Graduate Computer Architecture

Chapter 3 – Instruction Level Parallelism and Its Dynamic Exploitation

Overview

- Instruction level parallelism
- Dynamic Scheduling Techniques
  - Scoreboarding (Appendix A.8)
  - Tomasulo's Algorithm
- Reducing Branch Cost with Dynamic Hardware Prediction
  - Basic Branch Prediction and Branch-Prediction Buffers
  - Branch Target Buffers

Instruction Level Parallelism (ILP)

- Pipelining supports a limited sense of ILP
  - e.g. overlapped instructions, out of order completion, hazard issues, bypass logic, etc.
- Remember

Pipeline CPI = Ideal Pipeline CPI + Struct. Stalls
+RAW Stalls+WAR Stalls+WAW Stalls
+ Control Stalls

- NOTE: stall terms are really contributions to the CPI
  - better method is: (ideal cycles + cycles due to stalls)/IC = $CPI_{pipeline}$
- Now turn consideration to
  - methods which reduce the terms on the RHS of the equation
  - solution is of course a combo of HW and SW/Compiler methods

Pipelining

- Split instruction execution into independent stages
- Start new instruction every cycle
- Throughput is increased
- CPI approaches 1
- Ideal speedup equals number of pipe stages
- Amount of work per-stage must be balanced
- Example: DLX(5 stages)
  - IF (Instruction fectch)
  - ID (Instruction decode / register fetch cycle)
  - EXE (Execution/ effective address)
  - MEM (Memory access / branch completion)
  - WB (Write-back)

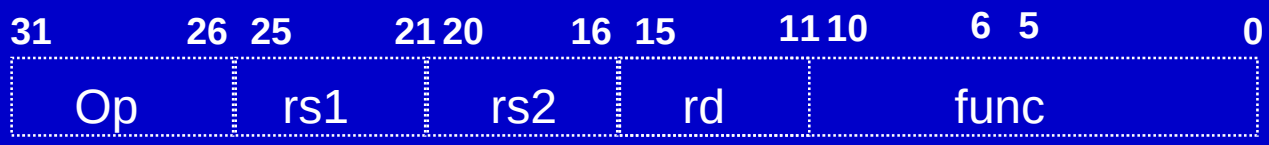
Computer Pipelines

- Computers execute billions of instructions, so instruction throughout is what matters
- Divide instruction execution up into several pipeline stages. For example

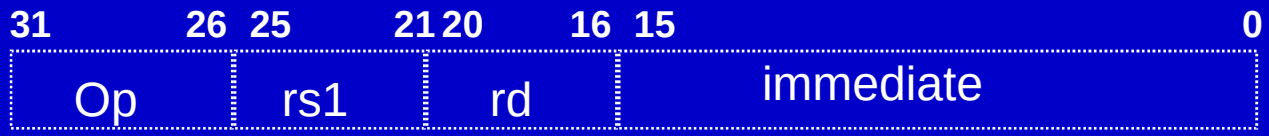
  IF ID EX MEM WB
- Simultaneously have different instructions in different pipeline stages
- The length of the longest pipeline stage determines the cycle time
- DLX desirable pipeline features:
  - all instructions same length
  - registers located in same place in instruction format
  - memory operands only in loads or stores

DLX Instruction Formats

Register-Register (R-type) **ADD R1, R2, R3**



Register-Immediate (I-type) **SUB R1, R2, #3**



Jump / Call (J-type) **JUMP end**

(jump, jump and link, trap and return from exception)

Multiple-Cycle DLX: Cycles 1 and 2

- Most DLX instruction can be implemented in 5 clock cycles
- The first two clock cycles are the same for every instruction.

1. Instruction fectch cycle (IF)

   load instruction

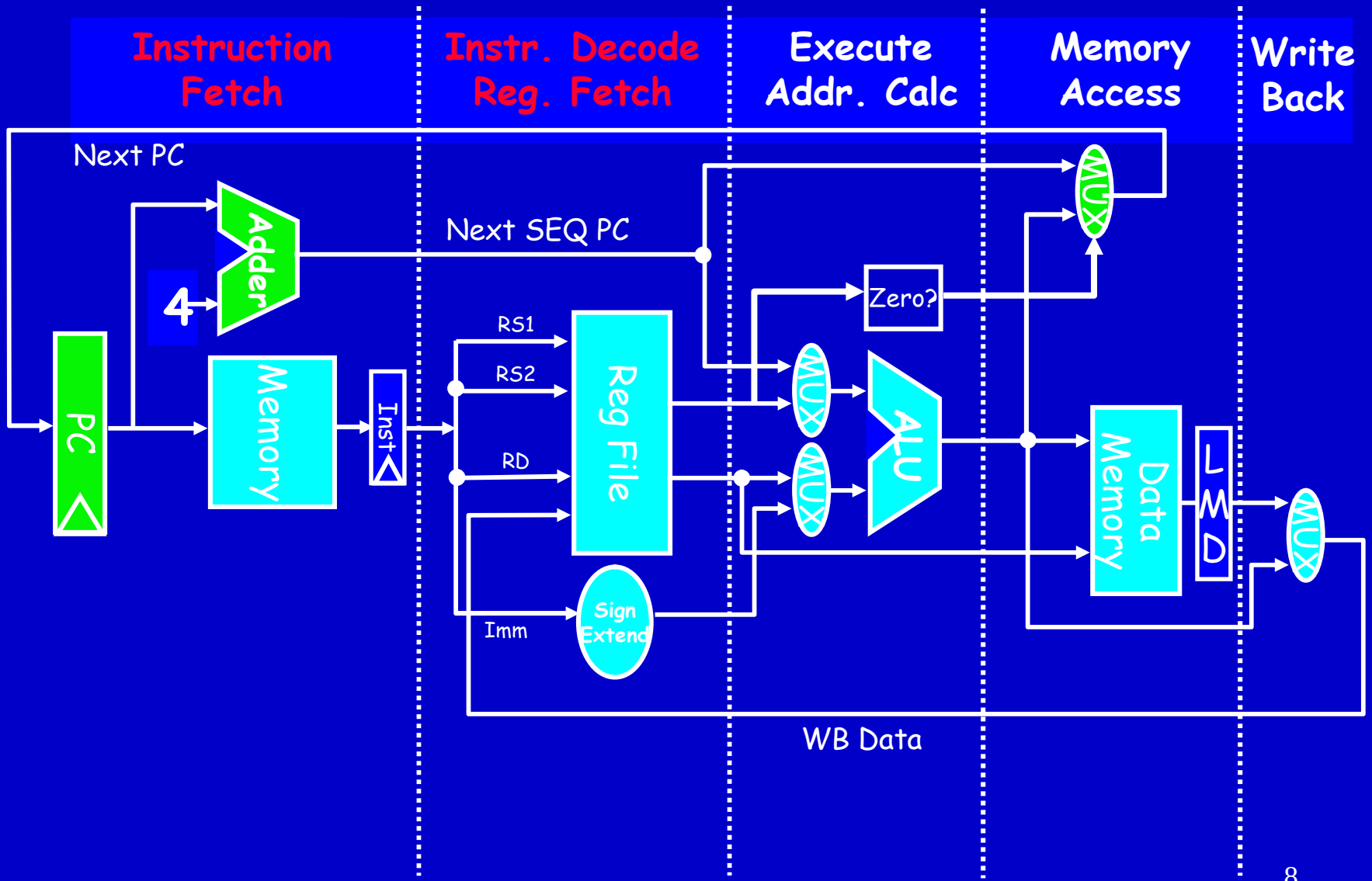
   update program counter

2. Instruction decode / register fetch cycle (ID)

   fetch source registers

   sign-extend immediate field

5 Steps of DLX Datapath

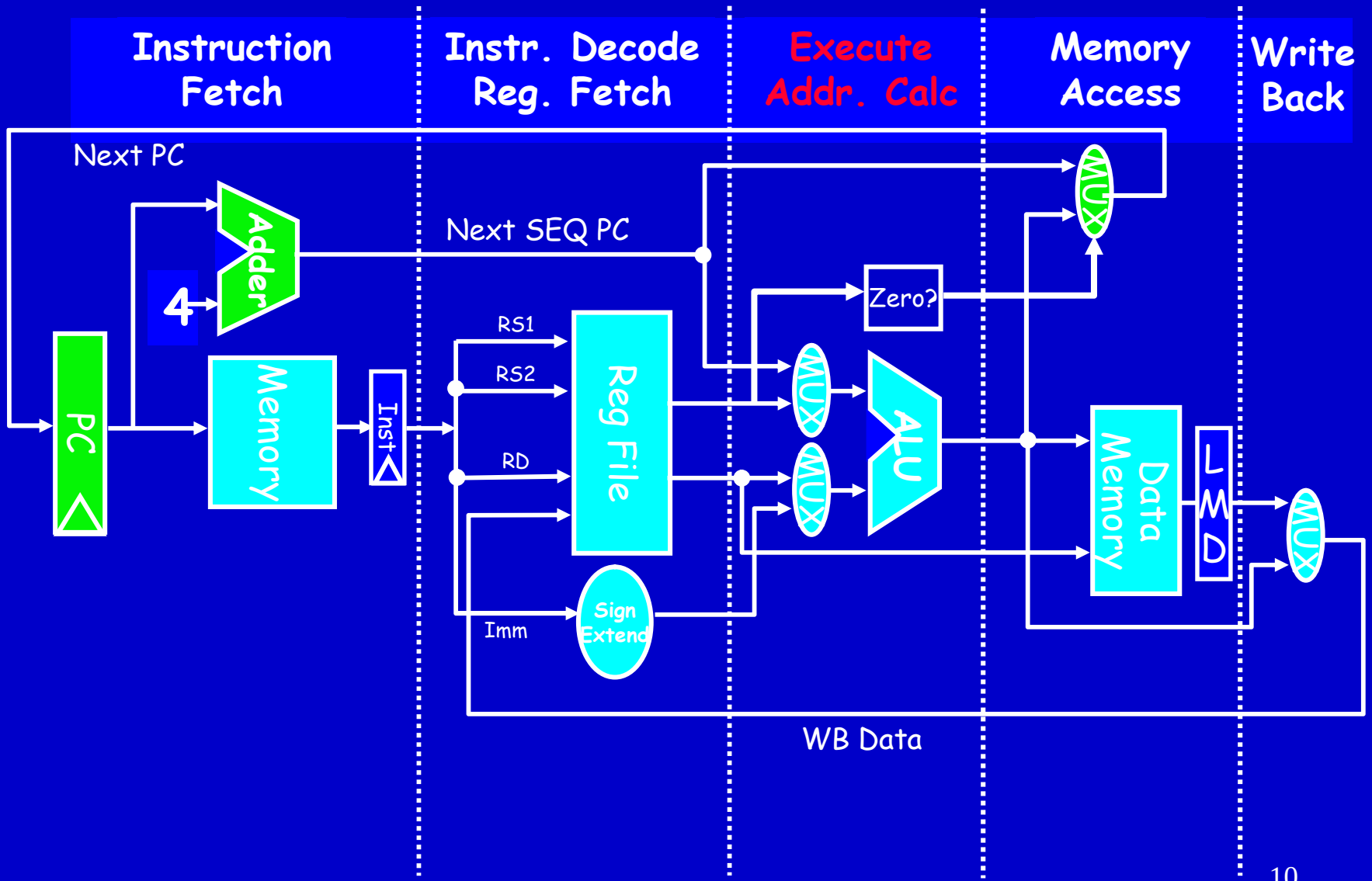


Multiple-Cycle DLX: Cycle 3

- The third cycle is known as the

  Execution/ effective address cycle (EX)
- The actions performed in this cycle depend on the type of operations.
  - Loads and Stores
    - calculate effective address
  - ALU operations
    - perform ALU operation
  - Branch
    - compute branch target
    - determine if the branch is taken

5 Steps of DLX Datapath



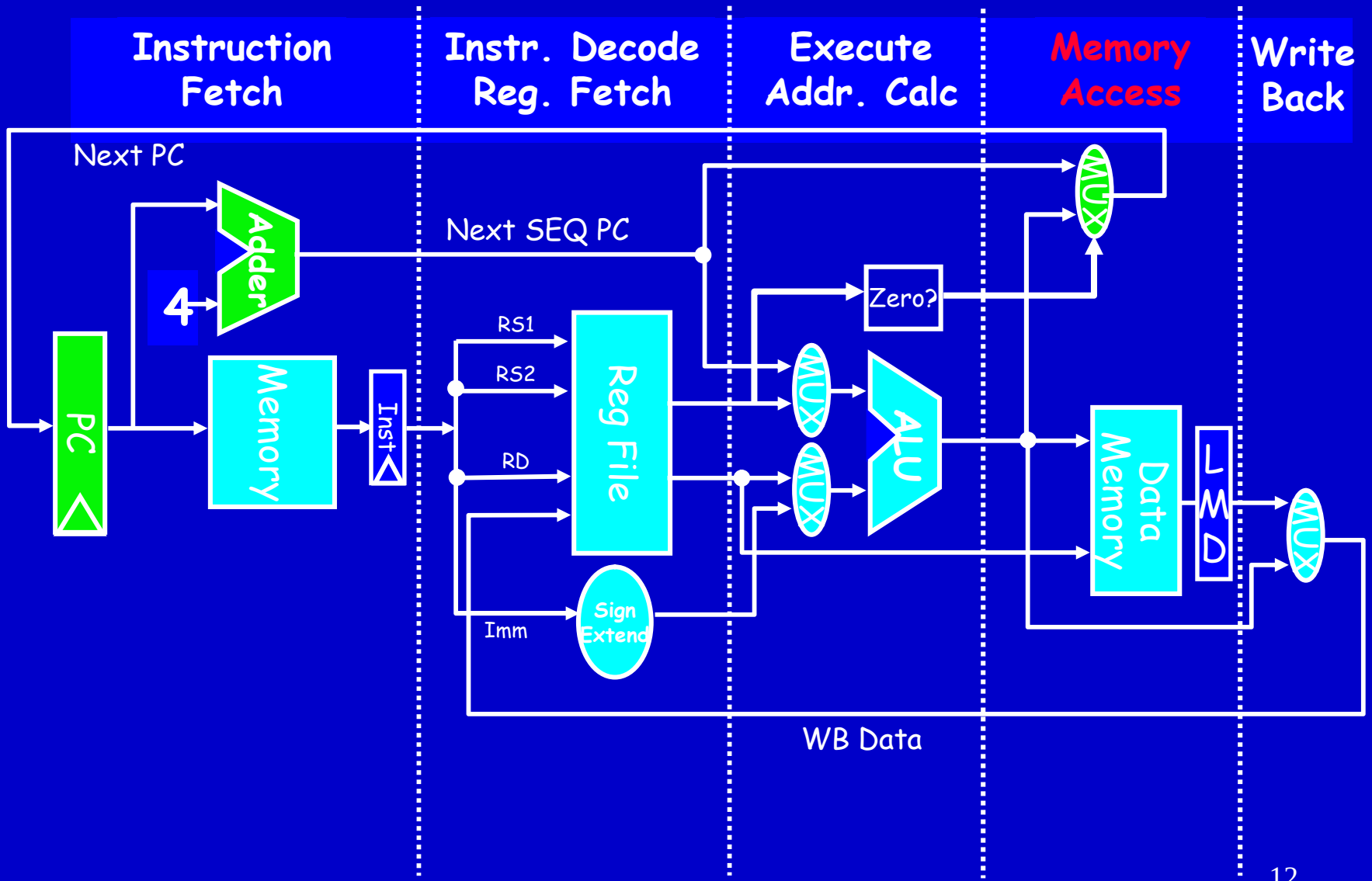
Multiple-Cycle DLX: Cycle 4

- The fourth cycle is known as the

  Memory access / branch completion cycle (MEM)

- The only DLX instructions active in this cycle are loads, stores, and branches
  - Loads
    - load memory onto processor
  - Stores
    - store data into memory
  - Branch
    - go to branch target or next instruction
  - ALU Operations
    - do nothing

