

Computer Architecture

Instruction-Level Parallelism

College of Computer Science
Chongqing University

Instruction-Level Parallelism

- ILP: overlap the execution of instructions
 - partially (through pipelining)
 - completely (through issuing on multiple functional units)
- Exploiting ILP
 - dynamically and hardware-intensive
 - mostly desktop and server markets (Pentium, MIPS, Alpha...)
 - static and software-intensive
 - embedded system markets (but also, Itanium...)
 - in practice these techniques are often combined
- Exploiting ILP and basic blocks
 - in a typical MIPS program, the average dynamic branch frequency is between 15% and 25% -> average length of a basic block is between 4 and 7 instructions:
 - necessary to exploit ILP across multiple basic blocks

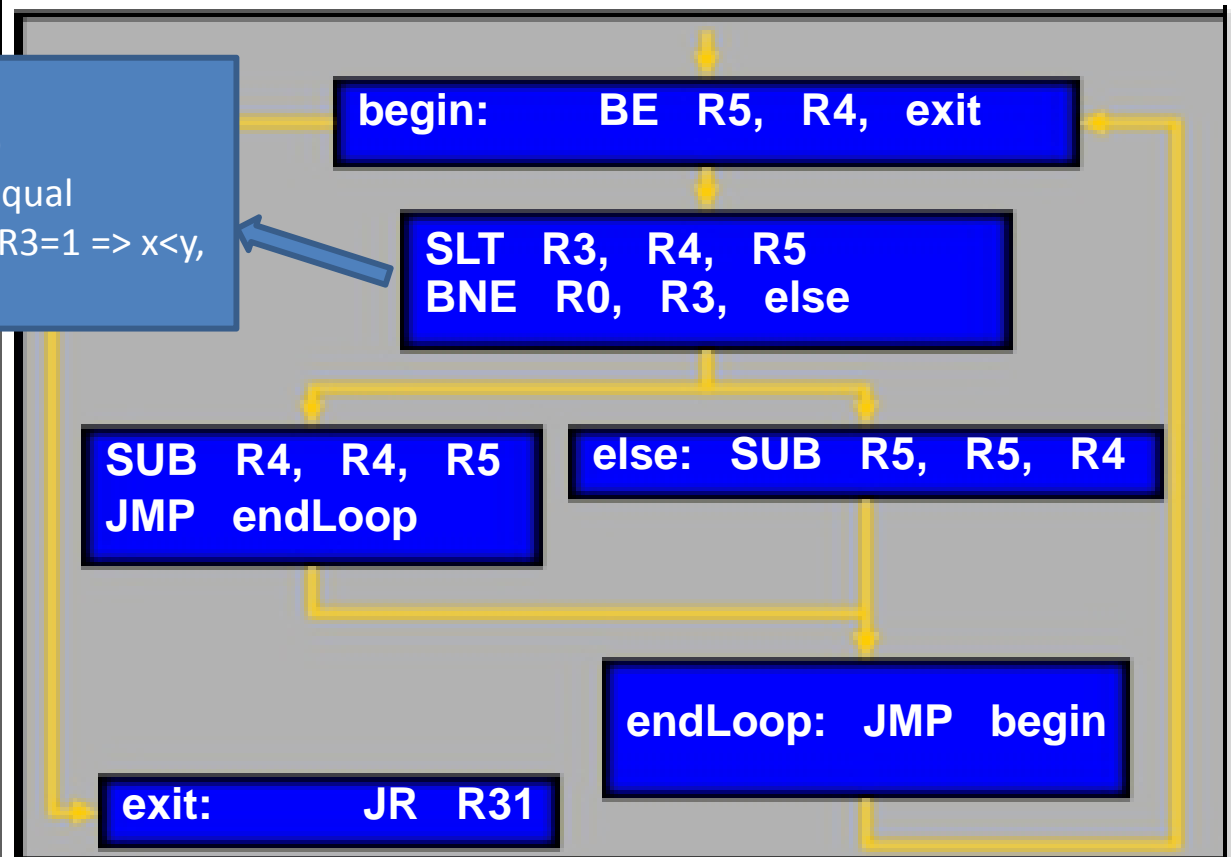
Background: Control Flow and Basic Blocks

```
int gcd (int x, int y) {  
    while (x != y) {  
        if (x < y)  
            y = y - x;  
        else  
            x = x - y;  
    }  
    return x;  
}
```

- **Basic Block** is a sequence of instructions without jump/branches (except possibly at the end) and without branch targets/labels (except possibly at the beginning)
- **Control-Flow Graph**

SLT: set on less than
(R3=1 when $x < y$)
BNE: branch if not equal
(when $R3 \neq 0 \Rightarrow R3=1 \Rightarrow x < y$,
go to else)

```
; R4 <- X, R5 <- y  
begin:    BE R5, R4, exit  
          SLT R3, R4, R5  
          BNE R0, R3, else  
          SUB R4, R4, R5  
          JMP endLoop  
else:     SUB R5, R5, R4  
endLoop:  JMP begin  
exit:     JR R31
```



Reducing CPI (or Increasing IPC)

$$\text{CPI} = \text{CPI}_{\text{ideal}} + \text{stalls}_{\text{structural}} + \text{stalls}_{\text{data Hazard}} + \text{stalls}_{\text{control}}$$

technique	reduces
forwarding/ bypassing	potential data-hazard stalls
delayed branches	control-hazard stalls
basic dynamic scheduling (scoreboarding)	data-hazard stalls from true dependencies
dynamic scheduling with register renaming	data-hazard, anti-dep. & output dep. stalls
dynamic branch prediction	control stalls
issuing multiple instruction per clock cycle	ideal CPI
speculation	data-hazard and control-hazard stalls
dynamic memory disambiguation	data-hazard stalls with memory
loop unrolling	control hazard stalls
basic compiler pipeline scheduling	data-hazard stalls
compiler dependency analysis	ideal CPI, data-hazard stalls
software pipelining & trace scheduling	ideal CPI, data-hazard stalls
compiler speculation	ideal CPI, data-hazard stalls

Dependences vs. Hazards

- If two instructions depend on each other
 - they must be executed in order
 - at most, they can be partially overlapped
- Dependences are a property of programs
 - a data dependency in the instruction sequence reflects a data dependency in the original source code
- Based on the given pipeline implementation, a data dependency may result in an actual hazard being detected and a possible stall (which reduces or eliminates the instruction overlap)
 - e.g. moving the branch test for the MIPS from the EX to ID does reduce the branch delay to one, but it may cause a stall in presence of a data dependency

```
DADDIU R1, R8, -8  
BNE R1, R2, Loop
```

Dependences vs. Hazards (cont.)

- Dependency \equiv hazard potential
 - occurrence of actual hazard and length of any stall is a property of the pipeline implementation
- Dependences determine the order in which results must be computed
- Dependences set an upper bound on the amount of instruction-level parallelism that can be exploited
- Strategies to overcome dependences
 - maintain them, but avoid hazards
 - eliminate them by transforming the code

Dynamic HW Scheduling: Motivations

```
DIV.D F0, F2, F4  
ADD.D F6, F0, F8  
SUB.D F8, F8, F14
```

- SUB.D does not depend on the previous instructions but it can not execute because ADD.D is stalling due to RAW hazard
- With **dynamic scheduling**, the HW rearranges the instruction execution order to reduce the stalls while maintaining data flow and exception behavior
 - it simplifies compiling
 - code compiled for a given pipeline can be run efficiently on another
 - it handles cases where dependences are unknown at compile time (due to memory references)
 - it is the basis for hardware speculative execution

Dynamic HW Scheduling vs. Static Pipeline Scheduling

- **Dynamic Scheduling (HW)**
 - tries to avoid stalls when dependences (which could generate hazards) are present
 - does not change the data flow
- **Static Scheduling (SW)**
 - the compiler tries to minimize stalls by separating dependent instructions so that they will not generate hazards
- **The two techniques can be combined as statically scheduled code can run on a processor with a dynamically scheduled pipeline**

Name Dependencies and the New Hazards due to Out-of-Order Execution

- **True data dependency:** an instruction produces a value for a following instruction
 - may lead to a **RAW hazard**
- **Name dependence:** no real value must be transmitted between two instructions, but they both use the same register or memory location
 - anti-dependence
 - may lead to a **WAR hazard**
 - output dependence
 - may lead to a **WAW hazard**

RAW: true dependency

```
DIV.D F0, F2, F4  
ADD.D F6, F0, F8
```

WAR: anti-dependency

```
ADD.D F6, F0, F8  
SUB.D F8, F8, F14
```

WAW: output dependency

```
ADD.D F6, F0, F8  
MUL.D F6, F10, F9
```

Control Dependencies

- Every instruction, except for those in the very first basic block of a program, is control-dependent on some set of branches
 - it cannot be moved before the branches on which it depends
 - it cannot be moved after the branches on which it doesn't depend
- Control dependences do not necessarily need to be preserved, but exception behavior and data flow must be preserved

```
DADDU R2, R3, R4
BEQZ  R2, T1
LW    R1, 0(R2)
T1: ...
```

LW may not be executed,
depending on BEQZ

```
DADDU R1, R2, R3
BEQZ  R4, T1
DSUBU R1, R5, R6
T1: ...
OR    R7, R1, R8
```

