Pipeline Hazards

Krste Asanovic
Laboratory for Computer Science
M.I.T.

http://www.csg.lcs.mit.edu/6.823

Reduced Instruction Set Computers

(Cocke, IBM; Patterson, UC Berkeley; Hennessy, Stanford)

Compilers have difficulty using complex instructions

- VAX: 60% of microcode for 20% of instructions, only responsible for 0.2% execution time
- IBM experiment retargets 370 compiler to use simple subset of ISA
  => Compiler generated faster code!

Simple instruction sets don't need microcode

- Use fast memory near processor as cache, not microcode storage

Design ISA for simple pipelined implementation

- Fixed length, fixed format instructions
- Load/store architecture with up to one memory access/instruction
- Few addressing modes, synthesize others with code sequence
- Register-register ALU operations
- Delayed branch

MIPS R2000

(One of first commercial RISCs, 1986)

Load/Store architecture

- 32x32-bit GPR (R0 is wired), HI & LO SPR (for multiply/divide)
- 74 instructions
- Fixed instruction size (32 bits), only 3 formats
- PC-relative branches, register indirect jumps
- Only base+displacement addressing mode
- No condition bits, compares write GPRs, branches test GPRs
- Delayed loads and branches

Five-stage instruction pipeline

- Fetch, Decode, Execute, Memory, Write Back
- CPI of 1 for register-to-register ALU instructions
- 8 MHz clock
- Tightly-coupled off-chip FP accelerator (R2010)

RISC/CISC Comparisons

(late 80s/early 90s)

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

R2000 vs VAX 8700 [Bhandarkar and Clark, ‘91]

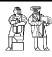
- R2000 has ~2.7x advantage with equivalent technology

Intel 80486 vs Intel i860 (both 1989)

- Same company, same CAD tools, same process
- i860 2-4x faster - even more on some floating-point tasks

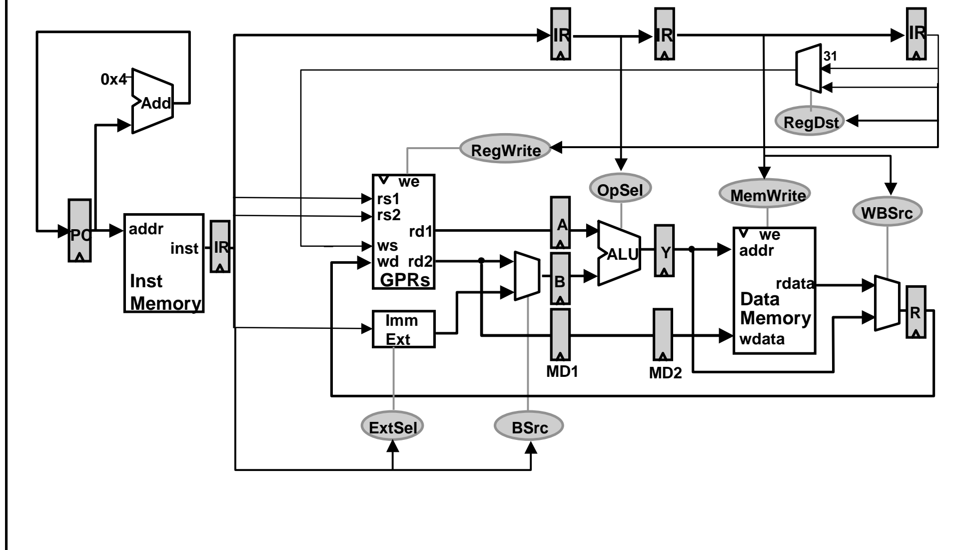
DEC nVAX vs Alpha 21064 (both 1992)

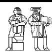
- Same company, same CAD tools, same process
- Alpha 2-4x faster



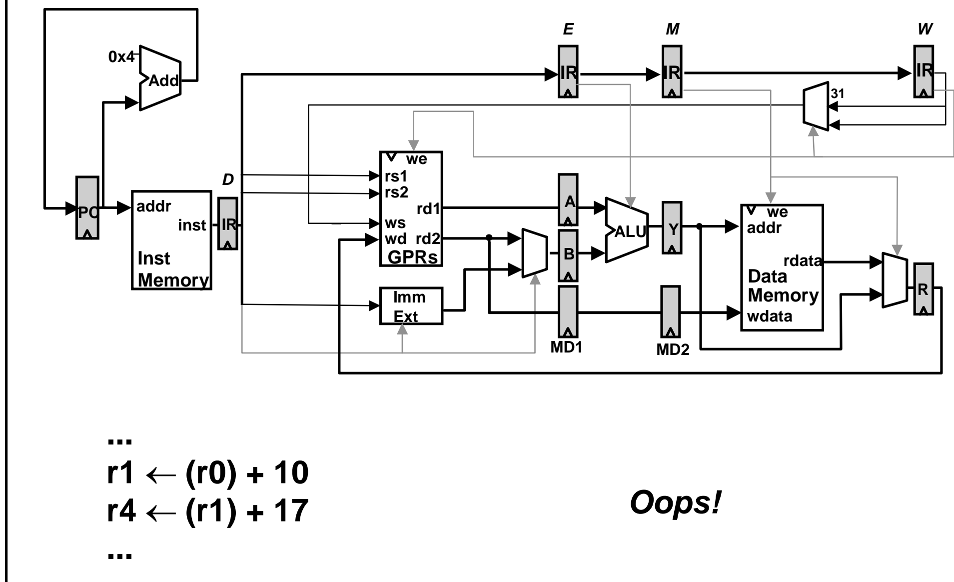
Pipelined DLX Datapath

without interlocks and jumps





Data Hazards



...
r1 ← (r0) + 10
r4 ← (r1) + 17
...

Oops!

