18-747 Lecture 4: Simple Superscalar Execution

James C. Hoe
Dept of ECE, CMU
September 10, 2001

Reading Assignments: S&L Ch2 pp 51-76, Ch3 pp1-36, MJ Ch1,Ch2 (Ch3)

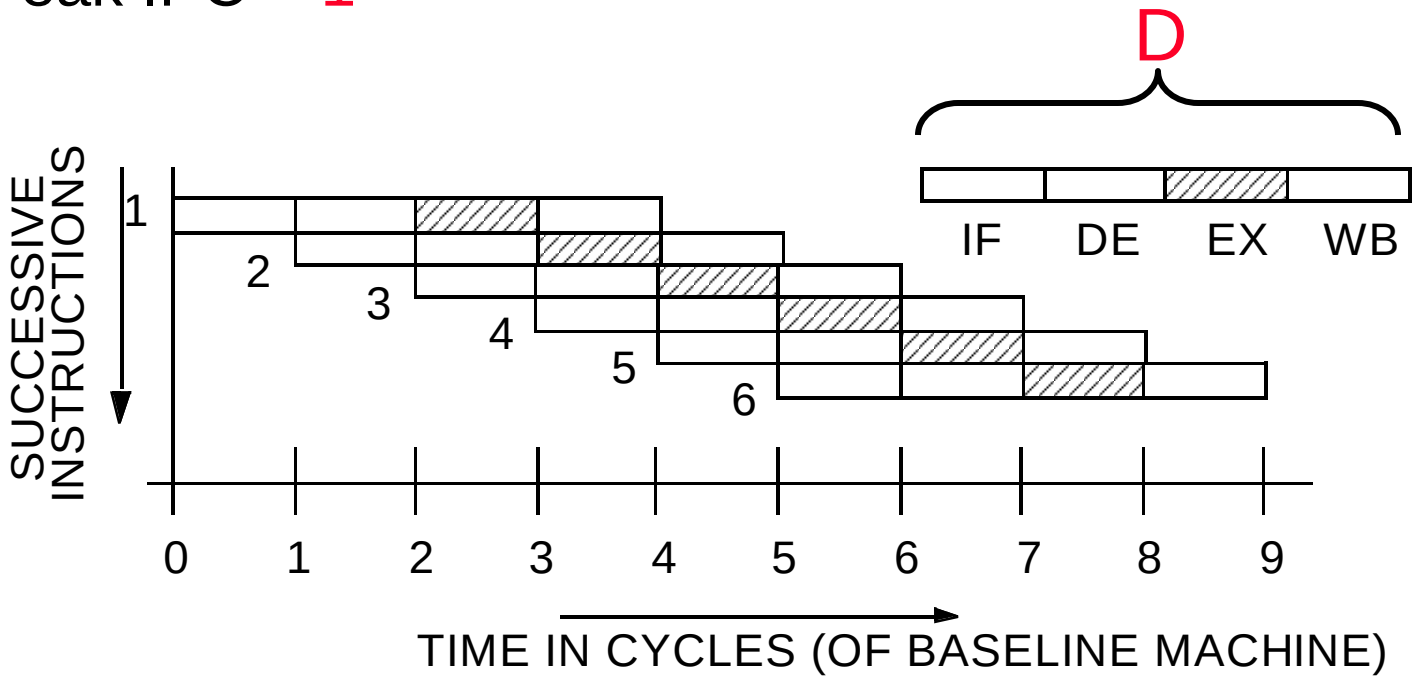
Announcements: First recitation this Friday, 2:30-3:30 DH1112 (this room)

Handouts: Handout #4 Project 0
SimpleScalar Tech. Report

Architectures for Instruction-Level Parallelism

Scalar Pipeline (baseline)

