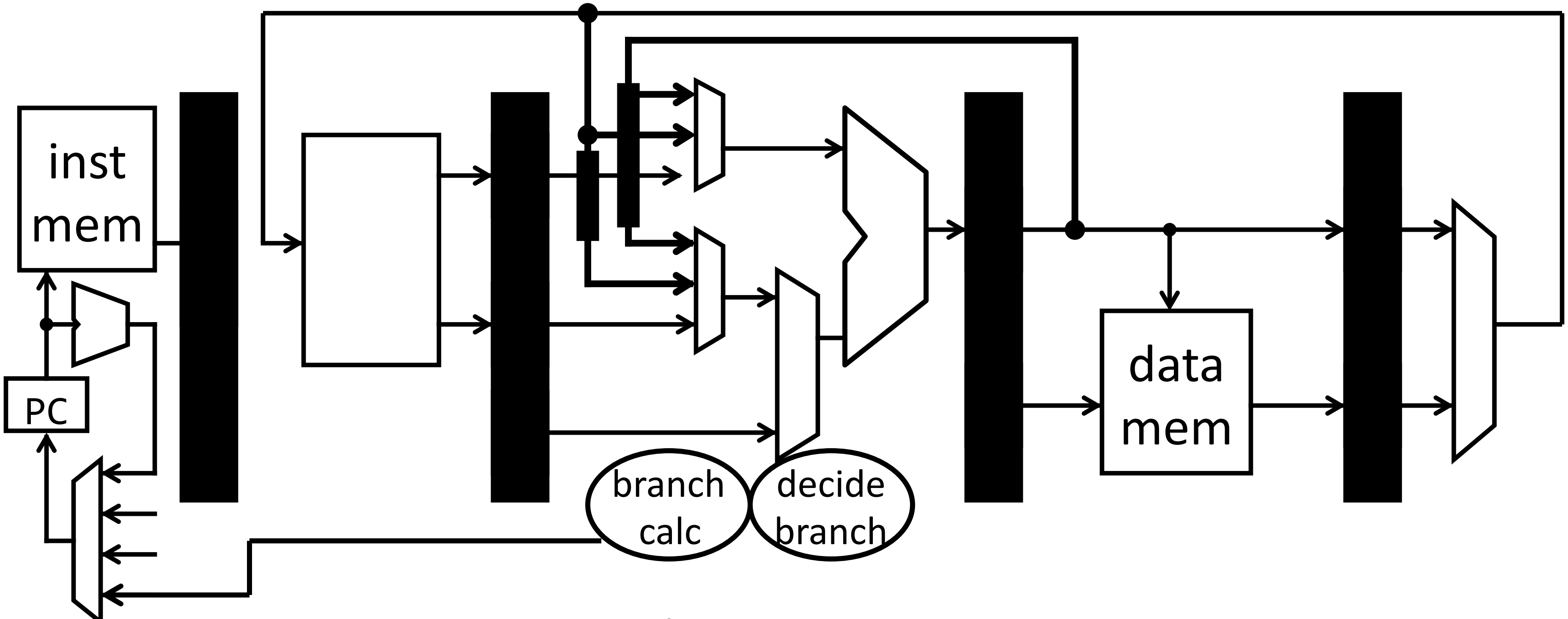
10: beq r1, r2, L	IF	ID	Ex	M	W				
14: add r3, r0, r3		IF	ID	NOP	NOP	NOP			
18: sub r5, r4, r6			IF	NOP	NOP	NOP	NOP		
1C: L: or r3, r2, r4				IF	ID	Ex	M	W	

