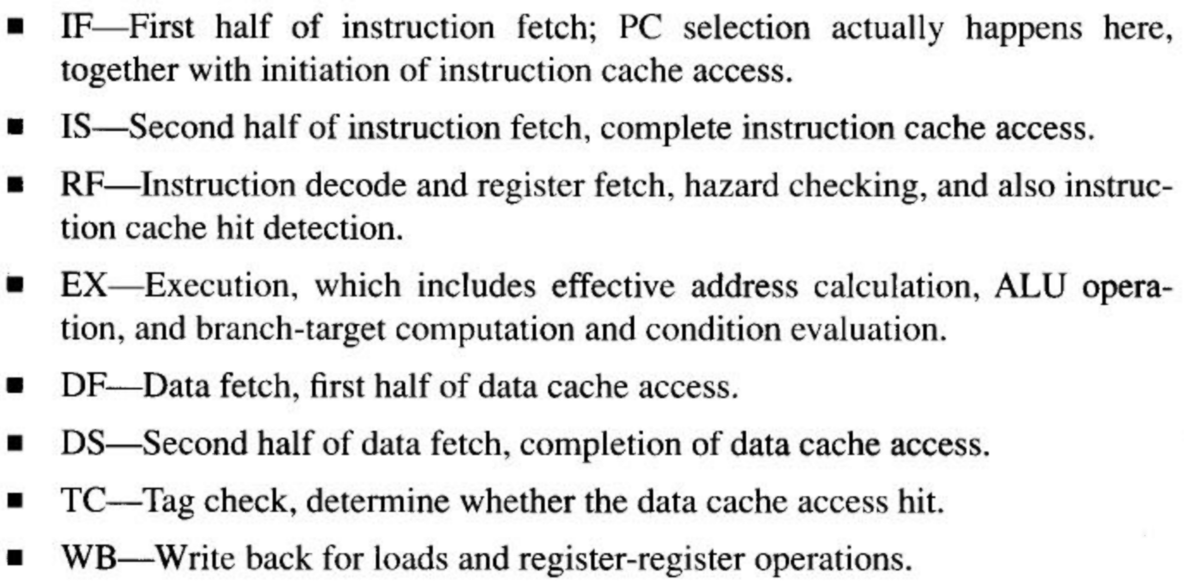
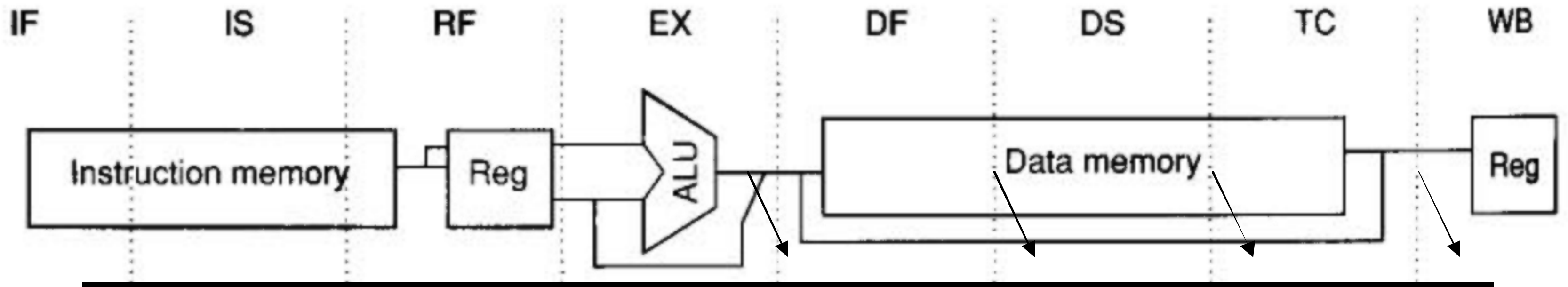
The MIPS R4000 Pipeline (circa 1991)

- ***M**icroprocessor without **I**nterlocked **P**ipeline **S**tages (MIPS)*
  - Challenge S
  - Challenge M
  - Crimson
  - Indigo
  - Indigo 2
  - Indy
- R4300 (embedded)
  - Nintendo-64



- IF—First half of instruction fetch; PC selection actually happens here, together with initiation of instruction cache access.
- IS—Second half of instruction fetch, complete instruction cache access.
- RF—Instruction decode and register fetch, hazard checking, and also instruction cache hit detection.
- EX—Execution, which includes effective address calculation, ALU operation, and branch-target computation and condition evaluation.
- DF—Data fetch, first half of data cache access.
- DS—Second half of data fetch, completion of data cache access.
- TC—Tag check, determine whether the data cache access hit.
- WB—Write back for loads and register-register operations.