- Instruction Parallelism = D
- Operation Latency = 1
- Peak IPC = 1



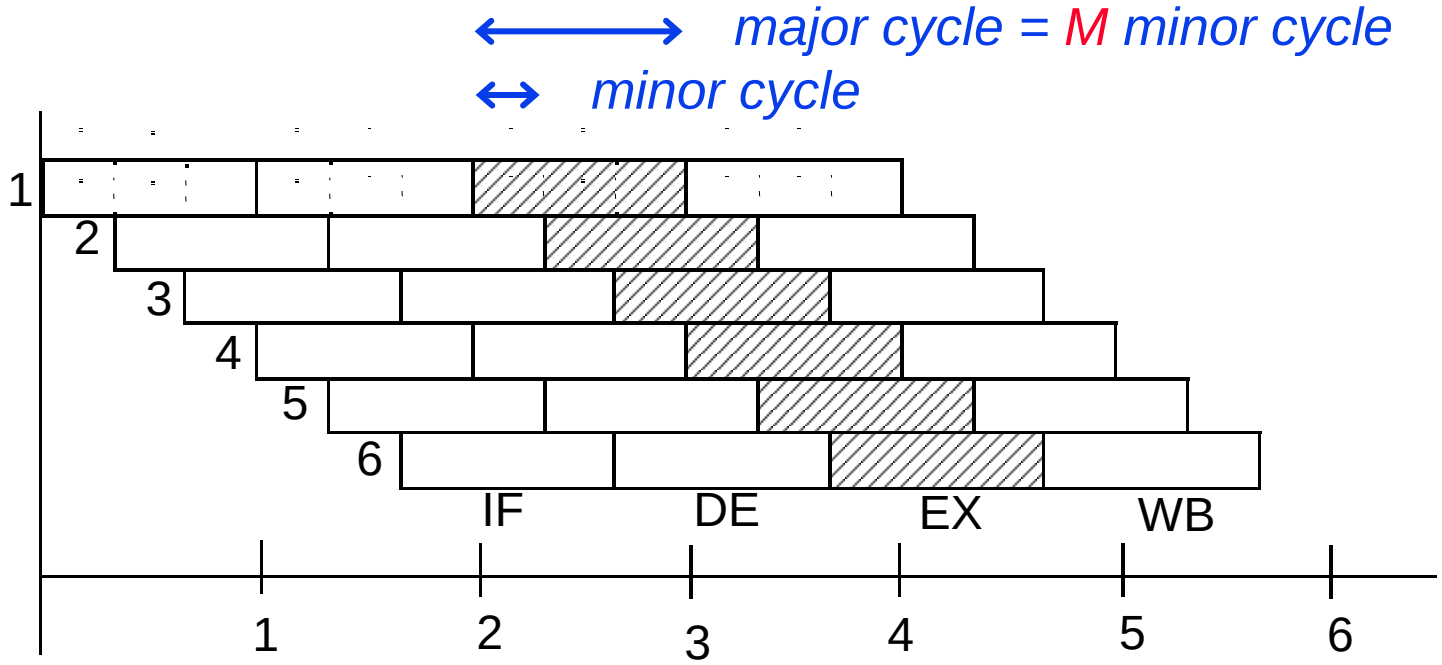
Superpipelined Machine

Superpipelined Execution

IP = *DxM*

OL = *M minor cycles*

Peak IPC = *1 per minor cycle (M per baseline cycle)*



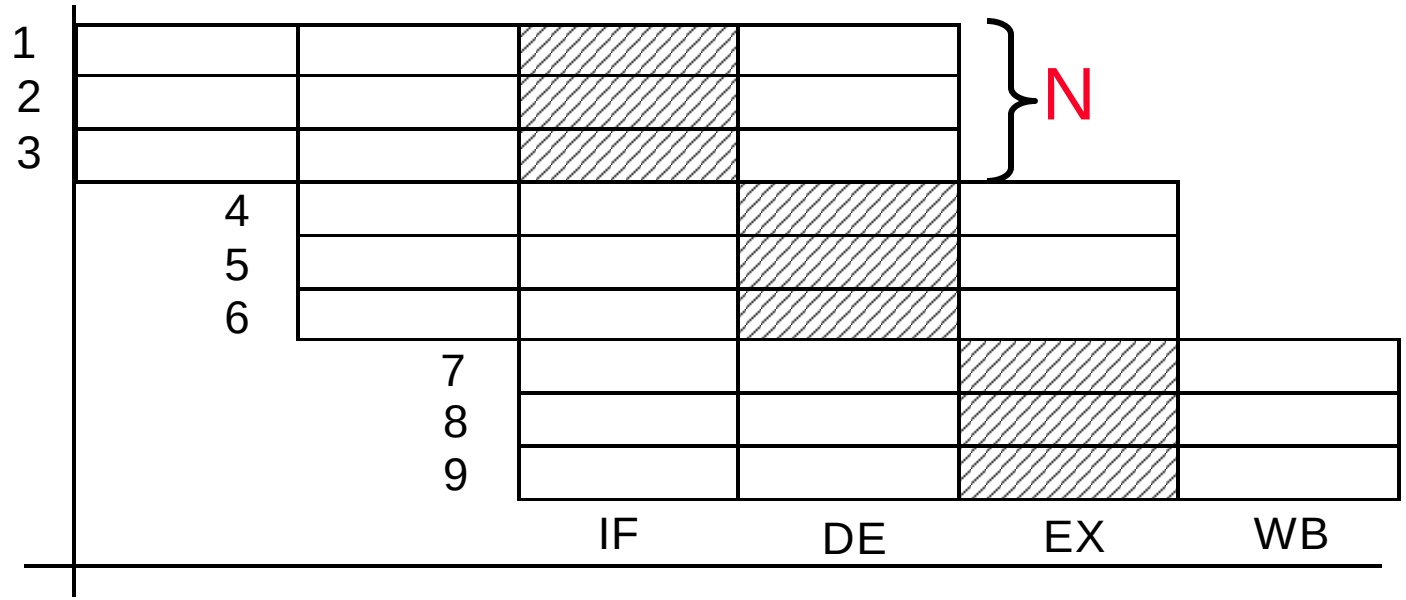
Superscalar Machines

Superscalar (Pipelined) Execution

IP = *DxN*

OL = *1 baseline cycles*

Peak IPC = *N per baseline cycle*



Superscalar and Superpipelined

Superscalar Parallelism

Operation Latency: 1

Issuing Rate: N

Superscalar Degree (SSD): N

(Determined by Issue Rate)

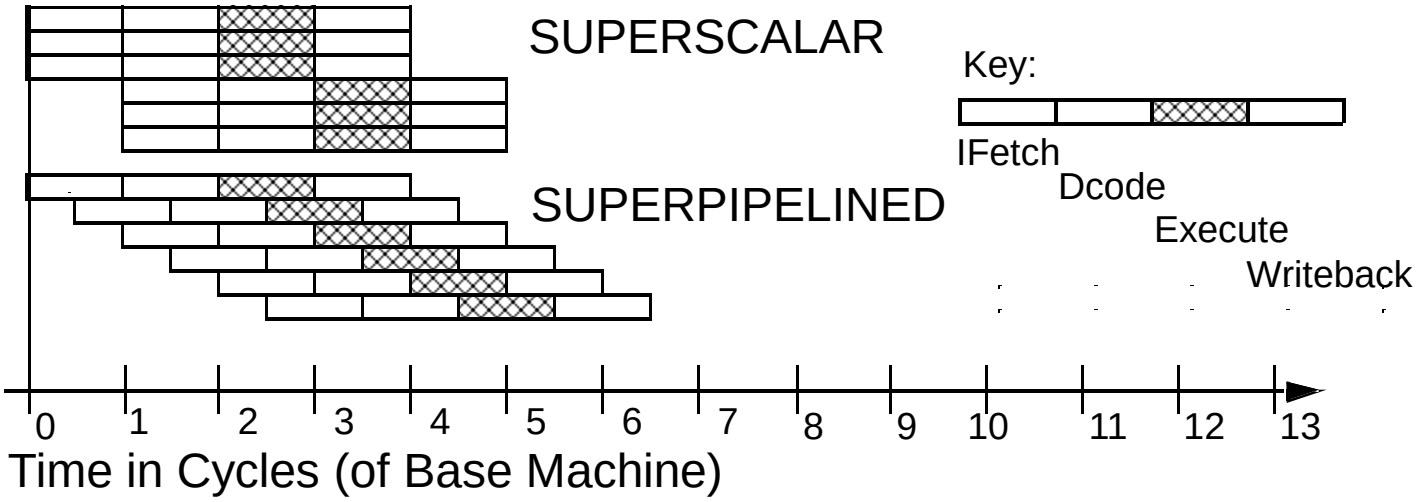
Superpipeline Parallelism

Operation Latency: M

Issuing Rate: 1

Superpipelined Degree (SPD): M

(Determined by Operation Latency)



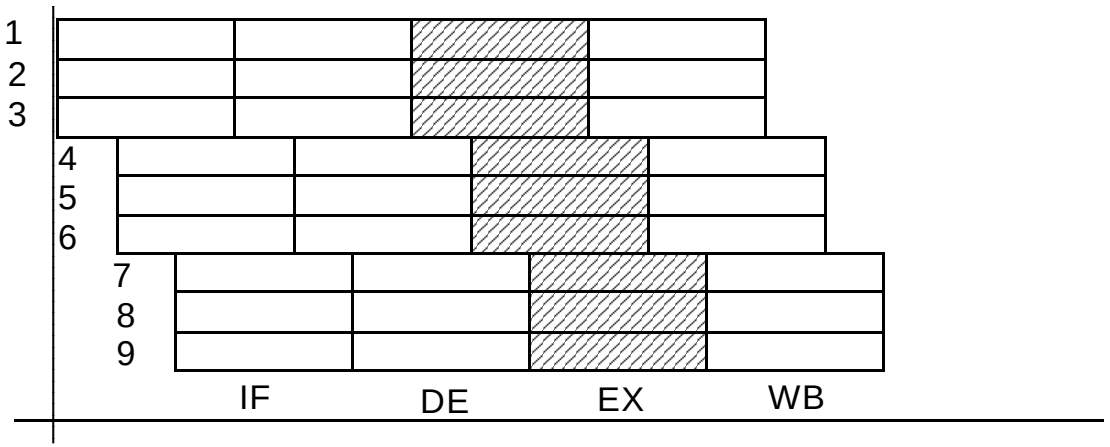
Superscalar and superpipelined machines of equal degree have roughly the same performance, i.e. if n = m then both have about the same IPC.

Limitations of Inorder Pipelines

- CPI of inorder pipelines degrades very sharply if the machine parallelism is increased beyond a certain point, *i.e. when NxM approaches average distance between dependent instructions*
- Forwarding is no longer effective

$\Rightarrow$ must stall more often

Pipeline may never be full due to frequent dependency stalls!!



What is Parallelism?