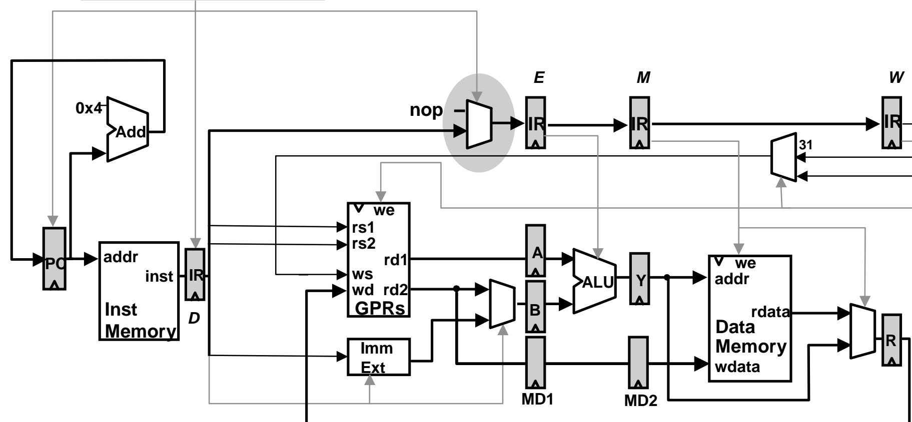
Resolving Data Hazards

1. **Interlocks**
   Freeze earlier pipeline stages until data becomes available

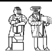
2. **Bypasses**
   If data is available somewhere in the datapath provide a *bypass* to get it to the right stage

Interlocks to resolve Data Hazards

Stall Condition

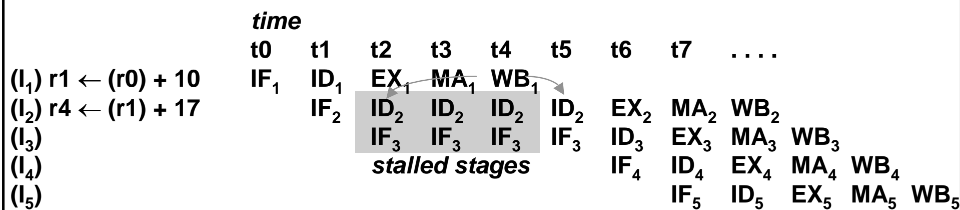


...
r1 ← (r0) + 10
r4 ← (r1) + 17
...

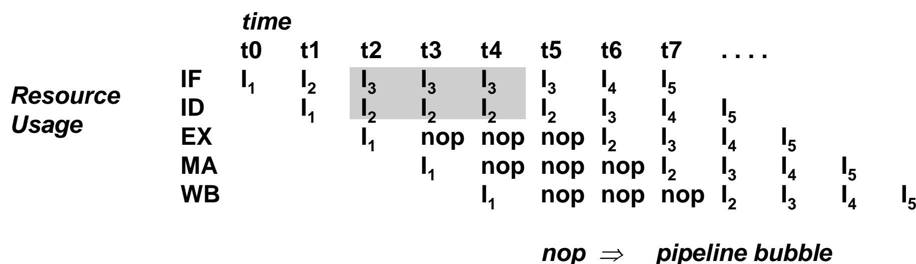


Krste Asanovic

Stalled Stages and Pipeline Bubbles



time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .			
(I_1) r1 ← (r0) + 10	IF_1	ID_1	EX_1	MA_1	WB_1							
(I_2) r4 ← (r1) + 17		IF_2	ID_2	ID_2	ID_2	ID_2	EX_2	MA_2	WB_2			
(I_3)			IF_3	IF_3	IF_3	IF_3	ID_3	EX_3	MA_3	WB_3		
(I_4)			*stalled stages*				IF_4	ID_4	EX_4	MA_4	WB_4	
(I_5)								IF_5	ID_5	EX_5	MA_5	WB_5



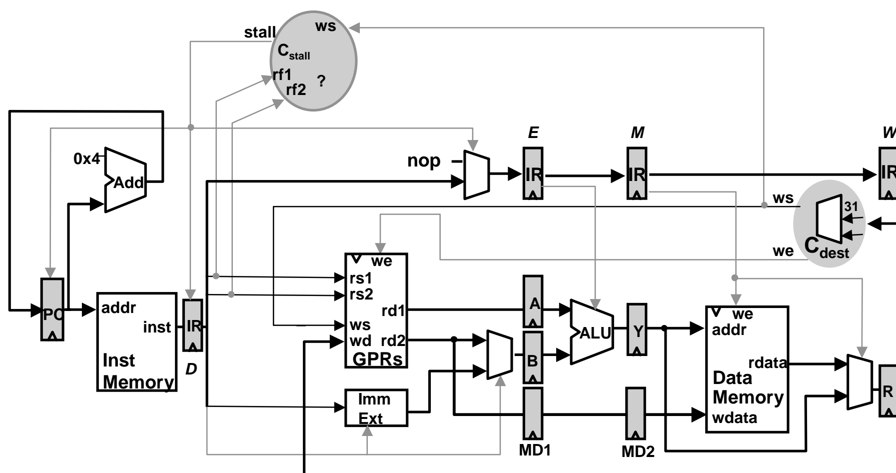
Resource Usage

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .			
IF	I_1	I_2	I_3	I_3	I_3	I_3	I_4	I_5				
ID		I_1	I_2	I_2	I_2	I_2	I_3	I_4	I_5			
EX			I_1	nop	nop	nop	I_2	I_3	I_4	I_5		
MA				I_1	nop	nop	nop	I_2	I_3	I_4	I_5	
WB					I_1	nop	nop	nop	I_2	I_3	I_4	I_5