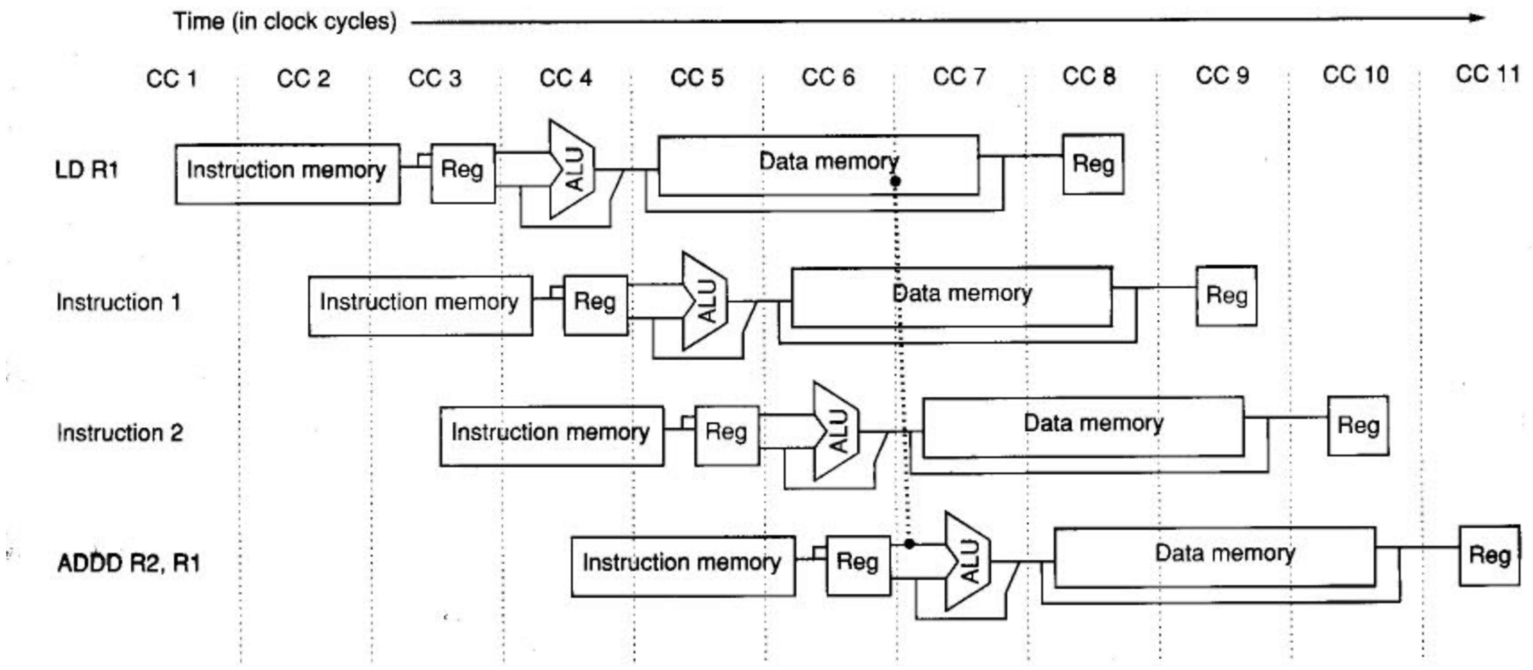
[redacted]



[redacted]

