18-447 Lecture 11: Pipelined Implementations: Hazards and Resolutions

James C. Hoe
Dept of ECE, CMU

February 25, 2009

Announcements: Project 1 due this week

Handouts:

Instruction Pipeline Reality

- Identical operations ... NOT!
  - ⇒ unifying instruction types
    - coalescing instruction types into one "multi-function" pipe
    - external fragmentation (some idle stages)
- Uniform Suboperations ... NOT!
  - ⇒ balance pipeline stages
    - stage quantization to yield balanced stages
    - internal fragmentation (some too-fast stages)
- Independent operations ... NOT!
  - ⇒ resolve data and resource hazards
    - duplicate contended resources
    - inter-instruction dependency detection and resolution

MIPS ISA features are engineered for improved pipelineability

Data Dependence

Data dependence

$r_3 \leftarrow r_1 \text{ op } r_2$
$r_5 \leftarrow r_3 \text{ op } r_4$

Read-after-Write (RAW)

Anti-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$
$r_1 \leftarrow r_4 \text{ op } r_5$

Write-after-Read (WAR)

Output-dependence

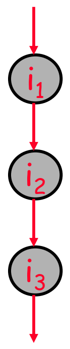
$r_3 \leftarrow r_1 \text{ op } r_2$
$r_5 \leftarrow r_3 \text{ op } r_4$
$r_3 \leftarrow r_6 \text{ op } r_7$

Write-after-Write (WAW)

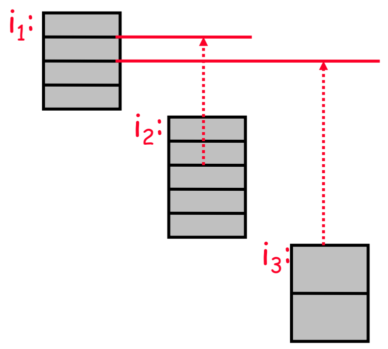
We discuss control-flow dependence in a later lecture

Dependencies and Pipelined Execution

Sequential and atomic instruction semantics



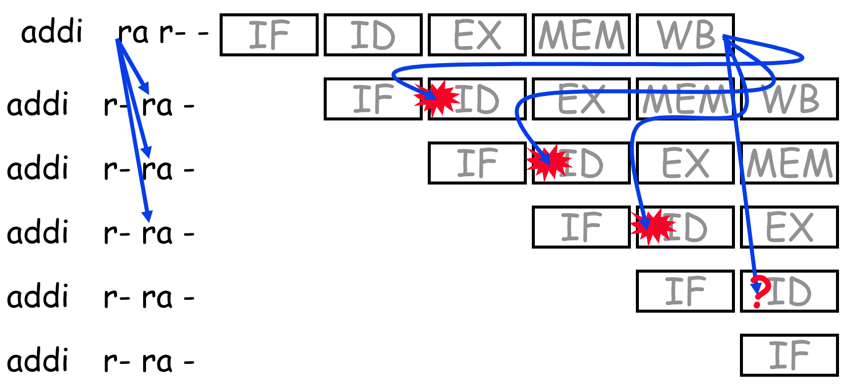
The true dependence between two instructions may only require ordering of certain sub-operations



This semantics is an overspecification. It defines what is correct but doesn't say to do it that way only

RAW Dependency and Hazard

- Following RAW dependencies lead to hazards in the 5-stage pipelined from L10

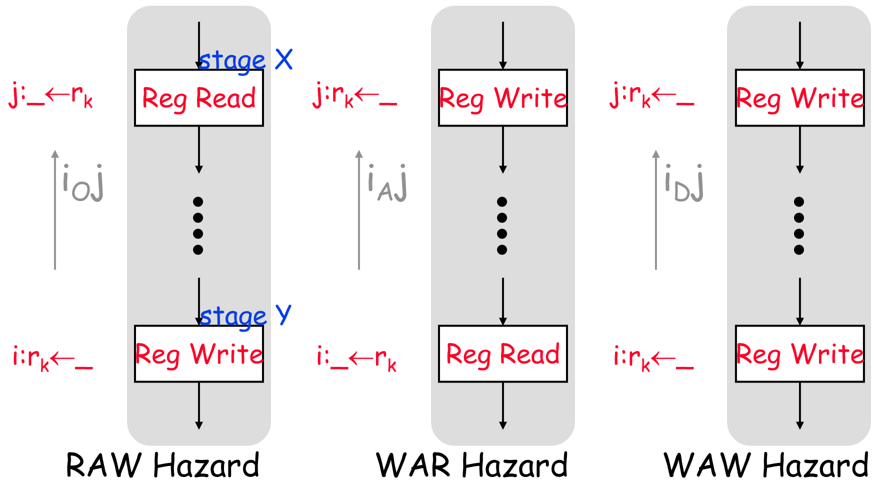


Register Data Hazard Analysis

	R/I-Type	LW	SW	Br	J	Jr
IF						
ID	read RF	read RF	read RF	read RF		read RF
EX						
MEM						
WB	write RF	write RF				