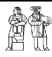
nop ⇒ ***pipeline bubble***

Krste Asanovic

Interlock Control Logic *worksheet*

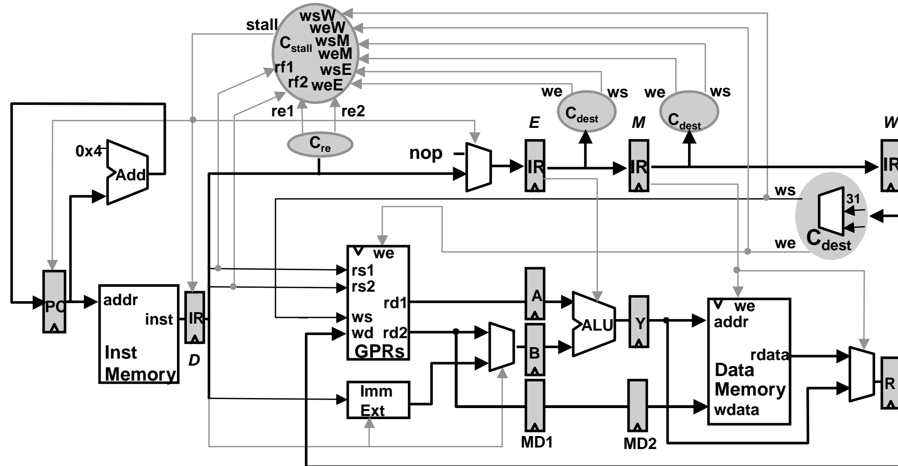


Compare the *source registers* of the instruction in the decode stage with the *destination register* of the *uncommitted* instructions.

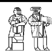


Interlock Control Logic

ignoring jumps & branches



Should we always stall if the rs field matches some rd?
not every instruction writes registers ⇒ we
not every instruction reads registers ⇒ re



Source & Destination Registers

R-type:	op	rf1	rf2	rf3	func
I-type:	op	rf1	rf2	immediate16	
J-type:	op	immediate26			

		source(s)	*destination*
ALU	rf3 ← (rf1) func (rf2)	rf1, rf2	rf3
ALUi	rf2 ← (rf1) op imm	rf1	rf2
LW	rf2 ← M [(rf1) + imm]	rf1	rf2
SW	M [(rf1) + imm] ← (rf2)	rf1, rf2	
B__Z	*cond* (rf1)		
	true: PC ← (PC) + imm	rf1	
	false: PC ← (PC) + 4	rf1	
J	PC ← (PC) + imm		
JAL	r31 ← (PC), PC ← (PC) + imm		31
JR	PC ← (rf1)	rf1	
JALR	r31 ← (PC), PC ← (rf1)	rf1	31

Deriving the Stall Signal

C_{dest}
ws = *Case* opcode
ALU ⇒ rf3
ALUi, LW ⇒ rf2
JAL, JALR ⇒ 31

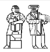
we = *Case* opcode
ALU, ALUi, LW,
JAL, JALR ⇒ on
... ⇒ off

C_{re}
re1 = *Case* opcode
ALU, ALUi, ⇒ on
⇒ off

re2 = *Case* opcode
⇒ on
⇒ off

stall =

Stall if the *source registers* of the instruction in the decode stage matches the *destination register* of the *uncommitted* instructions.



The Stall Signal

C_{dest}
ws = *Case* opcode
ALU ⇒ rf3
ALUi, LW ⇒ rf2
JAL, JALR ⇒ 31

we = *Case* opcode
ALU, ALUi, LW,
JAL, JALR ⇒ (ws ≠ 0)
... ⇒ off

C_{re}
re1 = *Case* opcode
ALU, ALUi, LW,
SW, BZ,
JR, JALR ⇒ on
J, JAL ⇒ off

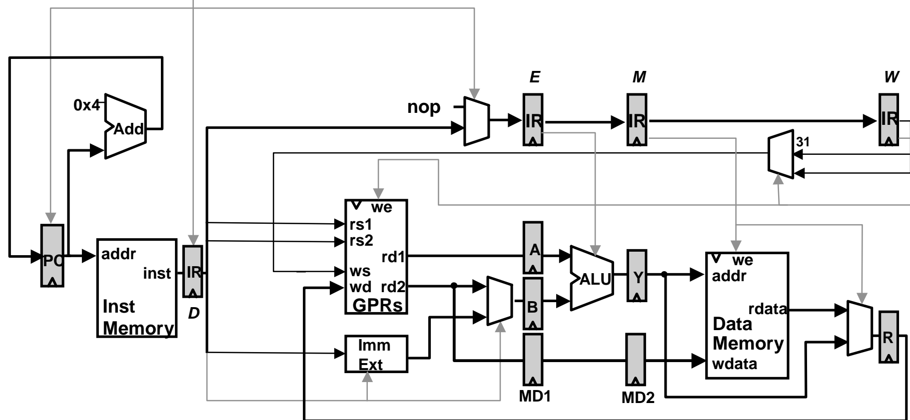
re2 = *Case* opcode
ALU, SW ⇒ on
... ⇒ off

C_{stall}

$$
\begin{aligned}
stall = & ((rf1_D = ws_E).we_E + \\
& (rf1_D = ws_M).we_M + \\
& (rf1_D = ws_W).we_W \quad) . re1_D \quad + \\
& ((rf2_D = ws_E).we_E + \\
& (rf2_D = ws_M).we_M + \\
& (rf2_D = ws_W).we_W \quad) . re2_D
\end{aligned}
$$

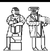
This is not the full story !

Hazards due to Loads & Stores

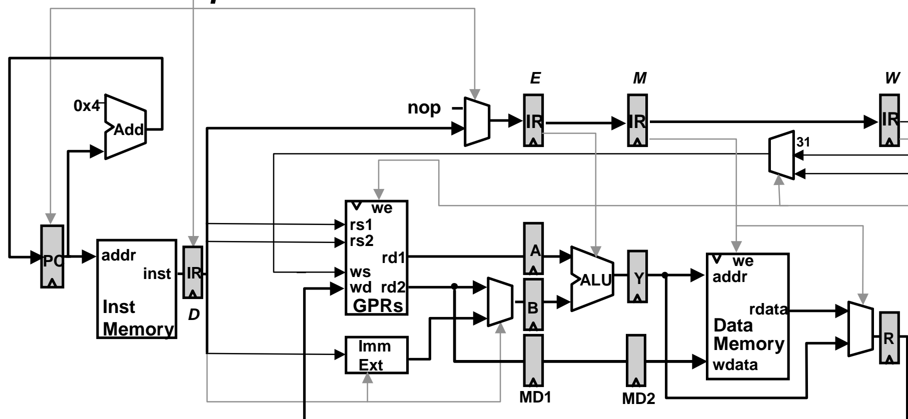


...
M[(r1)+7] ← (r2)
r4 ← M[(r3)+5]
...