- Work

  T_1 - time to complete a computation on a sequential system

- Critical Path

  T_∞ - time to complete the same computation on an infinitely-parallel system

- Average Parallelism

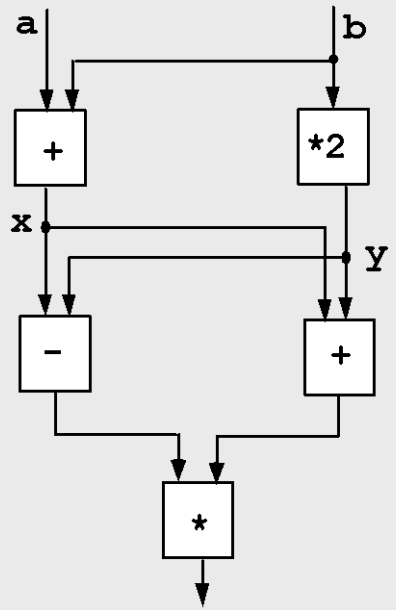
  $P_{avg} = T_1 / T_\infty$

- For a *p* wide system

$$T_p \geq \max\{ T_1/p, T_\infty \}$$

$$P_{avg} >> p \Rightarrow T_p \approx T_1/p$$

```
x = a + b;
y = b * 2
z =(x-y) * (x+y)
```



ILP: Instruction-Level Parallelism

- ILP is is a measure of the amount of inter-dependencies between instructions

- Average ILP = no. instruction / no. cyc required

  code1: ILP = 1

  i.e. must execute serially

  code2: ILP = 3

  i.e. can execute at the same time

```
code1:   r1 ← r2 + 1
         r3 ← r1 / 17
         r4 ← r0 - r3
```

```
code2:   r1 ← r2 + 1
         r3 ← r9 / 17
         r4 ← r0 - r10
```

Inter-instruction Dependences

- *Data dependence*

$r_3 \leftarrow r_1$ op r_2
$r_5 \leftarrow r_3$ op r_4

Read-after-Write (RAW)

- *Anti-dependence*

$r_3 \leftarrow r_1$ op r_2
$r_1 \leftarrow r_4$ op r_5

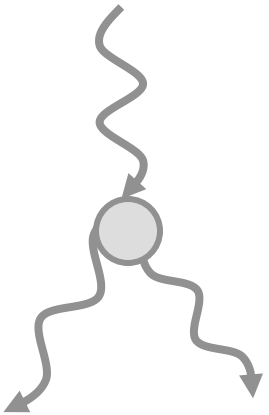
Write-after-Read (WAR)

- *Output dependence*

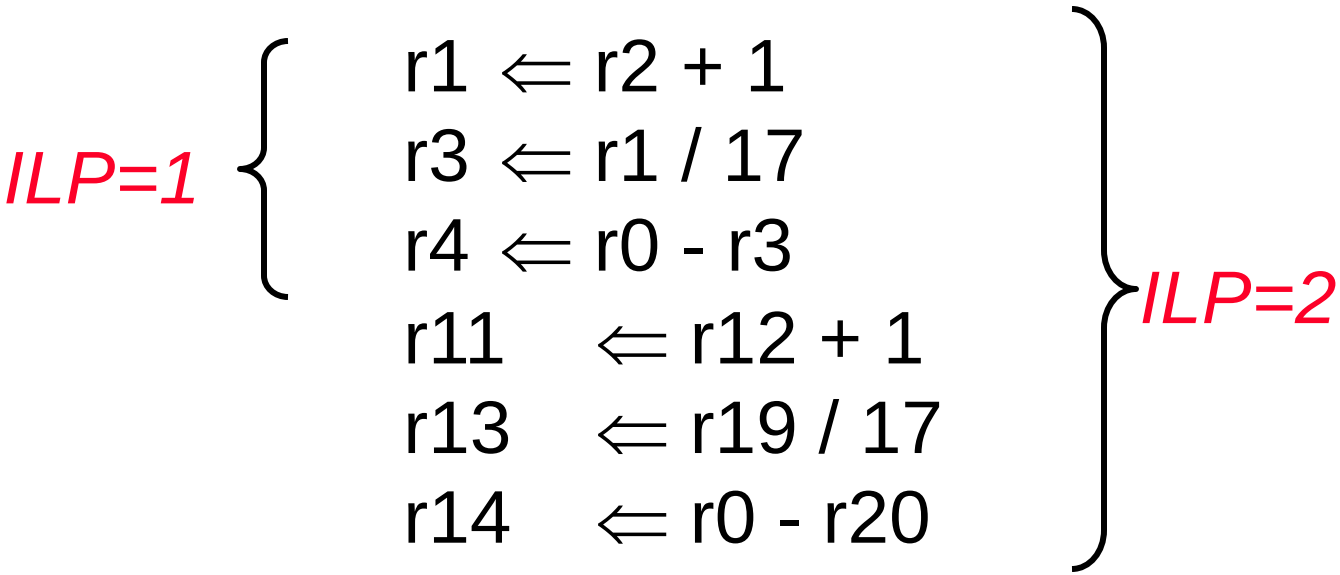
$r_3 \leftarrow r_1$ op r_2
$r_5 \leftarrow r_3$ op r_4
$r_3 \leftarrow r_6$ op r_7

Write-after-Write (WAW)