- For a given pipeline, when is there a register data hazard between 2 data dependent instructions?
  - dependence type: RAW, WAR, WAW?
  - instruction types involved?
  - distance between the two instructions?

Necessary Condition for Hazards



dist(i,j) $\leq$ dist(X,Y) $\Rightarrow$ Hazard!!
dist(i,j) > dist(X,Y) $\Rightarrow$ Safe

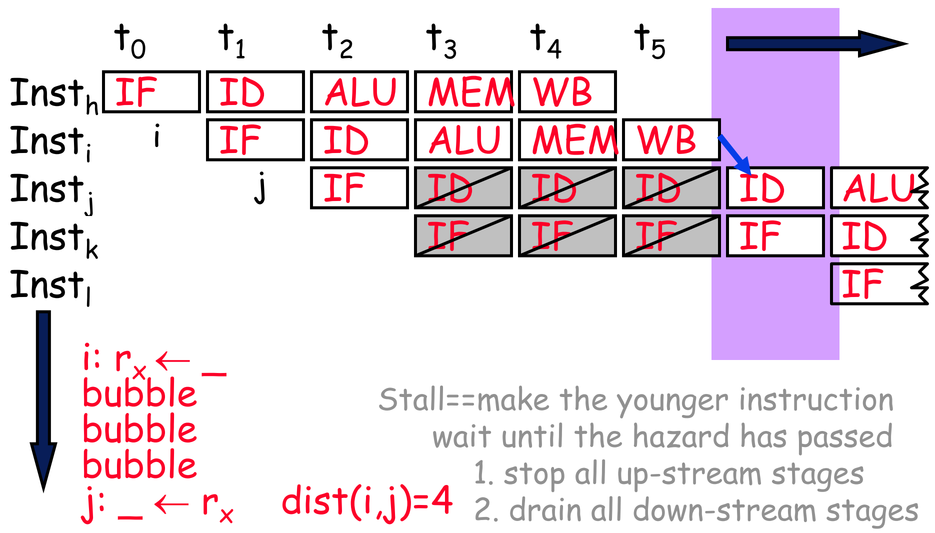
RAW Hazard Analysis Example

	R/I-Type	LW	SW	Br	J	Jr
IF						
ID	read RF	read RF	read RF	read RF		read RF
EX						
MEM						
WB	write RF	write RF				

- Instructions I_A and I_B (where I_A comes before I_B) have RAW hazard iff
  - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
  - dist(I_A, I_B) $\leq$ dist(ID, WB) = 3

What about WAW and WAR hazard?
What about memory data hazard?

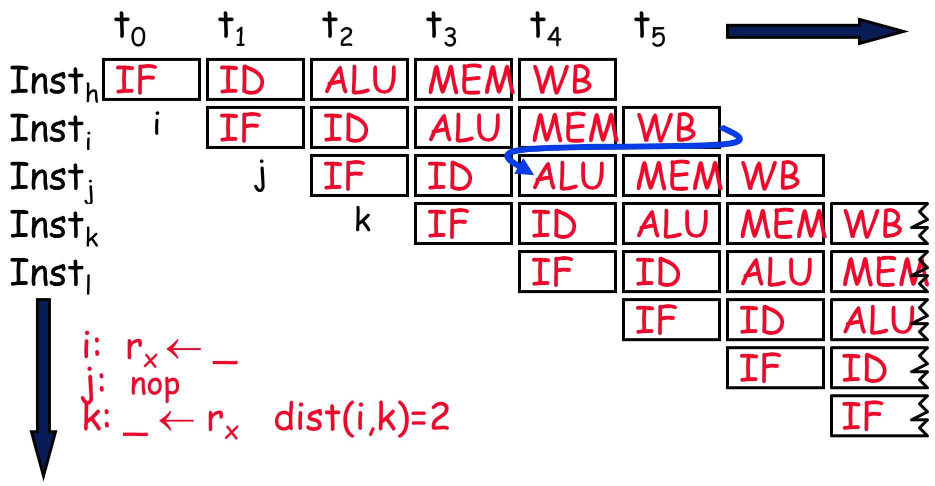
Pipeline Stall: a universal hazard resolution