The value of R1 depends
on the execution of BEQZ

Dynamic Scheduling & Data Hazards: The Scoreboard Approach

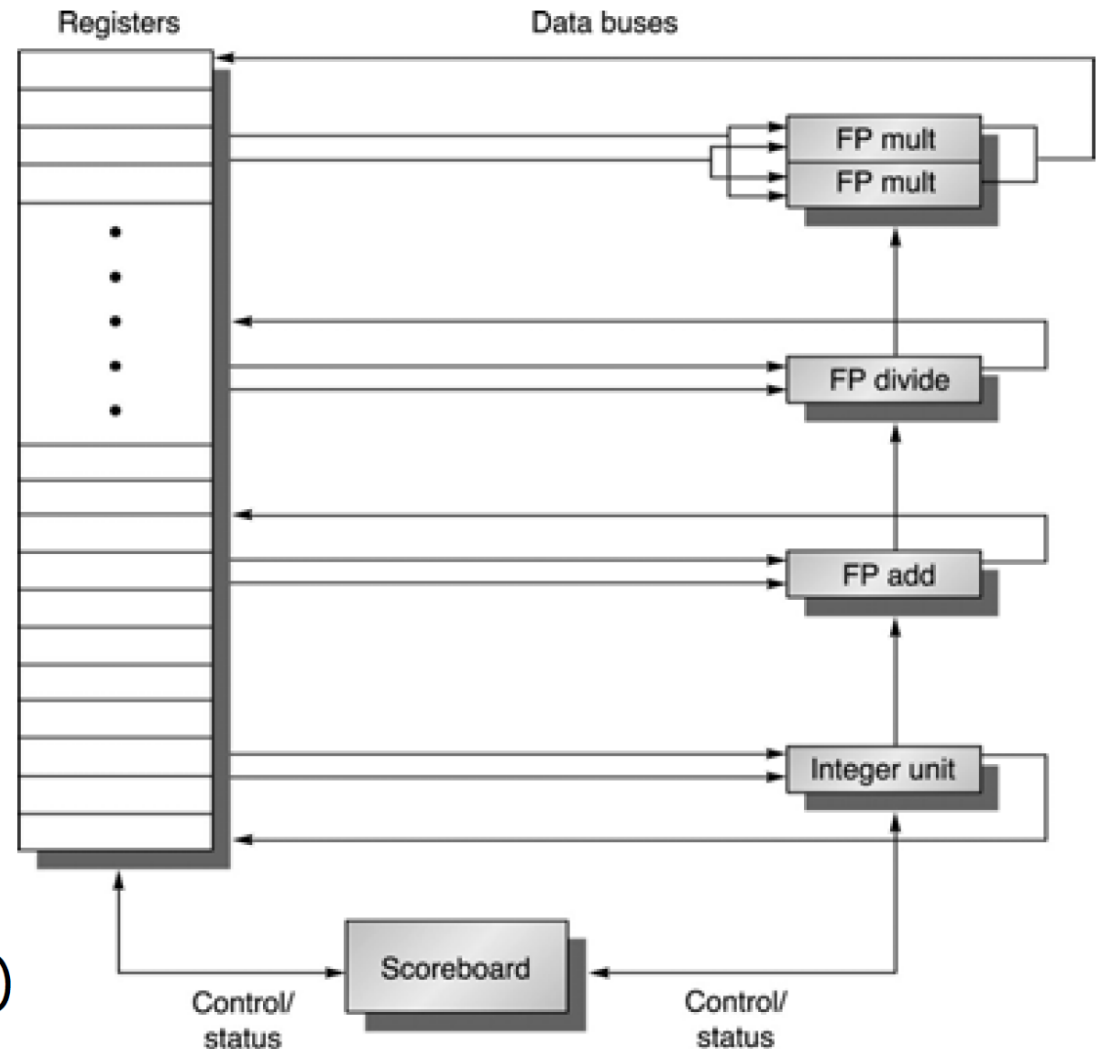
DIV.D	F0, F2, F4
ADD.D	F6, F0, F8
SUB.D	F8, F8, F14
MUL.D	F6, F10, F9

- true dependency between DIV.D and ADD.D (RAW hazard)
- anti-dependency between ADD.D and SUB.D (may lead to WAR hazard)
- output dependency between MUL.D and ADD.D (may lead to WAW hazard)

- Goal: to maintain $CPI \approx 1$ with out-of-order execution by
 - issuing an instruction ASAP
 - taking care of issuing/execution, including all hazard detections
 1. checking for structural hazards
 2. waiting for absence of data hazards
- Scoreboard was introduced in 1963 with CDC 6600
 - 7 units for integer operations
 - 5 memory reference units
 - 4 FP units

Assumption: Basic Architecture of MIPS Processor with Scoreboard

- Simpler structure than CDC 6600 original one
- For MIPS, focus on FP units (other units have small latencies)
- Assume
 - 1 integer unit
 - integer operations
 - memory references
 - branches
 - 1 FP adder (2 cycles)
 - 2 FP multipliers (10 cycles)
 - 1 FP divider (40 cycles)



MIPS Pipeline Stages with Scoreboard

- All hazard detection and resolution is centralized in the *scoreboard module*
 - when operands can be read
 - when execution can start
 - when result can be written
- In-order issuing, out-of-order execution and commit
- The ID, EX, and WB stages are replaced by four new stages

1. Issue (ID-1)

- in-order instruction decoding
- check/stall for structural hazards
- check/stall for WAW hazards

2. Read Operands (ID-2)

- wait for each operand availability
- resolve RAW hazards dynamically
- no forwarding, but wait for WB

3. Execution (EX)

- out-of-order execution
- inform scoreboard upon completion

4. Write Result (WB)

- check/stall for WAR hazards
- write result into register asap
- no statically-assigned write slot


MIPS Scoreboard Components

Instruction	Issue	Read Operands	Exec. Complete	Write Result
L.D F6, 34(R2)	<div>Instruction Status</div>			
L.D F2, 45(R3)				
MUL.D F0, F2, F4				
SUB.D F8, F6, F2				
DIV.D F10, F0, F6				
ADD.D F6, F8, F2				

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	<div>Functional Unit Status</div>								
Mult-1	<div>is it busy?</div>	<div>operation</div>	<div>destination register</div>	<div>source registers</div>		<div>functional units producing values for source registers</div>		<div>flags indicating when Fj and Fk are READY AND NOT READ YET</div>	
Mult-2									
Add									
Divide									

	F0	F2	F4	F6	F8	F10	...	F30
FU	<div>Register Result Status (RRS)</div>							

Scoreboard: Checking & Bookkeeping per Instruction (op D , $S1$, $S2$)

Instruction Status	Wait until	Bookkeeping
1. Issue	not Busy[FU] and not RRS[D]	$Busy[FU] \leftarrow true$; $Op[FU] \leftarrow op$; $Fi[FU] \leftarrow D$; $Fj[FU] \leftarrow S1$; $Fk[FU] \leftarrow S2$; $Qj \leftarrow RRS[S1]$; $Qk \leftarrow RRS[S2]$; $Rj \leftarrow (Qj == 0)$; $Rk \leftarrow (Qk == 0)$; $RRS[D] \leftarrow FU$
2. Read Operands	$Rj = true$ and $Rk = true$	$Rj \leftarrow false$; $Rk \leftarrow false$; $Qj \leftarrow 0$; $Qk \leftarrow 0$
3. Execution Complete	FU is done executing	
4. Write Result	$\forall f \{ \quad (Fj[f] \neq Fi[FU] \text{ or } Rj[f] = false) \quad \& \quad (Fk[f] \neq Fi[FU] \text{ or } Rk[f] = false) \quad \}$ 	$\forall f ((Qj[f] = FU) \Rightarrow Rj[f] \leftarrow true)$ $\forall f ((Qk[f] = FU) \Rightarrow Rk[f] \leftarrow true)$ $RSS[Fi[FU]] \leftarrow 0$; $Busy[FU] \leftarrow false$

If $(Fj[f] == Fi[Fu] \ \&\& \ Rj[f] == true)$, it cannot write, as another instruction has not read the old value yet --> avoid WAR

Scoreboard Example:

Clock Cycle = 1

Instruction		Issue	Read Operands		Exec. Complete		Write Result		
L.D F6, 34(R2)		1							
L.D F2, 45(R3)									
MUL.D F0, F2, F4									
SUB.D F8, F6, F2									
DIV.D F10, F0, F6									
ADD.D F6, F8, F2									

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	yes	L.D	F6		R2				true
Mult-1	no								
Mult-2	no								
Add	no								
Divide	no								

	F0	F2	F4	F6	F8	F10	...	F30
FU				Integer				

- issue first instruction

Scoreboard Example:

Clock Cycle = 1

Scoreboard: Checking & Bookkeeping per Instruction (op D, S1, S2)					Exec. Complete		Write Result	
Instruction Status	Wait until		Bookkeeping					
1. Issue	not Busy[FU] and not RRS[D]		Busy[FU] \leftarrow true; Op[FU] \leftarrow op; Fi[FU] \leftarrow D; Fj[FU] \leftarrow S1; Fk[FU] \leftarrow S2; Qj \leftarrow RRS[S1]; Qk \leftarrow RRS[S2]; Rj \leftarrow (Qj==0); Rk \leftarrow (Qk==0); RRS[D] \leftarrow FU					
2. Read Operands	Rj=true and Rk=true		Rj \leftarrow false; Rk \leftarrow false; Qj \leftarrow 0; Qk \leftarrow 0		Qj	Qk	Rj	Rk
3. Execution Complete	FU is done executing							true
4. Write Result	$\forall f \{ (Fj[f] \neq Fi[FU] \text{ or } Rj[f] = \text{false}) \text{ \& } (Fk[f] \neq Fi[FU] \text{ or } Rk[f] = \text{false}) \}$		$\forall f ((Qj[f] = FU) \Rightarrow Rj[f] \leftarrow \text{true})$ $\forall f ((Qk[f] = FU) \Rightarrow Rk[f] \leftarrow \text{true})$ RSS[Fi[FU]] \leftarrow 0; Busy[FU] \leftarrow false					
Add	no							
Divide	no							
	F0	F2	F4	F6	F8	F10	...	F30
FU				Integer				

- issue first instruction

Scoreboard Example:

Clock Cycle = 2

Instruction	Issue	Read Operands	Exec. Complete	Write Result
L.D F6, 34(R2)	1	2		
L.D F2, 45(R3)				
MUL.D F0, F2, F4				
SUB.D F8, F6, F2				
DIV.D F10, F0, F6				
ADD.D F6, F8, F2				

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	yes	L.D	F6		R2				false
Mult-1	no								
Mult-2	no								
Add	no								
Divide	no								

	F0	F2	F4	F6	F8	F10	...	F30
FU				Integer				

- read operand; second load instruction cannot be issued (struct. hazard)

Scoreboard Example:

Clock Cycle = 2

Scoreboard: Checking & Bookkeeping per Instruction (op D, S1, S2)						Exec. Complete		Write Result	
Instruction Status	Wait until		Bookkeeping						
1. Issue	not Busy[FU] and not RRS[D]		Busy[FU] \leftarrow true; Op[FU] \leftarrow op ; Fi[FU] \leftarrow D; Fj[FU] \leftarrow S1 ; Fk[FU] \leftarrow S2 ; Qj \leftarrow RRS[S1]; Qk \leftarrow RRS[S2]; Rj \leftarrow (Qj==0); Rk \leftarrow (Qk==0); RRS[D] \leftarrow FU						
2. Read Operands	Rj=true and Rk=true		Rj \leftarrow false; Rk \leftarrow false; Qj \leftarrow 0; Qk \leftarrow 0			Qj	Qk	Rj	Rk
3. Execution Complete	FU is done executing								false
4. Write Result	$\forall f \{ (Fj[f] \neq Fi[FU] \text{ or } Rj[f] = \text{false}) \text{ \& } (Fk[f] \neq Fi[FU] \text{ or } Rk[f] = \text{false}) \}$		$\forall f ((Qj[f] = FU) \Rightarrow Rj[f] \leftarrow \text{true})$ $\forall f ((Qk[f] = FU) \Rightarrow Rk[f] \leftarrow \text{true})$ RSS[Fi[FU]] \leftarrow 0; Busy[FU] \leftarrow false						
Add	no								
Divide	no								
	F0	F2	F4	F6	F8	F10	...	F30	
FU				Integer					