5 Steps of DLX Datapath



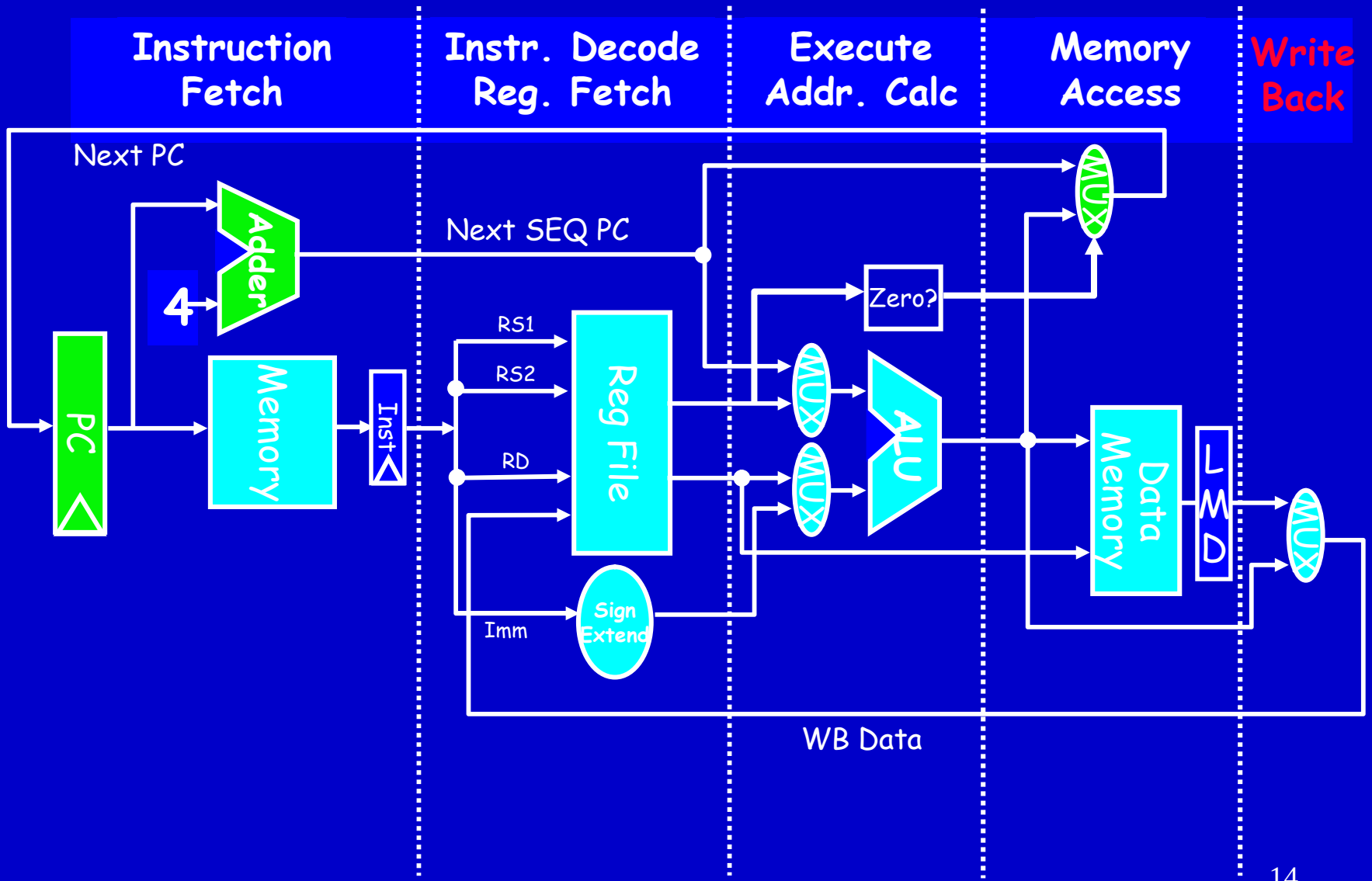
Multiple-Cycle DLX: Cycle 5

- The fifth cycle is known as the

  Write-back cycle (WB)

- During this cycles, results are written to the register file
  - Loads
    - write value from memory into register file
  - ALU Operations
    - write ALU result into register file
  - Stores and Branches
    - do nothing

5 Steps of DLX Datapath



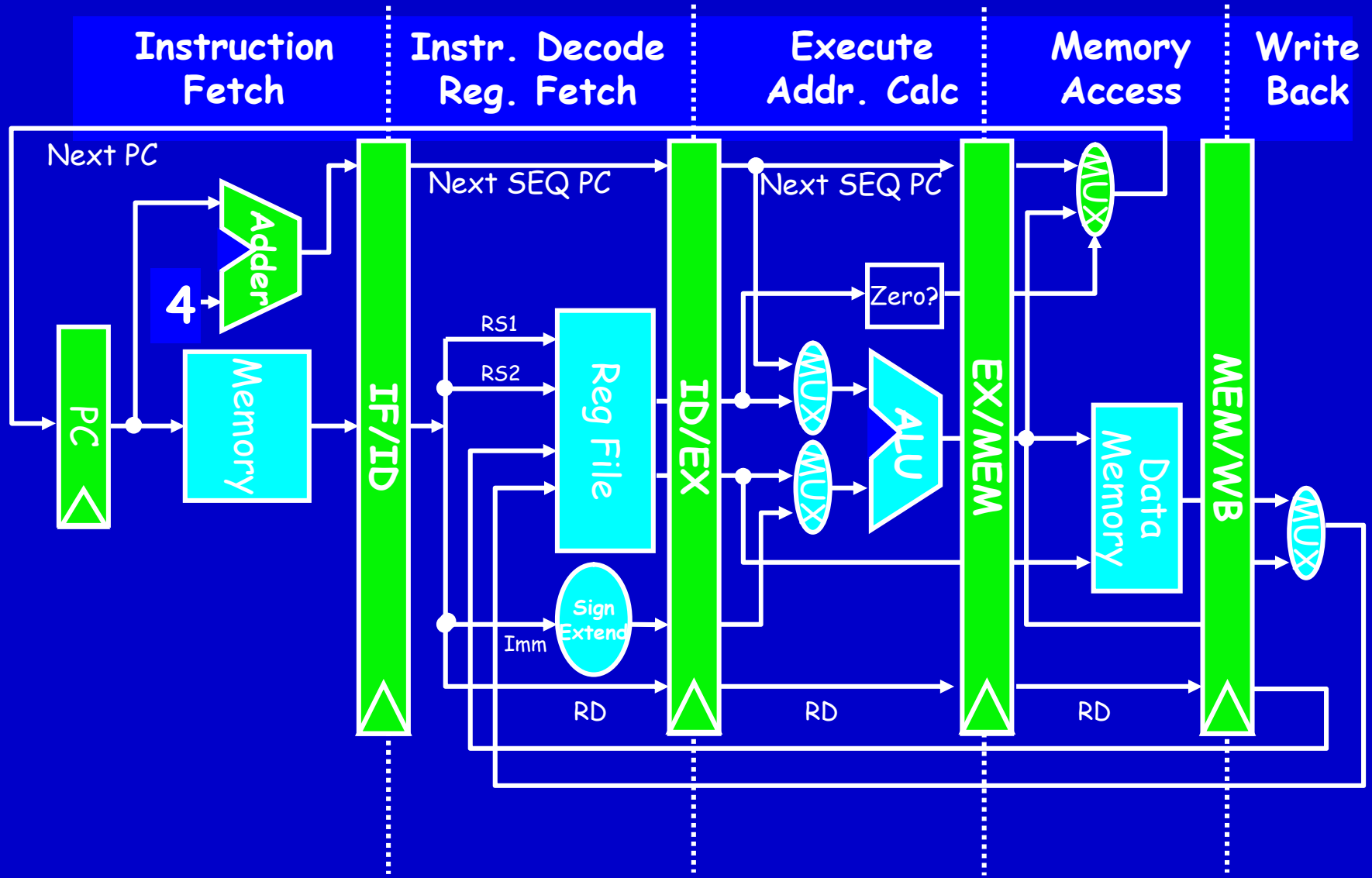
Pipelining DLX

- To reduce the CPI, DLX can be implemented using a five stage pipeline.

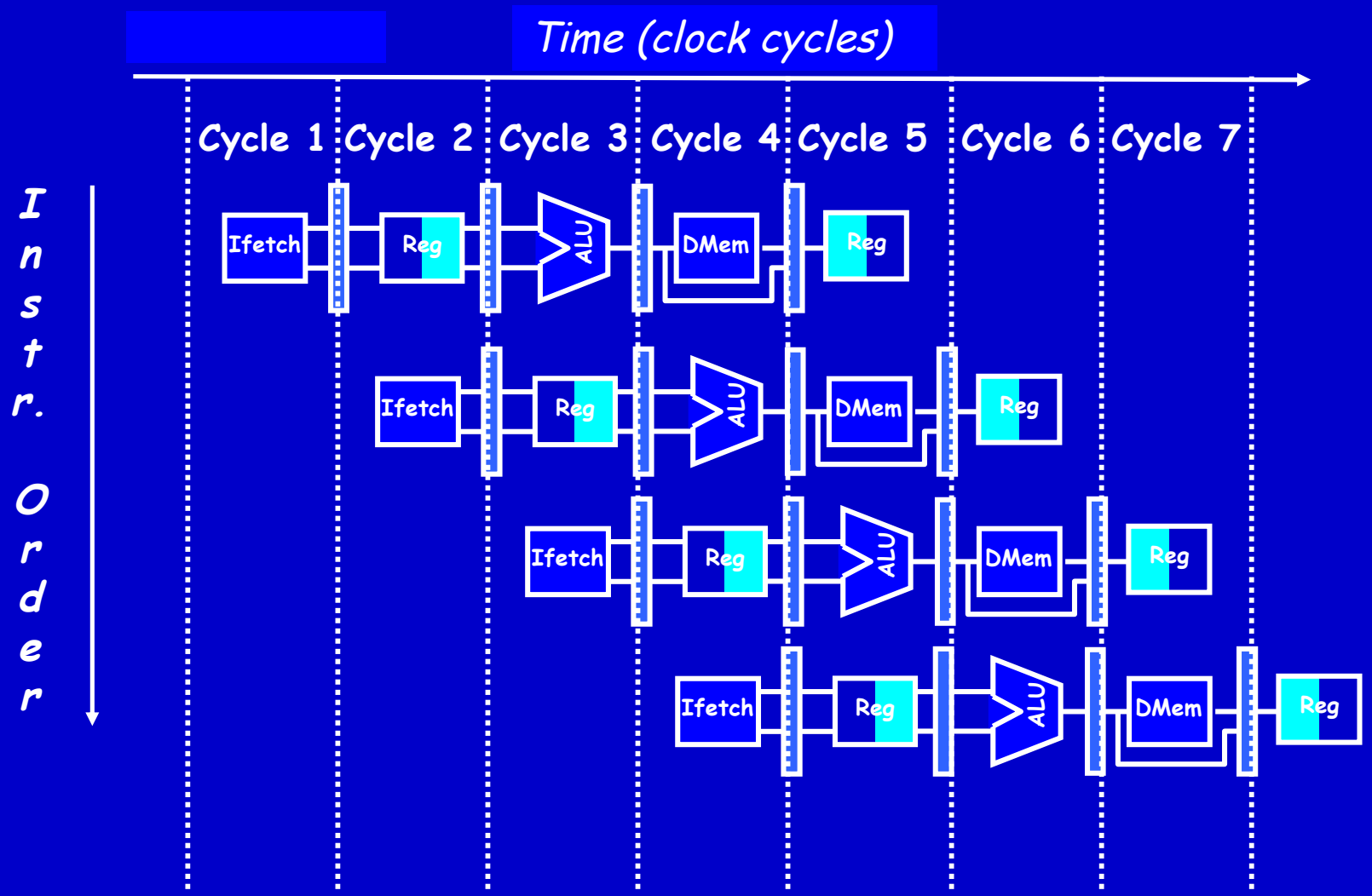
Inst.	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
1	IF	ID	EX	MEM	WB				
2		IF	ID	EX	MEM	WB			
3			IF	ID	EX	MEM	WB		
4				IF	ID	EX	MEM	WB	
5					IF	ID	EX	MEM	WB

- In this example, it takes 9 cycles execute 5 instructions for a CPI of 1.8.

5 Steps of DLX Datapath



Visualizing Pipelining



Its Not That Easy for Computers

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: Hardware cannot support this combination of instructions - two instructions need the same resource.
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Pipelining of branches & other instructions that change the PC
- Common solution is to stall the pipeline until the hazard is resolved, inserting one or more "bubbles" in the pipeline
- To do this, hardware or software must detect that a hazard has occurred.

Structural Hazards

- Structural hazards occur when two or more instructions need the same resource.
- Common methods for eliminating structural hazards are:
  - Duplicate resources
  - Pipeline the resource
  - Reorder the instructions
- It may be too expensive too eliminate a structural hazard, in which case the pipeline should stall.
- When the pipeline stalls, no instructions are issued until the hazard has been resolved.
- What are some examples of structural hazards?

Example: One or Two Memory Ports?

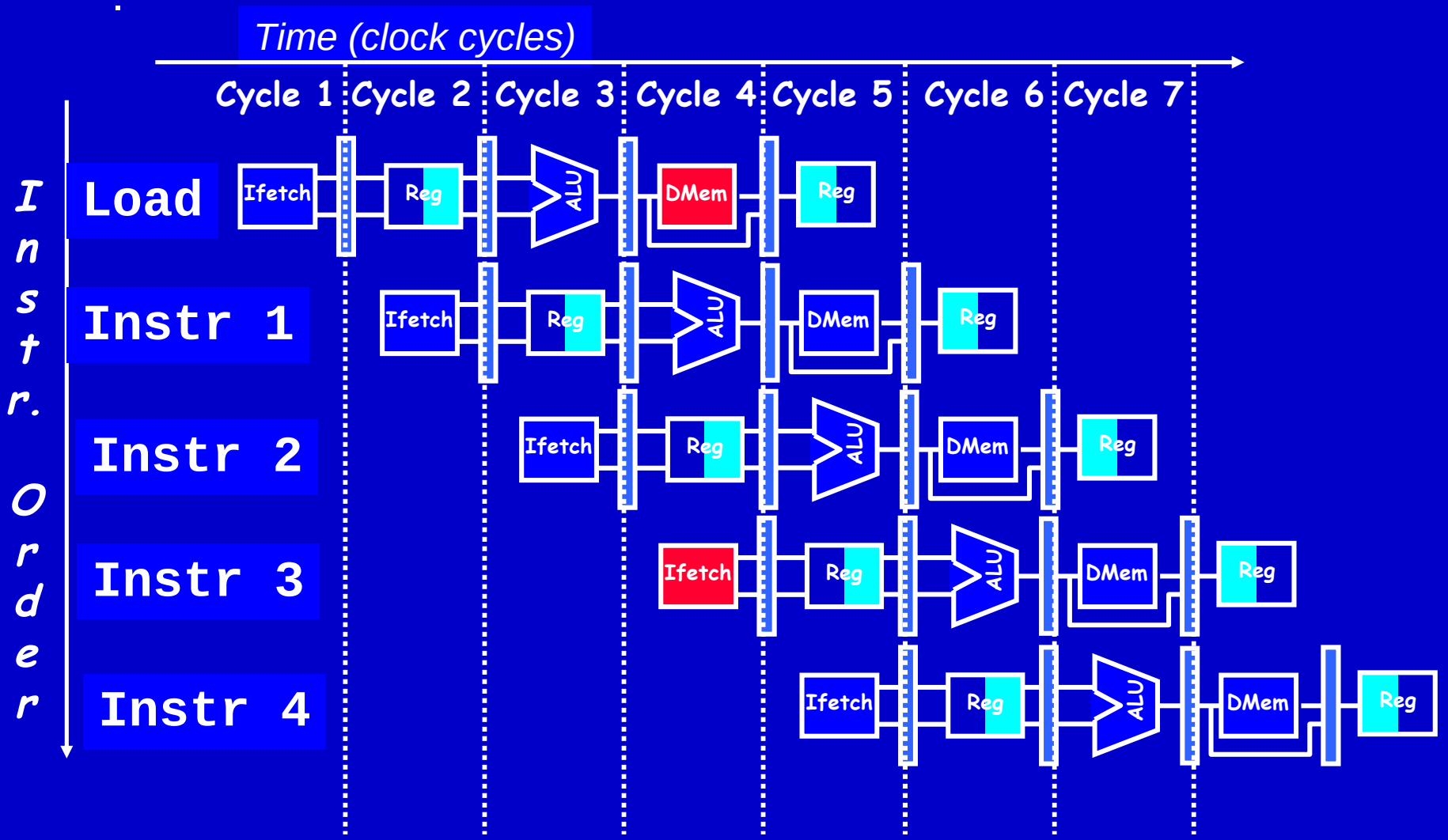
- Machine A: Dual ported memory (“Harvard Architecture”)
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

$$SpeedUp_A = Pipeline\ Depth/(1 + 0) \times (clock_{unpipe}/clock_{pipe})$$
$$= Pipeline\ Depth$$
$$SpeedUp_B = Pipeline\ Depth/(1 + 0.4 \times 1) \times (clock_{unpipe}/(clock_{unpipe} / 1.05)$$
$$= (Pipeline\ Depth/1.4) \times 1.05$$
$$= 0.75 \times Pipeline\ Depth$$
$$SpeedUp_A / SpeedUp_B = Pipeline\ Depth/(0.75 \times Pipeline\ Depth) = 1.33$$