- Control dependence



Scope of ILP Analysis

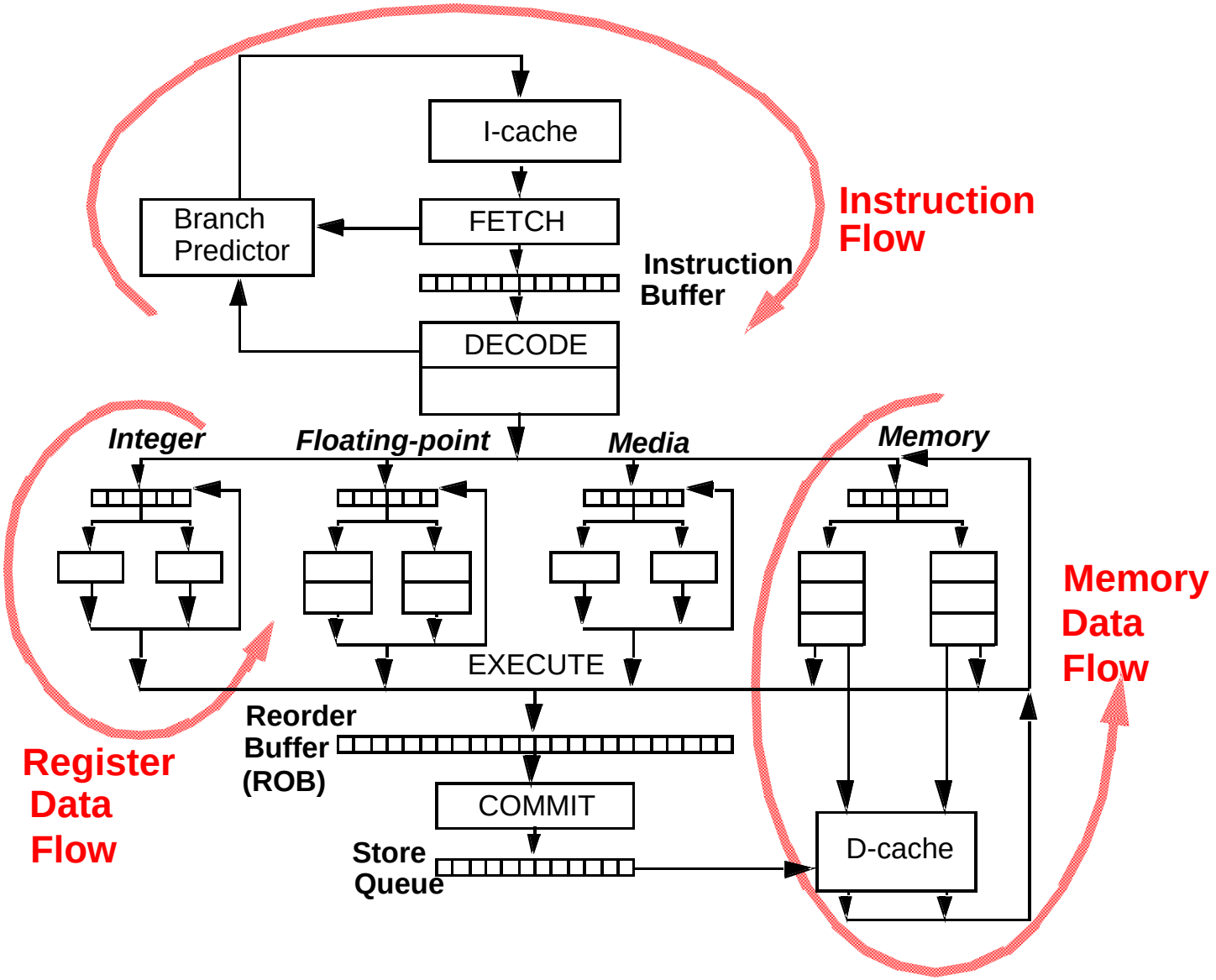


Out-of-order execution permits more ILP to be exploited

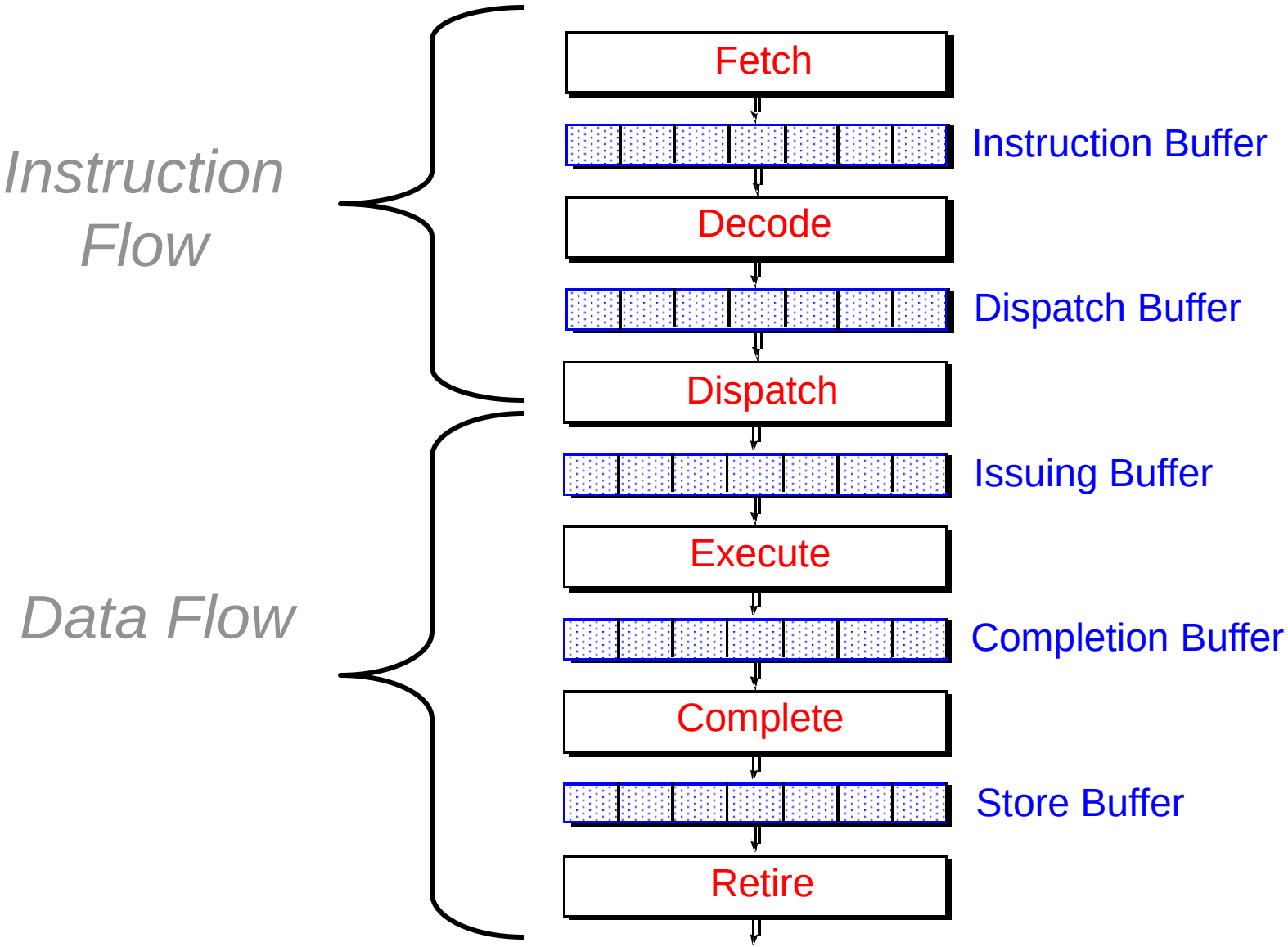
Purported Limits on ILP

Weiss and Smith [1984]	1.58
Sohi and Vajapeyam [1987]	1.81
Tjaden and Flynn [1970]	1.86
Tjaden and Flynn [1973]	1.96
Uht [1986]	2.00
Smith et al. [1989]	2.00
Jouppi and Wall [1988]	2.40
Johnson [1991]	2.50
Acosta et al. [1986]	2.79
Wedig [1982]	3.00
Butler et al. [1991]	5.8
Melvin and Patt [1991]	6
Wall [1991]	7
Kuck et al. [1972]	8
Riseman and Foster [1972]	51
Nicolau and Fisher [1984]	90