Control Hazards



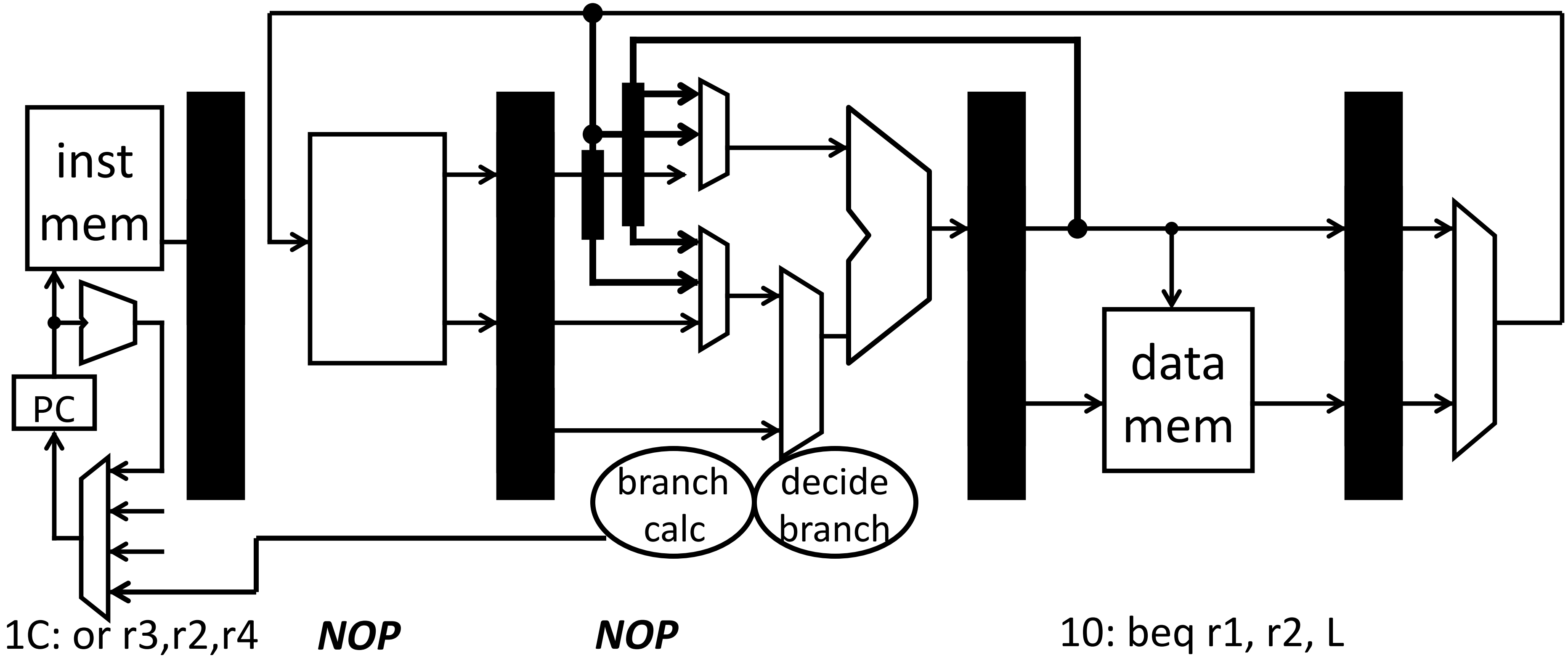
14: add r3,r0,r3   10: beq r1, r2, L

Control Hazards

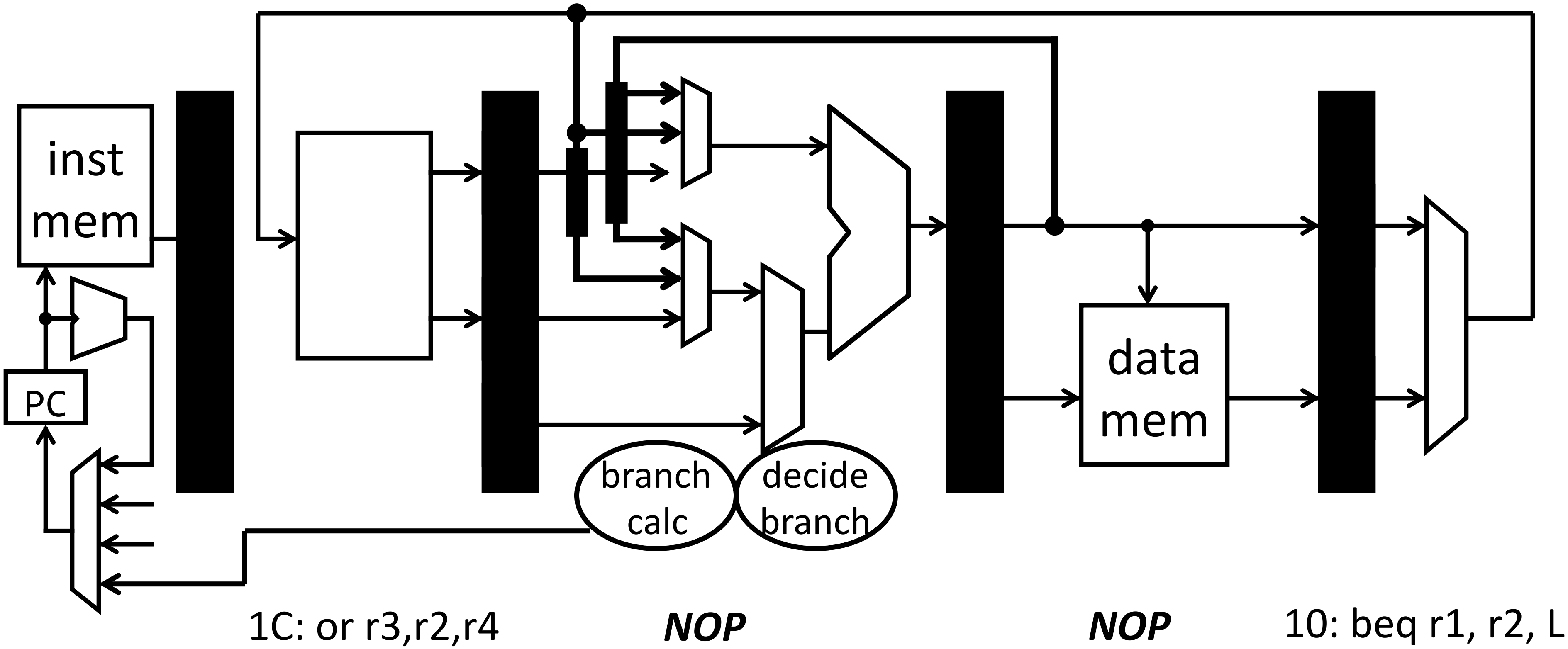


18: sub r5,r4,r6 14: add r3,r0,r3 10: beq r1, r2, L

Control Hazards



Control Hazards



Reduce the cost of control hazard?

Can we forward/bypass values for branches?

- We can move branch calc from EX to ID
- will require new bypasses into ID stage; or can just zap the second instruction

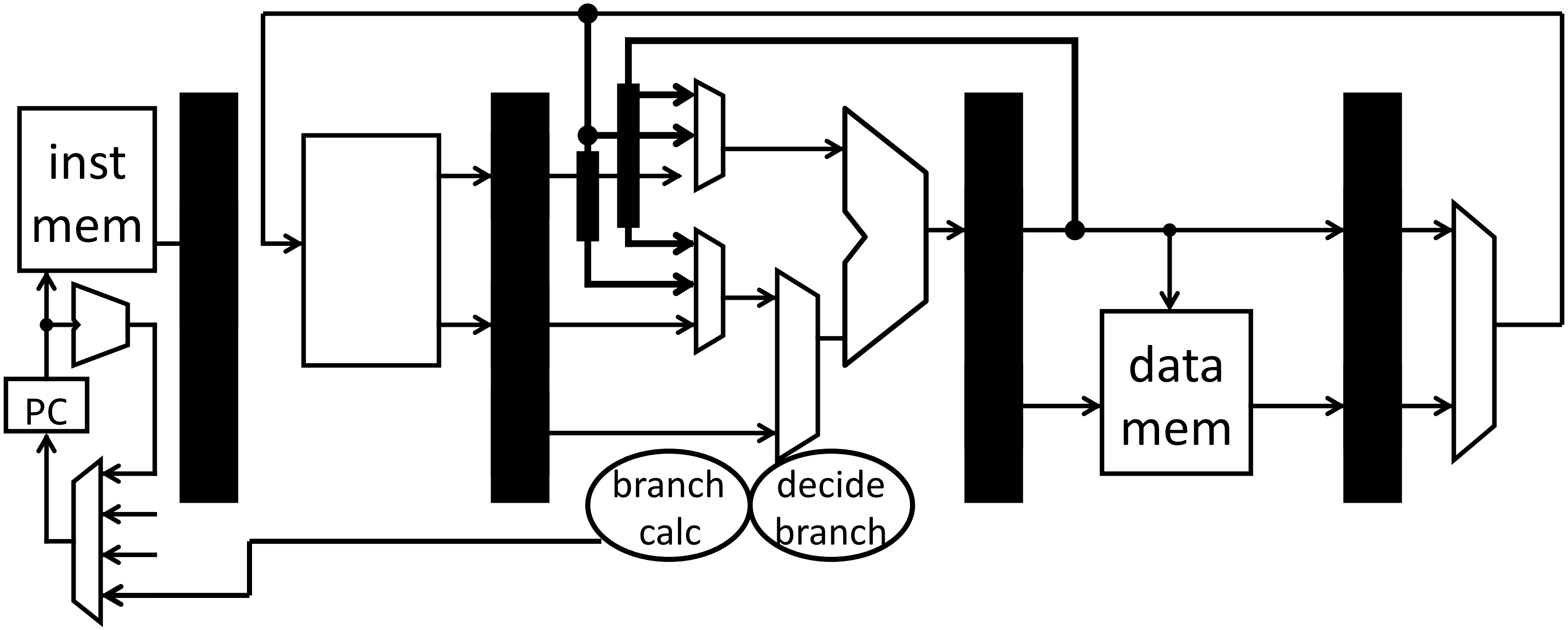
What happens to instructions following a branch, if branch taken?