• read operand; second load instruction cannot be issued (struct. hazard)

Scoreboard Example:

Clock Cycle = 3

Instruction	Issue	Read Operands	Exec. Complete	Write Result
L.D F6, 34(R2)	1	2	3	
L.D F2, 45(R3)				
MUL.D F0, F2, F4				
SUB.D F8, F6, F2				
DIV.D F10, F0, F6				
ADD.D F6, F8, F2				

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	yes	L.D	F6		R2				false
Mult-1	no								
Mult-2	no								
Add	no								
Divide	no								

	F0	F2	F4	F6	F8	F10	...	F30
FU				Integer				

- execute first load instruction; MUL.D waits too (issuing stage is stalled)

Scoreboard Example:

Clock Cycle = 3

Scoreboard: Checking & Bookkeeping per Instruction (op D, S1, S2)						Exec. Complete		Write Result	
						3			
Instruction Status	Wait until		Bookkeeping						
1. Issue	not Busy[FU] and not RRS[D]		Busy[FU] \leftarrow true; Op[FU] \leftarrow op; Fi[FU] \leftarrow D; Fj[FU] \leftarrow S1; Fk[FU] \leftarrow S2; Qj \leftarrow RRS[S1]; Qk \leftarrow RRS[S2]; Rj \leftarrow (Qj==0); Rk \leftarrow (Qk==0); RRS[D] \leftarrow FU						
2. Read Operands	Rj=true and Rk=true		Rj \leftarrow false; Rk \leftarrow false; Qj \leftarrow 0; Qk \leftarrow 0			Qj	Qk	Rj	Rk
3. Execution Complete	FU is done executing								false
4. Write Result	$\forall f \{ (Fj[f] \neq Fi[FU] \text{ or } Rj[f] = \text{false}) \text{ \& } (Fk[f] \neq Fi[FU] \text{ or } Rk[f] = \text{false}) \}$		$\forall f ((Qj[f] = FU) \Rightarrow Rj[f] \leftarrow \text{true})$ $\forall f ((Qk[f] = FU) \Rightarrow Rk[f] \leftarrow \text{true})$ RSS[Fi[FU]] \leftarrow 0; Busy[FU] \leftarrow false						
Add	no								
Divide	no								
	F0	F2	F4	F6	F8	F10	...	F30	
FU				Integer					

- execute first load instruction; MUL.D waits too (issuing stage is stalled)

Scoreboard Example:

Clock Cycle = 4

Instruction		Issue		Read Operands		Exec. Complete		Write Result	
L.D F6, 34(R2)		1		2		3		4	
L.D F2, 45(R3)									
MUL.D F0, F2, F4									
SUB.D F8, F6, F2									
DIV.D F10, F0, F6									
ADD.D F6, F8, F2									

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	no								
Mult-2	no								
Add	no								
Divide	no								

	F0	F2	F4	F6	F8	F10	...	F30
FU								

- first load instruction commits (writing F6)

Scoreboard Example:

Clock Cycle = 4

Scoreboard: Checking & Bookkeeping per Instruction (op D, S1, S2)						Exec. Complete		Write Result	
						3		4	
Instruction Status	Wait until		Bookkeeping						
1. Issue	not Busy[FU] and not RRS[D]		Busy[FU] \leftarrow true; Op[FU] \leftarrow op; Fi[FU] \leftarrow D; Fj[FU] \leftarrow S1; Fk[FU] \leftarrow S2; Qj \leftarrow RRS[S1]; Qk \leftarrow RRS[S2]; Rj \leftarrow (Qj==0); Rk \leftarrow (Qk==0); RRS[D] \leftarrow FU						
2. Read Operands	Rj=true and Rk=true		Rj \leftarrow false; Rk \leftarrow false; Qj \leftarrow 0; Qk \leftarrow 0			Qj	Qk	Rj	Rk
3. Execution Complete	FU is done executing								
4. Write Result	$\forall f \{ (Fj[f] \neq Fi[FU] \text{ or } Rj[f] = \text{false}) \text{ \& } (Fk[f] \neq Fi[FU] \text{ or } Rk[f] = \text{false}) \}$		$\forall f ((Qj[f] = FU) \Rightarrow Rj[f] \leftarrow \text{true})$ $\forall f ((Qk[f] = FU) \Rightarrow Rk[f] \leftarrow \text{true})$ RSS[Fi[FU]] \leftarrow 0; Busy[FU] \leftarrow false						
Add	no								
Divide	no								
	F0	F2	F4	F6	F8	F10	...	F30	
FU									

- first load instruction commits (writing F6)

Scoreboard Example:

Clock Cycle = 5

Instruction	Issue	Read Operands	Exec. Complete	Write Result
L.D F6, 34(R2)	1	2	3	4
L.D F2, 45(R3)	5			
MUL.D F0, F2, F4				
SUB.D F8, F6, F2				
DIV.D F10, F0, F6				
ADD.D F6, F8, F2				

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	yes	L.D	F2		R3				true
Mult-1	no								
Mult-2	no								
Add	no								
Divide	no								

	F0	F2	F4	F6	F8	F10	...	F30
FU		Integer						

- issue second load instruction

Scoreboard Example:

Clock Cycle = 5

Scoreboard: Checking & Bookkeeping per Instruction (op D, S1, S2)					Exec. Complete		Write Result	
					3		4	
Instruction Status	Wait until		Bookkeeping					
1. Issue	not Busy[FU] and not RRS[D]		Busy[FU] \leftarrow true; Op[FU] \leftarrow op; Fi[FU] \leftarrow D; Fj[FU] \leftarrow S1; Fk[FU] \leftarrow S2; Qj \leftarrow RRS[S1]; Qk \leftarrow RRS[S2]; Rj \leftarrow (Qj==0); Rk \leftarrow (Qk==0); RRS[D] \leftarrow FU					
2. Read Operands	Rj=true and Rk=true		Rj \leftarrow false; Rk \leftarrow false; Qj \leftarrow 0; Qk \leftarrow 0					
3. Execution Complete	FU is done executing							
4. Write Result	$\forall f \{ (Fj[f] \neq Fi[FU] \text{ or } Rj[f] = \text{false}) \text{ \& } (Fk[f] \neq Fi[FU] \text{ or } Rk[f] = \text{false}) \}$		$\forall f ((Qj[f] = FU) \Rightarrow Rj[f] \leftarrow \text{true})$ $\forall f ((Qk[f] = FU) \Rightarrow Rk[f] \leftarrow \text{true})$ RSS[Fi[FU]] \leftarrow 0; Busy[FU] \leftarrow false					
Add	no							
Divide	no							
	F0	F2	F4	F6	F8	F10	...	F30
FU		Integer						

• issue second load instruction

Scoreboard Example:

Clock Cycle = 6

Instruction		Issue	Read Operands		Exec. Complete		Write Result		
L.D F6, 34(R2)		1	2		3		4		
L.D F2, 45(R3)		5	6						
MUL.D F0, F2, F4		6							
SUB.D F8, F6, F2									
DIV.D F10, F0, F6									
ADD.D F6, F8, F2									

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	yes	L.D	F2		R3				false
Mult-1	yes	MUL.D	F0	F2	F4	Integer		false	true
Mult-2	no								
Add	no								
Divide	no								

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult-1	Integer						

- read operands of second load instruction; issue multiply instruction

Scoreboard Example:

Clock Cycle = 6

Scoreboard: Checking & Bookkeeping per Instruction (op D, S1, S2)						Exec. Complete		Write Result	
						3		4	
Instruction Status	Wait until		Bookkeeping						
1. Issue	not Busy[FU] and not RRS[D]		Busy[FU] \leftarrow true; Op[FU] \leftarrow op ; Fi[FU] \leftarrow D; Fj[FU] \leftarrow S1 ; Fk[FU] \leftarrow S2 ; Qj \leftarrow RRS[S1]; Qk \leftarrow RRS[S2]; Rj \leftarrow (Qj==0); Rk \leftarrow (Qk==0); RRS[D] \leftarrow FU						
2. Read Operands	Rj=true and Rk=true		Rj \leftarrow false; Rk \leftarrow false; Qj \leftarrow 0; Qk \leftarrow 0			Qj	Qk	Rj	Rk
3. Execution Complete	FU is done executing								false
4. Write Result	$\forall f \{ (Fj[f] \neq Fi[FU] \text{ or } Rj[f] = \text{false}) \text{ \& } (Fk[f] \neq Fi[FU] \text{ or } Rk[f] = \text{false}) \}$		$\forall f ((Qj[f] = FU) \Rightarrow Rj[f] \leftarrow \text{true})$ $\forall f ((Qk[f] = FU) \Rightarrow Rk[f] \leftarrow \text{true})$ RSS[Fi[FU]] \leftarrow 0; Busy[FU] \leftarrow false			Integer		false	true
Add	no								
Divide	no								
	F0	F2	F4	F6	F8	F10	...	F30	
FU	Mult-1	Integer							

- read operands of second load instruction; issue multiply instruction

Scoreboard Example:

Clock Cycle = 7

Instruction	Issue	Read Operands	Exec. Complete	Write Result
L.D F6, 34(R2)	1	2	3	4
L.D F2, 45(R3)	5	6	7	
MUL.D F0, F2, F4	6			
SUB.D F8, F6, F2	7			
DIV.D F10, F0, F6				
ADD.D F6, F8, F2				

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	yes	L.D	F2		R3				false
Mult-1	yes	MUL.D	F0	F2	F4	Integer		false	true
Mult-2	no								
Add	yes	SUB.D	F8	F6	F2		Integer	true	false
Divide	no								

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult-1	Integer			Add			

- execute second load instruction; stall multiply; issue subtraction

Scoreboard Example:

Clock Cycle = 7

Scoreboard: Checking & Bookkeeping per Instruction (op D, S1, S2)						Exec. Complete		Write Result																
<table><tr><th>Instruction Status</th><th>Wait until</th><th>Bookkeeping</th></tr><tr><td>1. Issue</td><td>not Busy[FU] and not RRS[D]</td><td>Busy[FU] ← true; Op[FU] ← op; Fi[FU] ← D; Fj[FU] ← S1; Fk[FU] ← S2; Qj ← RRS[S1]; Qk ← RRS[S2]; Rj ← (Qj==0); Rk ← (Qk==0); RRS[D] ← FU</td></tr><tr><td>2. Read Operands</td><td>Rj=true and Rk=true</td><td>Rj ← false; Rk ← false; Qj ← 0; Qk ← 0</td></tr><tr><td>3. Execution Complete</td><td>FU is done executing</td><td></td></tr><tr><td>4. Write Result</td><td>$\forall f \{ (Fj[f] \neq Fi[FU] \text{ or } Rj[f] = \text{false}) \text{ \& } (Fk[f] \neq Fi[FU] \text{ or } Rk[f] = \text{false}) \}$</td><td>$\forall f ((Qj[f] = FU) \Rightarrow Rj[f] \leftarrow \text{true})$ $\forall f ((Qk[f] = FU) \Rightarrow Rk[f] \leftarrow \text{true})$ RSS[Fi[FU]] ← 0; Busy[FU] ← false</td></tr></table>						Instruction Status	Wait until	Bookkeeping	1. Issue	not Busy[FU] and not RRS[D]	Busy[FU] ← true; Op[FU] ← op; Fi[FU] ← D; Fj[FU] ← S1; Fk[FU] ← S2; Qj ← RRS[S1]; Qk ← RRS[S2]; Rj ← (Qj==0); Rk ← (Qk==0); RRS[D] ← FU	2. Read Operands	Rj=true and Rk=true	Rj ← false; Rk ← false; Qj ← 0; Qk ← 0	3. Execution Complete	FU is done executing		4. Write Result	$\forall f \{ (Fj[f] \neq Fi[FU] \text{ or } Rj[f] = \text{false}) \text{ \& } (Fk[f] \neq Fi[FU] \text{ or } Rk[f] = \text{false}) \}$	$\forall f ((Qj[f] = FU) \Rightarrow Rj[f] \leftarrow \text{true})$ $\forall f ((Qk[f] = FU) \Rightarrow Rk[f] \leftarrow \text{true})$ RSS[Fi[FU]] ← 0; Busy[FU] ← false	3		4	
						Instruction Status	Wait until	Bookkeeping																
						1. Issue	not Busy[FU] and not RRS[D]	Busy[FU] ← true; Op[FU] ← op; Fi[FU] ← D; Fj[FU] ← S1; Fk[FU] ← S2; Qj ← RRS[S1]; Qk ← RRS[S2]; Rj ← (Qj==0); Rk ← (Qk==0); RRS[D] ← FU																
						2. Read Operands	Rj=true and Rk=true	Rj ← false; Rk ← false; Qj ← 0; Qk ← 0																
						3. Execution Complete	FU is done executing																	
4. Write Result	$\forall f \{ (Fj[f] \neq Fi[FU] \text{ or } Rj[f] = \text{false}) \text{ \& } (Fk[f] \neq Fi[FU] \text{ or } Rk[f] = \text{false}) \}$	$\forall f ((Qj[f] = FU) \Rightarrow Rj[f] \leftarrow \text{true})$ $\forall f ((Qk[f] = FU) \Rightarrow Rk[f] \leftarrow \text{true})$ RSS[Fi[FU]] ← 0; Busy[FU] ← false																						
7																								
		Qj	Qk	Rj	Rk																			
					false																			
		Integer		false	true																			
			Integer	true	false																			

Add	yes	SUB.D	F8	F6	F2		Integer	true	false
Divide	no								

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult-1	Integer			Add			

Scoreboard Example:

Clock Cycle = 8

Instruction	Issue	Read Operands	Exec. Complete	Write Result
L.D F6, 34(R2)	1	2	3	4
L.D F2, 45(R3)	5	6	7	8
MUL.D F0, F2, F4	6			
SUB.D F8, F6, F2	7			
DIV.D F10, F0, F6	8			
ADD.D F6, F8, F2				

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	yes	MUL.D	F0	F2	F4	Integer		true	true
Mult-2	no								
Add	yes	SUB.D	F8	F6	F2		Integer	true	true
Divide	yes	DIV.D	F10	F0	F6	Mult-1		false	true

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult-1				Add	Divide		

- issue division; second load instruction is done; F2 ready (but not read yet)

Scoreboard Example:

Clock Cycle = 8

Scoreboard: Checking & Bookkeeping per Instruction (op D, S1, S2)						Exec. Complete		Write Result						
Instruction Status						3		4						
1. Issue						7		8						
Wait until														
Bookkeeping														
not Busy[FU] and not RRS[D]														
Busy[FU] ← true; Op[FU] ←op; Fi[FU] ← D; Fj[FU] ←S1; Fk[FU] ←S2; Qj ← RRS[S1]; Qk ← RRS[S2]; Rj ← (Qj==0); Rk ← (Qk==0); RRS[D] ← FU														
2. Read Operands						Rj=true and Rk=true		Rj ← false; Rk ← false; Qj ← 0; Qk ← 0						
3. Execution Complete						FU is done executing								
4. Write Result						∀f { (Fj[f] ≠ Fi[FU] or Rj[f] = false) & (Fk[f] ≠ Fi[FU] or Rk[f] = false) }		∀f ((Qj[f] = FU) ⇒ Rj[f] ← true) & ∀f ((Qk[f] = FU) ⇒ Rk[f] ← true) RSS[Fi[FU]] ← 0; Busy[FU] ← false						
Add						yes	SUB.D	F8	F6	F2		Integer	true	true
Divide						yes	DIV.D	F10	F0	F6	Mult-1		false	true
						F0	F2	F4	F6	F8	F10	...	F30	
FU						Mult-1				Add	Divide			

• issue division; second load instruction is done; F2 ready (but not read yet)

Scoreboard Example:

Clock Cycle = 9

Instruction	Issue	Read Operands	Exec. Complete	Write Result
L.D F6, 34(R2)	1	2	3	4
L.D F2, 45(R3)	5	6	7	8
MUL.D F0, F2, F4	6	9		
SUB.D F8, F6, F2	7	9		
DIV.D F10, F0, F6	8			
ADD.D F6, F8, F2				

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	yes	MUL.D	F0	F2	F4			false	false
Mult-2	no								
Add	yes	SUB.D	F8	F6	F2			false	false
Divide	yes	DIV.D	F10	F0	F6	Mult-1		false	true

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult-1				Add	Divide		

- reading operands of multiplication and subtraction; addition not issued

Scoreboard Example: Clock Cycle = 9

Scoreboard: Checking & Bookkeeping per Instruction (op D , S1 , S2)						Exec. Complete		Write Result	
						3		4	
						7		8	

- reading operands of multiplication and subtraction; addition not issued

Scoreboard Example:

Clock Cycle = 10

Instruction	Issue	Read Operands	Exec. Complete	Write Result
L.D F6, 34(R2)	1	2	3	4
L.D F2, 45(R3)	5	6	7	8
MUL.D F0, F2, F4	6	9		
SUB.D F8, F6, F2	7	9		
DIV.D F10, F0, F6	8			
ADD.D F6, F8, F2				

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	yes	MUL.D	F0	F2	F4			false	false
Mult-2	no								
Add	yes	SUB.D	F8	F6	F2			false	false
Divide	yes	DIV.D	F10	F0	F6	Mult-1		false	true

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult-1				Add	Divide		

- multiplication (10 cycles) and subtraction (2 cycles) are executing

Scoreboard Example:

Clock Cycle = 11

Instruction		Issue	Read Operands		Exec. Complete		Write Result		
L.D F6, 34(R2)		1	2		3		4		
L.D F2, 45(R3)		5	6		7		8		
MUL.D F0, F2, F4		6	9						
SUB.D F8, F6, F2		7	9		11				
DIV.D F10, F0, F6		8							
ADD.D F6, F8, F2									

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	yes	MUL.D	F0	F2	F4			false	false
Mult-2	no								
Add	yes	SUB.D	F8	F6	F2			false	false
Divide	yes	DIV.D	F10	F0	F6	Mult-1		false	true

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult-1				Add	Divide		

- subtraction execution is completed

Scoreboard Example:

Clock Cycle = 12

Instruction	Issue	Read Operands	Exec. Complete	Write Result
L.D F6, 34(R2)	1	2	3	4
L.D F2, 45(R3)	5	6	7	8
MUL.D F0, F2, F4	6	9		
SUB.D F8, F6, F2	7	9	11	12
DIV.D F10, F0, F6	8			
ADD.D F6, F8, F2				

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	yes	MUL.D	F0	F2	F4			false	false
Mult-2	no								
Add	no								
Divide	yes	DIV.D	F10	F0	F6	Mult-1		false	true

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult-1					Divide		

- subtraction is done and result is written into F8

Scoreboard Example:

Clock Cycle = 13

Instruction		Issue	Read Operands		Exec. Complete		Write Result		
L.D F6, 34(R2)		1	2		3		4		
L.D F2, 45(R3)		5	6		7		8		
MUL.D F0, F2, F4		6	9						
SUB.D F8, F6, F2		7	9		11		12		
DIV.D F10, F0, F6		8							
ADD.D F6, F8, F2		13							
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	yes	MUL.D	F0	F2	F4			false	false
Mult-2	no								
Add	yes	ADD.D	F6	F8	F2			true	true
Divide	yes	DIV.D	F10	F0	F6	Mult-1		false	true
	F0	F2	F4	F6	F8	F10	...	F30	
FU	Mult-1			Add		Divide			

- addition is issued because FP adder is now available

Scoreboard Example:

Clock Cycle = 14

Instruction		Issue		Read Operands		Exec. Complete		Write Result	
L.D F6, 34(R2)		1		2		3		4	
L.D F2, 45(R3)		5		6		7		8	
MUL.D F0, F2, F4		6		9					
SUB.D F8, F6, F2		7		9		11		12	
DIV.D F10, F0, F6		8							
ADD.D F6, F8, F2		13		14					
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	yes	MUL.D	F0	F2	F4			false	false
Mult-2	no								
Add	yes	ADD.D	F6	F8	F2			false	false
Divide	yes	DIV.D	F10	F0	F6	Mult-1		false	true
	F0	F2	F4	F6	F8	F10	...	F30	
FU	Mult-1			Add		Divide			

- addition operands are read and execution starts (will take two cycles)

Scoreboard Example:

Clock Cycle = 15

Instruction		Issue		Read Operands		Exec. Complete		Write Result	
L.D F6, 34(R2)		1		2		3		4	
L.D F2, 45(R3)		5		6		7		8	
MUL.D F0, F2, F4		6		9					
SUB.D F8, F6, F2		7		9		11		12	
DIV.D F10, F0, F6		8							
ADD.D F6, F8, F2		13		14					
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	yes	MUL.D	F0	F2	F4			false	false
Mult-2	no								
Add	yes	ADD.D	F6	F8	F2			false	false
Divide	yes	DIV.D	F10	F0	F6	Mult-1		false	true
	F0	F2	F4	F6	F8	F10	...	F30	
FU	Mult-1			Add		Divide			

- executing addition (1 more cycle) and multiplication (still 4 more to go)

Scoreboard Example:

Clock Cycle = 16

Instruction		Issue	Read Operands		Exec. Complete		Write Result		
L.D F6, 34(R2)		1	2		3		4		
L.D F2, 45(R3)		5	6		7		8		
MUL.D F0, F2, F4		6	9						
SUB.D F8, F6, F2		7	9		11		12		
DIV.D F10, F0, F6		8							
ADD.D F6, F8, F2		13	14		16				
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	yes	MUL.D	F0	F2	F4			false	false
Mult-2	no								
Add	yes	ADD.D	F6	F8	F2			false	false
Divide	yes	DIV.D	F10	F0	F6	Mult-1		false	true
	F0	F2	F4	F6	F8	F10	...	F30	
FU	Mult-1			Add		Divide			

• addition execution is completed, but multiplication has still 3 more to go

Scoreboard Example:

Clock Cycle = 17

Instruction		Issue	Read Operands			Exec. Complete		Write Result	
L.D F6, 34(R2)		1	2			3		4	
L.D F2, 45(R3)		5	6			7		8	
MUL.D F0, F2, F4		6	9						
SUB.D F8, F6, F2		7	9			11		12	
DIV.D F10, F0, F6		8							
ADD.D F6, F8, F2		13	14			16			
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	yes	MUL.D	F0	F2	F4			false	false
Mult-2	no								
Add	yes	ADD.D	F6	F8	F2			false	false
Divide	yes	DIV.D	F10	F0	F6	Mult-1		false	true
	F0	F2	F4	F6	F8	F10	...	F30	
FU	Mult-1			Add		Divide			

- addition stalls (result is not written) to avoid WAR hazard on F6

Scoreboard Example:

Clock Cycle = 18

Instruction		Issue	Read Operands			Exec. Complete		Write Result	
L.D F6, 34(R2)		1	2			3		4	
L.D F2, 45(R3)		5	6			7		8	
MUL.D F0, F2, F4		6	9						
SUB.D F8, F6, F2		7	9			11		12	
DIV.D F10, F0, F6		8							
ADD.D F6, F8, F2		13	14			16			
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	yes	MUL.D	F0	F2	F4			false	false
Mult-2	no								
Add	yes	ADD.D	F6	F8	F2			false	false
Divide	yes	DIV.D	F10	F0	F6	Mult-1		false	true
	F0	F2	F4	F6	F8	F10	...	F30	
FU	Mult-1			Add		Divide			

• multiplication is completing, division waits for it, addition waits for division

Scoreboard Example:

Clock Cycle = 19

Instruction		Issue	Read Operands		Exec. Complete		Write Result		
L.D F6, 34(R2)		1	2		3		4		
L.D F2, 45(R3)		5	6		7		8		
MUL.D F0, F2, F4		6	9		19				
SUB.D F8, F6, F2		7	9		11		12		
DIV.D F10, F0, F6		8							
ADD.D F6, F8, F2		13	14		16				
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	yes	MUL.D	F0	F2	F4			false	false
Mult-2	no								
Add	yes	ADD.D	F6	F8	F2			false	false
Divide	yes	DIV.D	F10	F0	F6	Mult-1		false	true
	F0	F2	F4	F6	F8	F10	...	F30	
FU	Mult-1			Add		Divide			

- multiplication execution is finally completed

Scoreboard Example:

Clock Cycle = 20

Instruction		Issue		Read Operands		Exec. Complete		Write Result	
L.D F6, 34(R2)		1		2		3		4	
L.D F2, 45(R3)		5		6		7		8	
MUL.D F0, F2, F4		6		9		19		20	
SUB.D F8, F6, F2		7		9		11		12	
DIV.D F10, F0, F6		8							
ADD.D F6, F8, F2		13		14		16			
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	no								
Mult-2	no								
Add	yes	ADD.D	F6	F8	F2			false	false
Divide	yes	DIV.D	F10	F0	F6			true	true
	F0	F2	F4	F6	F8	F10	...	F30	
FU				Add		Divide			

- multiplication has committed (writing F0)

Scoreboard Example:

Clock Cycle = 21

Instruction		Issue	Read Operands		Exec. Complete		Write Result		
L.D F6, 34(R2)		1	2		3		4		
L.D F2, 45(R3)		5	6		7		8		
MUL.D F0, F2, F4		6	9		19		20		
SUB.D F8, F6, F2		7	9		11		12		
DIV.D F10, F0, F6		8	21						
ADD.D F6, F8, F2		13	14		16				
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	no								
Mult-2	no								
Add	yes	ADD.D	F6	F8	F2			false	false
Divide	yes	DIV.D	F10	F0	F6			false	false
	F0	F2	F4	F6	F8	F10	...	F30	
FU				Add		Divide			

- reading division operands

Scoreboard Example:

Clock Cycle = 22

Instruction		Issue	Read Operands		Exec. Complete		Write Result		
L.D F6, 34(R2)		1	2		3		4		
L.D F2, 45(R3)		5	6		7		8		
MUL.D F0, F2, F4		6	9		19		20		
SUB.D F8, F6, F2		7	9		11		12		
DIV.D F10, F0, F6		8	21						
ADD.D F6, F8, F2		13	14		16		22		
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	no								
Mult-2	no								
Add	no								
Divide	yes	DIV.D	F10	F0	F6			false	false
	F0	F2	F4	F6	F8	F10	...	F30	
FU						Divide			

• No more WAR hazards, addition can write result into F6

Scoreboard Example:

Clock Cycle = 61

Instruction		Issue	Read Operands			Exec. Complete	Write Result		
L.D F6, 34(R2)		1	2			3	4		
L.D F2, 45(R3)		5	6			7	8		
MUL.D F0, F2, F4		6	9			19	20		
SUB.D F8, F6, F2		7	9			11	12		
DIV.D F10, F0, F6		8	21			61			
ADD.D F6, F8, F2		13	14			16	22		
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	no								
Mult-2	no								
Add	no								
Divide	yes	DIV.D	F10	F0	F6			false	false
	F0	F2	F4	F6	F8	F10	...	F30	
FU						Divide			

- After 40 cycles division execution is completed

Scoreboard Example:

Clock Cycle = 62

Instruction		Issue	Read Operands			Exec. Complete	Write Result		
L.D F6, 34(R2)		1	2			3	4		
L.D F2, 45(R3)		5	6			7	8		
MUL.D F0, F2, F4		6	9			19	20		
SUB.D F8, F6, F2		7	9			11	12		
DIV.D F10, F0, F6		8	21			61	62		
ADD.D F6, F8, F2		13	14			16	22		

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	no								
Mult-2	no								
Add	no								
Divide	no								

	F0	F2	F4	F6	F8	F10	...	F30
FU								

- Division commits and the whole program is done


Scoreboard Example:

Clock Cycle = 62

Instruction		Issue		Read Operands		Exec. Complete		Write Result	
L.D F6, 34(R2)		1		2		3		4	
L.D F2, 45(R3)		5		6		7		8	
MUL.D F0, F2, F4		6		9		19		20	
SUB.D F8, F6, F2		7		9		11		12	
DIV.D F10, F0, F6		8		21		61		62	
ADD.D F6, F8, F2		13		14		16		22	
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult-1	no								
Mult-2	no								
Add	no								
Divide	no								
	F0	F2	F4	F6	F8	F10	...	F30	
FU									

- 62 cycles necessary (in-order **issue**; out-of-order **execution & commit**)

Scoreboard: Checking & Bookkeeping per Instruction (op D , $S1$, $S2$)

Instruction Status	Wait until	Bookkeeping
1. Issue	not Busy[FU] and not RRS[D]	$Busy[FU] \leftarrow true$; $Op[FU] \leftarrow op$; $Fi[FU] \leftarrow D$; $Fj[FU] \leftarrow S1$; $Fk[FU] \leftarrow S2$; $Qj \leftarrow RRS[S1]$; $Qk \leftarrow RRS[S2]$; $Rj \leftarrow (Qj == 0)$; $Rk \leftarrow (Qk == 0)$; $RRS[D] \leftarrow FU$
2. Read Operands	$Rj = true$ and $Rk = true$	$Rj \leftarrow false$; $Rk \leftarrow false$; $Qj \leftarrow 0$; $Qk \leftarrow 0$
3. Execution Complete	FU is done executing	
4. Write Result	$\forall f \{ (Fj[f] \neq Fi[FU] \text{ or } Rj[f] = false) \text{ \& } (Fk[f] \neq Fi[FU] \text{ or } Rk[f] = false) \}$ 	$\forall f ((Qj[f] = FU) \Rightarrow Rj[f] \leftarrow true)$ $\forall f ((Qk[f] = FU) \Rightarrow Rk[f] \leftarrow true)$ $RSS[Fi[FU]] \leftarrow 0$; $Busy[FU] \leftarrow false$

If $(Fj[f] == Fi[Fu] \ \&\& \ Rj[f] == true)$, it cannot write, as another instruction has not read the old value yet --> avoid WAR

Scoreboard: Benefits & Costs

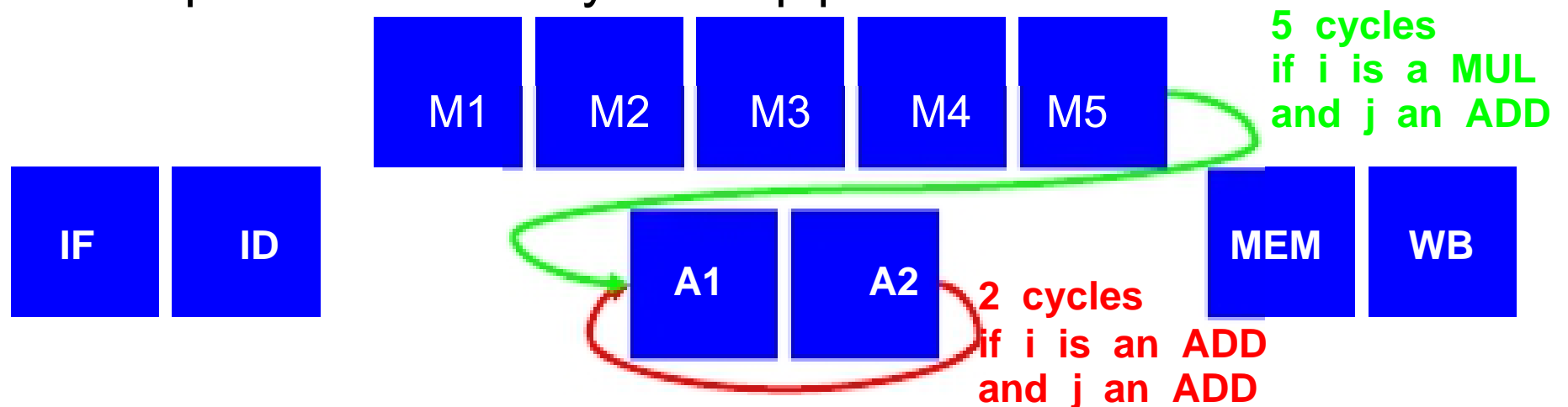
- The CDC 6600 designers measured
 - 1.7x performance improvement for FORTRAN programs
 - 2.5x performance improvement for assembly programs
 - but before breakthroughs in DRAM, SW scheduling, caches...
 - dynamic scheduling now popular with multiple-issue & speculation
- CDC 6600 scoreboard had about as much logic as one of the functional units
 - but four times as many buses as a simple in-order implementation
- In eliminating stalls, a scoreboard is limited by
 - # of scoreboard entries \geq window size (possible parallelism)
 - amount of parallelism among the instructions
 - recall that CDC 6600 windows do not extend beyond a basic block
 - # and type of functional units (structural hazards)
 - # of buses \geq # functional units proceeding in steps 2 & 4
 - presence of anti-dependences and output dependences

Compiler Technology to Improve ILP

- Loop Unrolling & Pipeline Static Scheduling
- Software Pipelining
- Compiler-Based Branch Prediction
- Applications
 - processors that use static issue
 - VLIW and VLIW-like
 - NXP Semiconductors (founded by Philips) Trimedia
 - Transmeta Crusoe
 - processors that combine dynamic scheduling and static scheduling
 - Itanium Processor Family (based on IA-64 ISA)
 - Explicit Parallel Instruction Computing (EPIC)

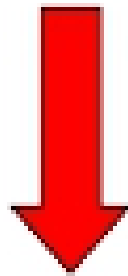
How to Keep a Pipeline Full?