Is there any possible data hazard in this instruction sequence?



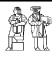
Hazards due to Loads & Stores



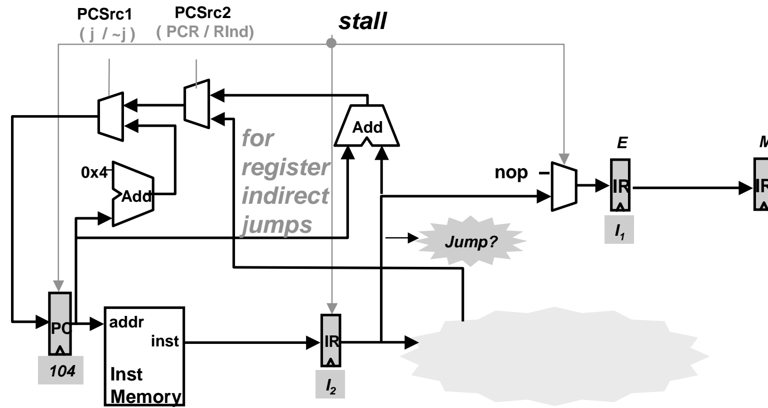
M[(r1)+7] ← (r2)
r4 ← M[(r3)+5]
...

(r1)+7 = (r3)+5 ⇒ *data hazard*

However, the hazard is avoided because *our memory system completes writes in one cycle !*



Complications due to Jumps

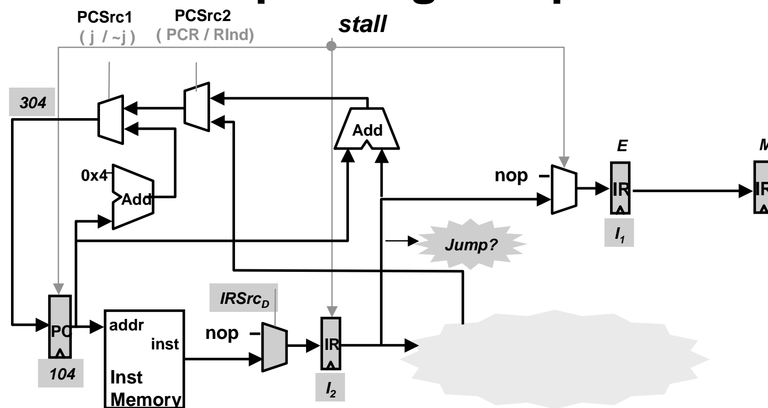


I_1	096	ADD	
I_2	100	J +200	
I_3	~~104~~	~~ADD~~	*kill*
I_4	304	SUB	

A jump instruction kills (not stalls) the following instruction

How?

Pipelining Jumps



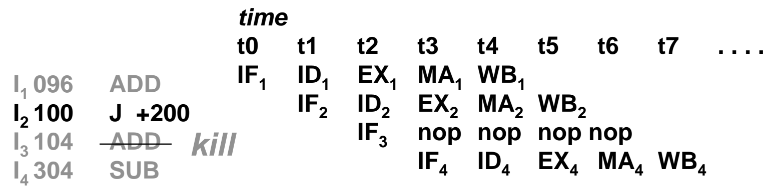
Killing the fetched instruction: Insert a mux before IR

I_1	096	ADD	
I_2	100	J +200	
I_3	~~104~~	~~ADD~~	*kill*
I_4	304	SUB	

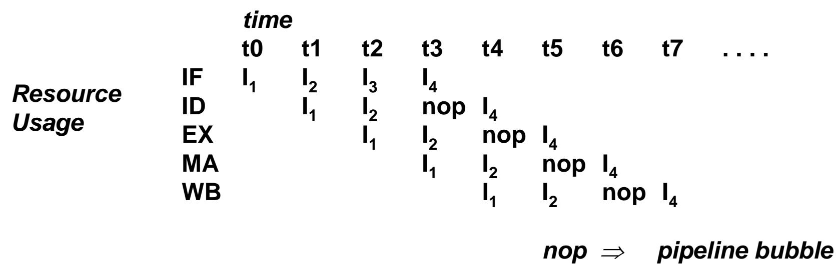
$IRSrc_D$ = *Case* $opcode_D$
J, JAL $\Rightarrow$ nop
... $\Rightarrow$ IM

Any interaction between stall and jump?

Jump Pipeline Diagrams



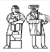
			time								
			t0	t1	t2	t3	t4	t5	t6	t7	. . . .
I_1 096	ADD		IF_1	ID_1	EX_1	MA_1	WB_1				
I_2 100	J +200			IF_2	ID_2	EX_2	MA_2	WB_2			
I_3 104	~~ADD~~	*kill*			IF_3	nop	nop	nop	nop		
I_4 304	SUB					IF_4	ID_4	EX_4	MA_4	WB_4	