- Machine A is 1.33 times faster

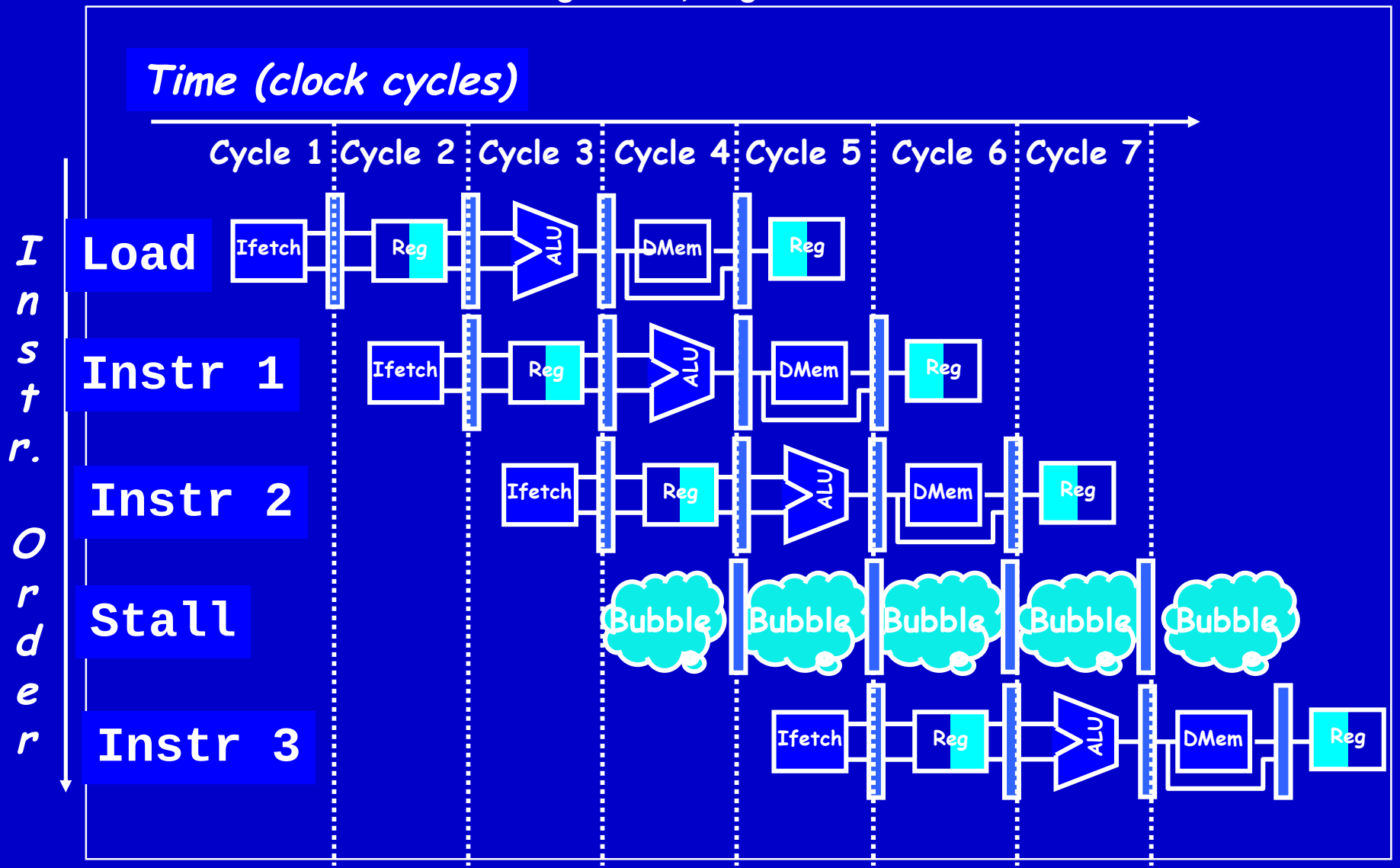
One Memory Port Structural Hazards

Figure 3.6, Page 142



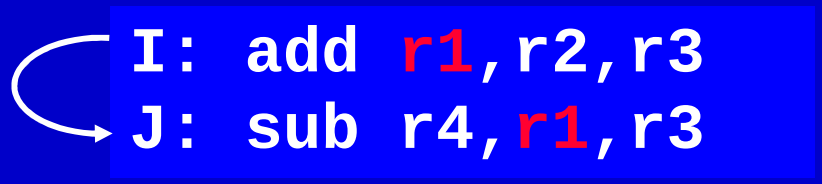
One Memory Port Structural Hazards

Figure 3.7, Page 143



Three Generic Data Hazards

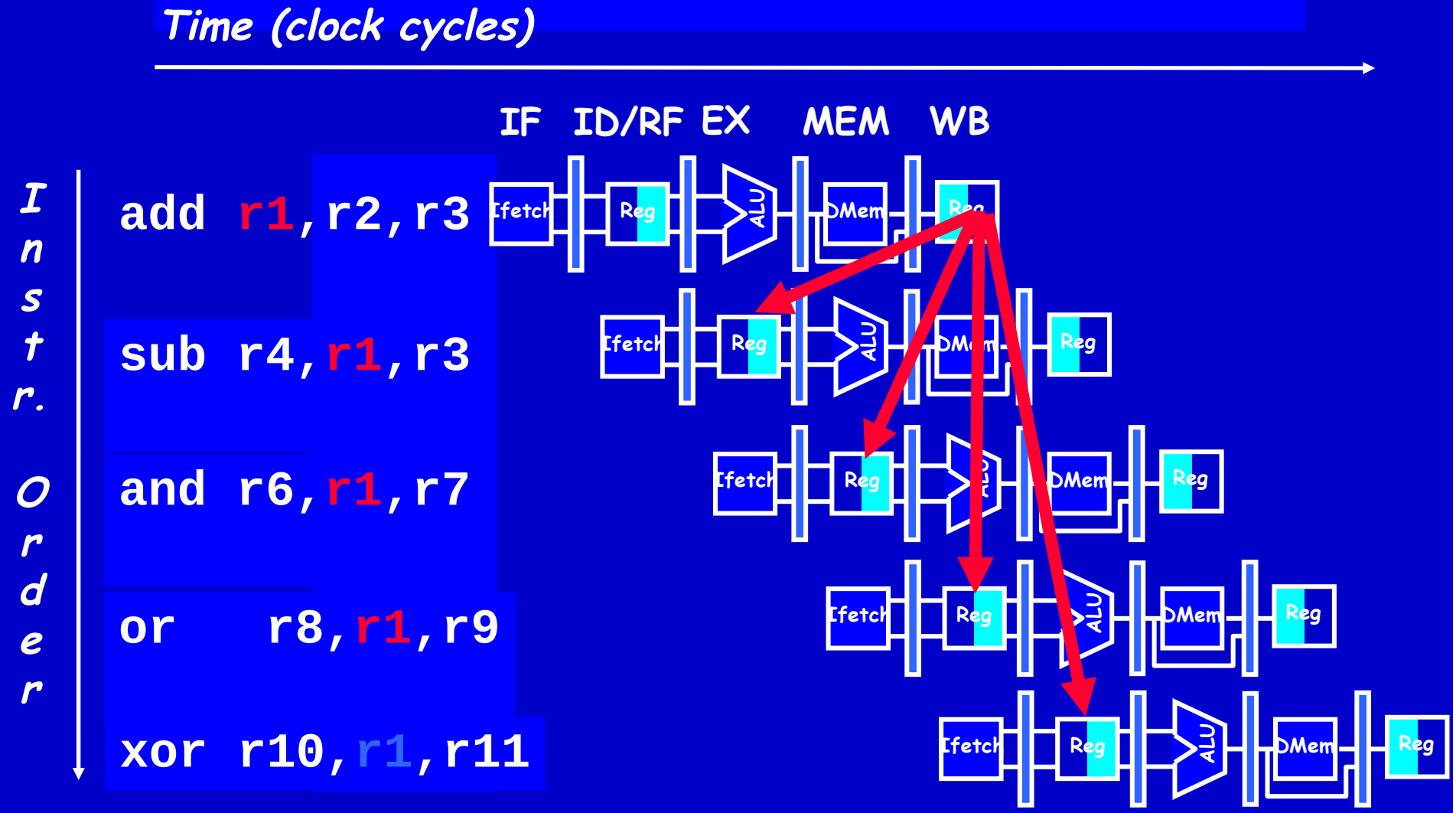
- Read After Write (RAW)
  Instr$_J$ tries to read operand before Instr$_I$ writes it



- Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

RAW Hazards on R1

Figure 3.9, page 147

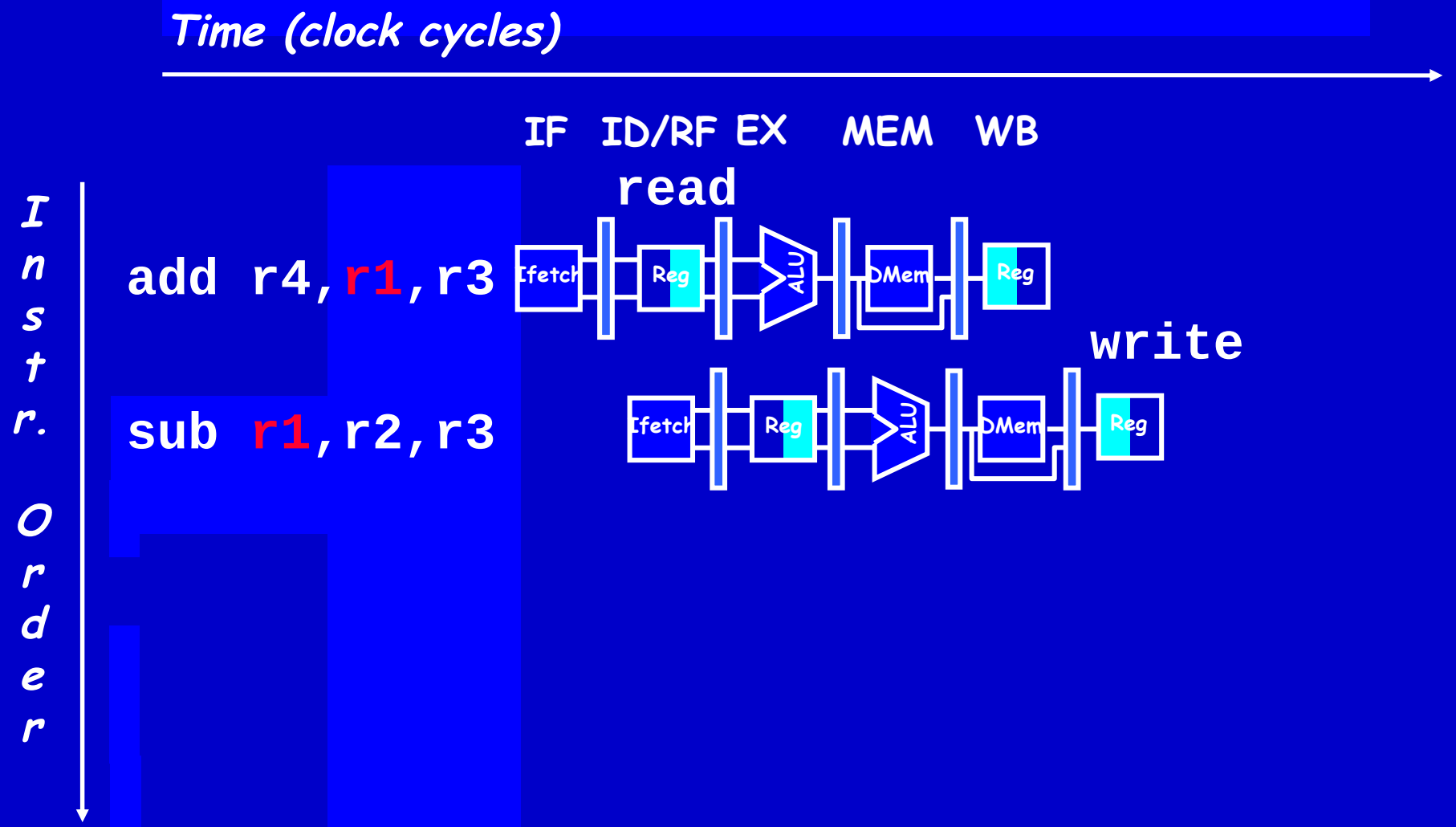


Three Generic Data Hazards

- Write After Read (WAR)
  Instr$_J$ writes operand *before* Instr$_I$ reads it

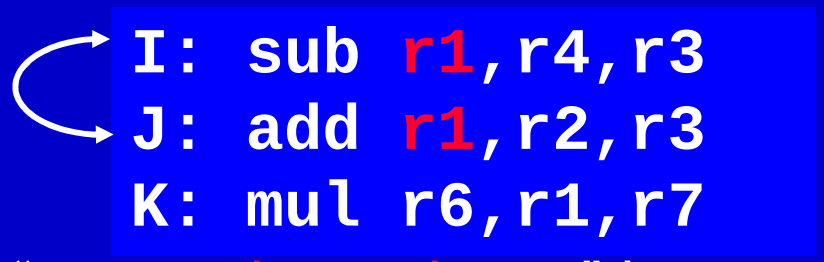
- Called an "anti-dependence" by compiler writers.
  This results from reuse of the name "r1".
- Can't happen in DLX 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5
- WAR hazards can happen if instructions execute out of order or access data late

No WAR Hazards on R1



Three Generic Data Hazards

- Write After Write (WAW)
  Instr$_J$ writes operand *before* Instr$_I$ writes it.



- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1".
- Can't happen in DLX 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
- Will see WAR and WAW in later more complicated pipes

Data Dependences - Stalls

- Another view
  - dependences are a program property
  - stalls are a pipeline or machine property
  - e.g. remove RAW stalls by forwarding
- Notation
  - data dependence ➔ RAW hazard
  - antidependence ➔ WAR hazard
  - output dependence ➔ WAW hazard
  - WAx ➔ WAR + WAW
- What's important
  - program semantics ➔ data flow
  - exception behavior

Methods

Technique	Reduces	Static or Dynamic
Loop Unrolling	Control Stalls	S, D?
Basic Pipeline Scheduling (we've seen some of this already)	RAW stalls	S, D?
Dynamic Scheduling with Scoreboarding	RAW stalls	D
Dynamic Scheduling with register renaming	WAx stalls	D
Dynamic Branch Prediction	Control Stalls	D
Issuing multiple instructions per cycle a.k.a Superscalar	Ideal CPI	S, D
Compiler dependence analysis	Ideal CPI and data stalls (RAW, WAW, WAR)	S
Software pipelining and trace scheduling	Ideal CPI and data stalls	S
Speculation	data and control stalls	S, D
Dynamic memory disambiguation	RAW stalls involving memory	D

CPI Equation

Pipeline CPI = Ideal pipeline CPI + Structural stalls + RAW stalls + WAR stalls + WAW stalls + Control stalls

Technique	Reduces
Forwarding and bypassing	Potential data hazard stalls (A.2)
Delayed branch and simple branch scheduling	Control hazard stalls (A.2)
Dynamic scheduling with scoreboarding	Data hazard stalls from true dependence (A.8)
Dynamic scheduling with register renaming	Data hazard stalls and stalls from antidependences and output dependence (3.2)
Dynamic branch prediction	Control stalls (3.4)
Issuing multiple instructions per cycle	Ideal CPI (3.6)
Hardware-based speculation	Data and control stalls (3.7)
Dynamic memory disambiguation	Data stalls involving memory (3.2, 3.7)
Loop unrolling	Control stalls (4.1)
Basic Compiler pipeline scheduling	Data hazard stalls (4.1)
Compiler dependence analysis	Ideal CPI and data stalls (4.4)
Software pipelining and trace scheduling	Ideal CPI and data stalls (4.3)
Compiler Speculation	All data and control stalls (4.4)

Instruction Level Parallelism

- Potential overlap among instructions
- Few possibilities in a basic block
  - Blocks are small (6-7 instructions)
  - Instructions are dependent
- Exploit ILP across multiple basic blocks
  - Iterations of a loop

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

  - Alternative to vector instructions

Loop Level Parallelism

- Consider adding two 1000 element arrays

  for (i=1, i<=1000, i=i+1)

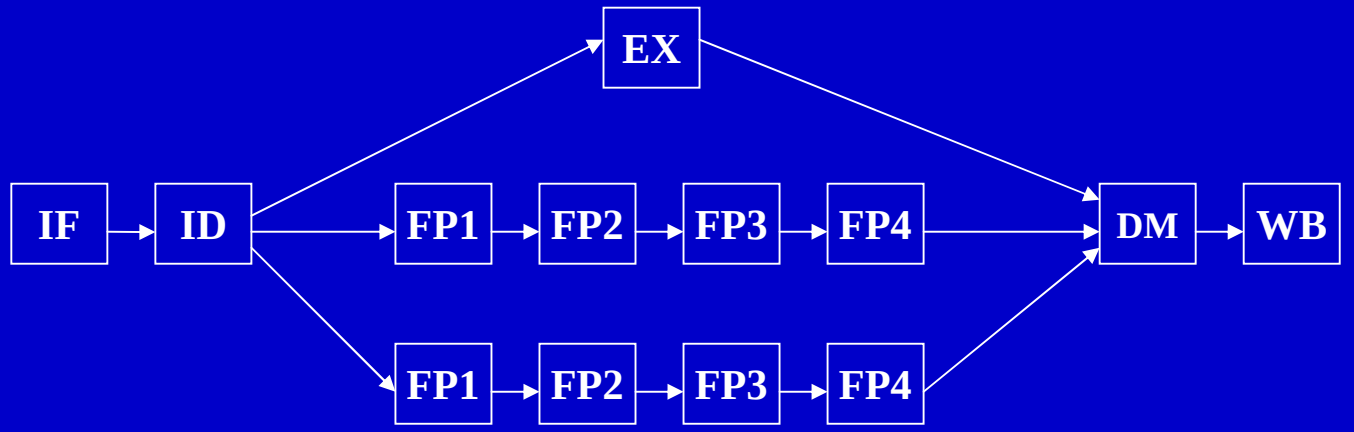
  x[i] = x[i] + y[i]

- Sure it's trivial but it illustrates the point
  - there is no dependence between data values produced in any iteration j and those needed in j+n for any j and n
  - truly independent - hence could be a 1000 way parallel program
  - independence means no stalls due to data hazards
  - problem is that we have to use that pesky branch instruction
    - prediction here is easy - BUT we still need to have a prediction scheme
- Vector Processor model
  - load vectors x and y (up to some machine dependent max)
  - then do *result-vec = xvec + yvec* in a single instruction

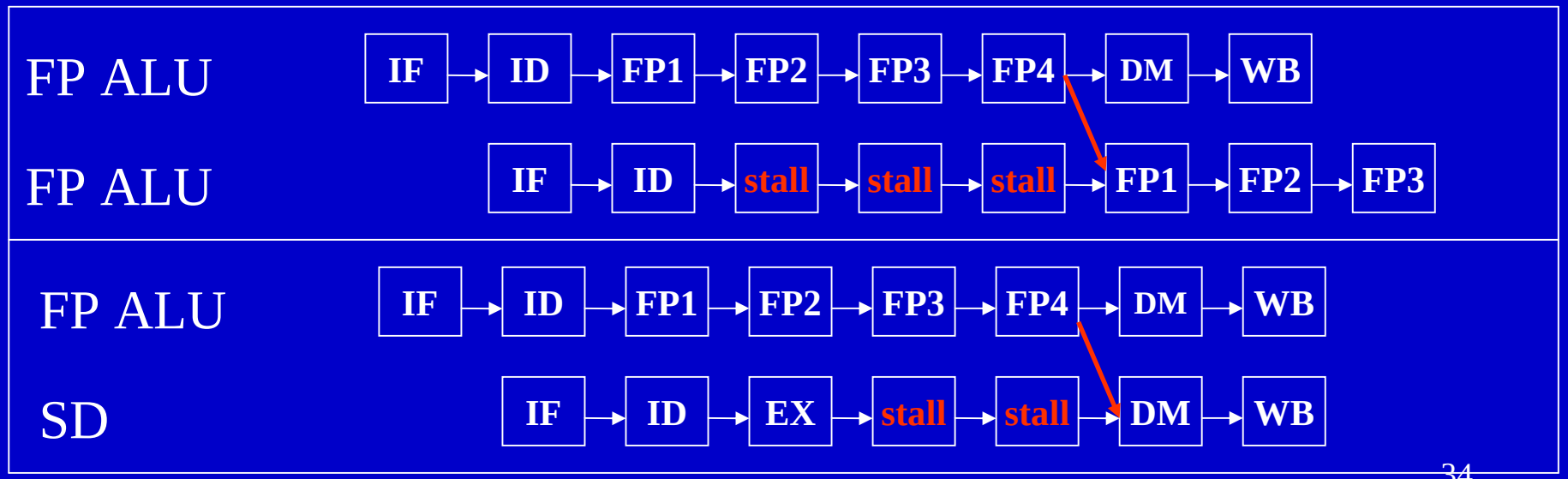
ILP within a Basic Block

- "Basic Block" definition
  - straight-line code - no branches out
  - single entry point at top
  - ➔ real code is a bunch of basic blocks connected by branch statements
- Notice
  - branch frequency is approx. 15% of total mix
  - implies basic block size between 6 and 7 instructions
  - machine instructions don't do much
  - there's probably little in the way of basic block parallelism
- Easiest target is the loop
  - already exploited by vector processors but using different mechanisms

Sample Pipeline



• • •



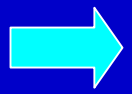
Basic Pipeline Scheduling

- Find sequences of unrelated instructions
- Compiler's ability to schedule
  - Amount of ILP available in the program
  - Latencies of the functional units
- Latency assumptions for the examples
  - Standard MIPS integer pipeline
  - No structural hazards (fully pipelined or duplicated units
  - Latencies of FP operations:

Instruction producing result	Instruction using result	Latency
FP ALU op	FP ALU op	3
FP ALU op	SD	2
LD	FP ALU op	1
LD	SD	0

Basic Scheduling

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



Sequential MIPS Assembly Code

```
Loop:   LD      F0, 0(R1)
        ADDD    F4, F0, F2
        SD      0(R1), F4
        SUBI    R1, R1, #8
        BNEZ    R1, Loop
```

Pipelined execution:

```
Loop:   LD      F0, 0(R1)     1
        stall                 2
        ADDD    F4, F0, F2    3
        stall                 4
        stall                 5
        SD      0(R1), F4     6
        SUBI    R1, R1, #8    7
        stall                 8
        BNEZ    R1, Loop      9
        stall                 10
```

Scheduled pipelined execution:

```
Loop:   LD      F0, 0(R1)     1
        SUBI    R1, R1, #8    2
        ADDD    F4, F0, F2    3
        stall                 4
        BNEZ    R1, Loop      5
        SD      8(R1), F4     6
```

Dynamic Pipeline Scheduling

- Hardware:
  - rearrange instruction stream to reduce stalls
  - can treat dependencies which are only known at run time
  - simplifies the compiler
    - reduces difference between hardware platforms from compiler perspective
    - minimizes the need for speculative slot filling - NP hard problem anyway
    - also mis-speculation causes power problems

Dynamic Scheduling

- Allow out of order issue
- Consider

DIVD F0, F2, F4
ADDD F10, F0, F8 hazards?
SUBD F14, F8, F14

  - DIVD has a long latency
  - ADDD has a data dependence on F0, SUBD does not
    - so swap them - static: compiler or dynamic: hardware
- Problems
  - big time if it's the hardware
    - so for now lets ignore precise interrupts
  - compiler defines program order so swap is free

Dynamic Scheduling

- Scheduling separates dependent instructions
  - Static – performed by the compiler
  - Dynamic – performed by the hardware
- Advantages of dynamic scheduling
  - Handles dependences unknown at compile time
  - Simplifies the compiler
  - Optimization is done at run time
- Disadvantages
  - Can not eliminate true data dependences

Dynamic vs. Static Scheduling

- Data hazards in a program to cause a processor to stall.
- With static scheduling the compiler tries to reorder these instructions during compile time to reduce pipeline stalls.
  - Uses less hardware
  - Can use more powerful algorithms
- With dynamic scheduling the hardware tries to rearrange the instructions during run-time to reduce pipeline stalls.
  - Simpler compiler
  - Handles dependencies not known at compile time
  - Allows code compiled for a different machine to run efficiently.

Out-of-order execution (1/2)

- Central idea of dynamic scheduling
  - In-order execution:

DIVD F0, F2, F4	IF ID DIV …..
ADDD F10, F0, F8	IF ID stall stall stall …
SUBD F12, F8, F14	IF stall stall …..

  - Out-of-order execution:

DIVD F0, F2, F4	IF ID DIV …..
SUBD F12, F8, F14	IF ID A1 A2 A3 A4 …
ADDD F10, F0, F8	IF ID stall …..

Out-of-Order Execution (2/2)

- Separate issue process in ID:
  - Issue
    - decode instruction
    - check structural hazards
    - in-order execution
  - Read operands
    - Wait until no data hazards
    - Read operands
- Out-of-order execution/completion
  - Exception handling problems
  - WAR hazards

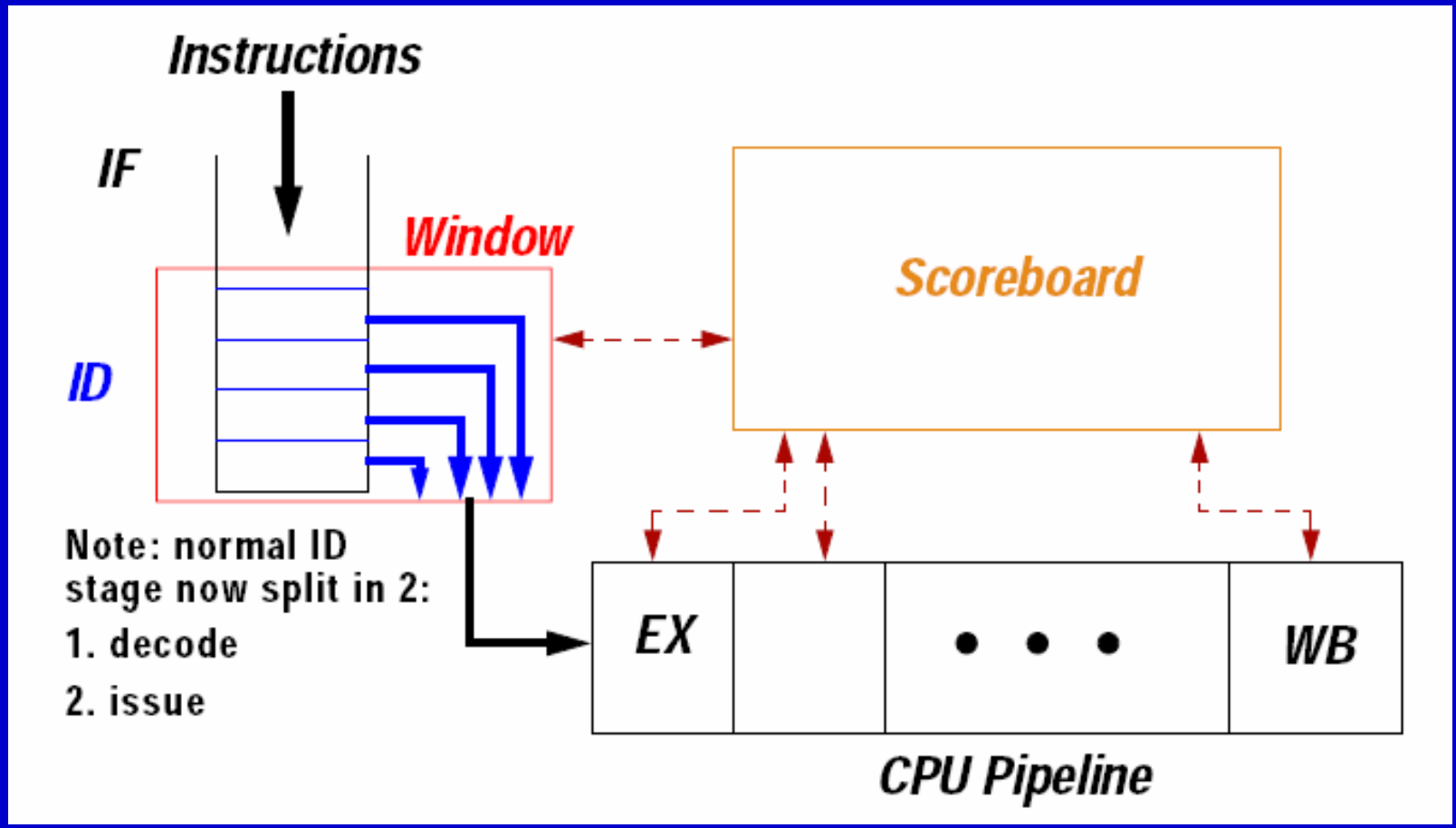
Scoreboard: a bookkeeping technique

- Out-of-order execution divides ID stage:
  1. Issue—decode instructions, check for structural hazards
  2. Read operands—wait until no data hazards, then read operands
- Scoreboards date to CDC6600 in 1963
- Instructions execute whenever not dependent on previous instructions and no hazards.
- CDC 6600: In order issue, out-of-order execution, out-of-order commit (or completion)
  - No forwarding!
  - Imprecise interrupt/exception model for now

Scoreboarding

- The scoreboard implements a centralized control scheme that
  - Detects all resource and data hazards
  - Allows instructions to execute out-of-order when no resource hazards or data dependencies
- First implemented in 1964 by the CDC 6600, which had 18 separate functional units
  - 4 FP units (2 multiply, 1 add, 1 divide)
  - 7 memory units (5 loads, 2 stores)
  - 7 integer units (add, shift, logical, compare, etc.)
- Dynamic DLX (much simpler)
  - 2 FP multiply (10 EX cycles)
  - 1 FP add (2 EX cycles)
  - 1 FP divide (40 EX cycles)
  - 1 integer unit (1 EX cycle)

Scoreboarding



Scoreboard Implications

- Out-of-order completion can lead to WAR and WAW hazards?
- Solution for WAW
  - Detect WAW hazard before reading operands
  - Stall write until other instruction completes
- Solutions for WAR
  - Detect WAR hazards before writing back to the register files and stall the write back
- This scoreboard does not take advantage of forwarding, since it waits until both results are written back to the register file