- Exploit Parallelism “in Time”
 - find sequences of independent instructions that can be overlapped in the pipeline
- Scheduling problem for a compiler
 - given “source instruction” **i** and “dependent instruction” **j**
 - avoid pipeline stalls by...
 - ...separating **j** from **i** by a distance in clock cycles that is equal to the latency of the pipeline execution of **i**



Example: a Simple Parallel Loop

```
for (i=1000; i>0; i=i-1) {  
    x[i] = x[i] + s;  
}
```



compiled into MIPS
assembly language

```
loop:   L.D      F0, 0(R1)      ; R1 = highest address of array element  
        ADD.D    F4, F0, F2     ; add scalar s (stored in F2)  
        S.D      F4, 0(R1)     ; store result  
        DADDUI   R1,R1,#-8      ; decrement pointer by 8 bytes (DW)  
        BNE      R1,R2, loop    ; 8(R2) points to x[1], last to process
```


Assumptions for the Following Examples: Latencies and Resources of MIPS Pipeline

instruction Producing result	instruction Consuming result	Latency in Clock cycles
FP ALU op	FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	1

- Additional assumption: no structural hazards
 - functional units are fully pipelined (or replicated as many times as the pipeline depth) so that an operation of any type can be issued on every clock cycle

Example: Scheduling the Simple Parallel

Loop 0

```

loop:  L.D
        ADD.D
        S.D
        DADDUI
        BNE
    
```

instruction
Producing result

FP ALU op
FP ALU op
Load double
Load double
Integer op

instruction
Consuming result

FP ALU op
Store double
FP ALU op
Store double
Integer op

Latency in
Clock cycles

3
2
1
0
1

- unscheduled execution

```

loop:  L.D      F0, 0(R1)
        stall
        ADD.D   F4, F0, F2
        stall
        stall
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #-8
        stall
        BNE     R1, R2, loop
    
```

- executionTime = 9 cycles

- scheduled execution

```

loop:  L.D      F0, 0(R1)
        DADDUI  R1, R1, #-8
        ADD.D   F4, F0, F2
        stall
        stall
        S.D     F4, 8(R1)
        BNE     R1, R2, loop
    
```

- executionTime = 7 cycles

• NOTE: execution time is expressed in 'clock cycles per loop iteration '

Scheduled Execution and Critical Path

- The clock cycle count for the loop is determined by the **critical path**, longest chain of dependent instructions
 - in the example, 6 clock cycles to execute sequentially the chain
 - L.D F0, 0(R1)
 - stall**
 - ADD.D F4, F0, F2
 - stall**
 - stall**
 - S.D F4, 8(R1)
- A good compiler (capable of some symbolic optimization) optimizes the “filling” of those stalls, for instance by altering and interchanging S.D with DADDUI

- scheduled execution**

```
loop:  L.D      F0, 0(R1)
       DADDUI  R1, R1, #-8
       ADD.D   F4, F0, F2
       stall
       stall
       S.D     F4, 8(R1)
       BNE     R1, R2, loop
```

- executionTime = 7 cycles

Loop Overhead and Loop Unrolling

- At each iteration of the loop, the only actual work is done on the array

```
L.D      F0, 0(R1)
ADD.D    F4, F0, F2
S.D      F4, 8(R1)
```

- Beside the stall cycle, **DADDUI** and **BNE** represent **loop overhead** cycles
- To reduce loop overhead, apply **loop unrolling** before scheduling the loop execution
 - replicate the loop body multiple times
 - merge unnecessary instructions
 - eliminate the name dependencies
 - adjust the control code

- scheduled execution**

```
loop:  L.D      F0, 0(R1)
       DADDUI   R1, R1, #-8
       ADD.D    F4, F0, F2
       stall
       stall
       S.D      F4, 8(R1)
       BNE      R1, R2, loop
```

- executionTime = 7 cycles**

Reducing Loop Overhead by Loop Unrolling before Scheduling – Step 1 (replication)

- original loop

```
loop:  L.D    F0, 0(R1)
        ADD.D  F4, F0, F2
        S.D    F4, 0(R1)
        DADDUI R1, R1, #-8
        BNE    R1, R2, loop
```



- unrolling the original loop 4 times (dropping 3 **BNE**)

```
loop:  L.D    F0, 0(R1)
        ADD.D  F4, F0, F2
        S.D    F4, 0(R1)
        DADDUI R1, R1, #-8

        L.D    F0, 0(R1)
        ADD.D  F4, F0, F2
        S.D    F4, 0(R1)
        DADDUI R1, R1, #-8

        L.D    F0, 0(R1)
        ADD.D  F4, F0, F2
        S.D    F4, 0(R1)
        DADDUI R1, R1, #-8

        L.D    F0, 0(R1)
        ADD.D  F4, F0, F2
        S.D    F4, 0(R1)
        DADDUI R1, R1, #-8
        BNE    R1, R2, loop
```

Execution Time after Step 1

- unrolling the original loop 4 times (dropping 3 **BNE**)

```

loop:  L.D      F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #-8
        L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #-8
        L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #-8
        L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #-8
        BNE     R1, R2, loop
    
```

```

loop:  L.D      F0, 0(R1)
        stall
        ADD.D   F4, F0, F2
        stall
        stall
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #-8
        stall
        L.D     F0, 0(R1)
        stall
        ADD.D   F4, F0, F2
        stall
        stall
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #-8
        stall
        L.D     F0, 0(R1)
        stall
        ADD.D   F4, F0, F2
        stall
        stall
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #-8
        stall
        BNE     R1, R2, loop
    
```

- executionTime = 33 cycles

Reducing Loop Overhead by Loop Unrolling before Scheduling – Step 2 (merging)

- unrolling the original loop 4 times (dropping 3 **BNE**)

```
loop:  L.D      F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #-8
        L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #-8
        L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #-8
        L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #-8
        BNE     R1, R2, loop
```



- merging **DADDUI** and adjusting the offsets

```
loop:  L.D      F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        L.D     F0, -8(R1)
        ADD.D   F4, F0, F2
        S.D     F4, -8(R1)
        L.D     F0, -16(R1)
        ADD.D   F4, F0, F2
        S.D     F4, -16(R1)
        L.D     F0, -24(R1)
        ADD.D   F4, F0, F2
        S.D     F4, -24(R1)
        DADDUI  R1, R1, #-32
        BNE     R1, R2, loop
```


Execution Time after Step 2

- merging **DADDUI** and adjusting the offsets

```

loop:  L.D      F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        L.D     F0, -8(R1)
        ADD.D   F4, F0, F2
        S.D     F4, -8(R1)
        L.D     F0, -16(R1)
        ADD.D   F4, F0, F2
        S.D     F4, -16(R1)
        L.D     F0, -24(R1)
        ADD.D   F4, F0, F2
        S.D     F4, -24(R1)
        DADDUI  R1, R1, #-32
        BNE     R1, R2, loop
    
```

```

loop:  L.D      F0, 0(R1)
        stall
        ADD.D   F4, F0, F2
        stall
        S.D     F4, 0(R1)
        L.D     F0, -8(R1)
        stall
        ADD.D   F4, F0, F2
        stall
        S.D     F4, -8(R1)
        L.D     F0, -16(R1)
        stall
        ADD.D   F4, F0, F2
        stall
        S.D     F4, -16(R1)
        L.D     F0, -24(R1)
        stall
        ADD.D   F4, F0, F2
        stall
        S.D     F4, -24(R1)
        DADDUI  R1, R1, #-32
        stall
        BNE     R1, R2, loop
    
```

- executionTime = 27 cycles

Reducing Loop Overhead by Loop Unrolling before Scheduling – Step 3 (renaming)

- merging **DADDUI** and adjusting the offsets

```
loop:  L.D      F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        L.D     F0, -8(R1)
        ADD.D   F4, F0, F2
        S.D     F4, -8(R1)
        L.D     F0, -16(R1)
        ADD.D   F4, F0, F2
        S.D     F4, -16(R1)
        L.D     F0, -24(R1)
        ADD.D   F4, F0, F2
        S.D     F4, -24(R1)
        DADDUI  R1, R1, #-32
        BNE     R1, R2, loop
```



- eliminating name dependences via register renaming

```
loop:  L.D      F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        L.D     F6, -8(R1)
        ADD.D   F8, F6, F2
        S.D     F8, -8(R1)
        L.D     F10, -16(R1)
        ADD.D   F12, F10, F2
        S.D     F12, -16(R1)
        L.D     F14, -24(R1)
        ADD.D   F16, F14, F2
        S.D     F16, -24(R1)
        DADDUI  R1, R1, #-32
        BNE     R1, R2, loop
```

Execution Time after Step 3

- eliminating name dependences via register renaming

```

loop:  L.D      F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        L.D     F6, -8(R1)
        ADD.D   F8, F6, F2
        S.D     F8, -8(R1)
        L.D     F10, -16(R1)
        ADD.D   F12, F10, F2
        S.D     F12, -16(R1)
        L.D     F14, -24(R1)
        ADD.D   F16, F14, F2
        S.D     F16, -24(R1)
        DADDUI  R1, R1, #-32
        BNE     R1, R2, loop
    
```

```

loop:  L.D      F0, 0(R1)
        stall
        ADD.D   F4, F0, F2
        stall
        stall
        S.D     F4, 0(R1)
        L.D     F6, -8(R1)
        stall
        ADD.D   F8, F6, F2
        stall
        stall
        S.D     F8, -8(R1)
        L.D     F10, -16(R1)
        stall
        ADD.D   F12, F10, F2
        stall
        stall
        S.D     F12, -16(R1)
        L.D     F14, -24(R1)
        stall
        ADD.D   F16, F14, F2
        stall
        stall
        S.D     F16, -24(R1)
        DADDUI  R1, R1, #-32
        stall
        BNE     R1, R2, loop
    
```