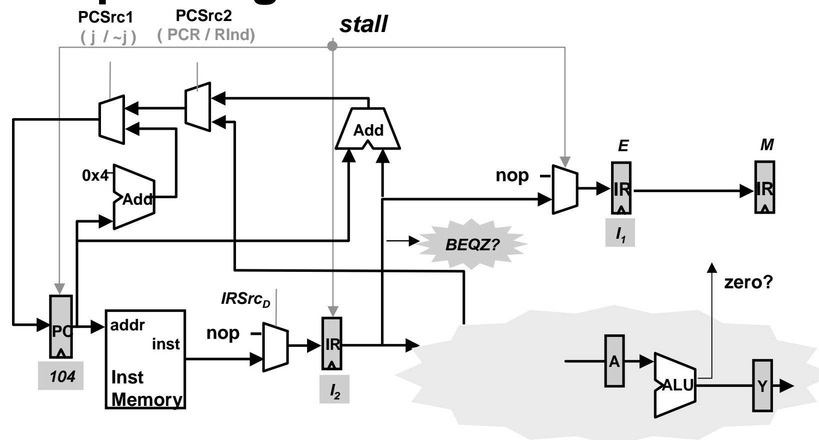
Resource Usage

	time								
	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
IF	I_1	I_2	I_3	I_4					
ID		I_1	I_2	nop	I_4				
EX			I_1	I_2	nop	I_4			
MA				I_1	I_2	nop	I_4		
WB					I_1	I_2	nop	I_4	

nop $\Rightarrow$ ***pipeline bubble***



Pipelining Conditional Branches

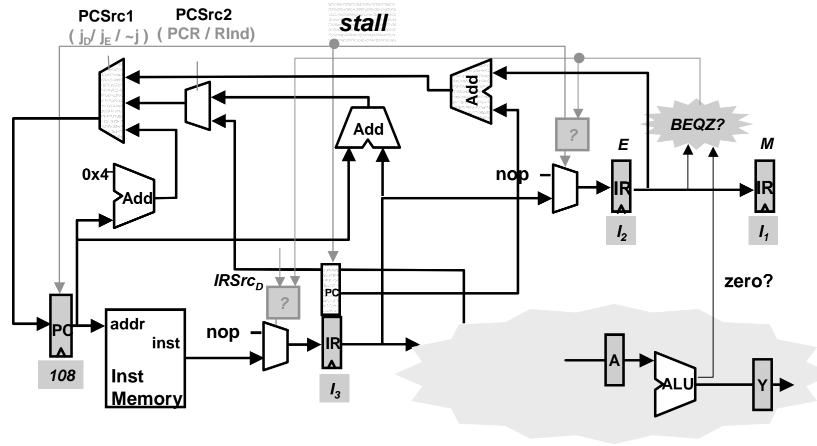


I_1 096 ADD
I_2 100 BEQZ r1, +200
I_3 104 ADD
I_4 304 ADD

Branch condition is not known until the execute stage

what action should be taken in the decode stage ?

Conditional Branches: *solution 1*



I_1	096	ADD
I_2	100	BEQZ r1, +200
I_3	104	ADD
I_4	304	ADD

If the branch is taken
- **kill the two following instructions**
- **the instruction at the decode stage is not valid**

⇒ ***stall signal is not valid***

New Stall Signal

$$stall = (((rf1_D = ws_E).we_E + (rf1_D = ws_M).we_M + (rf1_D = ws_W).we_W).re1_D$$
$$+ \; ((rf2_D = ws_E).we_E + (rf2_D = ws_M).we_M + (rf2_D = ws_W).we_W).re2_D)$$
$$. \; !((opcode_E = BEQZ).z + (opcode_E = BNEZ).!z)$$

Control Equations for PC Muxes

Solution 1

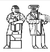
PCSrc1 = *Case* opcode$_E$
- **BEQZ.z, BNEZ.!z ⇒ j_E**
- **... ⇒ *Case* opcode$_D$**
  - **J, JAL, JR, JALR ⇒ j_D**
  - **... ⇒ ~j**

PCSrc2 = *Case* opcode$_D$
- **J, JAL ⇒ PCR**
- **JR, JALR ⇒ RegI**

Give priority to the older instruction, i.e., execute stage instruction over decode stage instruction

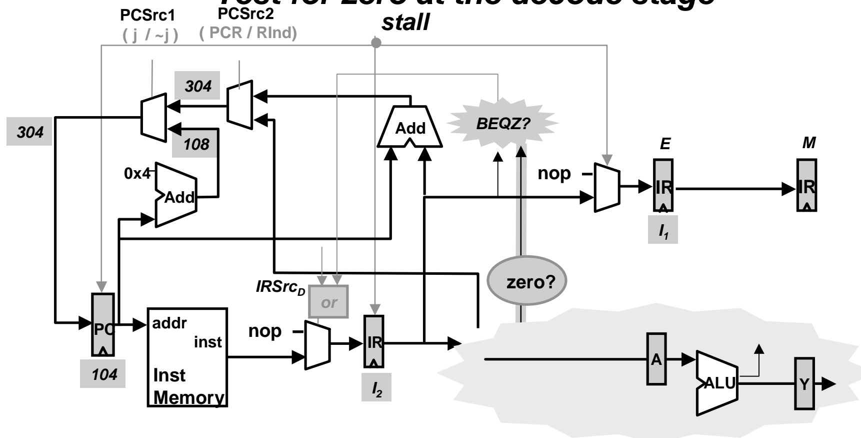
IRSrc$_D$ = *Case* opcode$_E$
- **BEQZ.z, BNEZ.!z ⇒ nop**
- **... ⇒ (*Case* opcode$_D$**
  - **J, JAL, JR, JALR ⇒ nop**
  - **... ⇒ IM)**

IRSrc$_E$ = *Case* opcode$_E$
- **BEQZ.z, BNEZ.!z ⇒ nop**
- **... ⇒ stall.nop + !stall.IR$_D$**



Conditional Branches: *solution 2*

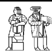
Test for zero at the decode stage



I_1 096 ADD
I_2 100 BEQZ r1, +200
I_3 104 ADD
I_4 304 ADD

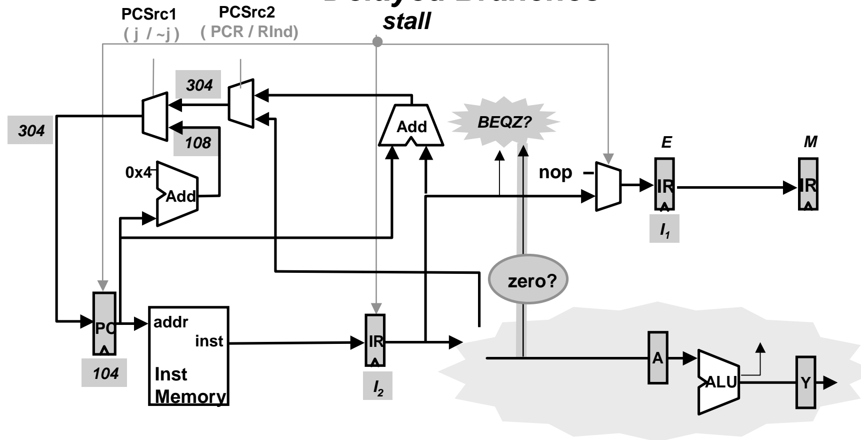
Need to kill only one instruction !