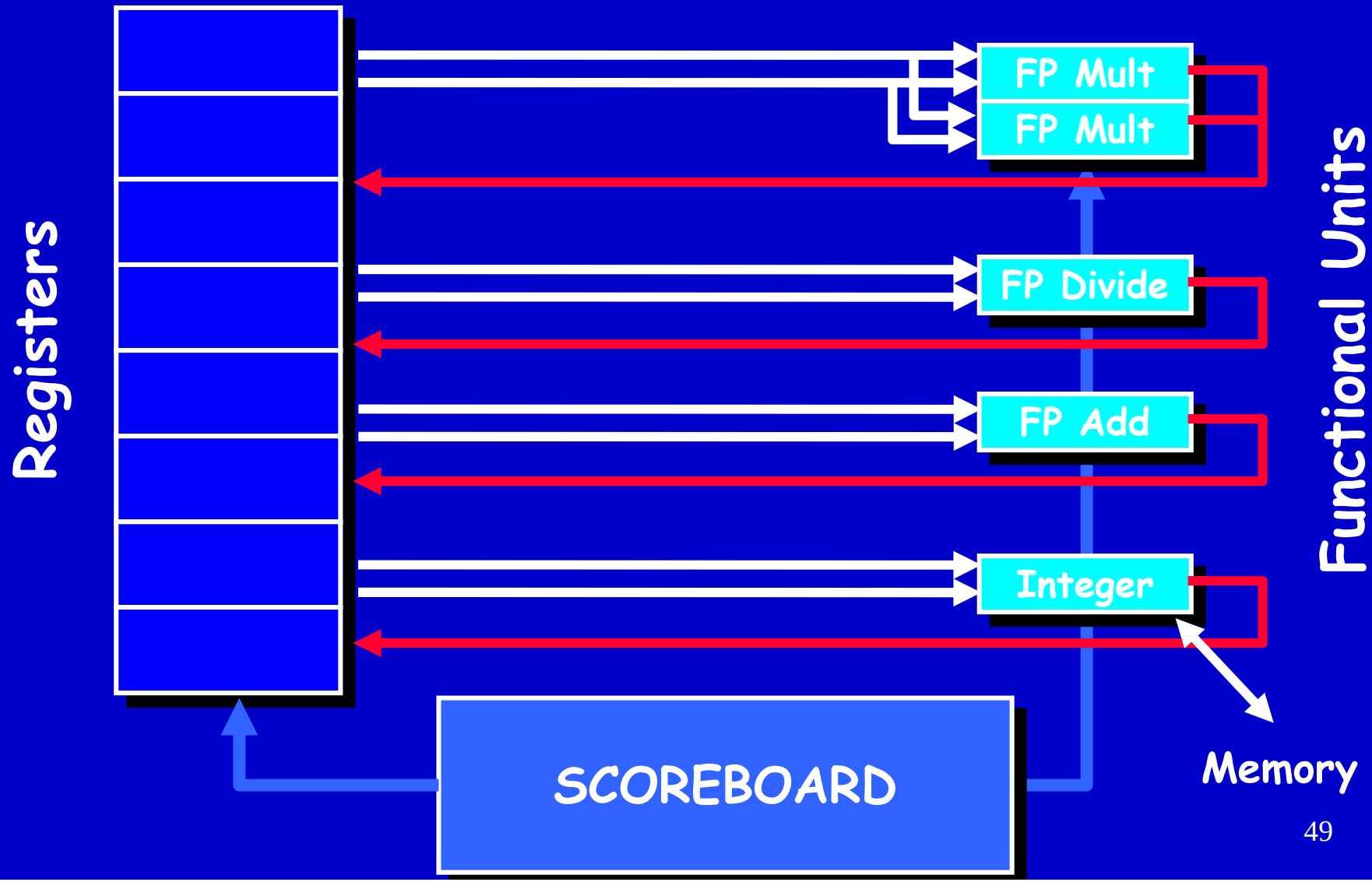
Scoreboard

- Allows out-of-order execution
  - Sufficient resources
  - No data dependencies
- Responsible for issue, execution and hazards
- Functional units with long delays
  - Duplicated
  - Fully pipelined

Scoreboards

- Centralized hazard check and interlock issue
- Load/Store architecture
  - Results
    - approx. 60% performance gain due to dynamic ordering
    - cost inflates by approx. 20%
    - still fastest machine of it’s day - good thing at $1.2M min.
    - 10 years ago technique too complex for single chip designs

MIPS with Scoreboard



Scoreboard Operation

- Scoreboard centralizes hazard management
  - Every instruction goes through the scoreboard
  - Scoreboard determines when the instruction can read its operands and begin execution
  - Monitors changes in hardware and decides when an stalled instruction can execute
  - Controls when instructions can write results
- New pipeline

ID		EX	WB
Issue	Read Regs	Execution	Write

Four Stages of Scoreboard Control

- Issue (ID1)
  - decode instructions & check for structural hazards (ID1)
    - Instructions issued in program order (for hazard checking)
    - Don't issue if structural hazard
    - Don't issue if instruction is output dependent on any previously issued but uncompleted instruction (no WAW hazards)
    - check for structural and WAW hazards
    - stall until structural and WAW hazards are resolved
    - free function unit AND no destination register conflict ==> issue else stall
      - e.g. check for structural and WAW hazards (one S.Hz might be Opnd and Result busses)
    - nothing out of order yet - if issue stalls then IF stalls when buffer fills
- Read operands (ID2)
  - wait until no RAW hazards
  - No forwarding of data in this model!
  - scoreboard monitors source operand availability
  - available if no earlier active is going to write it OR active writer is in write result stage
  - dynamic RAW resolution ==> things can get out of order at this point

Four Stages of Scoreboard Control

- Execution (EX)
  - starts when operands are received
  - may take multiple cycles so unit notifies the scoreboard when it's done
- Write result (WB)
  - finish execution
  - Stall until no WAR hazards with previous instructions:

Three Parts of the Scoreboard

1. Instruction status —which of 4 steps the instruction is in: ID1, ID1, EX, or WB.

2. Functional unit status —Indicates the state of the functional unit (FU). 9 fields for each functional unit
   - Busy—Indicates whether the unit is busy or not
   - Op—Operation to perform in the unit (e.g., + or –)
   - Fi—Destination register
   - Fj, Fk—Source-register numbers
   - Qj, Qk—Functional units producing source registers Fj, Fk
   - Rj, Rk—Flags indicating when Fj, Fk are ready

3. Register result status —Indicates which functional unit will write each register, if one exists. Blank when no pending instructions will write that register

Scoreboard Algorithm

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
Execution complete	Functional unit done	
Write result	∀f((Fj(f)≠Fi(FU) or Rj(f)=No) & (Fk(f)≠Fi(FU) or Rk(f)=No))	∀f(if Qj(f)=FU then Rj(f)← Yes); ∀f(if Qk(f)=FU then Rj(f)← Yes); Result(Fi(FU))← 0; Busy(FU)← No

Scoreboard Example for DLX

Instruction status

Instruction		*j*	*k*	*Issue*	*Read operanc*	*Executi complet*	*Write Result*
LD	F6	34+	R2				
LD	F2	45+	R3				
MULT	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU for j Qj*	*FU for k Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

Clock	*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
FU									

Scoreboard Example Cycle 1

Instruction status

Instruction		*j*	*k*	*Issue*	*Read operand*	*Executic complet*	*Write Result*
LD	F6	34+	R2	1			
LD	F2	45+	R3				
MULT	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU for j Qj*	*FU for k Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
1	*FU*				Integer					

Scoreboard Example Cycle 2

Instruction status

Instruction		j	k	Issue	Read operand	Executic complet	Write Result
LD	F6	34+	R2	1	2		
LD	F2	45+	R3				
MULT	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
2	FU				Integer					

- **Issue 2nd LD?**

Scoreboard Example Cycle 3

Instruction status

Instruction		j	k	Issue	Read operand	Executic complet	Write Result
LD	F6	34+	R2	1	2	3	
LD	F2	45+	R3				
MULT	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU				Integer					

- **Issue MULT?**

Scoreboard Example Cycle 4

Instruction status

Instruction		*j*	*k*	*Issue*	*Read operand*	*Executic complet*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3				
MULT	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU for j Qj*	*FU for k Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

Clock	*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
4 *FU*				Integer					

Scoreboard Example Cycle 5

Instruction status

Instruction		*j*	*k*	*Issue*	*Read operand*	*Executic complet*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5			
MULT	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU for j Qj*	*FU for k Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	Yes	Load	F2		R3				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
5	*FU*		Integer							

Scoreboard Example Cycle 6

Instruction status

Instruction		j	k	Issue	Read operand	Executic complet	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6		
MULT	F0	F2	F4	6			
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F2		R3				Yes
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
6	FU	Mult1	Integer							

Scoreboard Example Cycle 7

Instruction status

Instruction		j	k	Issue	Read operand	Executic complet	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	
MULT	F0	F2	F4	6			
SUBD	F8	F6	F2	7			
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F2		R3				Yes
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
	Divide	No								

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	Mult1	Integer			Add				

- **Read multiply operands?**

Scoreboard Example Cycle 8a

Instruction status

Instruction		*j*	*k*	*Issue*	*Read operand*	*Executic complet*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	
MULT	F0	F2	F4	6			
SUBD	F8	F6	F2	7			
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU for j Qj*	*FU for k Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	Yes	Load	F2		R3				Yes
	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

Register result status

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

Scoreboard Example Cycle 8b

Instruction status

Instruction		j	k	Issue	Read operand	Executic complet	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6			
SUBD	F8	F6	F2	7			
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

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

Scoreboard Example Cycle 9

Instruction status

Instruction		j	k	Issue	Read operand	Executic complet	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9		
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status

	Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
		Integer	No								
Note Remaining	10	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
		Mult2	No								
	2	Add	Yes	Sub	F8	F6	F2			Yes	Yes
		Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
9 FU	Mult1				Add	Divide			

- Read operands for MULT & SUB? Issue ADDD?

Scoreboard Example Cycle 11

Instruction status

Instruction		j	k	Issue	Read operand	Executic complet	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	No								
8	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
0	Add	Yes	Sub	F8	F6	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

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

Scoreboard Example Cycle 12

Instruction status

Instruction		*j*	*k*	*Issue*	*Read operand*	*Executic complet*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU for j Qj*	*FU for k Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
7	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
	Add	No								
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

Clock	*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
12 *FU*	Mult1					Divide			

- **Read operands for DIVD?**

Scoreboard Example Cycle 13

Instruction status

Instruction		j	k	Issue	Read operand	Executic complet	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13			

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	No								
6	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

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

Scoreboard Example Cycle 14

Instruction status

Instruction		j	k	Issue	Read operand	Executic complet	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14		

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	No								
5	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
2	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

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

Scoreboard Example Cycle 15

Instruction status

Instruction		*j*	*k*	*Issue*	*Read operand*	*Executic complet*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14		

Functional unit status

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU for j Qj*	*FU for k Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
4	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
1	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

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

Scoreboard Example Cycle 16

Instruction status

Instruction		*j*	*k*	*Issue*	*Read operand*	*Executic complet*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Functional unit status

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU for j Qj*	*FU for k Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
3	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
0	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

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

Scoreboard Example Cycle 17

Instruction status

Instruction		j	k	Issue	Read operand	Executic complet	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	No								
2	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
17 FU	Mult1			Add		Divide			

- **Write result of ADDD?**

Scoreboard Example Cycle 18

Instruction status

Instruction		*j*	*k*	*Issue*	*Read operand*	*Executic complet*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Functional unit status

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU for j Qj*	*FU for k Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
1	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

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

Scoreboard Example Cycle 19

Instruction status

Instruction		j	k	Issue	Read operand	Executic complet	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9	19	
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	No								
0	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

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

Scoreboard Example Cycle 20

Instruction status

Instruction		*j*	*k*	*Issue*	*Read operand*	*Executic complet*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Functional unit status

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU for j Qj*	*FU for k Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6			Yes	Yes

Register result status

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
20	*FU*				Add		Divide			

Scoreboard Example Cycle 21

Instruction status

Instruction		j	k	Issue	Read operand	Executic complet	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21		
ADDD	F6	F8	F2	13	14	16	

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6			Yes	Yes

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
21	*FU*				Add		Divide			

- **WAR Hazard is now gone...**

Scoreboard Example Cycle 22

Instruction status

Instruction		j	k	Issue	Read operand	Executic complet	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21		
ADDD	F6	F8	F2	13	14	16	22

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
40	Divide	Yes	Div	F10	F0	F6			Yes	Yes

Register result status

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

Scoreboard Example Cycle 61

Instruction status

Instruction		j	k	Issue	Read operand	Executic complet	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	
ADDD	F6	F8	F2	13	14	16	22

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
0	Divide	Yes	Div	F10	F0	F6			Yes	Yes

Register result status

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

Scoreboard Example Cycle 62

Instruction status

Instruction		j	k	Issue	Read operand	Executic complet	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	62
ADDD	F6	F8	F2	13	14	16	22

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
0	Divide	No								

Register result status

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
62	*FU*								

- **In-order issue; out-of-order execute & commit**

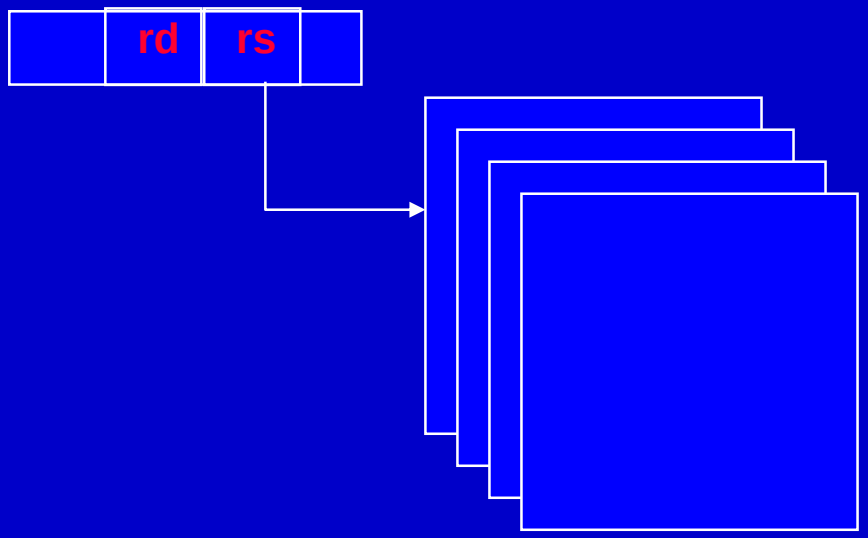
Scoreboard Limitations

- Amount of available ILP
- Number of scoreboard entries
  - Limited to a basic block (small *window*)
  - Extended beyond a branch
- Number and types of functional units
  - Structural hazards can increase with DS
- Presence of anti- and output- dependences
  - Lead to WAR and WAW stalls
    - Wait for WAR hazards
    - Prevent WAW hazards
- No forwarding hardware
- Small number of functional units (structural hazards), especially integer/load store units
- Do not issue on structural hazards

Another Dynamic Algorithm: Tomasulo Algorithm

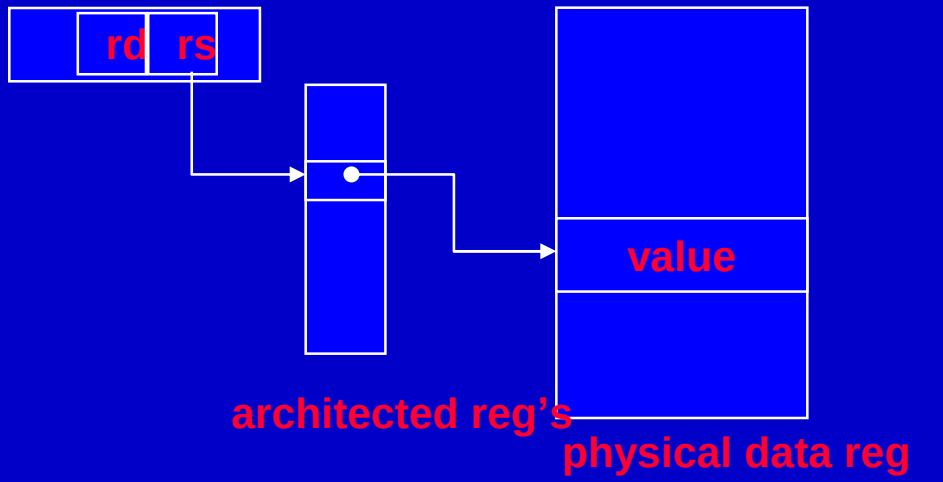
- For IBM 360/91 about 3 years after CDC 6600 (1966)
- Goal: High Performance without special compilers
- Differences between IBM 360 & CDC 6600 ISA
  - IBM has only 2 register specifiers/instr vs. 3 in CDC 6600
  - IBM has 4 FP registers vs. 8 in CDC 6600
  - IBM has memory-register ops
- Why Study? lead to Alpha 21264, HP 8000, MIPS 10000, Pentium II, PowerPC 604, …