t_0 t_1 t_2 t_3 t_4 t_5
$Inst_h$ IF ID ALU MEM WB
$Inst_i$ i IF ID ALU MEM WB
$Inst_j$ j IF ID ID ID ID ALU
$Inst_k$ IF IF IF IF ID
$Inst_l$ IF

i: $r_x \leftarrow$ _
bubble
bubble
bubble
j: _ $\leftarrow r_x$ dist(i,j)=4

Stall==make the younger instruction wait until the hazard has passed
1. stop all up-stream stages
2. drain all down-stream stages

What should happen in this case?


t_0 t_1 t_2 t_3 t_4 t_5
$Inst_h$ IF ID ALU MEM WB
$Inst_i$ i IF ID ALU MEM WB
$Inst_j$ j IF ID ALU MEM WB
$Inst_k$ k IF ID ALU MEM WB
$Inst_l$ IF ID ALU MEM
IF ID ALU
IF ID
IF

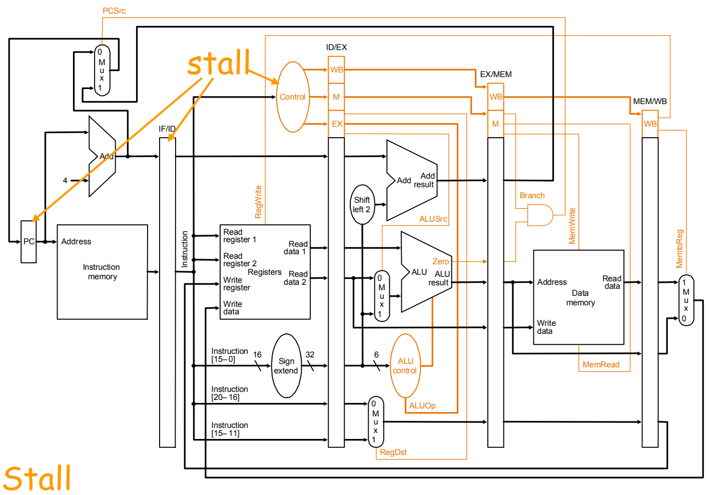
i: $r_x \leftarrow$ _
j: nop
k: _ $\leftarrow r_x$ dist(i,k)=2

Pipeline Stall

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	i	j	k	k	k	k	l				
ID	h	i	j	j	j	j	k	l			
EX		h	i	bub	bub	bub	j	k	l		
MEM			h	i	bub	bub	bub	j	k	l	
WB				h	i	bub	bub	bub	j	k	l

i: rx ← _
j: _ ← rx

Stall



- Stall
  - disable PC and IR latching
  - control should set RegWrite=0 and MemWrite=0

Based on figures from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Stall Conditions

- Instructions I_A and I_B (where I_A comes before I_B) have RAW hazard iff
  - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
  - $\text{dist}(I_A, I_B) \leq \text{dist}(\text{ID}, \text{WB}) = 3$

- In other words, must stall when I_B in ID stage wants to read a register to be written by I_A in EX, MEM or WB stage

Stall Condition

- Helper functions
  - rs(I) returns the rs field of I
  - use_rs(I) returns true if I requires RF[rs] and rs!=r0
- Stall when
  - (rs(IR_{ID})==$dest_{EX}$) && use_rs(IR_{ID}) && $RegWrite_{EX}$ or
  - (rs(IR_{ID})==$dest_{MEM}$) && use_rs(IR_{ID}) && $RegWrite_{MEM}$ or
  - (rs(IR_{ID})==$dest_{WB}$) && use_rs(IR_{ID}) && $RegWrite_{WB}$ or
  - (rt(IR_{ID})==$dest_{EX}$) && use_rt(IR_{ID}) && $RegWrite_{EX}$ or
  - (rt(IR_{ID})==$dest_{MEM}$) && use_rt(IR_{ID}) && $RegWrite_{MEM}$ or
  - (rt(IR_{ID})==$dest_{WB}$) && use_rt(IR_{ID}) && $RegWrite_{WB}$

It is crucial that the EX, MEM and WB stages continue to advance normally during stall cycles