- Still need to zap/flush instructions

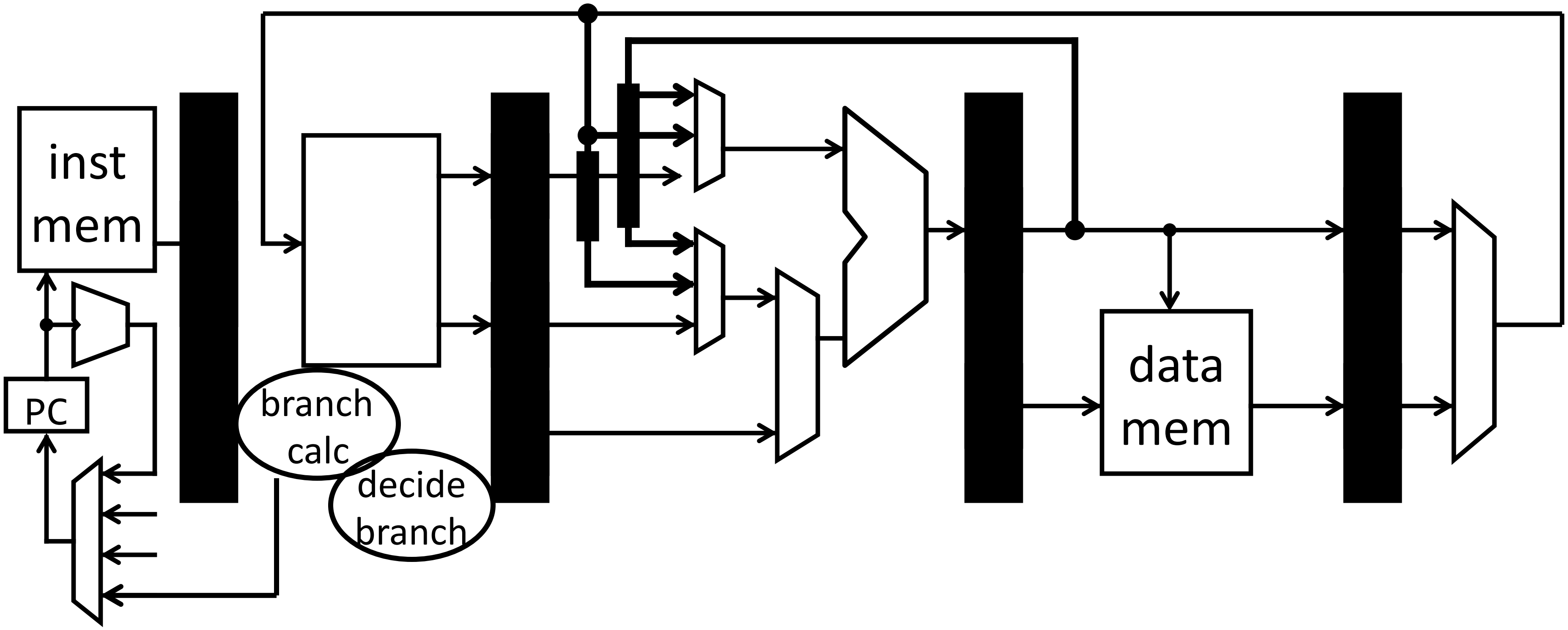
Is there still a performance penalty for branches

- Yes, need to stall, then may need to zap (flush) subsequent instructions that have already been fetched