Wouldn't work if DLX had general branch conditions (i.e., r1>r2)?



Conditional Branches: *solution 3*

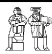
Delayed Branches



I1 096 ADD
I2 100 BEQZ r1, +200
I3 104 ADD (delay slot)
I4 304 ADD

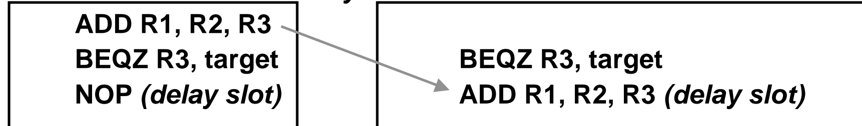
Change the semantics of branches and jumps
⇒ Instruction after branch *always* executed, regardless if branch taken or not taken.

Need not kill any instructions !

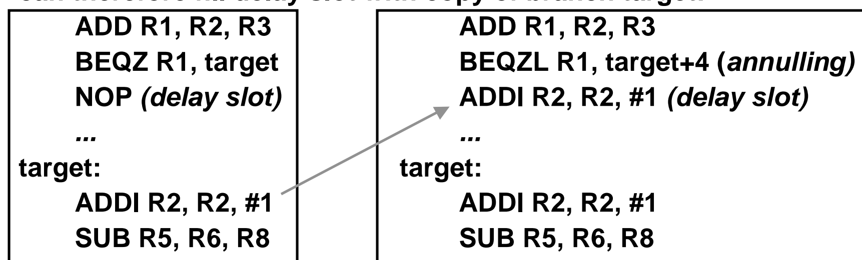


Annulling Branches

- **Plain delayed branches only allow compiler to fill delay slot with instructions that will *always* be executed:**



- **Annulling branches kill delay slot if branch is *not taken*. Compiler can therefore fill delay slot with copy of branch target:**



- **Can also have variant of annulling branch that kills delay slot if branch is *taken***

Branch Delay Slots

First introduced in pipelined microcode engines.
Adopted for early RISC machines with single-issue pipelines, made *visible* to user-level software.

Advantages

- **simplifies control logic for simple pipeline**
- **compiler helps reduce branch hazard penalties**
  - **~70% of single delay slots usefully filled**

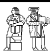
Disadvantages

- **complicates ISA specification and programming**
  - **adds extra "next-PC" state to programming model**
- **complicates control logic for more aggressive implementations**
  - **e.g., out-of-order superscalar designs**

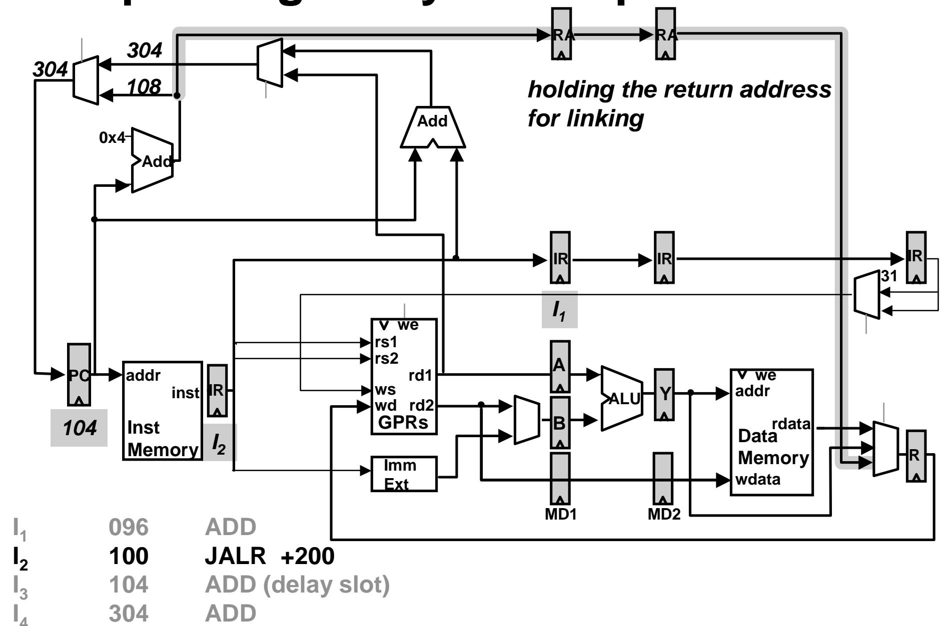
⇒ *Out of favor for new (post-1990) general-purpose ISAs*

Later lectures will cover advanced techniques for control hazards:

- **dynamic (run-time) schemes, e.g., branch prediction**
- **static (compile-time) schemes, e.g., predicated execution**



Pipelining Delayed Jumps & Links



Bypassing

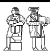
time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) r1 ← (r0) + 10	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) r4 ← (r1) + 17		IF_2	ID_2	ID_2	ID_2	ID_2	EX_2	MA_2	WB_2
(I_3)			IF_3	IF_3	IF_3	IF_3	ID_3	EX_3	MA_3
(I_4)			*stalled stages*				IF_4	ID_4	EX_4
(I_5)								IF_5	ID_5