Impact of Stall on Performance

- Each stall cycle corresponds to 1 lost ALU cycle
- For a program with N instructions and S stall cycles, Average IPC=N/(N+S)
- S depends on
  - frequency of RAW hazards
  - exact distance between the hazard-causing instructions
  - distance between hazards

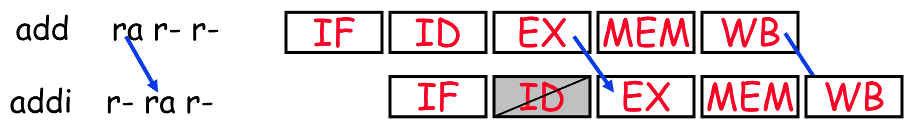
    suppose i_1,i_2 and i_3 all depend on i_0, once i_1's hazard is resolved, i_2 and i_3 must be okay too

Sample Assembly [p126, P&H]
for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { ...... }

```
                addi    $s1, $s0, -1           3 stalls
for2tst:        slti    $t0, $s1, 0            3 stalls
                bne     $t0, $zero, exit2
                sll     $t1, $s1, 2            3 stalls
                add     $t2, $a0, $t1          3 stalls
                lw      $t3, 0($t2)
                lw      $t4, 4($t2)            3 stalls
                slt     $t0, $t4, $t3          3 stalls
                beq     $t0, $zero, exit2
                .........
                addi    $s1, $s1, -1
                j       for2tst
exit2:
```

Data Forwarding or Register Bypassing

- It is intuitive to think of RF as state
  - "add rx ry rz" literally means get values from RF[ry] and RF[rz] respectively and put result in RF[rx]
- But, RF is just a part of a computing abstraction
  - "add rx ry rz" means 1. get the results of the last instructions to define the values of RF[ry] and RF[rz], respectively, and 2. until another instruction redefines RF[rx], younger instructions that refers to RF[rx] should use this instruction's result
- What matters is to maintain the correct "dataflow" between operations, thus



Resolving RAW Hazard by Forwarding

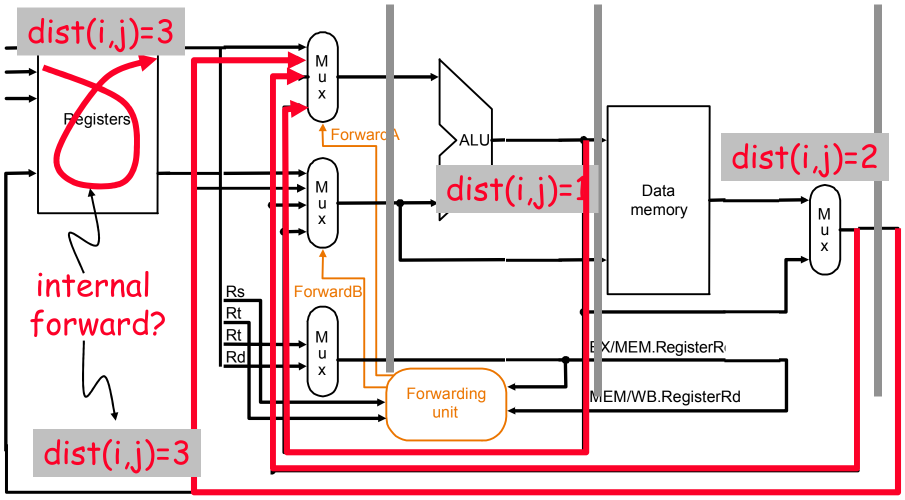
- Instructions I_A and I_B (where I_A comes before I_B) have RAW hazard iff
  - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
  - $\text{dist}(I_A, I_B) \leq \text{dist}(ID, WB) = 3$
- In other words, if I_B in ID stage reads a register written by I_A in EX, MEM or WB stage, then the operand required by I_B is not yet in RF

  ⇒ retrieve operand from datapath instead of the RF

  ⇒ retrieve operand from the youngest definition if multiple definitions are outstanding