- executionTime = 27 cycles

Reducing Loop Overhead by Loop Unrolling before Scheduling – Step 4 (scheduling)

- eliminating name dependences via register renaming

```
loop:  L.D      F0, 0(R1)
       ADD.D    F4, F0, F2
       S.D      F4, 0(R1)
       L.D      F6, -8(R1)
       ADD.D    F8, F6, F2
       S.D      F8, -8(R1)
       L.D      F10, -16(R1)
       ADD.D    F12, F10, F2
       S.D      F12, -16(R1)
       L.D      F14, -24(R1)
       ADD.D    F16, F14, F2
       S.D      F16, -24(R1)
       DADDUI   R1, R1, #-32
       BNE     R1, R2, loop
```



- scheduling

```
loop:  L.D      F0, 0(R1)
       L.D      F6, -8(R1)
       L.D      F10, -16(R1)
       L.D      F14, -24(R1)
       ADD.D    F4, F0, F2
       ADD.D    F8, F6, F2
       ADD.D    F12, F10, F2
       ADD.D    F16, F14, F2
       S.D      F4, 0(R1)
       S.D      F8, -8(R1)
       DADDUI   R1, R1, #-32
       S.D      F12, 16(R1)
       BNE     R1, R2, loop
       S.D      F16, 8(R1)
```

Scheduling after Step 4

- scheduling

```
loop:  L.D    F0, 0(R1)
       L.D    F6, -8(R1)
       L.D    F10, -16(R1)
       L.D    F14, -24(R1)
       ADD.D   F4, F0, F2
       ADD.D   F8, F6, F2
       ADD.D   F12, F10, F2
       ADD.D   F16, F14, F2
       S.D     F4, 0(R1)
       S.D     F8, -8(R1)
       DADDUI  R1, R1, #-32
       S.D     F12, 16(R1)
       BNE     R1, R2, loop
       S.D     F16, 8(R1)
```

```
loop:  L.D    F0, 0(R1)
       L.D    F6, -8(R1)
       L.D    F10, -16(R1)
       L.D    F14, -24(R1)
       ADD.D   F4, F0, F2
       ADD.D   F8, F6, F2
       ADD.D   F12, F10, F2
       ADD.D   F16, F14, F2
       S.D     F4, 0(R1)
       S.D     F8, -8(R1)
       DADDUI  R1, R1, #-32
       S.D     F12, 16(R1)
       BNE     R1, R2, loop
       S.D     F16, 8(R1)
```

- executionTime = 14 cycles

Final Performance Comparison

- original scheduled execution

```

Loop:  L.D      F0, 0(R1)
        DADDUI  R1, R1, #-8
        ADD.D   F4, F0, F2
        stall
        stall
        S.D     F4, 8(R1)
        BNE     R1, R2, loop
    
```

- executionTime = 7 cycles
- execution Time Per Element = 7 cycles
- e.g., an array of 1000 elements is processed in 7,000 cycles

- scheduled execution after loop unrolling

```

loop:  L.D      F0, 0(R1)
        L.D      F6, -8(R1)
        L.D      F10, -16(R1)
        L.D      F14, -24(R1)
        ADD.D    F4, F0, F2
        ADD.D    F8, F6, F2
        ADD.D    F12, F10, F2
        ADD.D    F16, F14, F2
        S.D      F4, 0(R1)
        S.D      F8, -8(R1)
        DADDUI   R1, R1, #-32
        S.D      F12, 16(R1)
        BNE     R1, R2, loop
        S.D      F16, 8(R1)
    
```

- executionTime = 14 cycles
- execution Time Per Element = 3.5 cycles
- e.g., an array of 1000 elements is processed in 3,500 cycles

Loop-unrolling speedup = 2

Loop Unrolling & Scheduling: Summary

- How is the final instruction sequence obtained?
- The compiler needs to determine that it is both possible and convenient to...
 - **unroll the loop** because the iterations are independent (except for the loop maintenance code)
 - **eliminate extra loop maintenance code** as long as the offsets are properly adjusted
 - **use different registers** to increase parallelism by removing name dependencies
 - **remove stalling cycles** by interleaving the instructions (and adjusting the offset or index increment consequently)
 - e.g. the compiler analyzes the memory addresses to determine that loads and stores from different iterations are independent

Example: Two-Issue Statically-Scheduled MIPS Superscalar

- Assume a two-issue statically-scheduled MIPS Superscalar implementation that can issue up to two instructions per clock cycles, i.e.
 - 1 integer/memory/control instruction
 - 1 FP instruction

Inst.Type								
non-FP	IF	ID	EX	MEM	WB			
FP	IF	ID	EX	MEM	WB			
non-FP		IF	ID	EX	MEM	WB		
FP		IF	ID	EX	MEM	WB		
non-FP			IF	ID	EX	MEM	WB	
FP			IF	ID	EX	MEM	WB	
non-FP				IF	ID	EX	MEM	WB
FP				IF	ID	EX	MEM	WB

Executing the Simple Loop on a Two-Issue Statically-Scheduled MIPS Superscalar

Integer Instruction	FP Instruction	Clock Cycle	• unrolled loop 5 times
loop: L.D F0, 0(R1)		1	
L.D F6, -8(R1)		2	• execTime = 12 cycles
L.D F10, -16(R1)	ADD.D F4, F0, F2	3	• execTimePer Element = 2.4 cycles
L.D F14, -24(R1)	ADD.D F8, F6, F2	4	
L.D F18, -32(R1)	ADD.D F12, F10, F2	5	
S.D F4, -0(R1)	ADD.D F16, F14, F2	6	• superscalar speedup = 1.5 (with respect to loop unrolling)
S.D F8, -8(R1)	ADD.D F20, F18, F2	7	
S.D F12, -16(R1)		8	
DADDUI R1, R1, #-40		9	• combined speedup = 2.9
S.D F16, 16(R1)		10	
BNE R1, R2, loop		11	
S.D F20, 8(R1)		12	• efficiency = =17/24 = 71%

Between Statically-Scheduled and Dynamically-Scheduled Superscalars

- **Superscalar processors** decide on the fly how many instructions to issue in a clock cycle
- Key is to detect any dependency
 - between candidate instructions for an issue packet
 - between any 'issue candidate' and any 'instruction already in the pipeline'
- **Statically-Scheduled Superscalar Processors**
 - rely on compiler assistance
- **Dynamically-Scheduled Superscalar Processors**
 - rely on specialized hardware
- **Very Long Instruction Word (VLIW) Processors**
 - let the compiler format the potential issue packet (possibly indicating that a dependency exists)

Multiple-Issue Processors

- Superscalars
 - issue a varying number of instructions per clock cycle
 - either statically scheduled by the compiler
 - using in-order execution
 - or dynamically scheduled by the hardware
 - using out-of-order execution (techniques like Scoreboard, Tomasulo's Algorithm...)
- VLIW (Very Long Instruction Words) Processors
 - issue a fixed number of instructions formatted either as
 - one large instruction, or
 - a fixed instruction packet with parallelism explicitly indicated by the instruction (EPIC for IA-64 in Itanium processors)
 - inherently statically scheduled by the compiler

Example: Ideal Pipeline Execution for Static Dual-Issue MIPS Datapath

Inst.Type								
ALU/branch	IF	ID	EX	MEM	WB			
load/store	IF	ID	EX	MEM	WB			
ALU/branch		IF	ID	EX	MEM	WB		
load/store		IF	ID	EX	MEM	WB		
ALU/branch			IF	ID	EX	MEM	WB	
load/store			IF	ID	EX	MEM	WB	
ALU/branch				IF	ID	EX	MEM	WB
load/store				IF	ID	EX	MEM	WB

VLIW: Very Long Instruction Word

- A “very long” (64-bit, 128-bit,...) instruction encodes several operations that can be executed in parallel on multiple independent functional units
 - packets in EPIC, molecules in Transmeta
- Same compiler/architecture concepts for either
 - multiple operations organized into a single instruction
 - set of instructions grouped by the compiler into a single packet while excluding dependencies
- The burden of detecting dependencies falls on the compiler (simpler HW than in a Superscalar)
- VLIW is effective with wide-issue processors
 - more than 2-way issue
 - 16/24 bits per field (112/168 bits of instruction length)

VLIW vs. Superscalars

- VLIW executes operations in parallel based on a fixed schedule determined when programs are compiled
 - no scheduling hardware like in superscalars
 - increase computational power with less hardware complexity (but greater compiler complexity) than that associated with most superscalar CPUs
- One VLIW instruction encodes multiple operations, at best one operation for each execution unit in the data path
 - in superscalar designs, instead, the number of execution units is invisible to the instruction set
 - each instruction encodes only one operation

Example: Executing the Simple Loop on a MIPS VLIW Processor

- Assuming a VLIW that can issue up to 2 memory references, 2 FP operations and 1 integer/branch operation per clock cycle

Memory Ref. 1	Memory Ref. 2	FP 1	FP Instruction	Other
L.D F0, 0(R1)	L.D F6, -8(R1)			
L.D F10, -16(R1)	L.D F14, -24(R1)			
L.D F18, -32(R1)	L.D F22, -40(R1)	ADD.D F4, F0, F2	ADD.D F8, F6, F2	
L.D F26, -48(R1)		ADD.D F12, F10, F2	ADD.D F16, F14, F2	
		ADD.D F20, F18, F2	ADD.D F24, F22, F2	
S.D F4, -0(R1)	S.D F8, -8(R1)	ADD.D F28, F26, F2		
S.D F12, -16(R1)	S.D F16, -24(R1)			DADDUI R1,R1,#-56
S.D F20, 24(R1)	S.D F24, 16(R1)			
S.D F28, 8(R1)				BNE R1,R2, loop