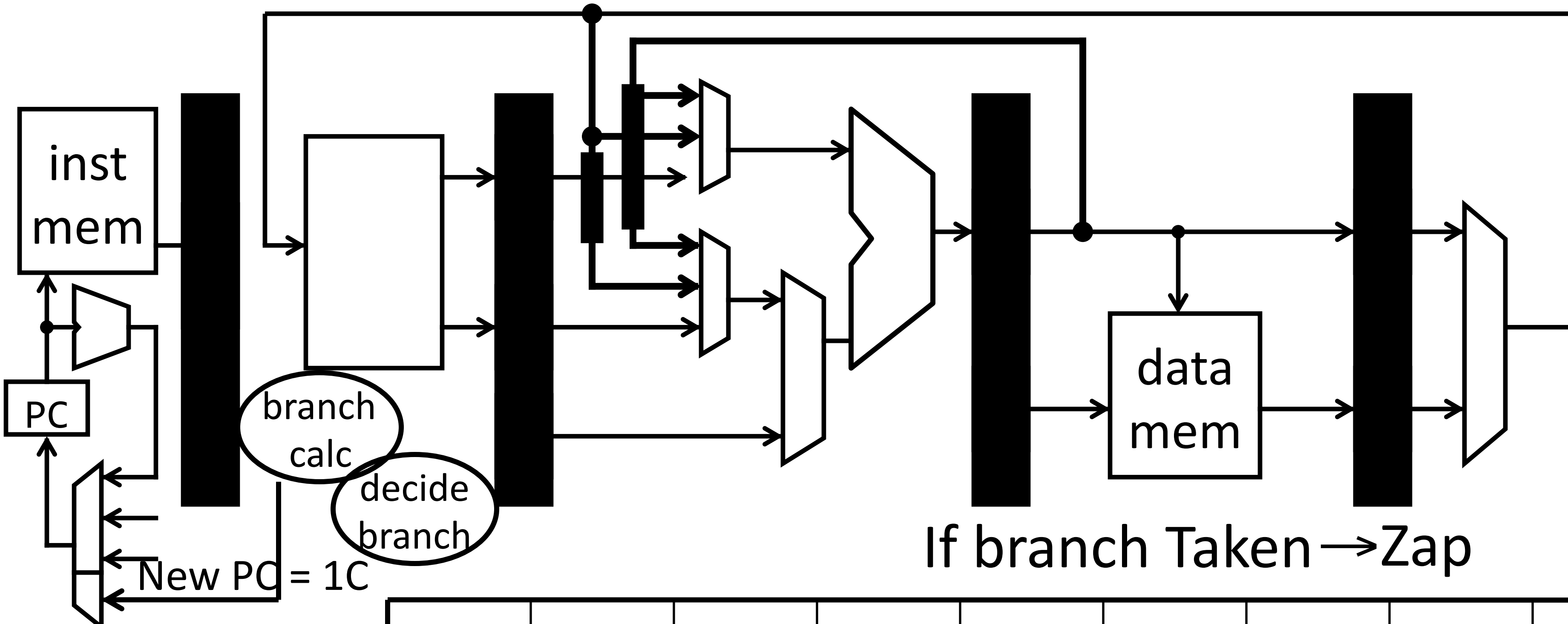
Control Hazards



Control Hazards



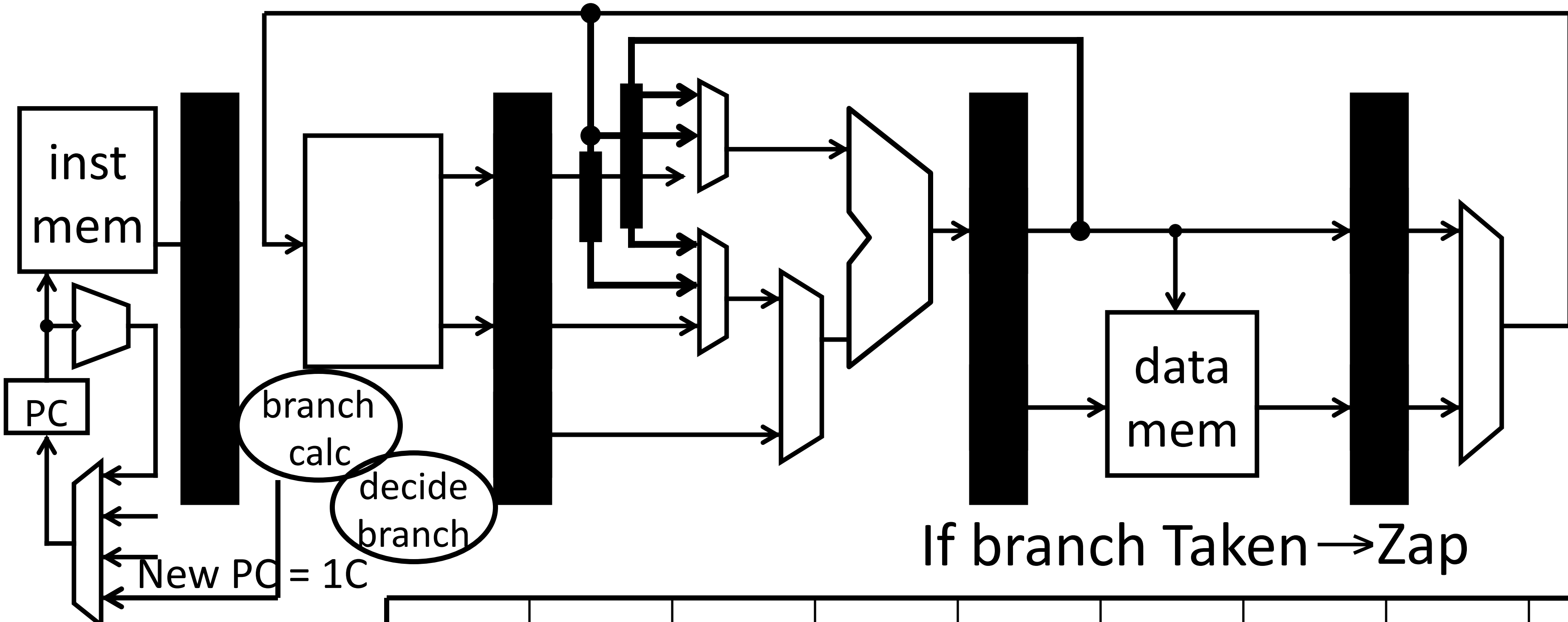
Control Hazards



If branch Taken → Zap

10:	beq r1, r2, L	IF	ID	Ex	M	W			
14:	add r3, r0, r3		IF	NOP	NOP	NOP	NOP		
18:	sub r5, r4, r6								
1C:	L: or r3, r2, r4			IF	ID	Ex	M	W	