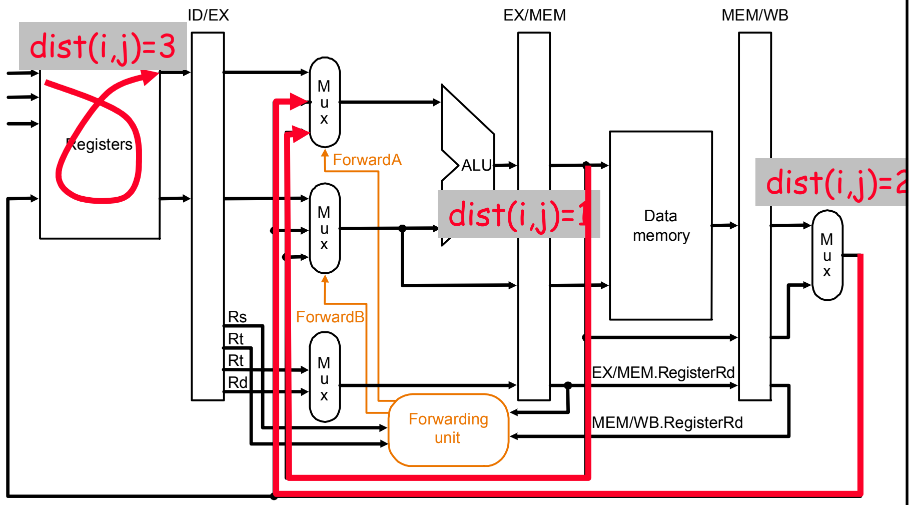
Electrical & Computer ENGINEERING

Forwarding Paths (v1)



Electrical & Computer ENGINEERING

Forwarding Paths (v2)



b. With forwarding

Assumes RF forwards internally

Forwarding Logic (for v2)

```
if ($rs_{EX}$!=0) && ($rs_{EX}$==$dest_{MEM}$) && $RegWrite_{MEM}$ then
    forward operand from MEM stage   // dist=1
else if ($rs_{EX}$!=0) && ($rs_{EX}$==$dest_{WB}$) && $RegWrite_{WB}$
    then
    forward operand from WB stage    // dist=2
else
    use $A_{EX}$ (operand from register file) // dist >= 3
```

Ordering matters!! Must check youngest match first

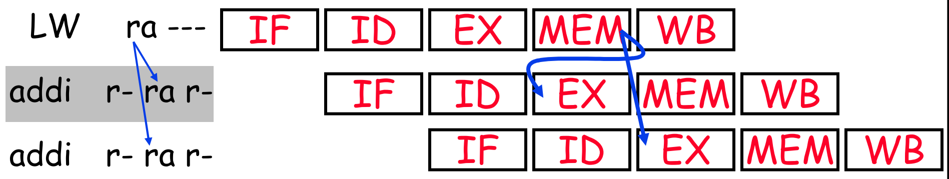
Why doesn't use_rs( ) appear in the forwarding logic?

Data Hazard Analysis (with Forwarding)

	R/I-Type	LW	SW	Br	J	Jr
IF						
ID						use
EX	use produce	use	use	use		
MEM		produce	(use)			
WB						

- Even with data-forwarding, RAW dependence on an immediate preceding LW instruction produces a hazard

Load Delay Slot



- R2000 defined load with arch. latency of 1 inst
  - the instruction immediately following a load (in the "delay slot") still sees the old register value (this is the behavior if we don't do anything special beyond forwarding)
  - ISA feature tailored to the 5-stage pipelined microarchitecture Warning!! Implementation exposed!!
- If loads are defined normally, i.e., atomic
  - a dependent immediate successor to LW must stall 1 cycle in ID
  - Stall = (rs(IR_{ID})==$dest_{EX}$) && use_rs(IR_{ID}) && $MemRead_{EX}$

Sample Assembly [p126, P&H]

for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { ...... }

```
          addi   $s1, $s0, -1
for2tst:  slti   $t0, $s1, 0
          bne    $t0, $zero, exit2
          sll    $t1, $s1, 2
          add    $t2, $a0, $t1
          lw     $t3, 0($t2)
          lw     $t4, 4($t2)
          nop
          slt    $t0, $t4, $t3
          beq    $t0, $zero, exit2
          .........
          addi   $s1, $s1, -1
          j      for2tst
exit2:
```

Terminology

- Dependencies
  - ordering requirement between instructions
- Pipeline Hazards:
  - (potential) violations of dependencies
- Hazard Resolution:
  - static $\Rightarrow$ schedule instructions at compile time to avoid hazards
  - dynamic $\Rightarrow$ detect hazard and adjust pipeline operation

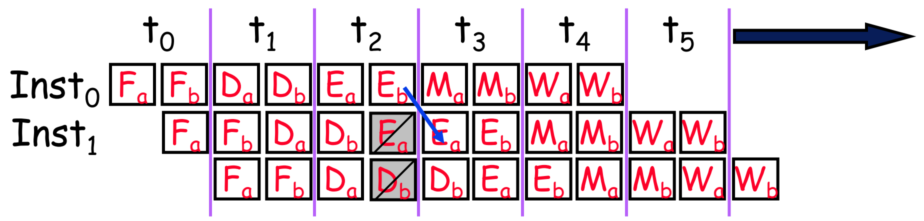
    Stall, Flush or Forward
- Pipeline Interlock:
  - hardware mechanisms for dynamic hazard resolution
  - detect and enforce dependences at run time

Why not very deep pipelines?