Each *stall or kill* introduces a bubble in the pipeline

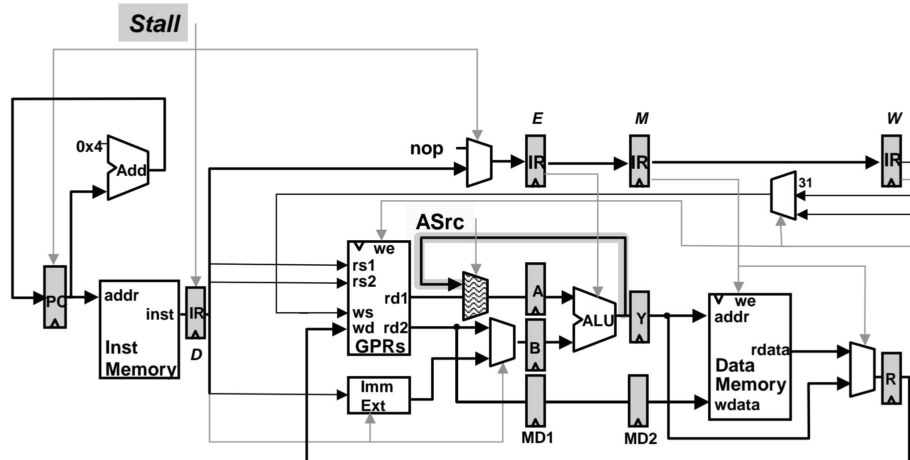
⇒ *CPI > 1*

A new datapath, i.e., *a bypass*, can get the data from the output of the ALU to its input

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) r1 ← (r0) + 10	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) r4 ← (r1) + 17		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3)			IF_3	ID_3	EX_3	MA_3	WB_3		
(I_4)				IF_4	ID_4	EX_4	MA_4	WB_4	
(I_5)					IF_5	ID_5	EX_5	MA_5	WB_5



Adding Bypasses



...

(I_1) r1 ← (r0) + 10

(I_2) r4 ← (r1) + 17

...

Of course you can add many more bypasses!

The Bypass Signal

deriving it from the stall signal

C_{dest}

ws = *Case* opcode
- ALU ⇒ rf3
- ALUi, LW ⇒ rf2
- JAL, JALR ⇒ R31

we = *Case* opcode
- ALU, ALUi, LW ⇒ (ws ≠ R0)
- JAL, JALR ⇒ on
- ... ⇒ off

C_{re}

re1 = *Case* opcode
- ALU, ALUi, LW, SW, BZ, JR, JALR ⇒ on
- J, JAL ⇒ off

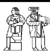
re2 = *Case* opcode
- ALU, SW ⇒ on
- ... ⇒ off

C_{stall}

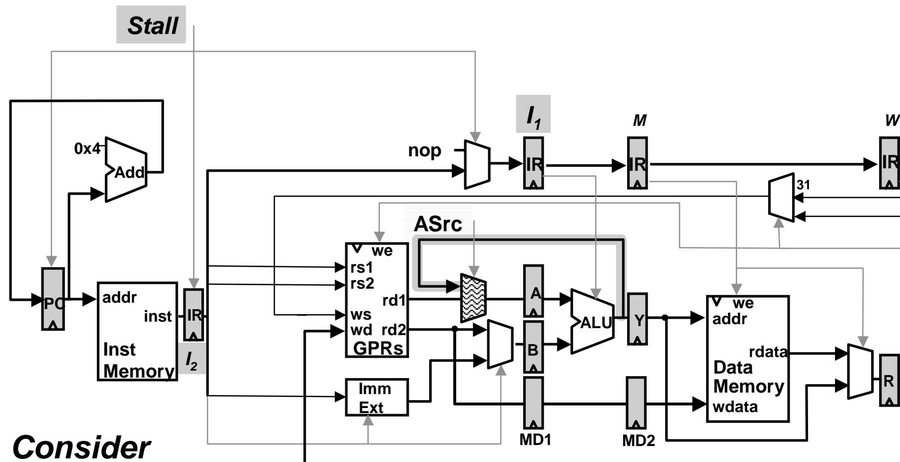
stall = (~~$(rf1_D = ws_E).we_E$ +~~
$(rf1_D = ws_M).we_M$ +
$(rf1_D = ws_W).we_W$) . $re1_D$ +
$((rf2_D = ws_E).we_E$ +
$(rf2_D = ws_M).we_M$ +
$(rf2_D = ws_W).we_W$) . $re2_D$

C_{bypass} ASrc = $(rf1_D = ws_E).we_E . re1_D$

Is this correct ?