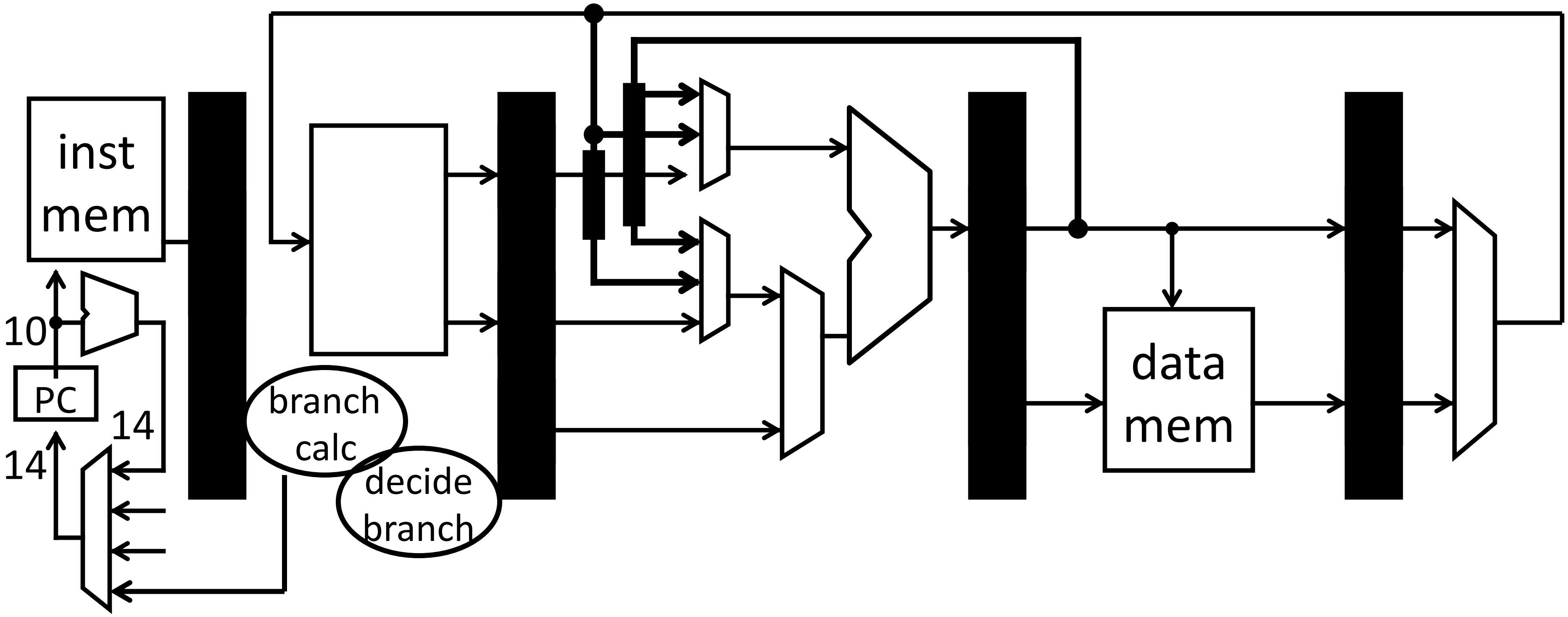
Control Hazards



If branch Taken → Zap

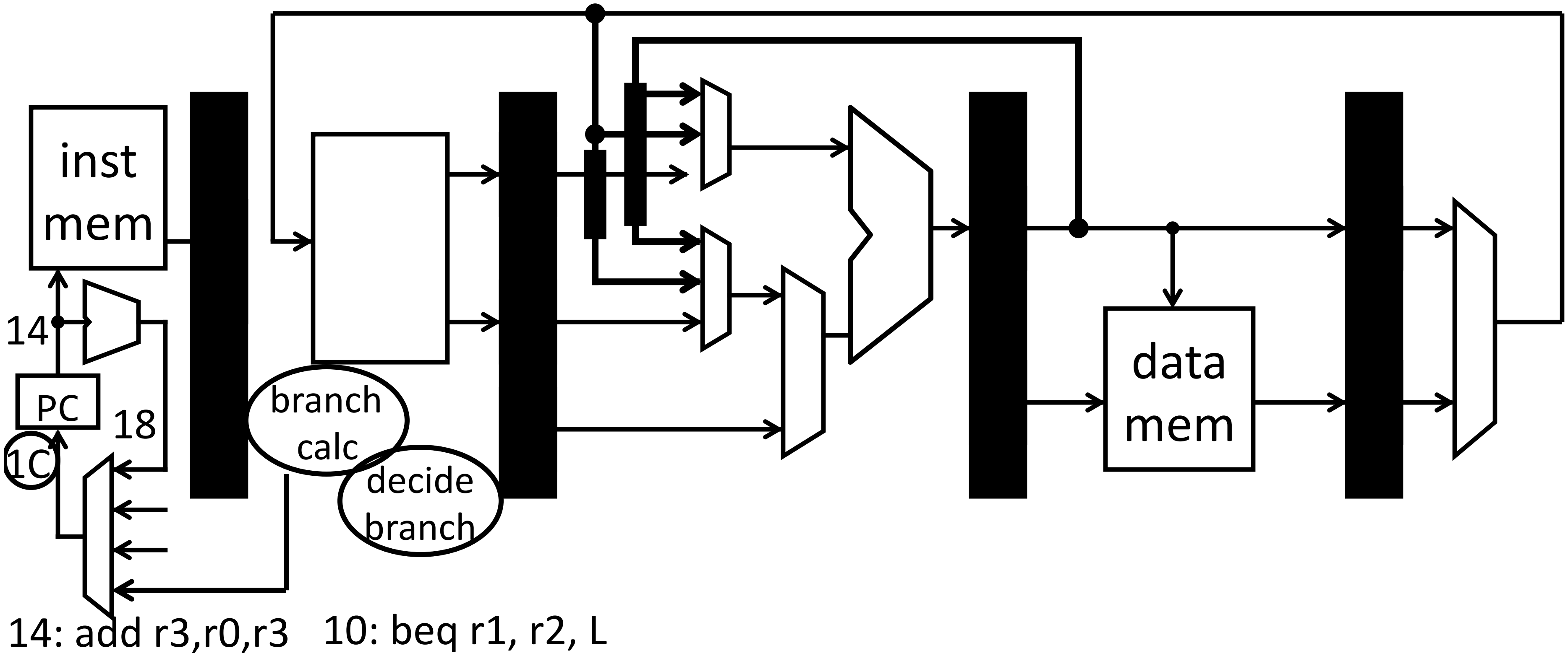
10:	beq r1, r2, L	IF	ID	Ex	M	W			
14:	~~add r3, r0, r3~~		~~IF~~	~~NOP~~	~~NOP~~	~~NOP~~	~~NOP~~		
18:	sub r5, r4, r6								
1C:	L: or r3, r2, r4			IF	ID	Ex	M	W	