2-cycle delay for Loads

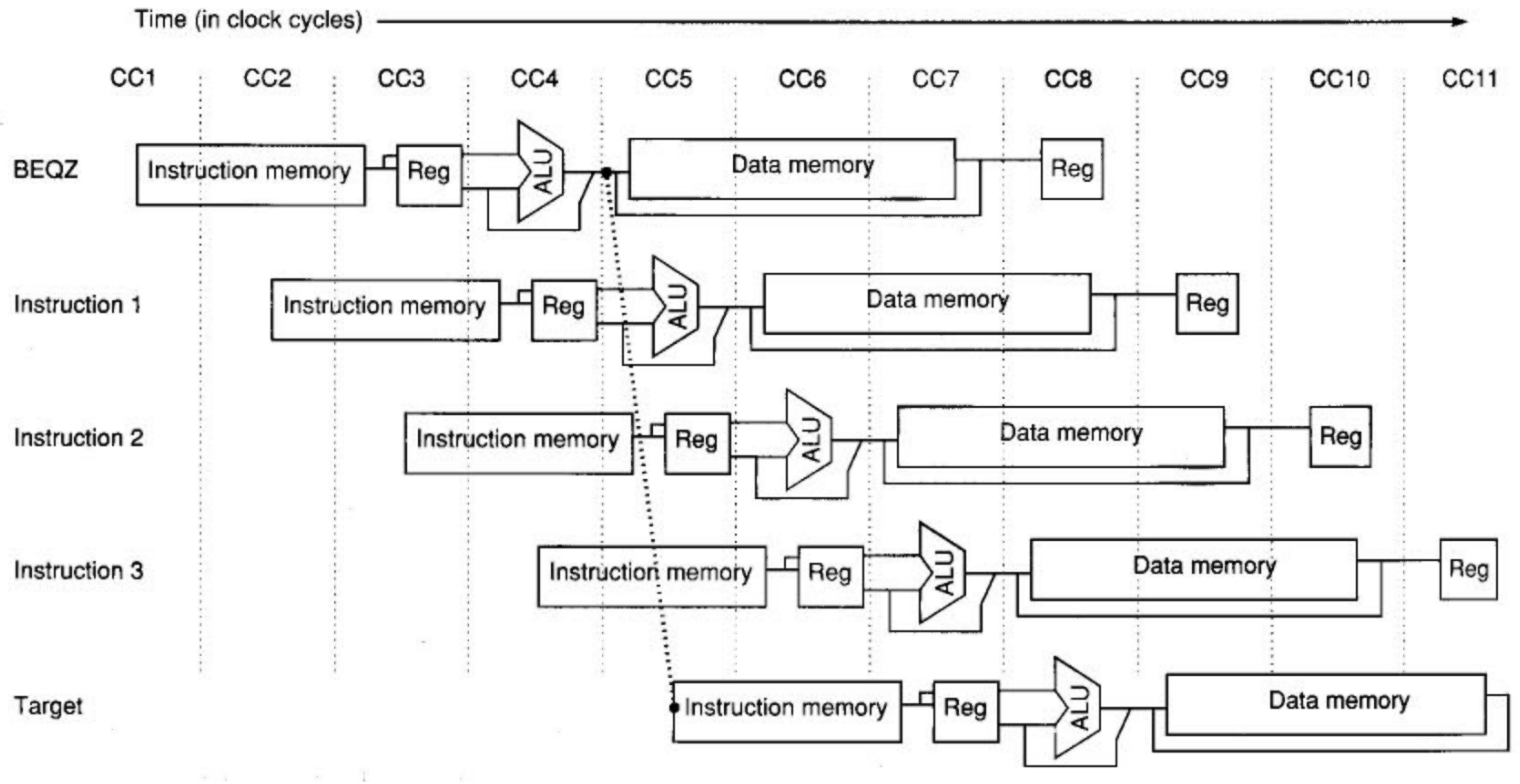
- Data is not available until end of DS



Instruction number	1	2	3	4	5	6	7	8	9
LD R1,...	IF	IS	RF	EX	DF	DS	TC	WB	
DADD R2,R1,...		IF	IS	RF	stall	stall	EX	DF	DS
DSUB R3,R1,...			IF	IS	stall	stall	RF	EX	DF
OR R4,R1,...				IF	stall	stall	IS	RF	EX

3-cycle branch delay

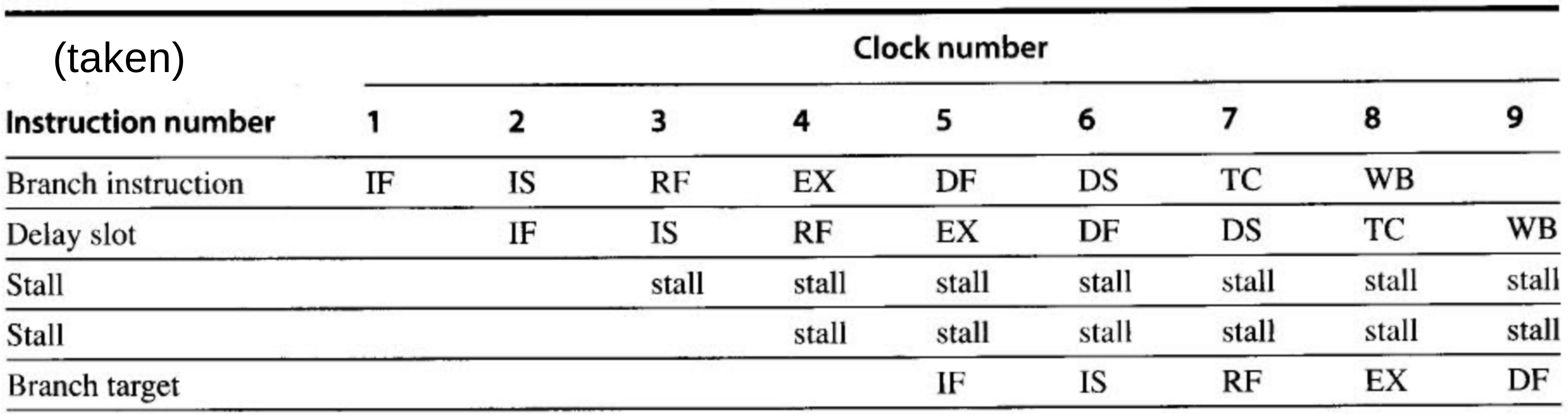
- Branch computed during EX (cycle 4)



MIPS 4000 branch delay implementation

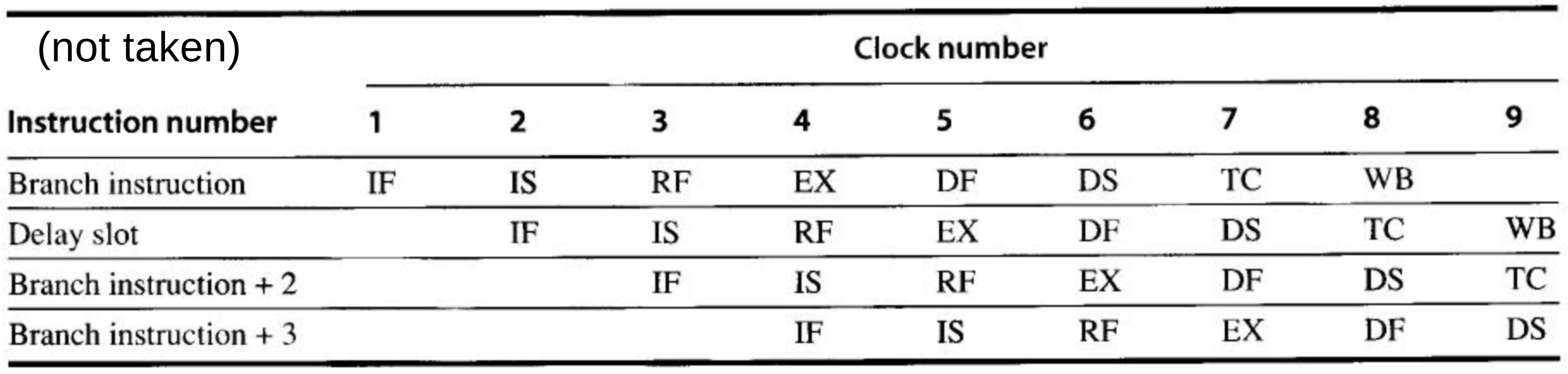
- Single cycle delayed branch for prediction
- Predicted not taken for other cycles

(taken)



	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Branch instruction	IF	IS	RF	EX	DF	DS	TC	WB	
Delay slot		IF	IS	RF	EX	DF	DS	TC	WB
Stall			stall	stall	stall	stall	stall	stall	stall
Stall				stall	stall	stall	stall	stall	stall
Branch target					IF	IS	RF	EX	DF