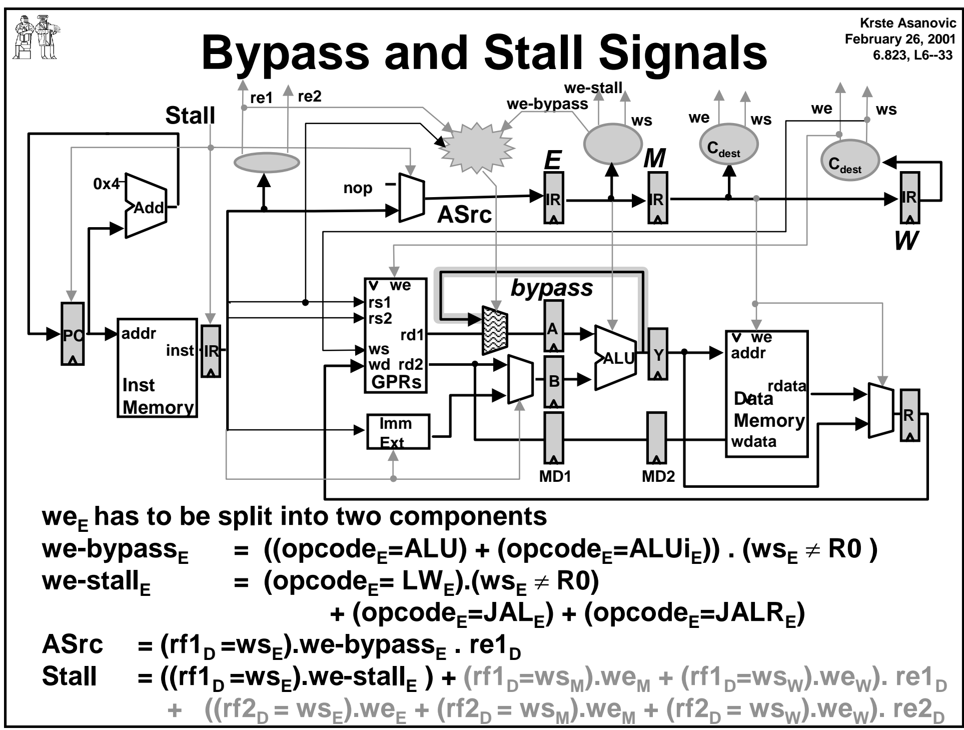
Usefulness of a Bypass

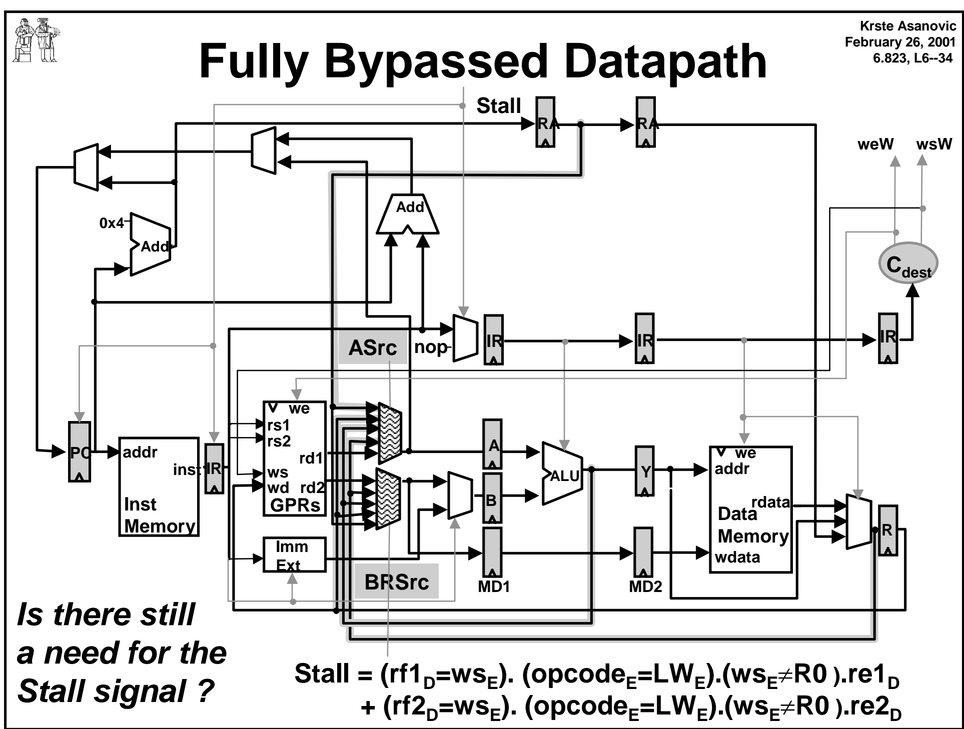


Consider

	...	...	...
(I_1)	r1 ← (r0) + 10	r1 ← M[(r0) + 10]	JAL 500
(I_2)	r4 ← (r1) + 17	r4 ← (r1) + 17	r4 ← (r31) + 17

Where can this bypass help?


Bypass and Stall Signals
Krste Asanovic
February 26, 2001
6.823, L6--33
Stall
re1
re2
we-stall
we-bypass
ws
we
ws
we
ws
E
M
Cdest
Cdest
0x4
Add
nop
ASrc
IR
IR
IR
W
bypass
PC
addr
inst
IR
Inst
Memory
we
rs1
rs2
rd1
ws
wd
rd2
GPRs
A
B
ALU
Y
we
addr
rdata
Data
Memory
wdata
R
Imm
Ext
MD1
MD2
weE has to be split into two components
we-bypassE = ((opcodeE=ALU) + (opcodeE=ALUiE)) . (wsE ≠ R0)
we-stallE = (opcodeE= LWE).(wsE ≠ R0)
+ (opcodeE=JALE) + (opcodeE=JALRE)
ASrc = (rf1D =wsE).we-bypassE . re1D
Stall = ((rf1D =wsE).we-stallE) + (rf1D=wsM).weM + (rf1D=wsW).weW). re1D
+ ((rf2D = wsE).weE + (rf2D = wsM).weM + (rf2D = wsW).weW). re2D


Fully Bypassed Datapath
Krste Asanovic
February 26, 2001
6.823, L6--34
Stall
RA
RA
weW
wsW
Add
0x4
Add
Cdest
ASrc
nop
IR
IR
IR
PC
addr
inst
IR
Inst
Memory
we
rs1
rs2
rd1
ws
wd
rd2
GPRs
Imm
Ext
A
B
ALU
Y
we
addr
rdata
Data
Memory
wdata
R
MD1
MD2
BRSrc
Is there still
a need for the
Stall signal ?
Stall = (rf1D=wsE). (opcodeE=LWE).(wsE≠R0).re1D
+ (rf2D=wsE). (opcodeE=LWE).(wsE≠R0).re2D

Why an Instruction may not be dispatched every cycle (CPI>1)

- Full bypassing may be too expensive to implement
  - typically all frequently used paths provided
  - some infrequently used bypass paths may increase cycle time and/or cost more than their benefit in reducing CPI
- Loads have two cycle latency
  - Instruction after load cannot use load result
  - MIPS-I ISA defined *load delay slots*, a software-visible pipeline hazard (compiler schedules independent instruction or inserts NOP to avoid hazard). Removed in MIPS-II.
- Conditional branches may cause bubbles
  - kill following instruction(s) if no delay slots

(Machines with software-visible delay slots may execute significant number of NOP instructions inserted by compiler code scheduling)