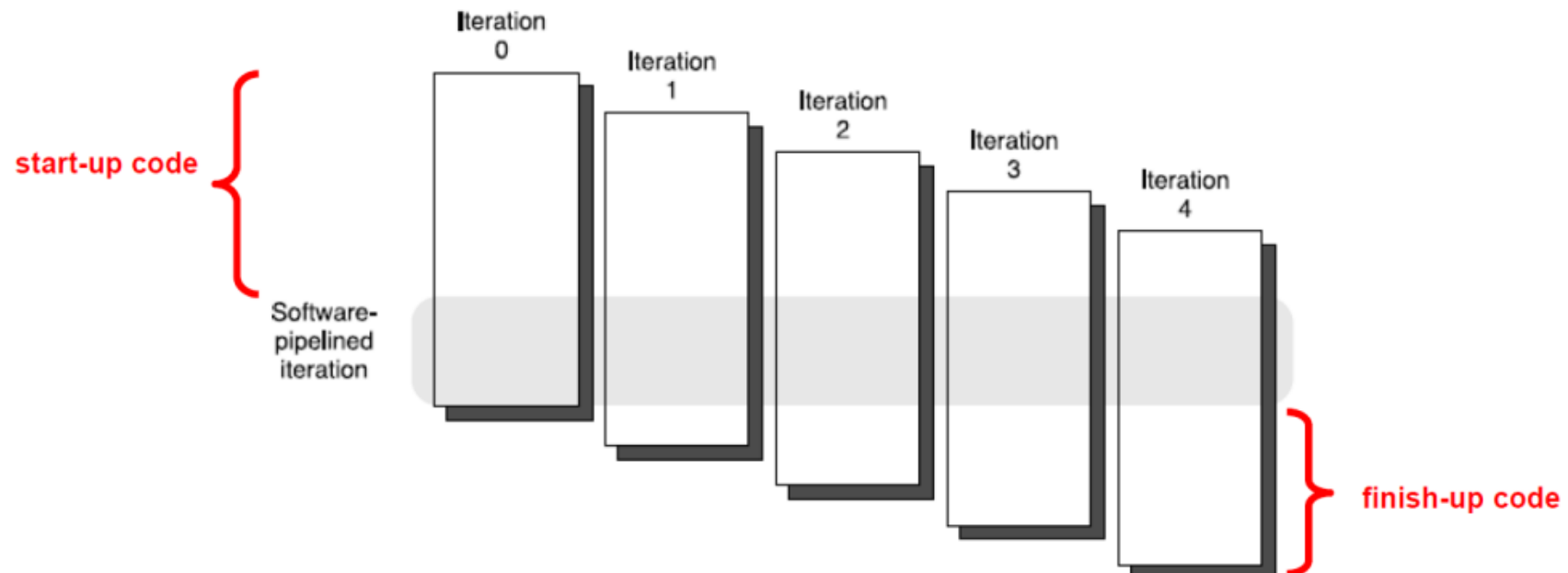
- unrolled loop 7 times (this eliminates stalls, i.e. empty issue cycles)
- execTime=9cycles
- execTimePerElement=9/7=1.29cycles
- VLIW speedup=2.71(with respect to loop unrolling)
- combined speedup(with loop unrolling)=5.43
- efficiency(% of busy slots)=23/45=51%

VLIWs: Techniques to Increase Efficiency

- To keep functional units busy, there must be enough parallelism in a code sequence
 - loop unrolling & scheduling
 - expand a loop by grouping instructions from multiple iterations and interleaving them
 - software pipelining & scheduling
 - keep the same number of instructions per loop but take instructions from different iterations and interleave them
 - local scheduling
 - if unrolling generates straight-line code
 - global scheduling
 - if necessary to schedule across branches
 - structurally more complex and harder to optimize

Software Pipelining

- Observation: if iterations from a loop are independent, then we can get ILP by taking instructions from different iterations
- Software pipelining: interleave instructions from different loop iterations
 - reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (Tomasulo in SW)



LOOP: 1 L.D F0,0(R1)
 2 ADD.D F4,F0,F2
 3 S.D F4,0(R1)
 4 DADDUI R1,R1,#8
 5 BNE R1,R2,LOOP

(5/5)
 2 RAWs!

Software Pipelining Example

1 L.D F0,0(R1)

2 ADD.D F4,F0,F2

3 S.D F4,0(R1)

4 L.D F0,-8(R1)

5 ADD.D F4,F0,F2

6 S.D F4,-8(R1)

7 L.D F0,-16(R1)

8 ADD.D F4,F0,F2

9 S.D F4,-16(R1)

Before: Unrolled 3 times (11/11)

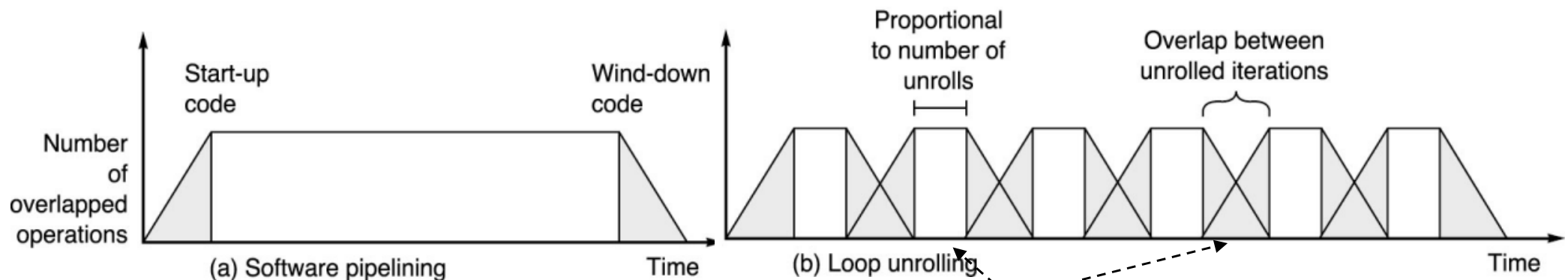
1 L.D F0,0(R1)
 2 ADD.D F4,F0,F2
 3 S.D F4,0(R1)
 4 L.D F0,-8(R1)
 5 ADD.D F4,F0,F2
 6 S.D F4,-8(R1)
 7 L.D F0,-16(R1)
 8 ADD.D F4,F0,F2
 9 S.D F4,-16(R1)
 10 DADDUI R1,R1,#24
 11 BNE R1,R2,LOOP

After: Software Pipelined (5/11)

1 L.D F0,0(R1)
 2 ADD.D F4,F0,F2
 4 L.D F0,-8(R1) No RAW! (2 WARs)
 3 S.D F4,0(R1); Stores M[i]
 5 ADD.D F4,F0,F2; Adds to M[i-1]
 7 L.D F0,-16(R1); loads M[i-2]
 10 DADDUI R1,R1,#8
 11 BNE R1,R2,LOOP
 6 S.D F4,-8(R1)
 8 ADD.D F4,F0,F2
 9 S.D F4,-16(R1)

Software Pipelining vs. Loop Unrolling

- Software pipelining
 - Can be thought of as symbolic loop unrolling
 - May use loop-unrolling to figure out how to pipeline the loop
- Loop unrolling reduces the overhead of the loop
 - the branch and counter update code
 - but every time the inner unrolled loop still need be initiated
- Software pipelining reduces the time when the loop is not running at peak speed to once per loop
 - major advantage: consumes less code space
 - In practice, compilation using software pipelining is quite difficult
- The best performance can come from doing both



25 loops with 4 unrolled iterations each
= 100 iterations

Example: Combining SW Pipelining and Loop Unrolling on a MIPS VLIW Processor

- assuming a VLIW that can issue up to 2 memory references, 2 FP operations and 1 integer/branch operation per clock cycle

MemoryRef.1	MemoryRef.2	FP1	FP2	Other
L.D F0,-48(R1)	S.D F4,0(R1)	ADD.D F4,F0,F2		
L.D F6,-56(R1)	S.D F8,-8(R1)	ADD.D F8,F6,F2		DADDUI R1,R1,#-24
L.D F10,-64(R1)	S.D F12,-16(R1)	ADD.D F12,F10,F2		BNE R1,R2,loop

- software pipelining with loop unrolling across 9 iterations of original loop
 - stores for iterations: i, i-8, i-16
 - adds for iterations: i-24, i-32, i-40
 - loads for iterations: i-48, i-56, i-64
- execTime=3cycles
- execTimePerElement=1cycle
- speedup=1.29(with respect to VLIW with loop unrolling only)
- efficiency(% of busy slots)=11/15=73%
- need much less FP registers(7 instead of 15)

Software Pipelining: Summary

- Major advantage over straight loop unrolling
 - SW pipelining consumes less code space
- Can be applied together with loop unrolling
 - they reduce different types of overhead
- Compilation using SW pipelining is quite difficult
 - many loops require significant transformations
 - trade-offs in terms of overhead versus efficiency of the SW-pipelined loop are complex to analyze
 - register management can be tricky
- Some modern processors add specialized HW support for SW pipelining
 - Intel IA-64 ISA

VLIWs: Handling Stalls

- Early VLIWs operated in lockstep
 - no hazard-detection hardware
 - a stall in any functional unit causes the entire pipeline to stall (since the units must be kept synchronized)
- In more recent VLIWs, the hardware allows for unsynchronized execution (the functional units operate more independently)
 - e.g. in NXP Trimedia processor
 - HW does not detect hazards (if they are present, then the execution will be incorrect)
 - compiler explicitly includes NOPs when a field cannot be used (due to intra-packet or inter-packet dependencies)
 - the code stream is compressed in memory and instructions are expanded only when they are fetched from the I-cache

Beyond VLIWs:

Handling Backward Compatibility

- In a strict VLIW approach, the code sequence makes use of both the ISA and the detailed pipeline structure (including both functional units and their latencies)
 - this makes migrating between successive implementations difficult
 - instead, superscalars, which may also require recompilation for performance purposes, do allow to run old binary files
- EPIC architectures (such as IA-64) provide solutions to many of the early VLIW approach problems
 - multiple software instructions are grouped in bundles
 - each of the bundles has information indicating if this set of operations is depended upon by the subsequent bundle
 - future implementations can be built to issue multiple bundles in parallel
 - the dependency information is calculated by the compiler, so the hardware does not have to perform operand dependency checking
 - other architectural improvements such as
 - predicated execution
 - very large architectural register files to avoid register renaming

In-class Quiz #2

Quiz 2

The following loop computes $Y[i] = a \times X[i] + Y[i]$, the key step in a Gaussian elimination. Assume the pipeline latencies from the table.

Loop:

L.D	F0, 0(R1)	;load X[i]
MUL.D	F0, F0, F2	;multiply a*X[i]
L.D	F4, 0(R2)	;load Y[i]
ADD.D	F0, F0, F4	;add a*X[i]+Y[i]
S.D	0(R2), F0	;store Y[i]
DSUBUI	R1, R1, #8	;decrement X index
DSUBUI	R2, R2, #8	;decrement Y index
BNEZ	R1, loop	;loop if not done

Instruction producing result	Instruction using result	Latency in clock cycle
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

- Unroll the loop as many times as necessary to schedule the loop body (from the first instruction to fifth instruction) without any delays. Show the schedule.
- Assume there is one stall between the “Integer ALU op” and “branch instruction”, and the branch instruction is a 1-cycle delayed branch. How to schedule code to remove all the stalls in the loop and what is the execution time per iteration?