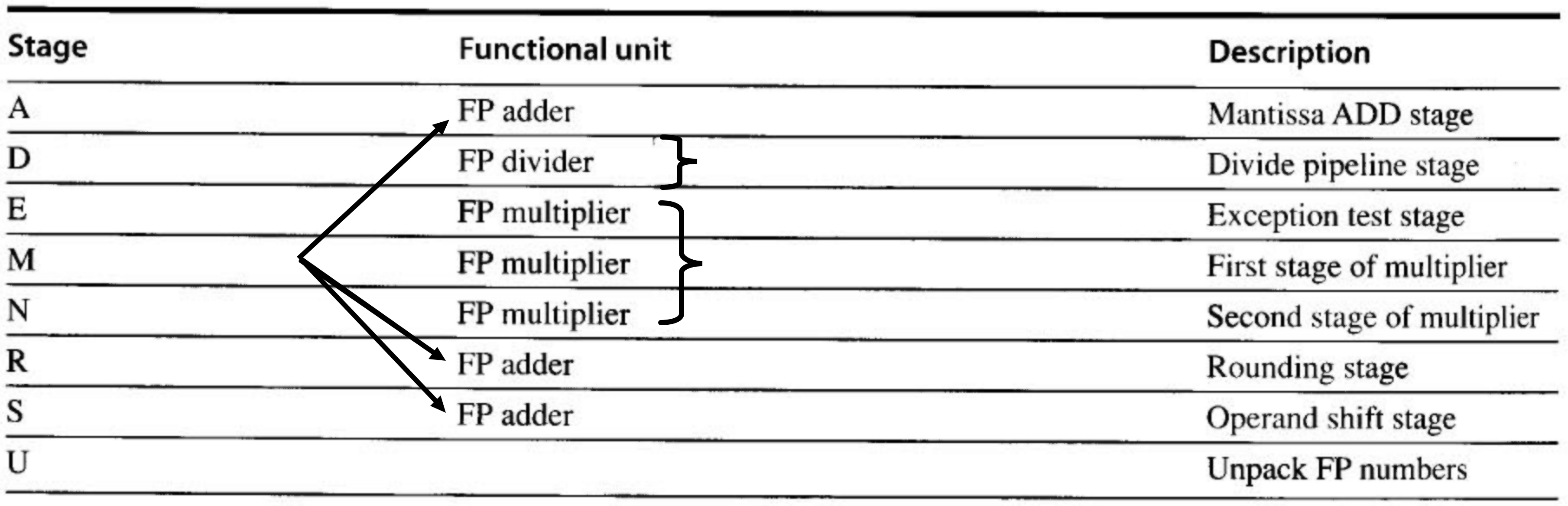
(not taken)



	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Branch instruction	IF	IS	RF	EX	DF	DS	TC	WB	
Delay slot		IF	IS	RF	EX	DF	DS	TC	WB
Branch instruction + 2			IF	IS	RF	EX	DF	DS	TC
Branch instruction + 3				IF	IS	RF	EX	DF	DS

8-stage FP Pipeline

- 3 functional units: FP Add, FP Mult, FP Divide
- Length varies from 2 -> 112 cycles for FP double (negate vs. sqrt)
- 8 stages of FP functional unit to perform op
  - Can use a stage 0 or more times depending on op



Stage	Functional unit	Description
A	FP adder	Mantissa ADD stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

Common latency and initiation intervals for FP

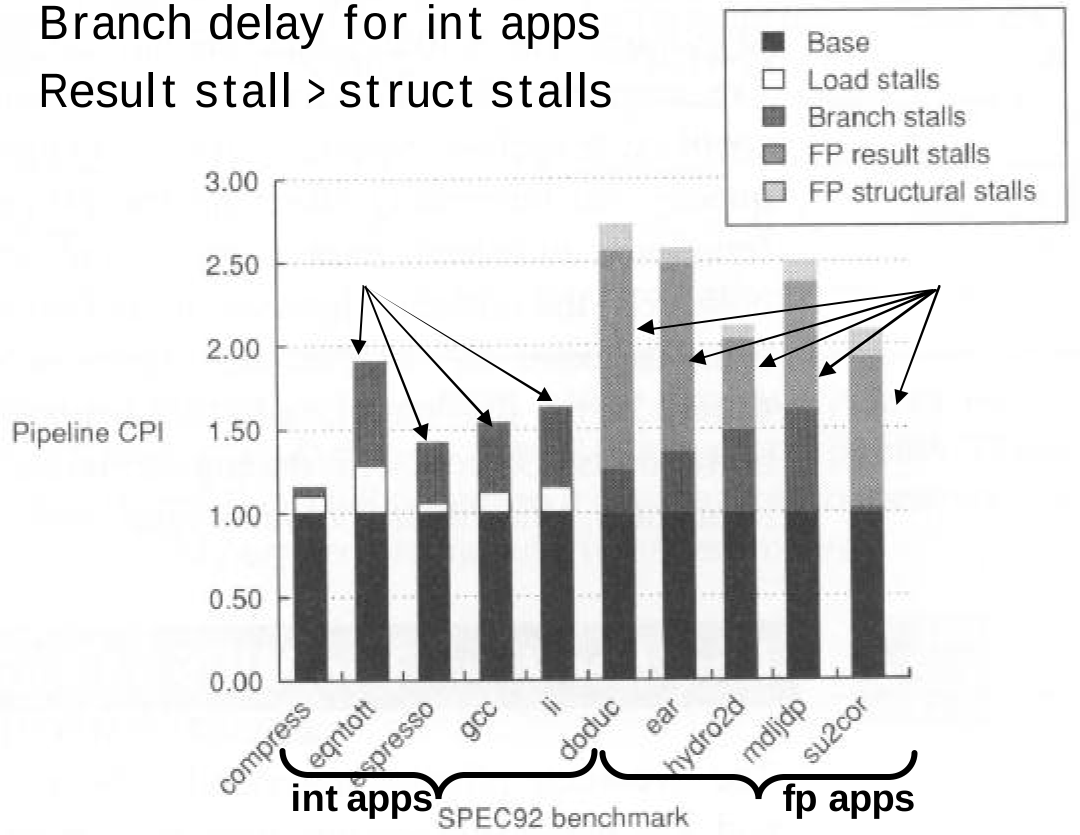
FP instruction	Latency	Initiation interval	Pipe stages
Add, subtract	4	3	U, S + A, A + R, R + S
Multiply	8	4	U, E + M, M, M, M, N, N + A, R
Divide	36	35	U, A, R, D^{27}, D + A, D + R, D + A, D + R, A, R
Square root	112	111	U, E, (A+R)^{108}, A, R
Negate	2	1	U, S
Absolute value	2	1	U, S
FP compare	3	2	U, A, R