Control Hazards

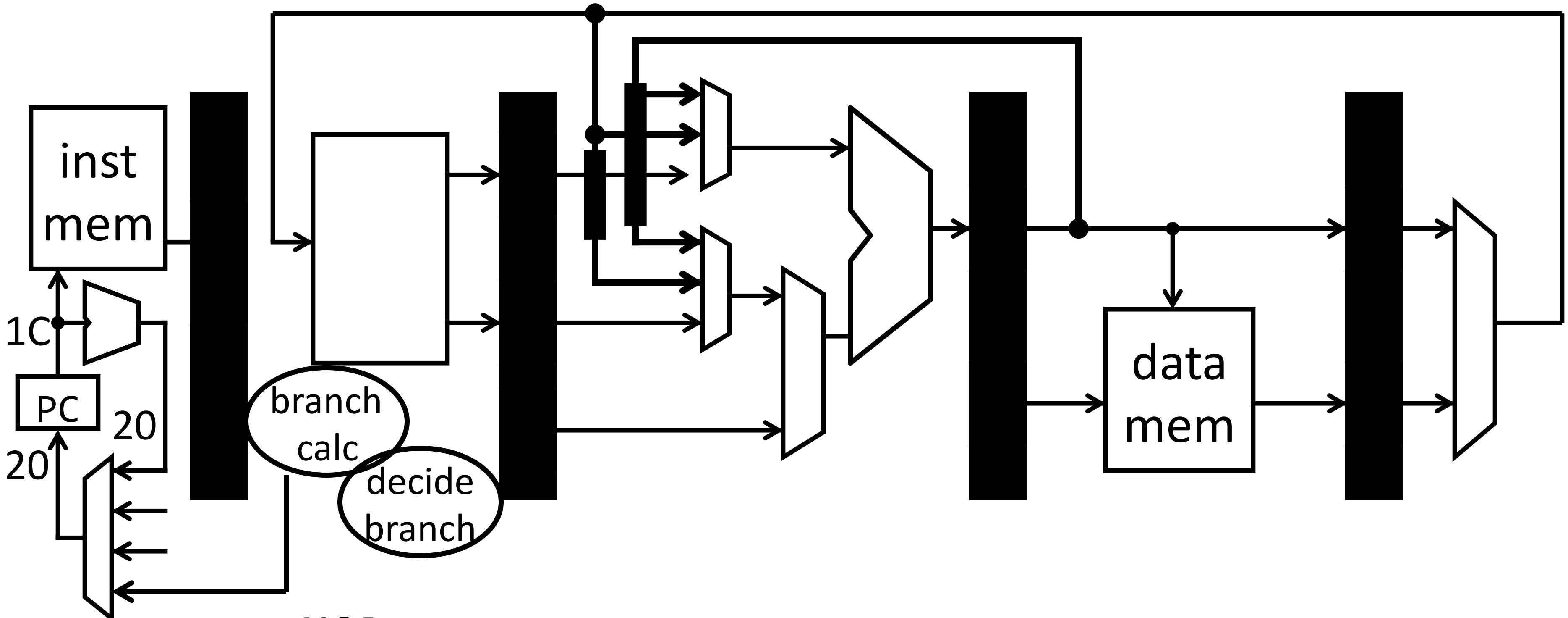


10: beq r1,r2,L

Control Hazards

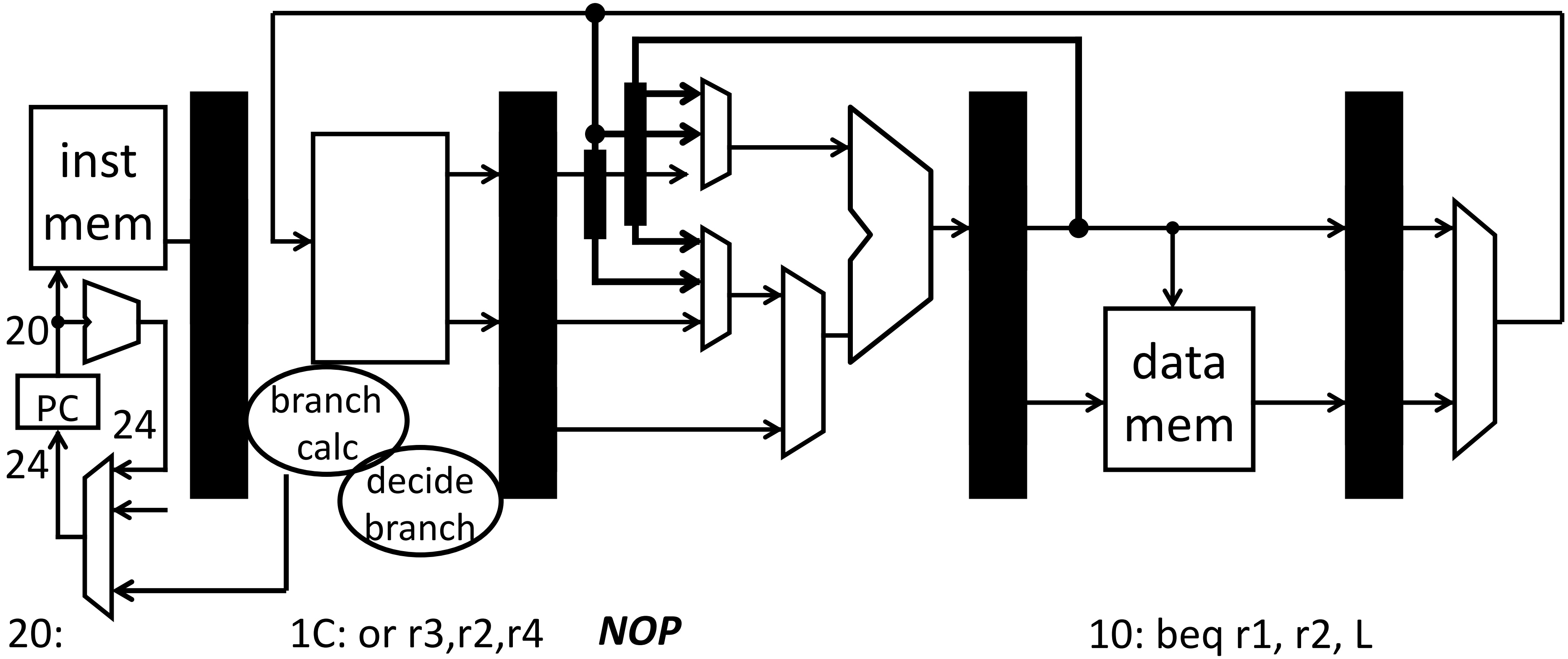


Control Hazards



1C: or r3,r2,r4 ***NOP*** 10: beq r1, r2, L

Control Hazards



Control Hazards

Control Hazards

- instructions are fetched in stage 1 (IF)
- branch and jump decisions occur in stage 3 (EX)

i.e. next PC is not known until ***2 cycles*** *after* branch/jump

- Can optimize and move branch and jump decision to stage 2 (ID)

i.e. next PC is not known until ***1 cycles*** *after* branch/jump

Stall (+ Zap)

- prevent PC update
- clear IF/ID pipeline register
  - instruction just fetched might be wrong one, so convert to nop
- allow branch to continue into EX stage

Takeaway

Control hazards occur because the PC following a control instruction is not known until control instruction computes if branch should be taken or not

If branch taken, then need to zap/flush instructions. There still a performance penalty for branches: Need to stall, then may need to zap (flush) subsequent instructions that have already been fetched