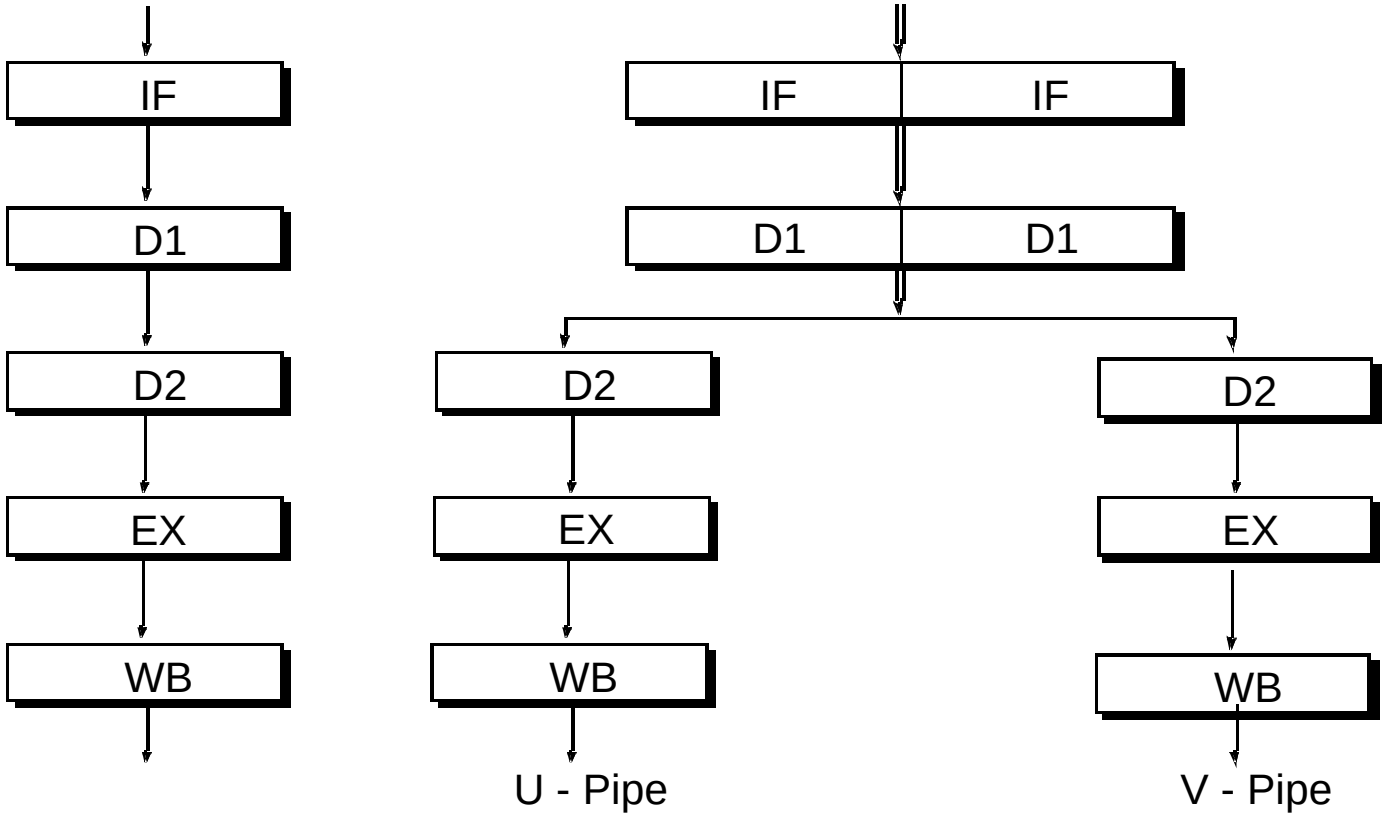
Flow Path Model of Superscalars



Superscalar Pipeline Design



Inorder Pipelines

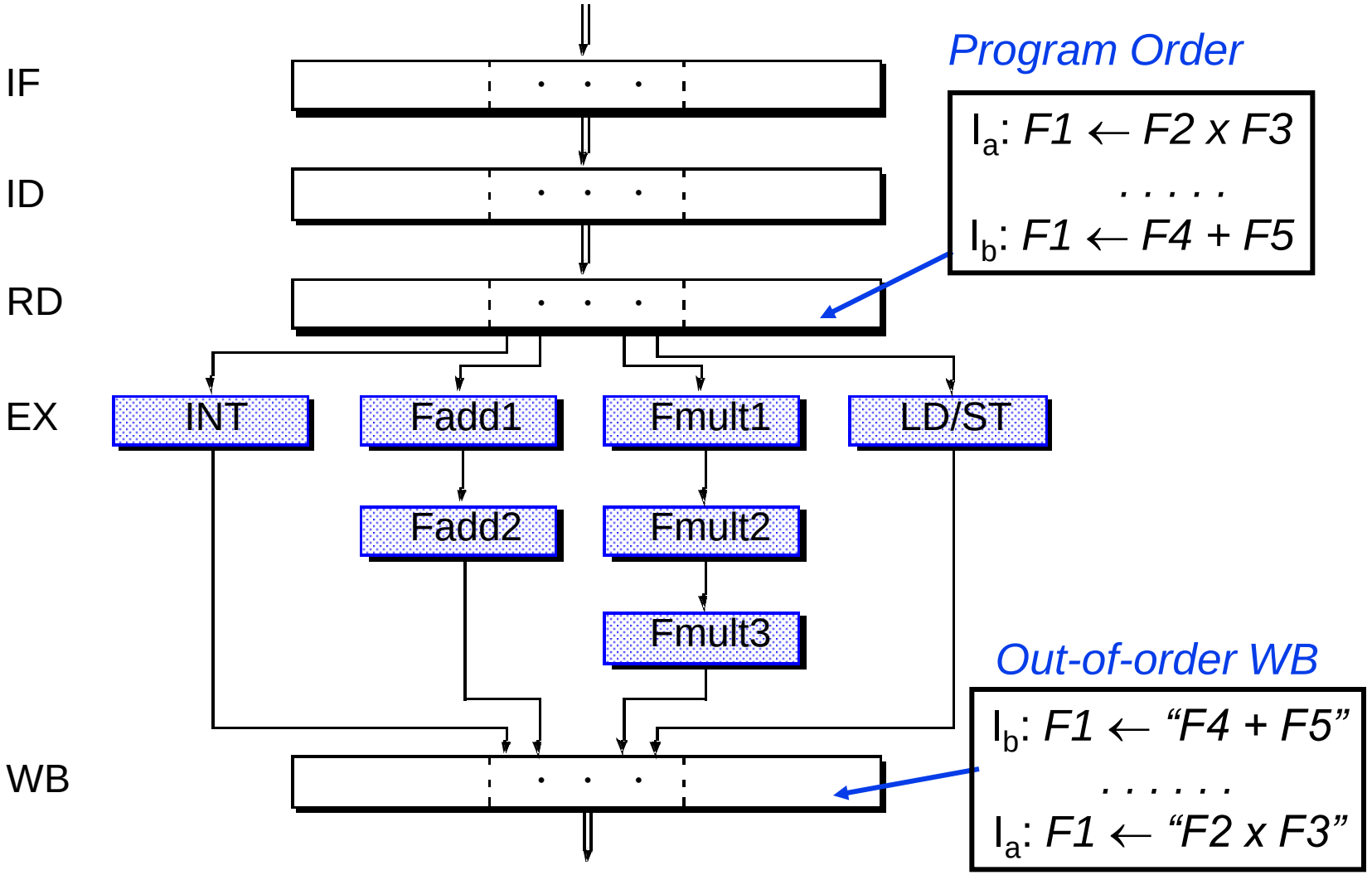


Intel i486

Intel Pentium

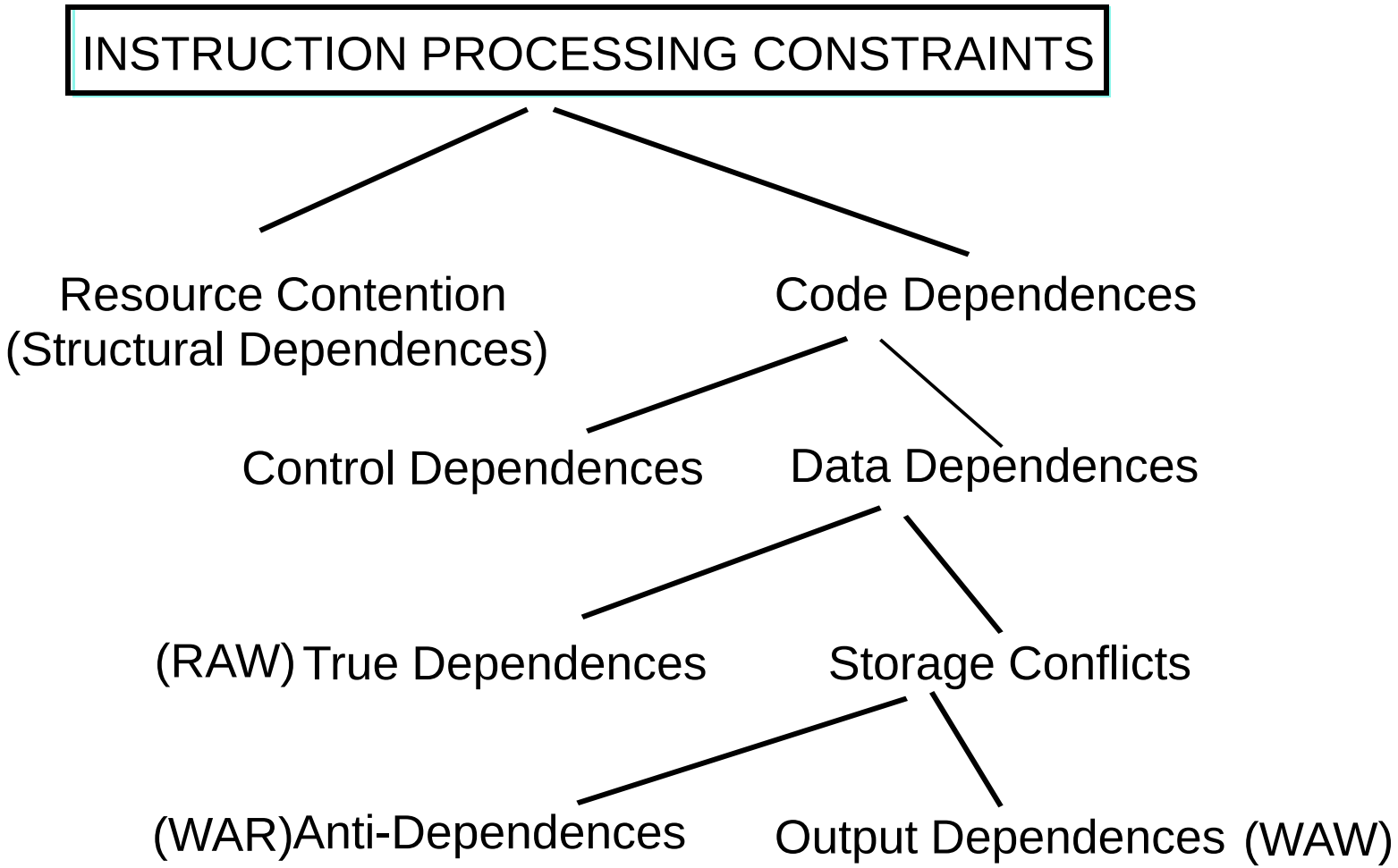
Inorder pipeline, no WAW no WAR (almost always true)