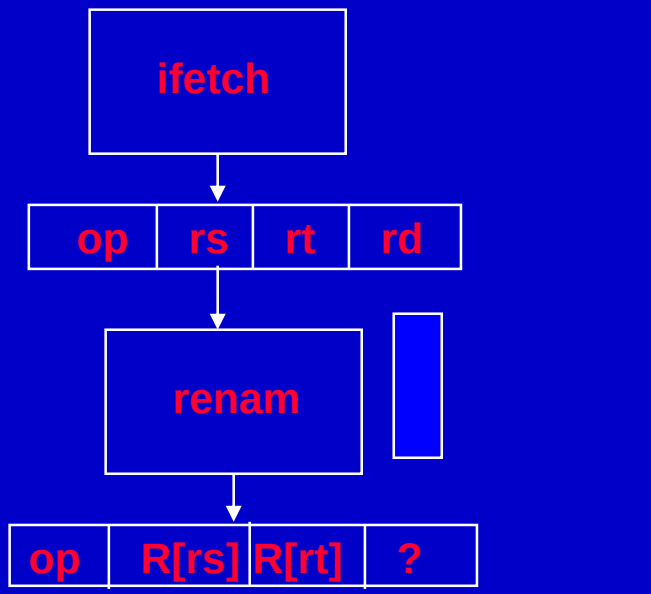
Register Renaming (Conceptual)



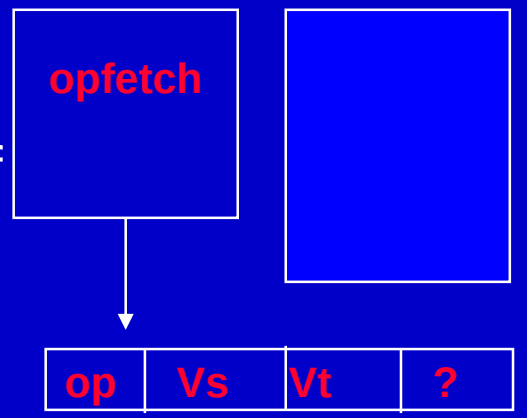
- Imagine if each write to register Ri created a new instance of that register
  - kth instance Ri.k
- Later references to source register treated as Ri.k
- Next use as a destination creates Ri.k+1

Register Renaming (less Conceptual)



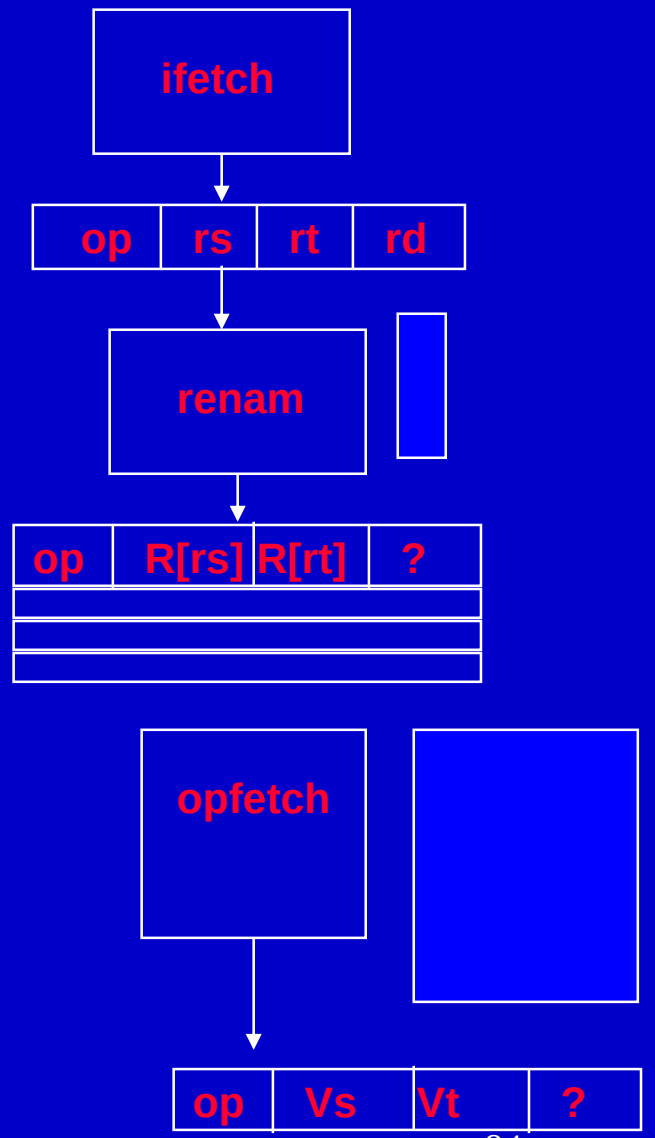


- Separate the functions of the register
- Reg identifier in instruction is mapped to "physical register" id for current instance of the register
  - Physical reg set may be larger than allocated
- What are the rules for allocating / deallocating physical registers?



Reg renaming

- Source Reg s:
  - physical reg P=R[s]
- Destination reg d:
  - Old physical register R[d] "terminates"
  - R[d] :=get_free
- Free physical register when
  - No longer referenced by any architected register (terminated)
  - No incomplete instructions waiting to read it
    - Easy with in-order
    - Out of order?



Broadcasting result value

- Series of instructions issued and waiting for value to be produced by logically preceding instruction.
- CDC6600 has each come back and read the value once it is placed in register file
- Alternative: broadcast value and reg # to all the waiting instructions
  - One that match grab the value

Temporary renaming

- Value “currently” bound to register is not present in the register file, instead…
- To be produced by particular instruction in the datapath
  - Designated by function unit that will produce value, or
  - Nearest matching instruction ahead in the datapath (in-order), or
  - With an associated “tag”

Register Renaming in Tomasulo

- Provided by the reservation station
  - Buffer the operands of instructions waiting to issue
- A reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register.
- Pending instructions designate the reservation station that will provide their input.
- When successive writes to a register overlap in execution, only the last one is actually used to update the register.
- As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation station, which provide register renaming.
- Since there can be more reservation stations than real registers, Tomasulo can eliminate hazard arising from name dependence.

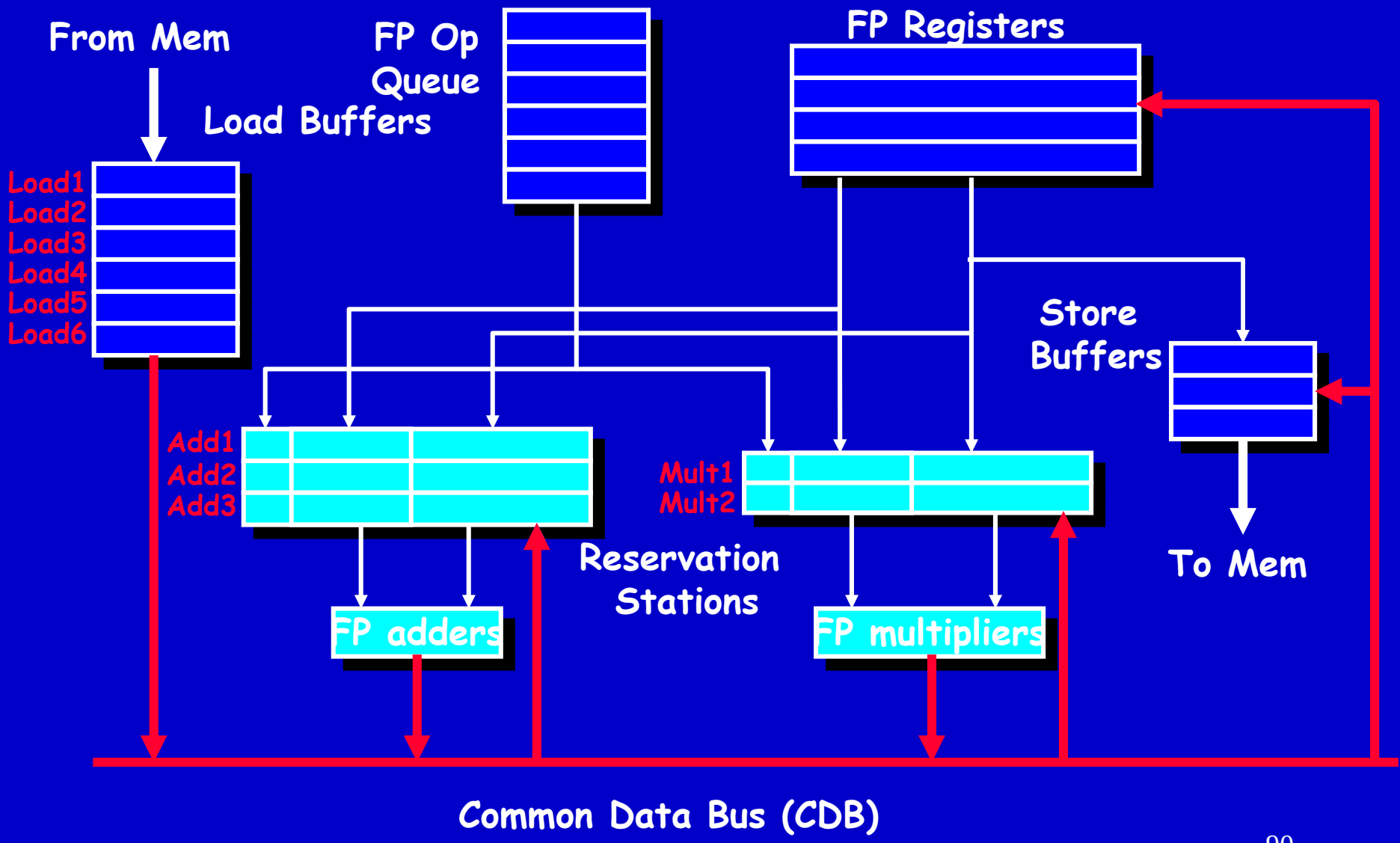
Tomasulo Approach

- Another approach to eliminate stalls
  - Combines scoreboard with
  - Register renaming (to avoid WAR and WAW)
- Designed for the IBM 360/91
  - High FP performance for the whole 360 family
  - Four double precision FP registers
  - Long memory access and long FP delays
- Can support overlapped execution of multiple iterations of a loop

Tomasulo Algorithm vs. Scoreboard

- Control & buffers distributed with Function Units (FU) vs. centralized in scoreboard;
  - FU buffers called "reservation stations"; have pending operands
- Registers in instructions replaced by values or pointers to reservation stations(RS); called register renaming ;
  - avoids WAR, WAW hazards
  - More reservation stations than registers, so can do optimizations compilers can't
- Results to FU from RS, not through registers, over Common Data Bus that broadcasts results to all FUs
- Load and Stores treated as FUs with RSs as well
- Integer instructions can go past branches, allowing FP ops beyond basic block in FP queue

Tomasulo Organization



Reservation Station Components

Op—Operation to perform in the unit (e.g., + or –)
Qj, Qk—Reservation stations producing source registers
Vj, Vk—Value of Source operands
Rj, Rk—Flags indicating when Vj, Vk are ready
Busy—Indicates reservation station and FU is busy

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Qi—a field in register file, the number of the reservation station that contains the operation whose result should be stored into this register

Tomasulo's Algorithm

Instruction status	Wait until	Action or bookkeeping

Issue	Station or buffer empty	`if (Register['S1'].Qi ≠0)`
		`  {RS[r].Qj← Register['S1'].Qi}`
		`else {RS[r].Vj← S1; RS[r].Qj← 0};`
		`if (Register[S2].Qi≠0)`
		`  {RS[r].Qk← Register[S2].Qi};`
		`else {RS[r].Vk← S2; RS[r].Qk← 0}`
		`RS[r].Busy← yes;`
		`Register['D'].Qi=r;`

Execute	(RS[r].Qj=0) and (RS[r].Qk=0)	None—operands are in Vj and Vk

Write result	Execution completed at r and CDB available	`∀x(if (Register[x].Qi=r) {Fx← result;`
		`  Register[x].Qi← 0});`
		`∀x(if (RS[x].Qj=r) {RS[x].Vj← result;`
		`  RS[x].Qj ← 0});`
		`∀x(if (RS[x].Qk=r) {RS[x].Vk← result;`
		`  RS[x].Qk ← 0});`
		`∀x(if (Store[x].Qi=r) {Store[x].V← result;`
		`  Store[x].Qi ← 0});`
		`RS[r].Busy← No`

An enhanced and detailed design in Fig. 3.5 of the textbook

Tomasulo Algorithm for Dynamic Scheduling

- For IBM 360/91 in 1967 - about 3 years after CDC 6600
- Goal: High performance without special compilers
- Differences between IBM 360 & CDC 6600
  - IBM has only 2 register specifiers/instr vs. 3 in CDC 6600
  - IBM has register-memory instructions
  - IBM has 4 FP registers vs. 8 in CDC 6600
  - IBM has pipelined functional units (3 adds, 2 multiplies)
- Tomasulo algorithm is designed to handle name dependencies (WAW and WAR hazards) efficiently

```
SUB     F1, F2, F0
DIVF    F2, F3 , F2
ADDF    F3, F0, F0
MULF    F3, F1, F1
```

Tomosulo Algorithm

- Differences from Scoreboarding
  - Distributed hazard detection and control (through reservation stations)
  - Results are bypassed to function units
  - Common data bus (CDB) broadcasts results to all FUs.
  - HW renaming of registers to avoid WAR, WAW hazards
  - Load and Stores treated as FUs as well
  - Registers in instructions replaced by pointers to reservation station buffers
- Lead to concepts used in Alpha 21264, HP 8000, MIPS 10000, Pentium II, PowerPC 604, …

Three Stages of Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue

   If reservation station free, Tomasulo issues instr & sends operands (renames registers).

2. Execution—operate on operands (EX)

   When both operands ready then execute;
   if not ready, watch CDB for result

3. Write result—finish execution (WB)

   Write on Common Data Bus to all awaiting units;
   mark reservation station available.

- Normal data bus: data + destination (“go to” bus)
- Common data bus: data + source (“come from” bus)
  - 64 bits of data + 4 bits of Functional Unit source address
  - Write if matches expected Functional Unit (produces result)
  - Does the broadcast

Tomasulo Summary

- Advantages
  - Prevents register from being the bottleneck
  - Eliminates WAR, WAW hazards
  - Allows loop unrolling in HW
- Common Data Bus
  - Broadcasts results to multiple instructions
  - Central bottleneck
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation

Tomasulo Example

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
0	*FU*									

Tomasulo Example Cycle 1

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
1	FU				Load1					

Tomasulo Example Cycle 2

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3	2		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
2	FU		Load2		Load1					

Note: Unlike 6600, can have multiple loads outstanding

Tomasulo Example Cycle 3

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU	Mult1	Load2		Load1					

- Note: registers names are removed ("renamed") in Reservation Stations; MULT issued vs. scoreboard
- Load1 completing; what is waiting for Load1?

Tomasulo Example Cycle 4

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	Yes	SUBD	M(A1)			Load2
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	Mult1	Load2		M(A1)	Add1				

- Load2 completing; what is waiting for Load2?

Tomasulo Example Cycle 5

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
5	FU	Mult1	M(A2)		M(A1)	Add1	Mult2			

Tomasulo Example Cycle 6

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
6	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

- **Issue ADDD here vs. scoreboard?**

Tomasulo Example Cycle 7

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
0	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	Mult1	M(A2)		Add2	Add1	Mult2			

- Add1 completing; what is waiting for it?

Tomasulo Example Cycle 8

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
2	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
8	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Tomasulo Example Cycle 9

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
1	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
9	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Tomasulo Example Cycle 10

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
10	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

- Add2 completing; what is waiting for it?