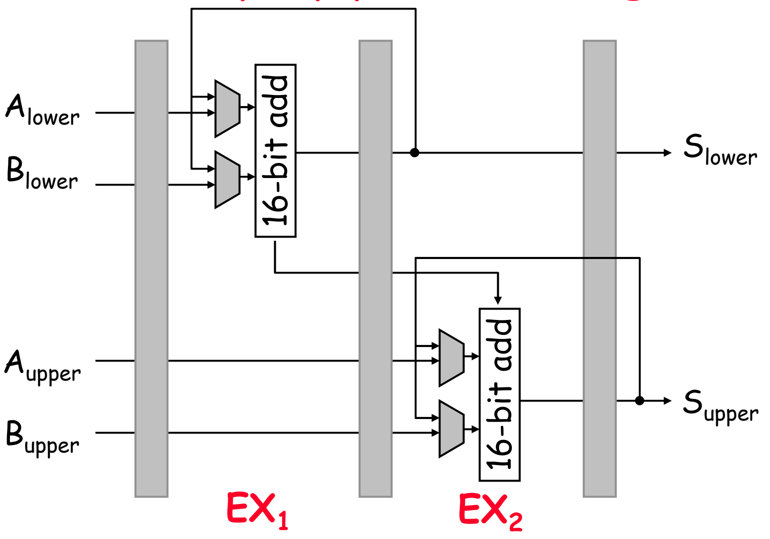
- 5-stage pipeline still has plenty of combinational delay between registers
- "Superpipelining" $\Rightarrow$ increase pipelining such that even intrinsic operations (e.g. ALU, RF access, memory access) require multiple stages
- What's the problem?

$Inst_0$: $r1 \leftarrow r2 + r3$

$Inst_1$: $r4 \leftarrow r1 + 2$

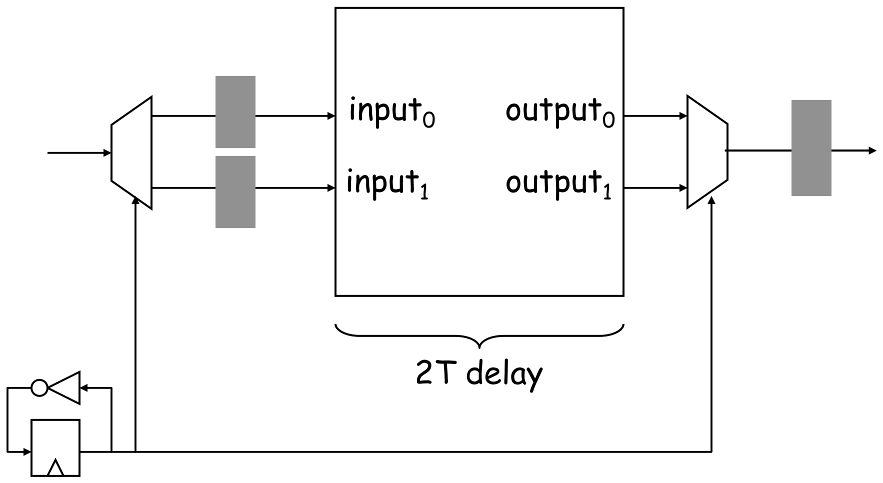

t_0
t_1
t_2
t_3
t_4
t_5
$Inst_0$
$Inst_1$
F_a F_b D_a D_b E_a E_b M_a M_b W_a W_b
F_a F_b D_a D_b E_a E_a E_b M_a M_b W_a W_b
F_a F_b D_a D_b D_b E_a E_b M_a M_b W_a W_b

Intel P4's Superpipelined Integer ALU



32-bit addition pipelined over 2 stages, BW=1/latency$_{16\text{-bit-add}}$
No stall between back-to-back dependencies

What if you really can't superpipeline?



If you can't double the bandwidth by pipelining, doubling the resource also doubles the bandwidth