		Clock cycle												
Operation	**Issue/stall**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Add	Issue	U	S + A	A + R	R + S									
Multiply	Issue		U	E + M	M	M	M	N	N + A	R				
(OR)	Issue			U	M	M	M	M	N	N + A	R			

		Clock cycle												
Operation	**Issue/stall**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Add	Issue	U	S + A	A + R	R + S									
Divide	Stall		U	A	R	D	D	D	D	D	D	D	D	D
(OR)	Issue			U	A	R	D	D	D	D	D	D	D	D
(OR)	Issue				U	A	R	D	D	D	D	D	D	D

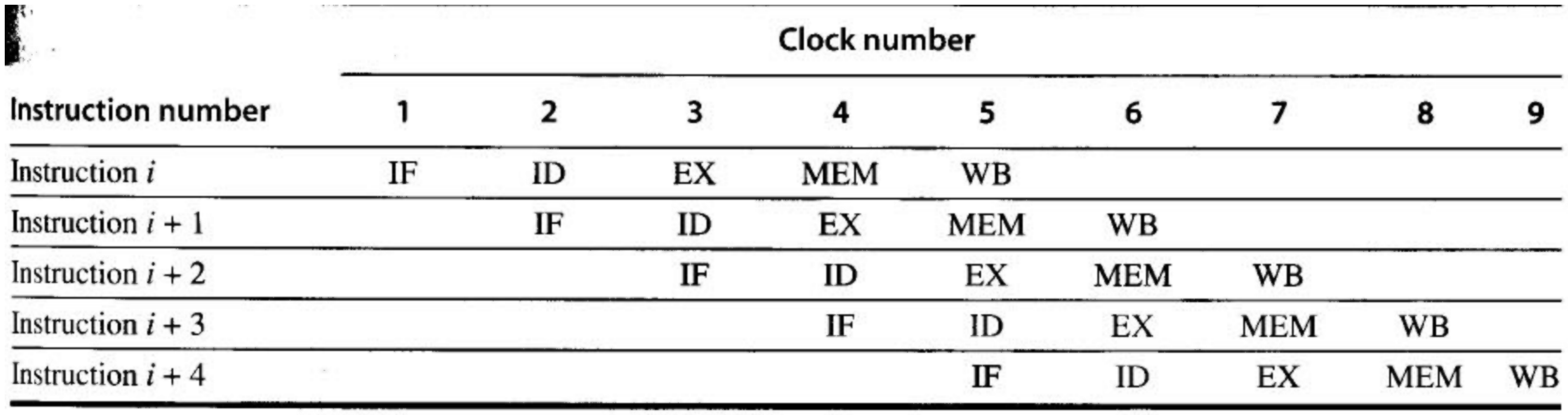
MIPS 4000 Performance Summary

- Major causes of pipeline stalls
  - Load stalls (e.g. instr in next 2 cycles depends on load result)
  - Branch stalls (e.g. taken branches = 2 cycles, poor delay slot use)
  - FP result stalls (e.g. instr following depends on FP operand)
  - FP structural stalls (e.g. stalls due to single write port, multiple WBs)
    - Branch delay for int apps
    - Result stall > struct stalls



Instruction-level Parallelism

- Pipelining is a form of instruction-level parallelism
- Main idea: overlap the execution of instructions to improve performance
- Instructions are evaluated simultaneously or *in parallel*



	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Remember: It is not this simple.

Pipeline CPI = Ideal Pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls

Approaches to Exploiting ILP

- Consider a basic block
  - Sequential series of instructions without branch
  - Branch frequency at least 1 in 7 instr (for MIPS)
  - Potential overlap in single basic block is small
  - Must exploit ILP across basic blocks
- Loop-level parallelism

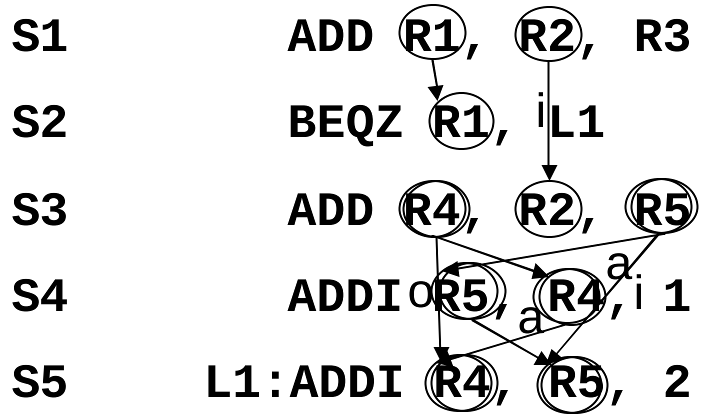
```
for (i=1; i<=1000; i=i+1)
    x[i] = x[i] + y[i];
```