Tomasulo Example Cycle 11

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
11	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- Write result of ADDD here vs. scoreboard?
- All quick instructions complete in this cycle!

Tomasulo Example Cycle 12

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
12	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 13

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 14

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
14	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 15

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
15	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 16

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
16	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

skip a couple of cycles

Tomasulo Example Cycle 55

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
1	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
55	FU	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 56

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
56	FU	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

- Mult2 is completing; what is waiting for it?

Tomasulo Example Cycle 57

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	57
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
56	FU	M*F4	M(A2)		(M-M+M	(M-M)	Result			

- Once again: In-order issue, out-of-order execution and completion.

Compare to Scoreboard Cycle 62

Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	62
ADDD	F6	F8	F2	13	14	16	22

Issue	*Exec Comp*	*Write Result*
1	3	4
2	4	5
3	15	16
4	7	8
5	56	57
6	10	11

- Why take longer on scoreboard/6600?
  - Structural Hazards
  - Lack of forwarding

Tomasulo v. Scoreboard (IBM 360/91 v. CDC 6600)

Pipelined Functional Units	Multiple Functional Units
(6 load, 3 store, 3 +, 2 x/÷)	(1 load/store, 1 + , 2 x, 1 ÷)
window size: ≤ 14 instructions	≤ 5 instructions
No issue on structural hazard	same
WAR: renaming avoids	stall completion
WAW: renaming avoids	stall issue
Broadcast results from FU	Write/read registers
Control: reservation stations	central scoreboard

Tomasulo Drawbacks

- The scheme requires a large amount of hardware
- Many associative stores (CDB) at high speed
- Performance limited by the single Common Data Bus
  - Multiple CDBs
    - Each CDB must interact with each reservation station
    - The associate tag-matching hardware would need to be duplicated at station for each CDB.

Discussion: Load/Store ordering

- In 360/91 loads allowed to bypass stores or loads with different addresses
- Stores must wait for “logically preceding” loads and stores to same address
  - Record original program order?
  - Serialize through effective address calculation?

Loop Iterations

Loop: LD F0, 0(R1)

MULTD F4,F0,F2

SD 0(R1), F4

SUBI R1, R1, #8

BNEZ R1, Loop

		Instruction status			
Instruction		**From iteration**	**Issue**	**Execute**	**Write result**
LD	F0,0(R1)	1	√	√	
MULTD	F4,F0,F2	1	√		
SD	0(R1),F4	1	√		
LD	F0,0(R1)	2	√	√	
MULTD	F4,F0,F2	2	√		
SD	0(R1),F4	2	√		

	Reservation stations					
Name	**Busy**	**Fm**	**Vj**	**Vk**	**Qj**	**Qk**
Add1	No					
Add2	No					
Add3	No					
Mult1	Yes	MULT		Regs[F2]	Load1	
Mult2	Yes	MULT		Regs[F2]	Load2	

	Register status								
Field	**F0**	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**...**	**F30**
Qi	Load2		Mult2						

	Load buffers		
Field	**Load 1**	**Load 2**	**Load 3**
Address	Regs[R1]	Regs[R1]-8	
Busy	Yes	Yes	No

	Store buffers		
Field	**Store 1**	**Store 2**	**Store 3**
Qi	Mult1	Mult2	
Busy	Yes	Yes	No
Address	Regs[R1]	Regs[R1]-8	

Loop Iterations

- The loop is unrolled dynamically by the hardware, using the reservation station.

Tomasulo Loop Example

```
Loop:LD      F0   0    R1
     MULTD   F4   F0   F2
     SD      F4   0    R1
     SUBI    R1   R1   #8
     BNEZ    R1   Loop
```

- Assume Multiply takes 4 clocks
- Assume first load takes 8 clocks (cache miss), second load takes 1 clock (hit)
- To be clear, will show clocks for SUBI, BNEZ
- Reality: integer instructions ahead

Loop Example

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1			
1	MULTD	F4	F0	F2			
1	SD	F4	0	R1			
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
0	**80**	*Fu*									

Loop Example Cycle 1

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2			
1	SD	F4	0	R1			
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
1	**80**	*Fu*	Load1								

Loop Example Cycle 2

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1			
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F4)	Load1	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
2	**80**	*Fu*	Load1		Mult1						

Loop Example Cycle 3

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F4)	Load1	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
3	**80**	*Fu*	Load1		Mult1						

- Implicit renaming sets up “DataFlow”

Loop Example Cycle 4

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F4)	Load1	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
4	**80**	*Fu*	Load1		Mult1						

- Dispatching SUBI Instruction

Loop Example Cycle 5

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F4)	Load1	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
5	**72**	*Fu*	Load1		Mult1						

- And, BNEZ instruction

Loop Example Cycle 6

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F4)	Load1	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
6	**72**	*Fu*	Load2		Mult1						

- Notice that F0 never sees Load from location 80

Loop Example Cycle 7

Instruction status:

					Exec	Write				
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*		*Busy*	*Addr*	*Fu*
1	LD	F0	0	R1	1			Load1	Yes	80	
1	MULTD	F4	F0	F2	2			Load2	Yes	72	
1	SD	F4	0	R1	3			Load3	No		
2	LD	F0	0	R1	6			Store1	Yes	80	Mult1
2	MULTD	F4	F0	F2	7			Store2	No		
2	SD	F4	0	R1				Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*	*Code:*			
	Add1	No						LD	F0	0	R1
	Add2	No						MULTD	F4	F0	F2
	Add3	No						SD	F4	0	R1
	Mult1	Yes	Multd		R(F2)	Load1		SUBI	R1	R1	#8
	Mult2	Yes	Multd		R(F2)	Load2		BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
7	**72**	*Fu*	Load2		Mult2						

- Register file completely detached from computation
- First and Second iteration completely overlapped

Loop Example Cycle 8

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
8	**72**	*Fu*	Load2		Mult2						

Loop Example Cycle 9

Instruction status:

					Exec	Write					
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*		*Busy*	*Addr*	*Fu*
1	LD	F0	0	R1	1	9		Load1	Yes	80	
1	MULTD	F4	F0	F2	2			Load2	Yes	72	
1	SD	F4	0	R1	3			Load3	No		
2	LD	F0	0	R1	6			Store1	Yes	80	Mult1
2	MULTD	F4	F0	F2	7			Store2	Yes	72	Mult2
2	SD	F4	0	R1	8			Store3	No		

Reservation Stations:

					S1	*S2*	*RS*				
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*	*Code:*			
	Add1	No						LD	F0	0	R1
	Add2	No						MULTD	F4	F0	F2
	Add3	No						SD	F4	0	R1
	Mult1	Yes	Multd		R(F2)	Load1		SUBI	R1	R1	#8
	Mult2	Yes	Multd		R(F2)	Load2		BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
9	**72**	*Fu*	Load2		Mult2						

- Load1 completing: who is waiting?
- Note: Dispatching SUBI

Loop Example Cycle 10

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	Multd	M[80]	R(F2)		
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
10	**64**	*Fu*	Load2		Mult2						

- Load2 completing: who is waiting?
- Note: Dispatching BNEZ

Loop Example Cycle 11

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*S1 Qj*	*S2 Qk*	*RS*
	Add1	No						
	Add2	No						
	Add3	No						
3	Mult1	Yes	Multd	M[80]	R(F2)			
4	Mult2	Yes	Multd	M[72]	R(F2)			

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
11	**64**	*Fu*	Load3		Mult2						

- Next load in sequence

Loop Example Cycle 12

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*S1 Qj*	*S2 Qk*	*RS*
	Add1	No						
	Add2	No						
	Add3	No						
2	Mult1	Yes	Multd	M[80]	R(F2)			
3	Mult2	Yes	Multd	M[72]	R(F2)			

Code:

```
LD    F0  0   R1
MULTD F4  F0  F2
SD    F4  0   R1
SUBI  R1  R1  #8
BNEZ  R1  Loop
```

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
12	**64**	*Fu*	Load3		Mult2						

- Why not issue third multiply?

Loop Example Cycle 13

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*S1 Qj*	*S2 Qk*	*RS*
	Add1	No						
	Add2	No						
	Add3	No						
1	Mult1	Yes	Multd	M[80]	R(F2)			
2	Mult2	Yes	Multd	M[72]	R(F2)			

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
13	**64**	*Fu*	Load3		Mult2						

Loop Example Cycle 14

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	Multd	M[80]	R(F2)		
1	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
14	**64**	*Fu*	Load3		Mult2						

- Mult1 completing. Who is waiting?

Loop Example Cycle 15

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

```
LD    F0  0   R1
MULTD F4  F0  F2
SD    F4  0   R1
SUBI  R1  R1  #8
BNEZ  R1  Loop
```

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
15	**64**	*Fu*	Load3		Mult2						

- Mult2 completing. Who is waiting?

Loop Example Cycle 16

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	[72]*R2
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
16	**64**	*Fu*	Load3		Mult1						

Loop Example Cycle 17

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	[72]*R2
Store3	Yes	64	Mult1

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
17	**64**	*Fu*	Load3		Mult1						

Loop Example Cycle 18

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3	18	
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	[72]*R2
Store3	Yes	64	Mult1

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
18	**64**	*Fu*	Load3		Mult1						

Loop Example Cycle 19

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3	18	19
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8	19	

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	No		
Store2	Yes	72	[72]*R2
Store3	Yes	64	Mult1

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
19	**64**	*Fu*	Load3		Mult1						

Loop Example Cycle 20

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3	18	19
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8	19	20

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	No		
Store2	No		
Store3	Yes	64	Mult1

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

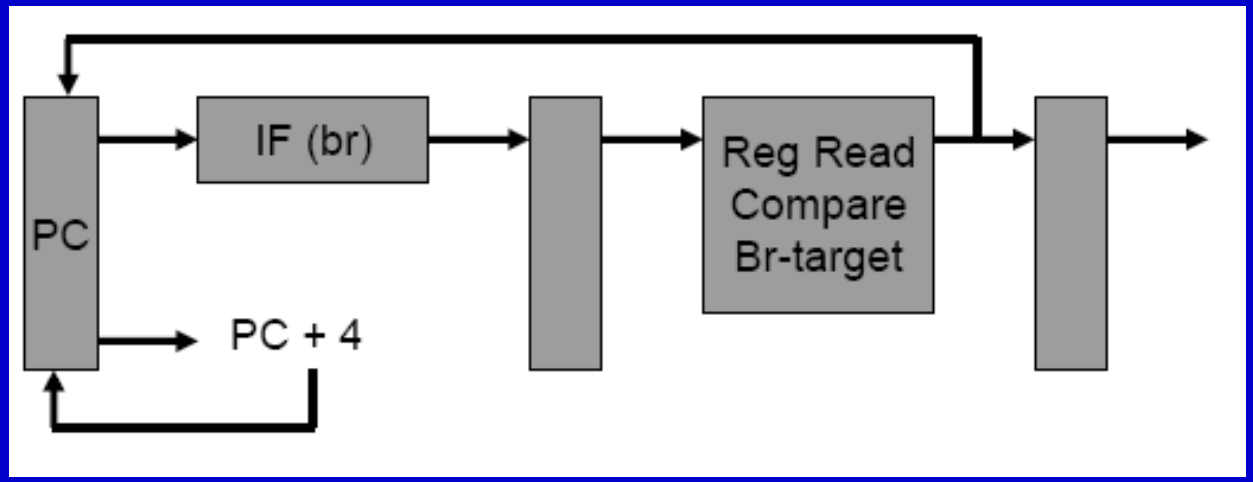
Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
20	**64**	*Fu*	Load3		Mult1						

3.4 Reducing Branch Costs with Dynamic Hardware Prediction

Control Hazards

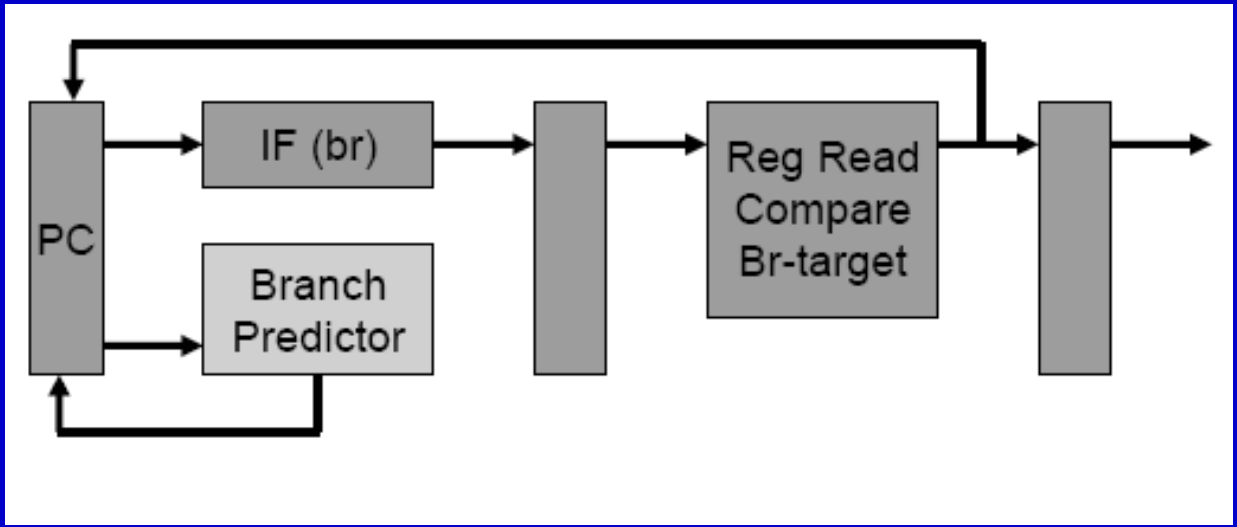
- In the 5-stage in-order processor: assume always taken or assume always not taken; if the branch goes the other way, squash mis-fetched instructions
- Modern out-of-order processors: dynamic branch prediction
- Branch predictor: a cache of recent branch outcomes

Pipeline without Branch Predictor



- In the 5-stage pipeline, a branch completes in two cycles
- If the branch went the wrong way, one incorrect instr is fetched
- One stall cycle per incorrect branch

Pipeline with Branch Predictor



- In the 5-stage pipeline, a branch completes in two cycles
- If the branch went the wrong way, one incorrect instr is fetched
- One stall cycle per incorrect branch

Dynamic Hardware Prediction

- Importance of control dependences
  - Branches and jumps are frequent
- Schemes to attack control dependences
  - Static
    - Basic (stall the pipeline)
    - Predict-not-taken and predict-taken
    - Delayed branch and canceling branch
  - **Dynamic predictors**
- The prediction will depend on the behavior of the branch at run time and the branch changes its own behavior during execution.
- Goal: allow the processor resolve the outcome of a branch early, preventing control dependences.
- Effectiveness of dynamic prediction schemes
  - Accuracy
  - Cost of the branch when the direction is incorrect

Basic Branch Prediction Buffers

- A branch-prediction buffer is a small memory indexed by the lower potion of the address of the branch instruction.
  - The memory contains whether the branch was recently taken or not.
  - Any branch with the same low-order address can modify the content.
- If the hint turns out to be wrong, the prediction bit is inverted and stored back.
- It is effectively a cache.