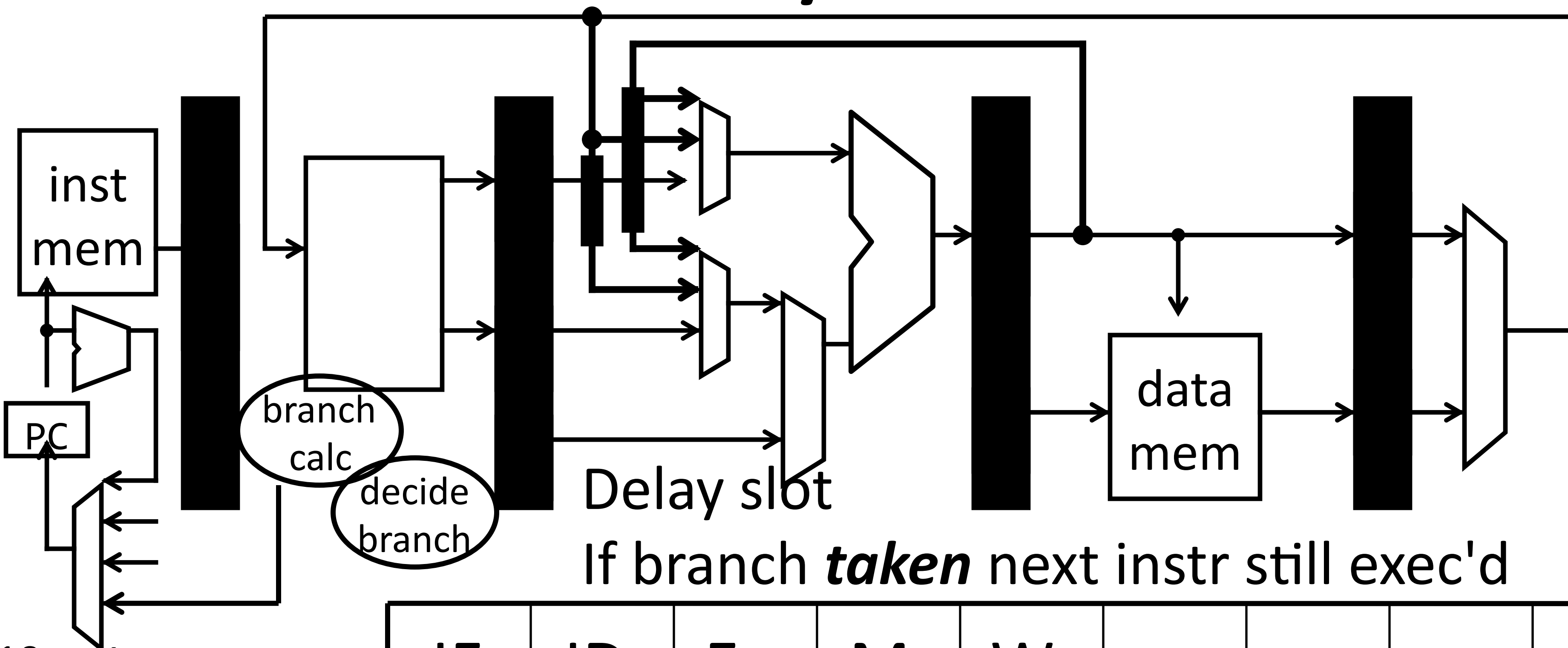
We can reduce cost of a control hazard by moving branch decision and calculation from Ex stage to ID stage. This reduces the cost from flushing two instructions to only flushing one.

Reduce cost of control hazard more?

Delay Slot

- ISA says N instructions after branch/jump ***always*** executed
  - MIPS has 1 branch delay slot
  - i.e. whether branch taken or not, instruction following branch is ***always*** executed

Delay Slot

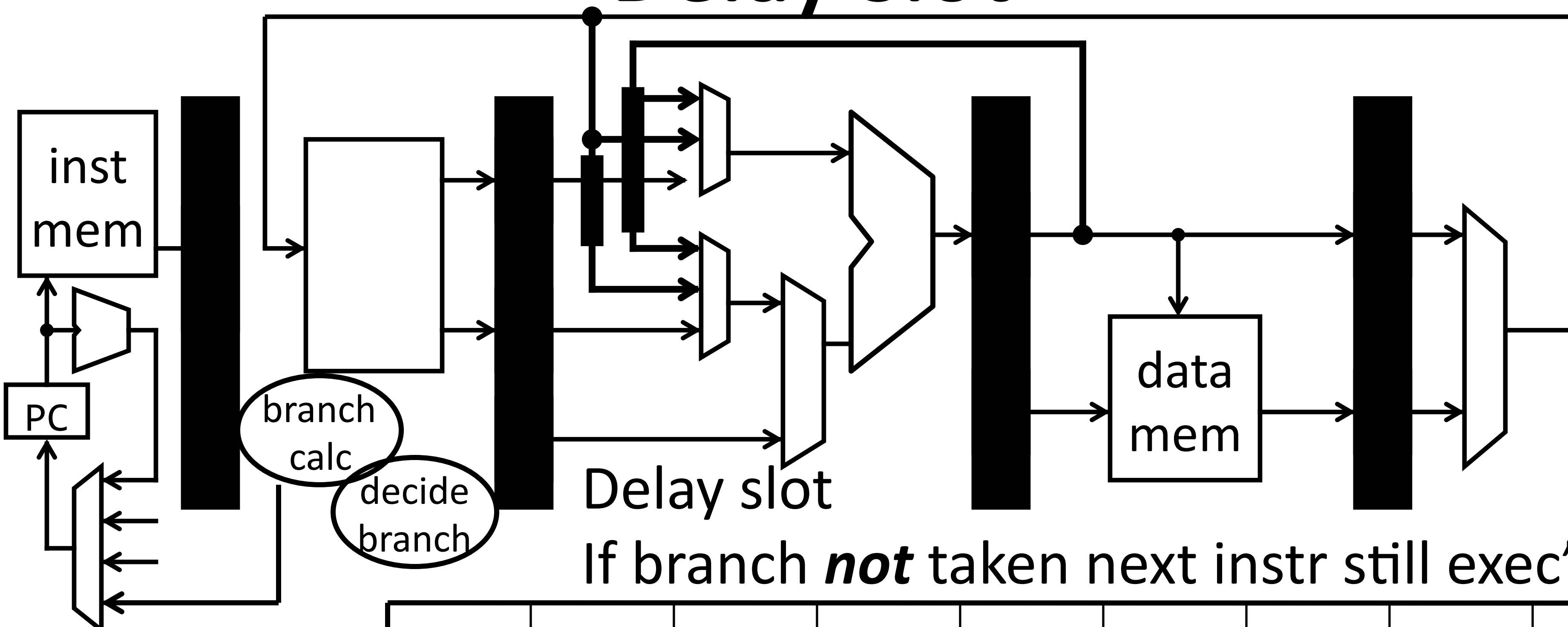


Delay slot

If branch ***taken*** next instr still exec'd

10: beq r1, r2, L	IF	ID	Ex	M	W			
14: add r3, r0, r3		IF	ID	Ex	M	W		
18: sub r5, r4, r6								
1C: L: or r3, r2, r4			IF	ID	Ex	M	W	

Delay Slot



Delay slot

If branch ***not*** taken next instr still exec'd

10:	beq r1, r2, L	IF	ID	Ex	M	W			
14:	add r3, r0, r3		IF	ID	Ex	M	W		
18:	sub r5, r4, r6			IF	ID	Ex	M	W	
1C:	L: or r3, r2, r4				IF	ID	Ex	M	W