Out-of-order Pipelining 101

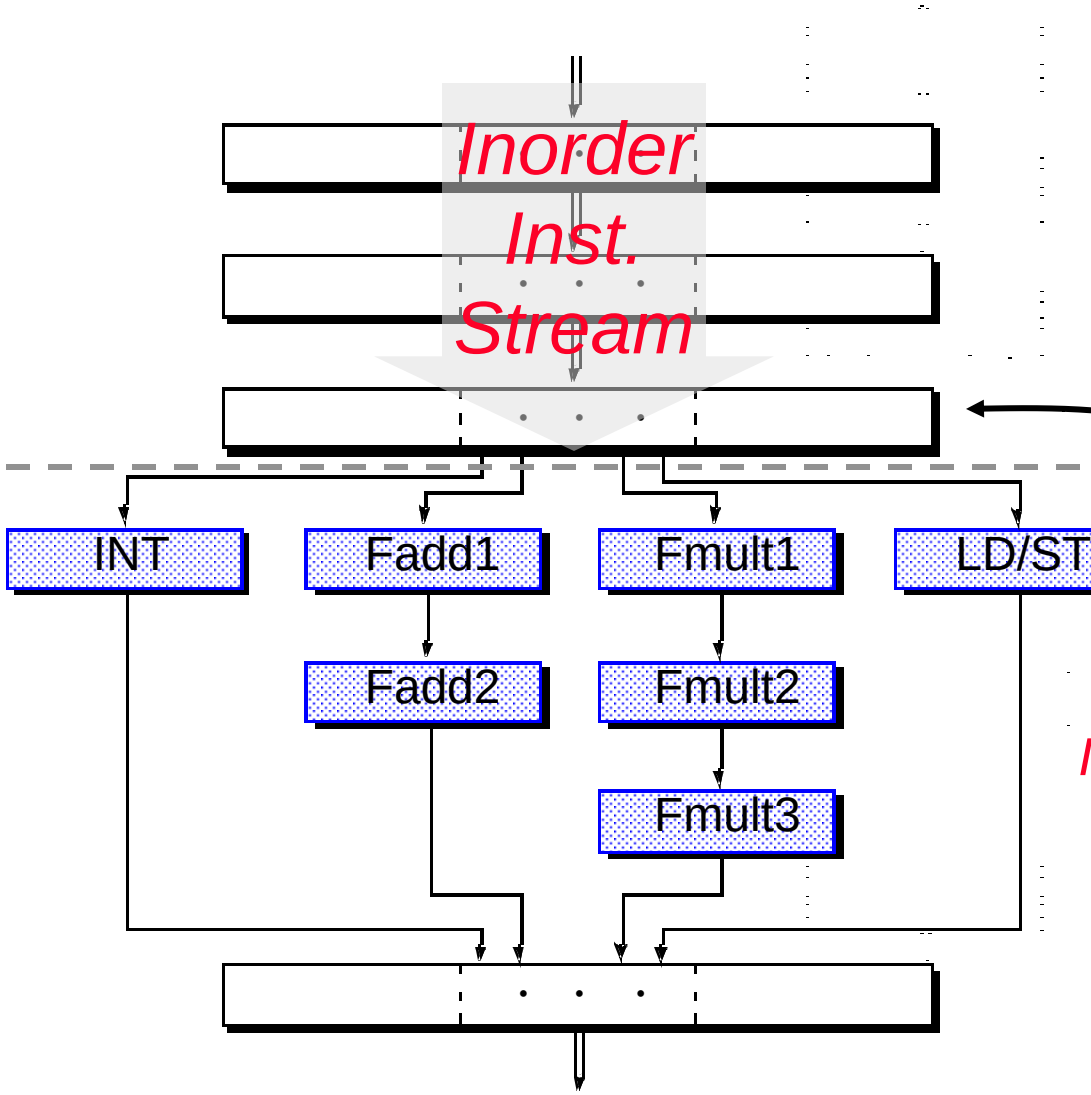


What is the value of F1? WAW!!!

Superscalar Execution Check List



In-order Issue into Diversified Pipelines



RD ← Fn (RS, RT)

Dest. Reg. | Func Unit | Source Registers

Issue stage needs to check:

1. *Structural Dependence*
2. *RAW Hazard*
3. *WAW Hazard*
4. *WAR Hazard*

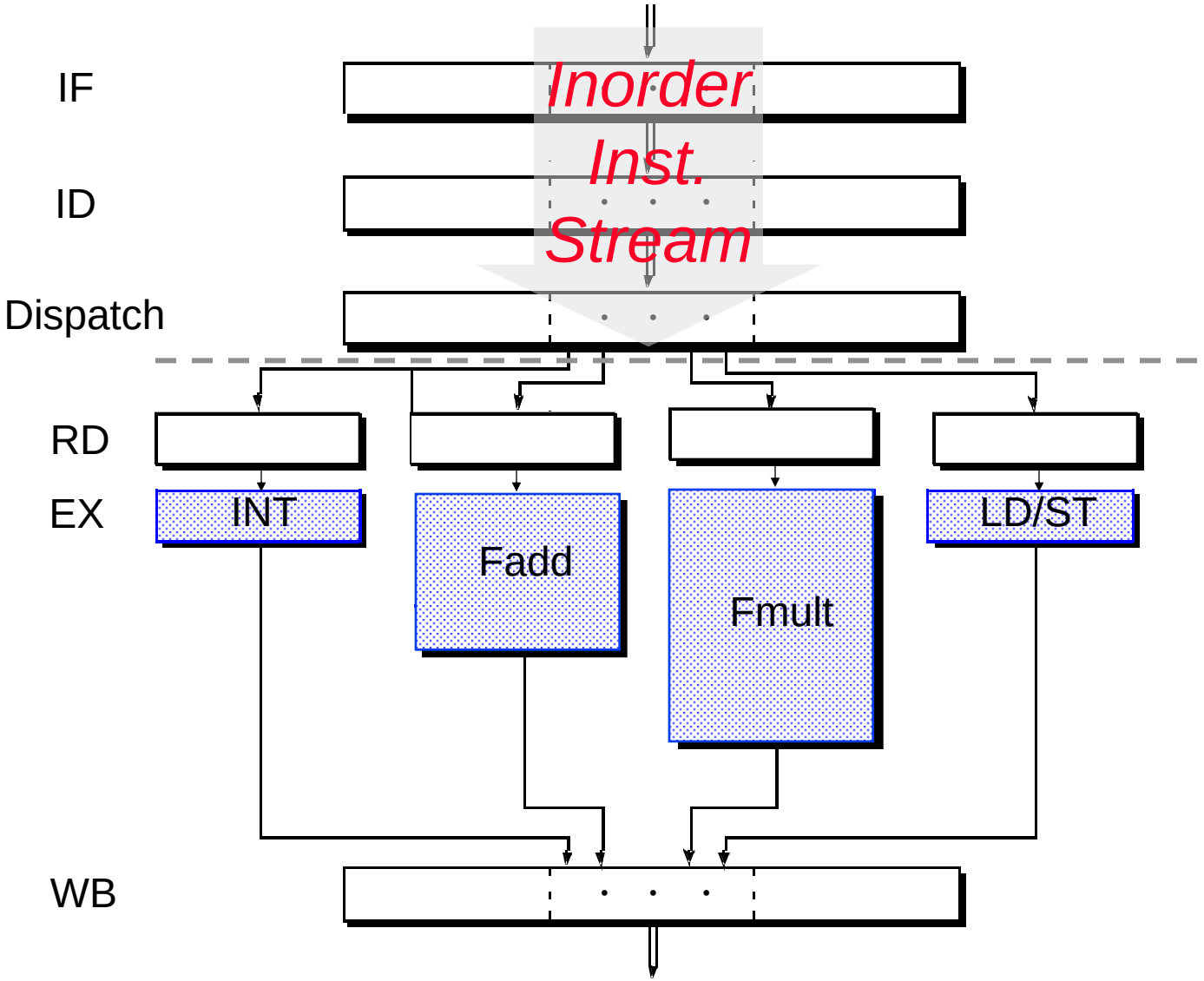
Simple Scoreboarding

- Scoreboard: a bit-array, 1-bit for each GPR
  - if the bit is not set, the register has valid data
  - if the bit is set, the register has stale data

    i.e. some outstanding inst is going to change it

- Dispatch in Order: RD ← Fn (RS, RT)
  - if SB[RS] or SB[RT] is set ⇒ RAW, stall
  - if SB[RD] is set is set ⇒ WAW, stall
  - else dispatch to FU, set SB[RD]
- Complete out-of-order
  - update GPR[RD], clear SB[RD]

Out-of-Order Issue



Scoreboarding for Out-of-Order Issue

- Scoreboard: one entry per GPR

  (what do we need to record?)

- Dispatch in order: "RD ← Fn (RS, RT)"
  - if FU is busy ⇒ structural hazard, stall
  - if SB[RD] is set is set ⇒ WAW, stall
  - if SB[RS] or SB[RT] is set is set ⇒ RAW *(what to do??)*
- Issue out-of-order: *(when?)*
- Complete out-of-order
  - update GPR[RD], clear SB[RD]

  (what about WAR?)

Scoreboard for Out-of-Order Issue

[*H&P pp242~251*]

RD ← Fn (RS, RT)”

Function Unit Status									
Name	Busy	Op	Fi_i	Fj	Fk	Qj	Qk	Rj	Rk
Integer		**Fn**	**RD**	**RS**	**RT**			**Yes**	**No**
FAdd									
FMult									
LD/ST									

Which FU is computing the new value if not ready?

Register Results Status (a.k.a Scoreboard)								
	R0	R1	R2	R3	R4	R5	R6	.....
FU								

Which FU is going to update the register?

Scoreboard Management: "RD ← Fn (RS, RT)"

Status	Wait until	Bookkeeping
Dispatch	not busy (FU) and not Result ('RD')	Busy(FU) ← yes; Op(FU) ← Fn; Fi(FU) ← 'RD'; Fj(FU) ← 'RS'; Fk(FU) ← 'RT'; Qj(FU) ← Result('RS'); Qk(FU) ← Result('RT'); Rj(FU) ← not Qj(FU); Rk(FU) ← not Qk(FU); Result('RD') ← FU;
Issue (Read operands)	Rj(FU) and Rk(FU)	Rj(FU) ← no; Rk(FU) ← no; Qj(FU) ← 0; Qk(FU) ← 0;
Execution Complete	Functional unit done	
Write Result	∀*f* ((Fj(*f*)≠Fi(FU) or Rj(*f*)==No) and (Fk(*f*)≠Fi(FU) or Rk(*f*)==No))	∀*f* (if Qj(*f*)==FU then Rj(*f*) ← yes); ∀*f* (if Qk(*f*)==FU then Rk(*f*) ← yes); Result(Fi(FU)) ← 0; Busy(FU) ← no;

Legends: FU -- the fxn unit used by the instruction;
Fj(X) -- content of entry Fj for fxn unit X;
Result(X) -- register result status entry for register X;

Scoreboarding Example 1/3

Instruction Status				
Instruction	Dispatch	Read Operands	Execution Complete	Write Result
LD F6, 43 (R2)	X	X	X	X
LD F2, 45(R3)	X	X	X	
MULTD F0, F2, F4	X			
SUBD F8, F6, F2	X			
DIVD F10, F0, F6	X			
ADDD F6, F8, F2				

Function Unit Status									
Name	Busy	Op	Fi_i	Fj	Fk	Qj	Qk	Rj	Rk
Integer (1)	Yes	LD	F2	R3				No	
Mult1(10)	Yes	MULTD	F0	F2	F4	Integer		No	Yes
Mult2(10)	No								
Add(2)	Yes	SUBD	F8	F6	F2		Integer	Yes	No
Div(40)	Yes	DIVD	F10	F0	F6	Mult1		No	Yes

Register Results Status (a.k.a Scoreboard)								
	F0	F2	F4	F6	F8	F10	F12	.....
FU	Mult1	Integer			Add	Divide		

Scoreboarding Example 2/3

Instruction Status				
Instruction	Dispatch	Read Operands	Execution Complete	Write Result
LD F6, 43 (R2)	X	X	X	X
LD F2, 45(R3)	X	X	X	X
MULTD F0, F2, F4	X	X	X	
SUBD F8, F6, F2	X	X	X	X
DIVD F10, F0, F6	X			
ADDD F6, F8, F2	X	X	X	

Function Unit Status									
Name	Busy	Op	Fi_i	Fj	Fk	Qj	Qk	Rj	Rk
Integer (1)	No								
Mult1(10)	Yes	MULTD	F0	F2	F4			No	No
Mult2(10)	No								
Add(2)	Yes	ADDD	F6	F8	F2			No	No
Div(40)	Yes	DIVD	F10	F0	F6	Mult1		No	Yes