  - Static Exploitation of ILP: Compiler unrolls loop at compile-time
  - Dynamic Exploitation of ILP: Hardware unrolls loop at run-time
- Problem: Dependencies kill parallelism.
  - Dependent instructions must be executed in-order

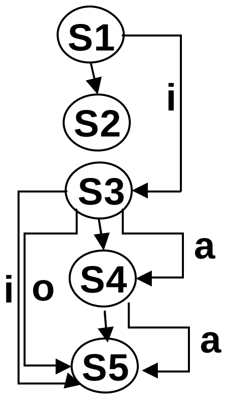
Dependence Relations

- Dependence → a relation between two statements (S1, S2) that constrains program order
- Two types: (assume S2 follows S1)
  - Control Dependence: constraint due to control flow
  - Data Dependence: constraint from flow of data
    - Flow dependence: S1 sets a value that S2 uses (aka true dependence)
    - Antidependence: S1 uses a value that S2 sets
    - Output dependence: S1 and S2 set the values of some variable
    - Input dependence: S1 and S2 read the value of some variable

- Example:



```
S1         ADD R1, R2, R3
S2         BEQZ R1, L1
S3         ADD R4, R2, R5
S4         ADDI R5, R4, 1
S5      L1:ADDI R4, R5, 2
```



Data Dependence Graph

Overcoming Data Dependencies

Data dependences (such as these) are properties of <u>code</u>.

For HW: interlocking pipeline must stall issue to ensure correct completion.

For SW: compiler must schedule instructions to ensure correct completion.

1. Data dependence indicates potential for a hazard
2. Determines order in which results must be completed
3. Sets a bound on achievable parallelism

- How to overcome data hazards while maintaining correctness?
  1. Maintain dependence but avoid hazard (code scheduling)
  2. Eliminate dependences by transforming codes

- Why is dependence detection difficult?
  - Between Regs: Straightforward except in branching
  - Between Mem locations:
    - Not as easy to detect using simple comparison
    - False positive: 20(R4) and 20(R4) refer to diff locations
    - False negative: 100(R4) and 20(R6) refer to same location

Name Dependences

Other data dependences are properties of machine/compiler.

Name dependence: Two instructions use the same register or memory location (name), but no flow of data between the instructions associated with name

- "False" or artificial dependence created between two instructions
- No value being "transmitted" between instructions

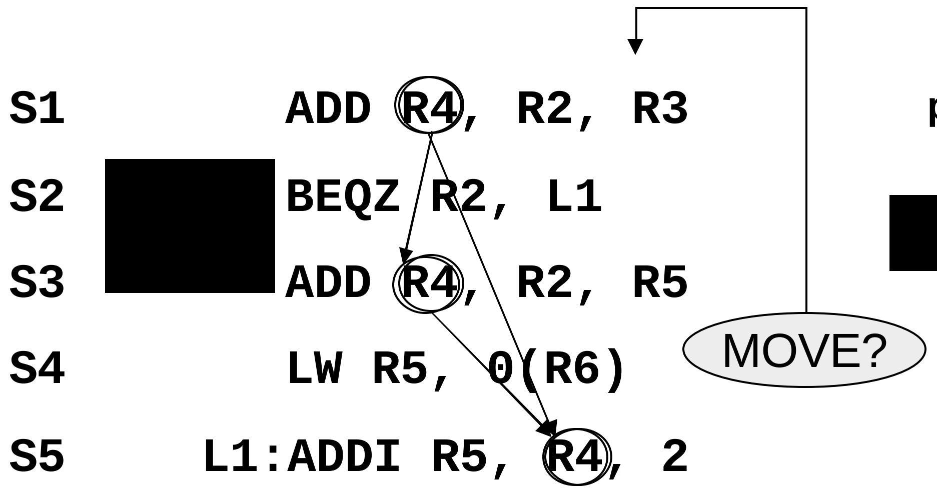
- Types of name dependences between instr ***i*** preceding instr ***j***
  - Antidependence: ***j*** writes a name ***i*** reads (e.g. WAR pipeline hazard)
  - Output dependence: ***j*** writes a name ***i*** writes (e.g. WAW pipeline hazard)
- How to overcome?
  - Register renaming (static by compiler or dynamic by HW)

Pipeline Data Hazard Types

- Pipeline hazard occurs when dependence exists and instr's are close enough that instr overlap changes operand and access order
- Named by "ordering that must be preserved"
- Types of data hazards (***i*** followed be ***j***)
  - Read After Write (RAW)
    - ***j*** tries to read operand before ***i*** writes it
  - Write After Read (WAR)
    - ***j*** tries to write operand before ***i*** reads it
  - Write After Write (WAW)
    - ***j*** tries to write operand before ***i*** writes it

Control Dependences

- Simple approach to preserve control dependence
  - Program order + wait to execute branch target until known
  - Problem: limits performance of branches
  - Instruction order + branch order not absolute requirements
- Critical correctness properties
  - Exception behavior – any changes in instruction order must preserve behavior of any exceptions raised in the program
  - Data flow
  - Example:



```
S1          ADD R4, R2, R3
S2  [redacted]  BEQZ R2, L1
S3          ADD R4, R2, R5
S4          LW R5, 0(R6)
S5      L1:ADDI R5, R4, 2
```

What if LW causes protection exception?