Control Hazards

Control Hazards

- instructions are fetched in stage 1 (IF)
- branch and jump decisions occur in stage 3 (EX)

i.e. next PC is not known until 2 cycles after branch/jump

- Can optimize and move branch and jump decision to stage 2 (ID)

i.e. next PC is not known until ***1 cycles*** *after* branch/jump

Stall (+ Zap)

- prevent PC update
- clear IF/ID pipeline register
  - instruction just fetched might be wrong one, so convert to nop
- allow branch to continue into EX stage

Delay Slot

- ISA says N instructions after branch/jump always executed
  - MIPS has 1 branch delay slot

Takeaway

Control hazards occur because the PC following a control instruction is not known until control instruction computes if branch should be taken or not. If branch taken, then need to zap/ flush instructions. There still a performance penalty for branches Need to stall, then may need to zap (flush) subsequent instructions that have already been fetched.

We can reduce cost of a control hazard by moving branch decision and calculation from Ex stage to ID stage. This reduces the cost from flushing two instructions to only flushing one.

Delay Slots can potentially increase performance due to control hazards by putting a useful instruction in the delay slot since the instruction in the delay slot will ***always*** be executed. Requires software (compiler) to make use of delay slot. Put nop in delay slot if not able to put useful instruction in delay slot.

Reduce cost of Ctrl Haz even further?

Speculative Execution

- *"Guess"* direction of the branch
  - Allow instructions to move through pipeline
  - Zap them later if wrong guess
- Useful for long pipelines

Speculative Execution: Loops

Pipeline so far

- "Guess" (predict) that the branch will not be taken

We can do better!

- Make prediction based on last branch
- Predict "take branch" if last branch "taken"
- Or Predict "do not take branch" if last branch "not taken"

- Need one bit to keep track of last branch

Speculative Execution: Loops

What is accuracy of branch predictor?

Wrong twice per loop!

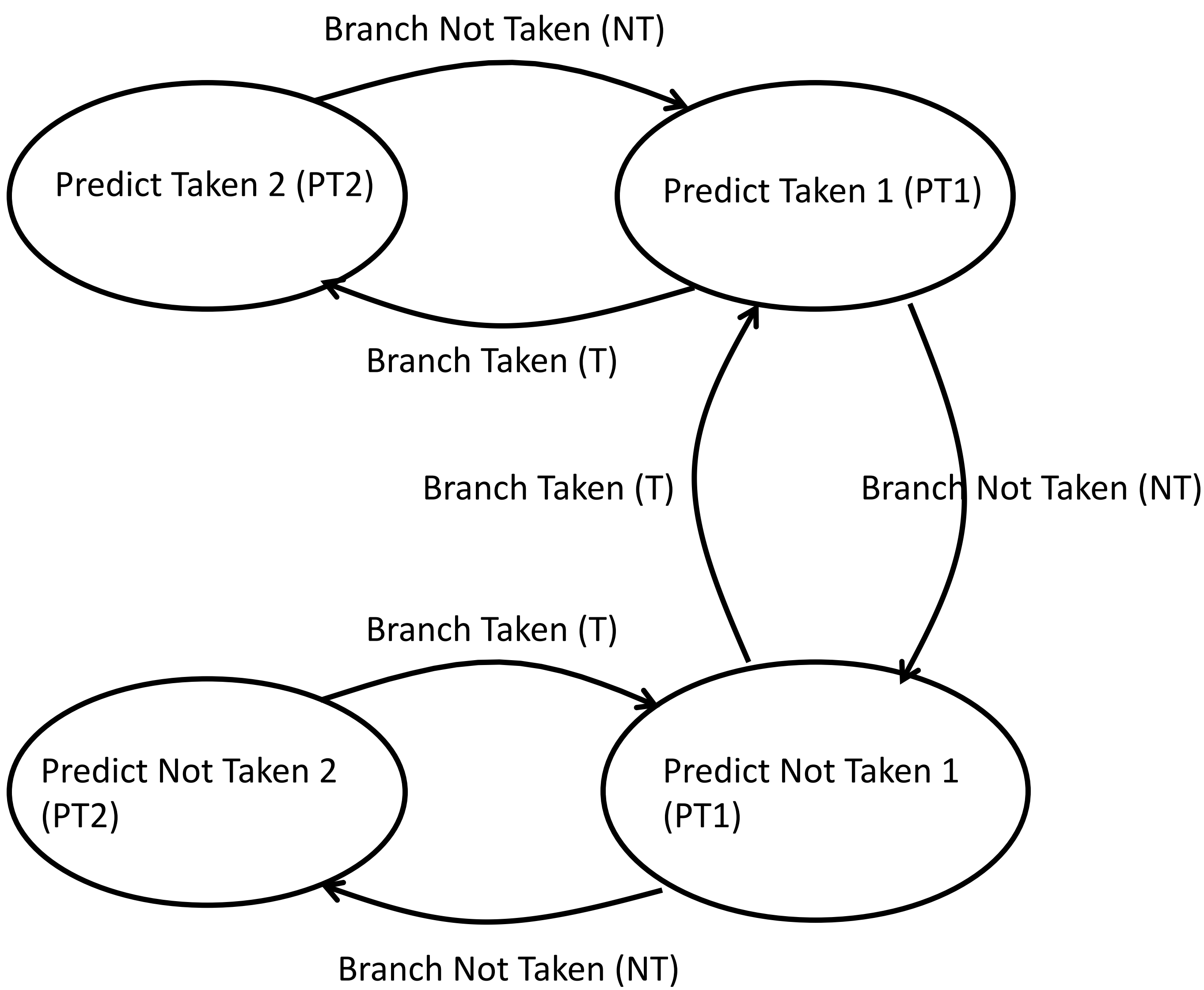
Once on loop enter and exit

We can do better with 2 bits

```
While (r3 ≠ 0) {.... r3--;}
Top:    BEQZ r3, End

        J Top
End:
```

Speculative Execution: Branch Execution



Summary

Control hazards

- Is branch taken or not?
- Performance penalty: stall and flush

Reduce cost of control hazards

- Move branch decision from Ex to ID
  - 2 nops to 1 nop
- Delay slot
  - Compiler puts useful work in delay slot. ISA level.
- Branch prediction
  - Correct. Great!
  - Wrong. Flush pipeline. Performance penalty

Hazards Summary

Data hazards

Control hazards

Structural hazards

- resource contention
- so far: impossible because of ISA and pipeline design