Register Results Status (a.k.a Scoreboard)								
	F0	F2	F4	F6	F8	F10	F12	.....
FU	Mult1			Add		Divide		

Scoreboarding Example 3/3

Instruction Status				
Instruction	Dispatch	Read Operands	Execution Complete	Write Result
LD F6, 43 (R2)	X	X	X	X
LD F2, 45(R3)	X	X	X	X
MULTD F0, F2, F4	X	X	X	X
SUBD F8, F6, F2	X	X	X	X
DIVD F10, F0, F6	X	X	X	
ADDD F6, F8, F2	X	X	X	X

Function Unit Status									
Name	Busy	Op	Fi_i	Fj	Fk	Qj	Qk	Rj	Rk
Integer (1)	No								
Mult1(10)	No								
Mult2(10)	No								
Add(2)	No								
Div(40)	Yes	DIVD	F10	F0	F6			No	No

Register Results Status (a.k.a Scoreboard)								
	F0	F2	F4	F6	F8	F10	F12	.....
FU						Divide		

Limitations of Scoreboarding

- Consider a scoreboard processor with infinitely wide datapath

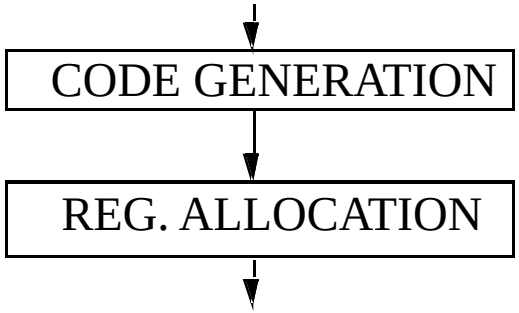
  In the best case, how many instructions can be simultaneously outstanding?

- Hints
  - no structural hazards
  - can always write a RAW-free code sequence

    addi r1,r0,1; addi r2,r0,1; addi r3,r0,1; …….
  - think about x86 ISA with only 8 registers

Contribution to Register Recycling

COMPILER REGISTER ALLOCATION



Single Assignment, Symbolic Reg.

Map Symbolic Reg. to Physical Reg.
Maximize Reuse of Reg.

INSTRUCTION LOOPS

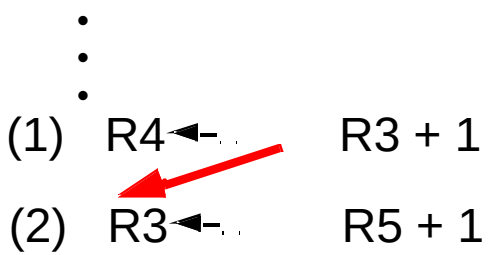
```
9   $34:  mul   $14   $7,    40
10        addu  $15,  $4,    $14
11        mul   $24,  $9,    4
12        addu  $25,  $15,   $24
13        lw    $11,  0($25)
14        mul   $12,  $9,    40
15        addu  $13,  $5,    $12
16        mul   $14,  $8,    4
17        addu  $15,  $13,   $14
18        lw    $24,  0($15)
19        mul   $25,  $11,   $24
20        addu  $10,  $10,   $25
21        addu  $9,   $9,    1
22        ble   $9,   10,    $34
```

For (k=1;k<= 10; k++)
t += a [i] [k] * b [k] [j] ;

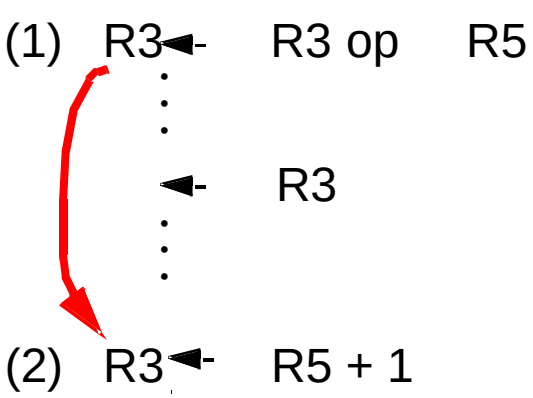
Reuse Same Set of Reg. in Each Iteration

Overlapped Execution of Different Iterations

Resolving False Dependences



Must Prevent (2) from completing before (1) is dispatched



Must Prevent (2) from completing before (1) completes

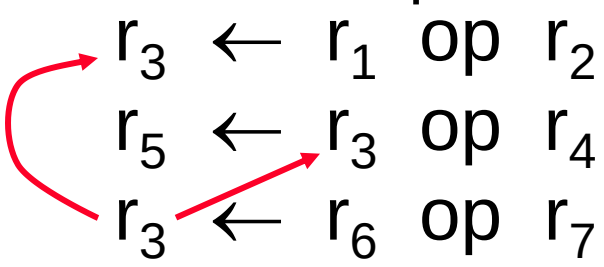
Stalling: delay Dispatching (or write back) of the 2nd instruction

Copy Operands: Copy not-yet-used operand to prevent being overwritten (WAR)

Register Renaming: use a different register (WAW & WAR)

Register Renaming

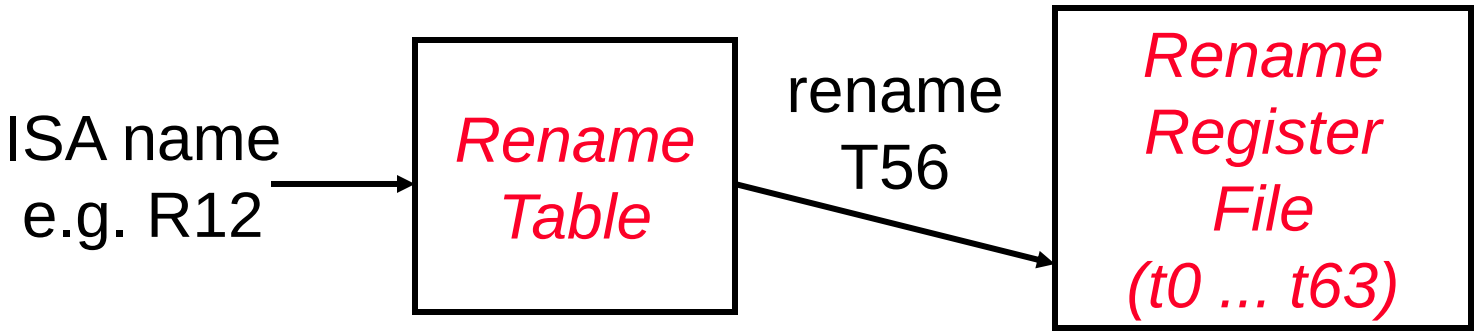
- Anti and output dependencies are false dependencies



- The dependence is on name/location rather than data
- Given infinite number of registers, anti and output dependencies can always be eliminated

Original	Renamed
r1 ← r2 / r3	r1 ← r2 / r3
r4 ← r1 * r5	r4 ← r1 * r5
r1 ← r3 + r6	r8 ← r3 + r6
r3 ← r1 - r4	r9 ← r8 - r4

Hardware Register Renaming



- maintain bindings from ISA reg. names to rename registers
- When issuing an instruction that updates 'RD':
  - allocate an unused rename register TX
  - recording binding from 'RD' to TX
- When to remove a binding? When to de-allocate a rename register?

R1 ← R2 / R3
R4 ← R1 * R5
R1 ← R3 + R6

To be continued next lecture!!