[redacted]

Dynamic Scheduling

- Forwarding or bypassing decreases effective pipeline stalls
- Hazard detection unit causes stalls due to remaining dependences
- Such dependences often cannot be detected at compile time
- If we maintain exceptions and data flow
  - Hardware can rearrange order of instruction execution
  - Further reduce stalls in the pipeline (exploit more ILP)
- Design of Dynamic Scheduling will be more complex
  - But simpler for well-designed ISA
  - Performance gain must be significant to warrant complexity

Our pipeline so far...

- In-order instruction issue
  - Once one instruction stalls, no later instruction can proceed
- Structural and data hazards checked in ID stage (MIPS 5-stage)
  - To begin execution when operands are available (dynamic scheduling)
    - Separate ID into Issue (IS) + Register Fetch (RF) [MIPS 4000]
    - Check structural hazards on IS (in-order)
    - Check data hazards on RF
    - Result: in-order issue, out-of-order execution, out-of-order completion
- Dynamically scheduled pipeline
  - Instructions Fetched (IF) and Issued (IS) in-order
  - Instructions can stall or bypass each other in Register Fetch (RF)
  - How to implement?
- Scoreboarding
  - Goal: maintain one instruction per clock cycle

Our MIPS Scoreboard Architecture

Control similar for two funct units vs. super-pipelined depth of 2

- MIPS 5-stage: IF→ID→EX→MEM→WB

- MIPS R4000:

Decode + struct hazards

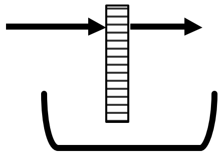
Data hazards + read operands

IF→IS+RF→EX→DF+DS+TC→WB

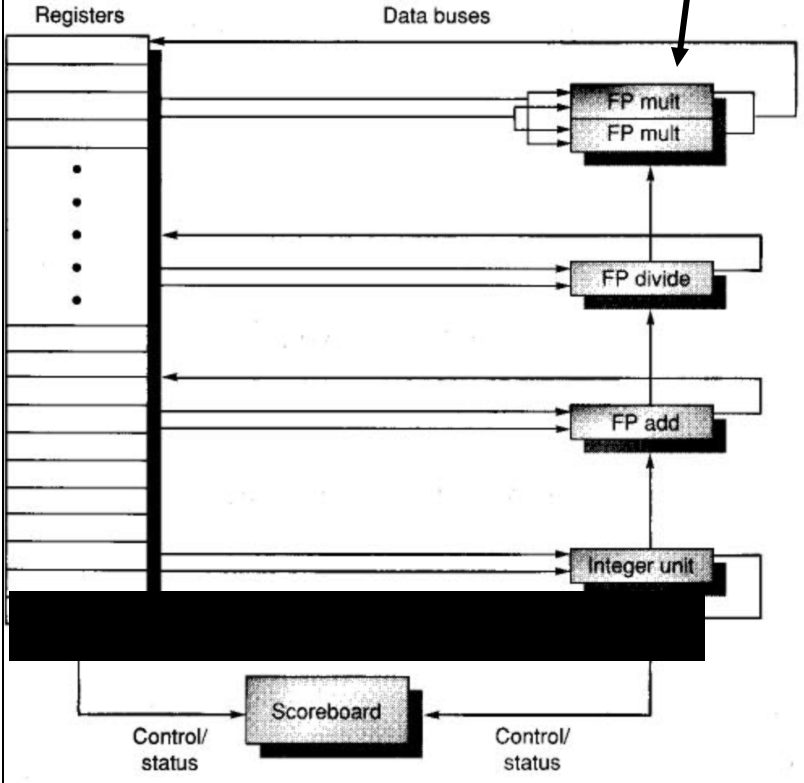
Out-of-order execution and completion

[redacted]

Centralized hazard detection and resolution



In-order fetch+issue (w/ buffer)



[redacted]

[redacted]

FP Operations on scoreboard in MIPS

- Four steps of Scoreboard for MIPS FP pipeline
  - IS
    - Check for functional unit free
    - Check for active instruction with same destination register (WAW)
    - Issue or stall and update internal data structure
    - Note: issue stage will stall IF stage unless buffer present between
  - RF
    - Check for pending writes to source operands (RAW)
    - Allow register read or stall and update internal data structure
  - EX
    - On operands perform operation (may be multiple cycles)
    - Notify scoreboard (update internal data structure) upon completion
  - WB
    - Check for writes to pending read operations (WAR)
    - Allow register write or stall and update internal data structure

Scoreboard Components

(6 instructions)

Instruction status

Instruction		IS:	RF:	EX:	WB:
L.D	F6,34(R2)	√	√	√	√
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				

Which step is instruction currently in?

(5 funct units)

Functional unit status

Name	**Busy**	**Op**	**Fi**	**Fj**	**Fk**	**Qj**	**Qk**	**Rj**	**Rk**
Integer	Yes	Load	F2	R3				No	
Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes
			(dest reg)	(source regs)		(funct unit producing source regs)		(operands ready and not yet read?)	

Register result status

	F0	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**...**	**F30**
FU	Mult1	Integer			Add	Divide			

Just before MUL.D writes result

(6 instructions)

Instruction status

Instruction			**IS:**	**RF:**	**EX:**	**WB:**
L.D	F6,34(R2)		√	√	√	√
L.D	F2,45(R3)	RAW to MUL.D, ADD.D, SUB.D	√	√	√	√
MUL.D	F0,F2,F4	RAW to DIV.D	√	√	√	
SUB.D	F8,F6,F2	RAW to ADD.D (struct also)	√	√	√	√
DIV.D	F10,F0,F6	WAR to ADD.D	√			
ADD.D	F6,F8,F2	WAR w/ SUB.D	√	√	√	

Assume ADD = 2 cyc, MUL = 10 cyc, DIV = 40 cyc.

(5 funct units)

Functional unit status

Name	**Busy**	**Op**	**Fi**	**Fj**	**Fk**	**Qj**	**Qk**	**Rj**	**Rk**
Integer	No								
Mult1	Yes	Mult	F0	F2	F4			No	No
Mult2	No								
Add	Yes	Add	F6	F8	F2			No	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes
			(dest reg)	(source regs)		(funct unit producing source regs)		(operands ready and not yet read?)	

Register result status

	F0	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**...**	**F30**
FU	Mult 1			Add		Divide			

Just before DIV.D writes result

(6 instructions)

Instruction status

Instruction			IS:	RF:	EX:	WB:
L.D	F6,34(R2)		√	√	√	√
L.D	F2,45(R3)	RAW to MUL.D, ADD.D, SUB.D	√	√	√	√
MUL.D	F0,F2,F4	RAW to DIV.D	√	√	√	√
SUB.D	F8,F6,F2	RAW to ADD.D (struct also)	√	√	√	√
DIV.D	F10,F0,F6	WAR to ADD.D	√	√	√	
ADD.D	F6,F8,F2	WAR w/ SUB.D	√	√	√	√

Assume ADD = 2 cyc, MUL = 10 cyc, DIV = 40 cyc.

(5 funct units)

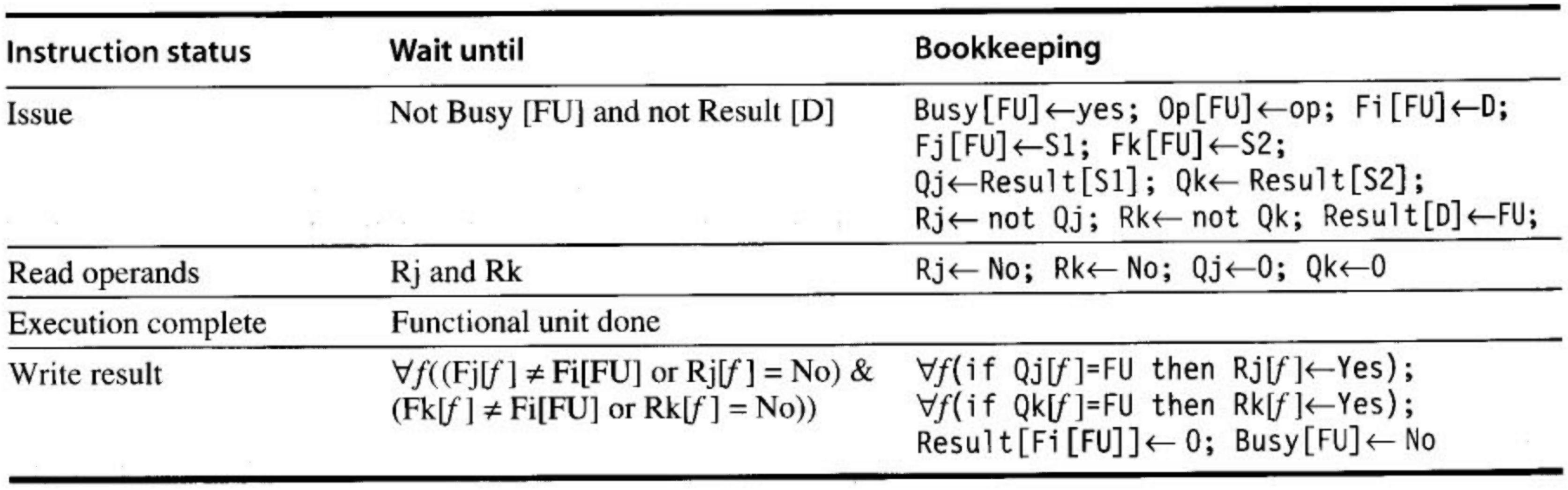
Functional unit status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	No								
Mult2	No								
Add	No								
Divide	Yes	Div	F10	F0	F6			No	No
			(dest reg)	(source regs)		(funct unit producing source regs)		(operands ready and not yet read?)	

Register result status

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU						Divide			

Summary of Scoreboard Checks and Actions



Instruction status	Wait until	Bookkeeping
Issue	Not Busy [FU] and not Result [D]	`Busy[FU]←yes; Op[FU]←op; Fi[FU]←D; Fj[FU]←S1; Fk[FU]←S2; Qj←Result[S1]; Qk← Result[S2]; Rj← not Qj; Rk← not Qk; Result[D]←FU;`
Read operands	Rj and Rk	`Rj← No; Rk← No; Qj←0; Qk←0`
Execution complete	Functional unit done	
Write result	$\forall f$((Fj[f] ≠ Fi[FU] or Rj[f] = No) & (Fk[f] ≠ Fi[FU] or Rk[f] = No))	$\forall f$(`if Qj[`f`]=FU then Rj[`f`]←Yes);` $\forall f$(`if Qk[`f`]=FU then Rk[`f`]←Yes);` `Result[Fi[FU]]← 0; Busy[FU]← No`

Scoreboard Limitations

- Amount of parallelism available
  - Later we look at studies of available ILP
- Window: number of scoreboard entries
  - How far ahead can we look?
  - Branches? Increased complexity?
  - Increase window size helps, but complexity → cycle time
- Number and types of functional units
  - Complexity vs. potential for ILP vs. structural hazards
  - Increase funct units helps, but complexity → cycle time
- Presence of anti and output dependences
  - WAW important with branch prediction

- Chapter 3 continues exploitation of ILP with hardware…