Performance Shortcoming

- If a branch is almost always taken, will predict incorrectly twice, rather than once, when it not taken.
  - Mispredict on the first and last loop iterations

Dynamic Branch Prediction

- Performance = *ƒ*(accuracy, cost of misprediction)
- Branch History Table: Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check
- Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 iterations before exit):
  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts exit instead of looping

1-Bit Prediction

- For each branch, keep track of what happened last time and use that outcome as the prediction
- What are prediction accuracies for branches 1 and 2 below:

```
while (1) {
    for (i=0;i<10;i++) {        branch-1
      …
    }
    for (j=0;j<20;j++) {        branch-2
      …
    }
}
```

2-Bit Prediction

- For each branch, maintain a 2-bit saturating counter:

  if the branch is taken: counter = min(3,counter+1)

  if the branch is not taken: counter = max(0,counter-1)
- If (counter >= 2), predict taken, else predict not taken
- Advantage: a few atypical branches will not influence the prediction (a better measure of “the common case”)
- Especially useful when multiple branches share the same counter (some bits of the branch PC are used to index into the branch predictor)
- Can be easily extended to N-bits (in most processors, N=2)

N-bit Branch Prediction Buffers

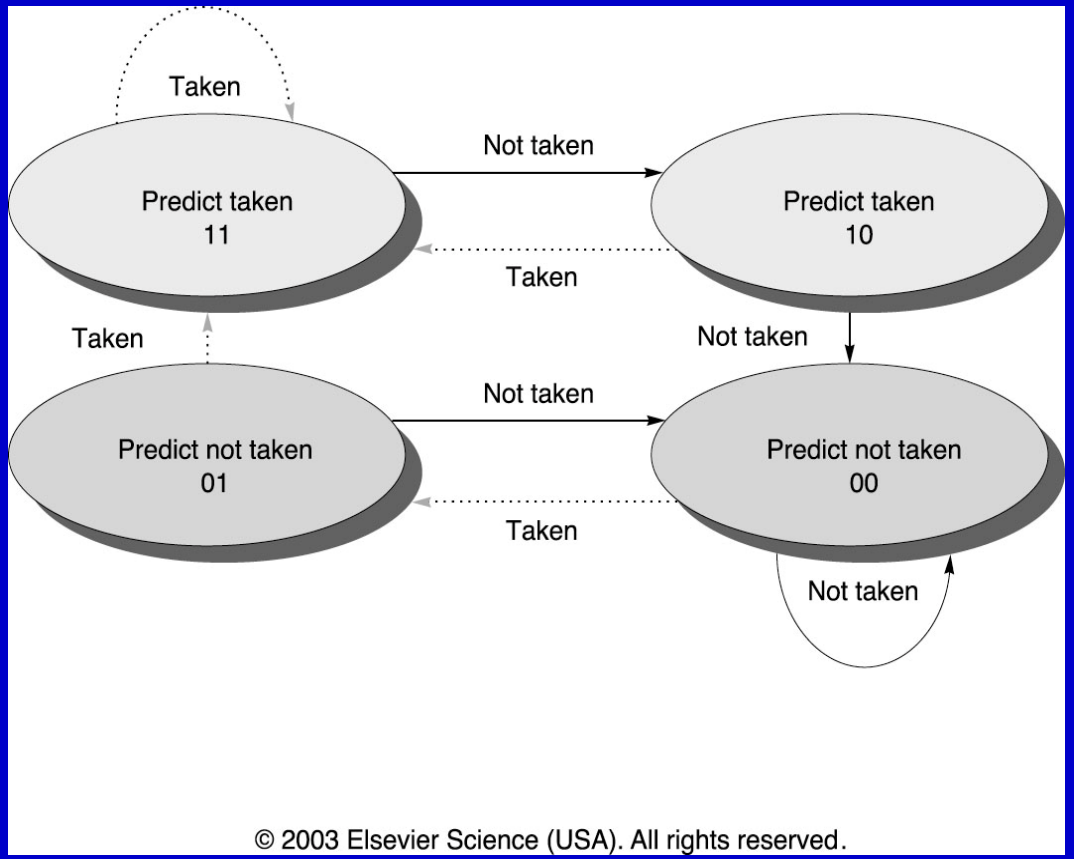
- When the counter is greater than or equal to one-half of its maximum value (2^n-1), the branch is predicted as taken.
- The counter is increased on a taken branch and decremented on an untaken branch.
- A branch buffer can be implemented as a small cache accessed during the IF stage.

N-bit Branch Prediction Buffers

Use an n-bit saturating counter

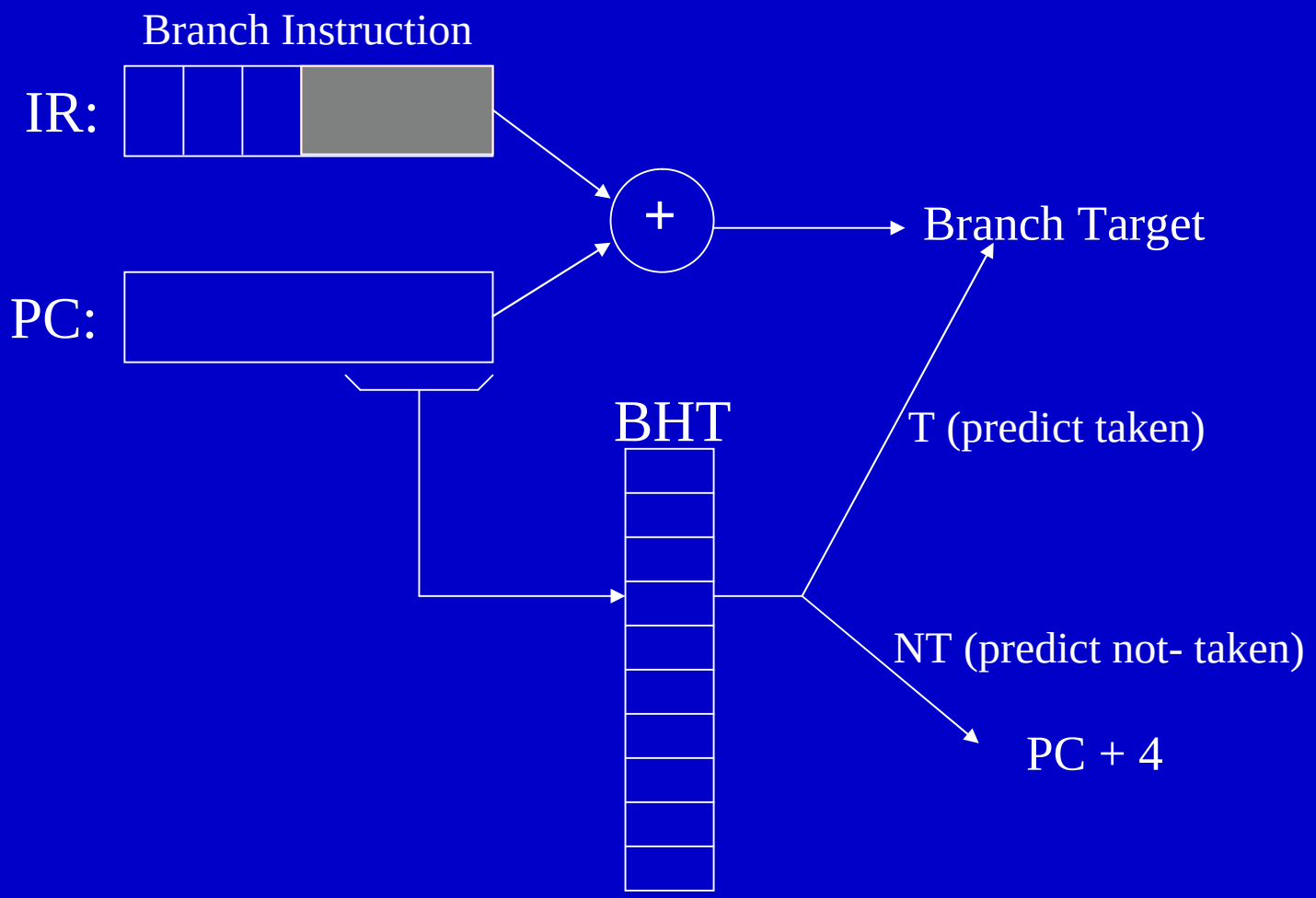
Only the loop exit causes a misprediction

2-bit predictor almost as good as any general n-bit predictor



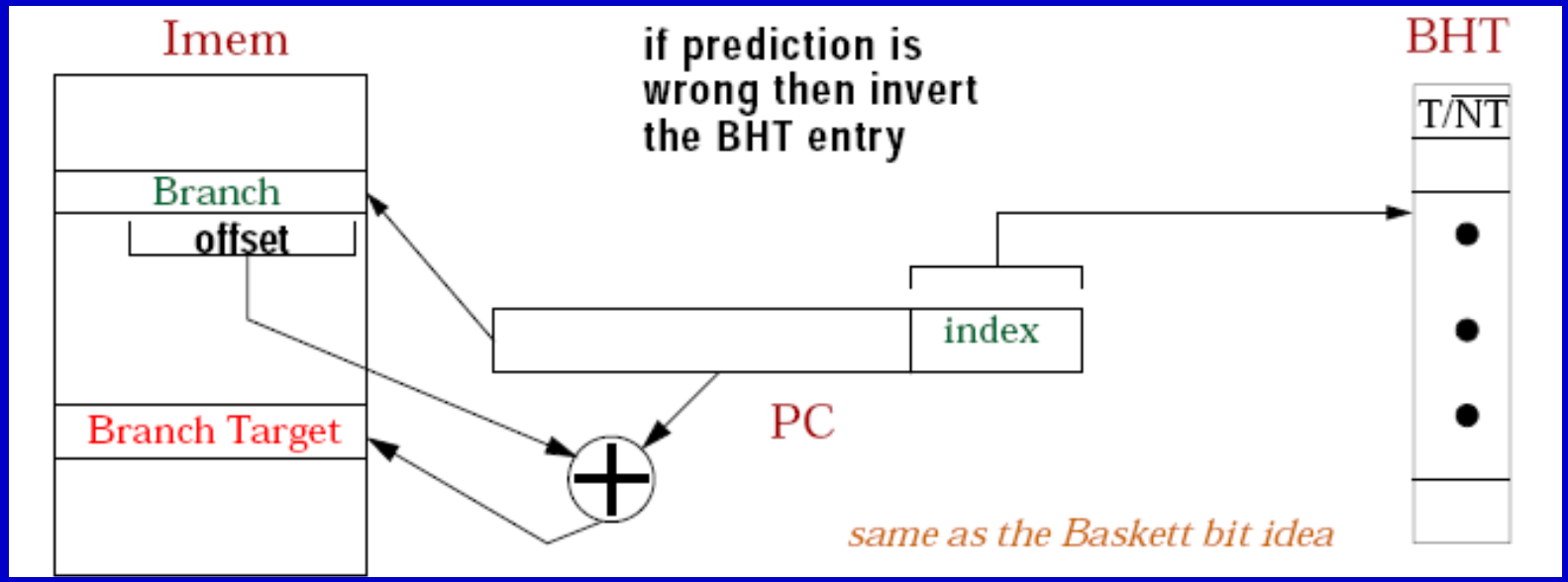
Basic Branch Prediction Buffers

a.k.a. Branch History Table (BHT) - Small direct-mapped cache of T/NT bits

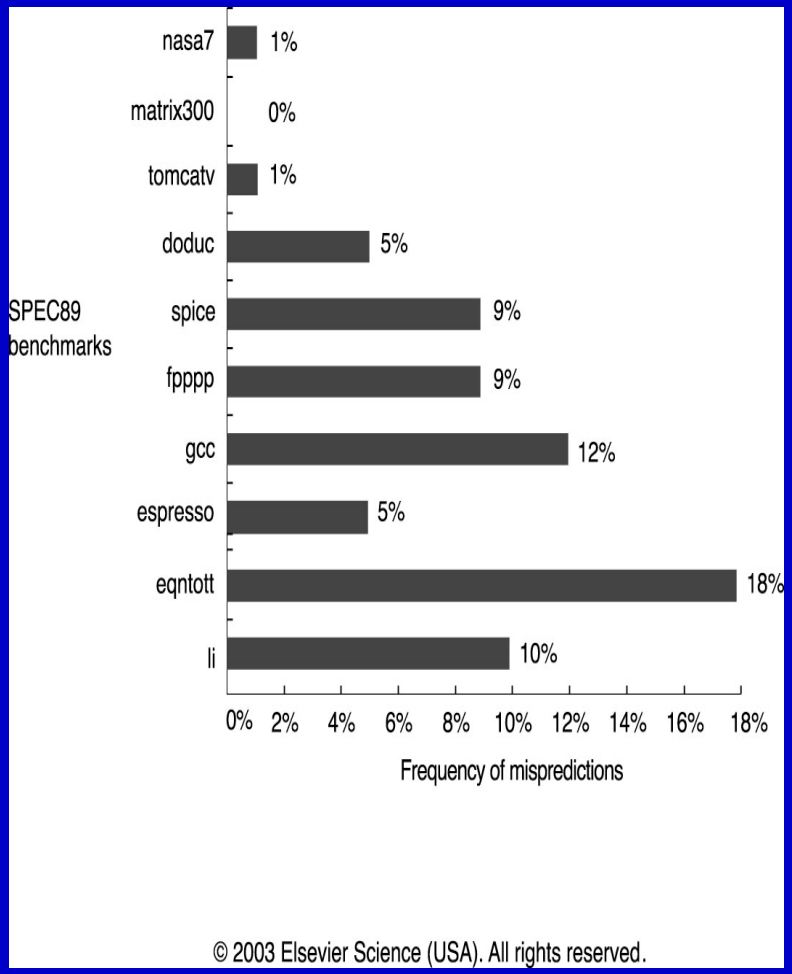


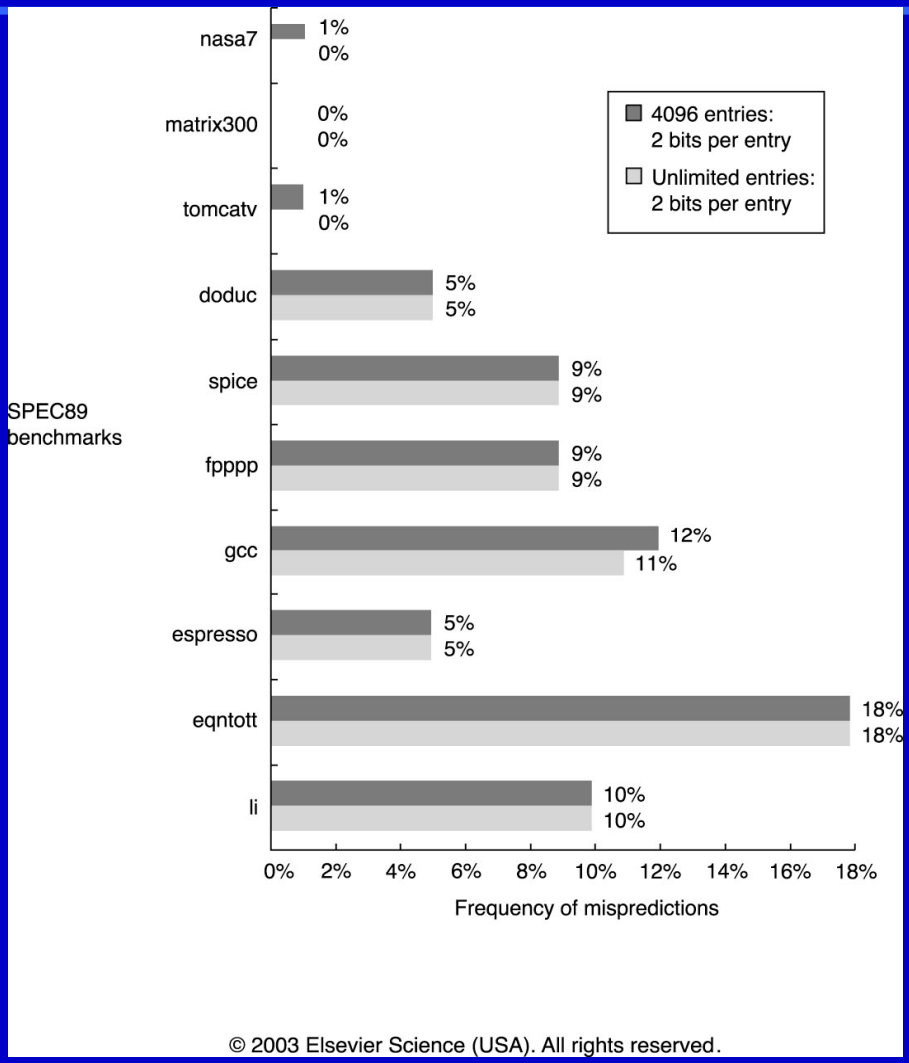
Branch History Table (or Prediction Buffer)

- small direct-mapped cache with a *which-way-last-time* bit
- low-order address bits index entry



Prediction Accuracy of a 4K-entry 2-bit Prediction Buffer





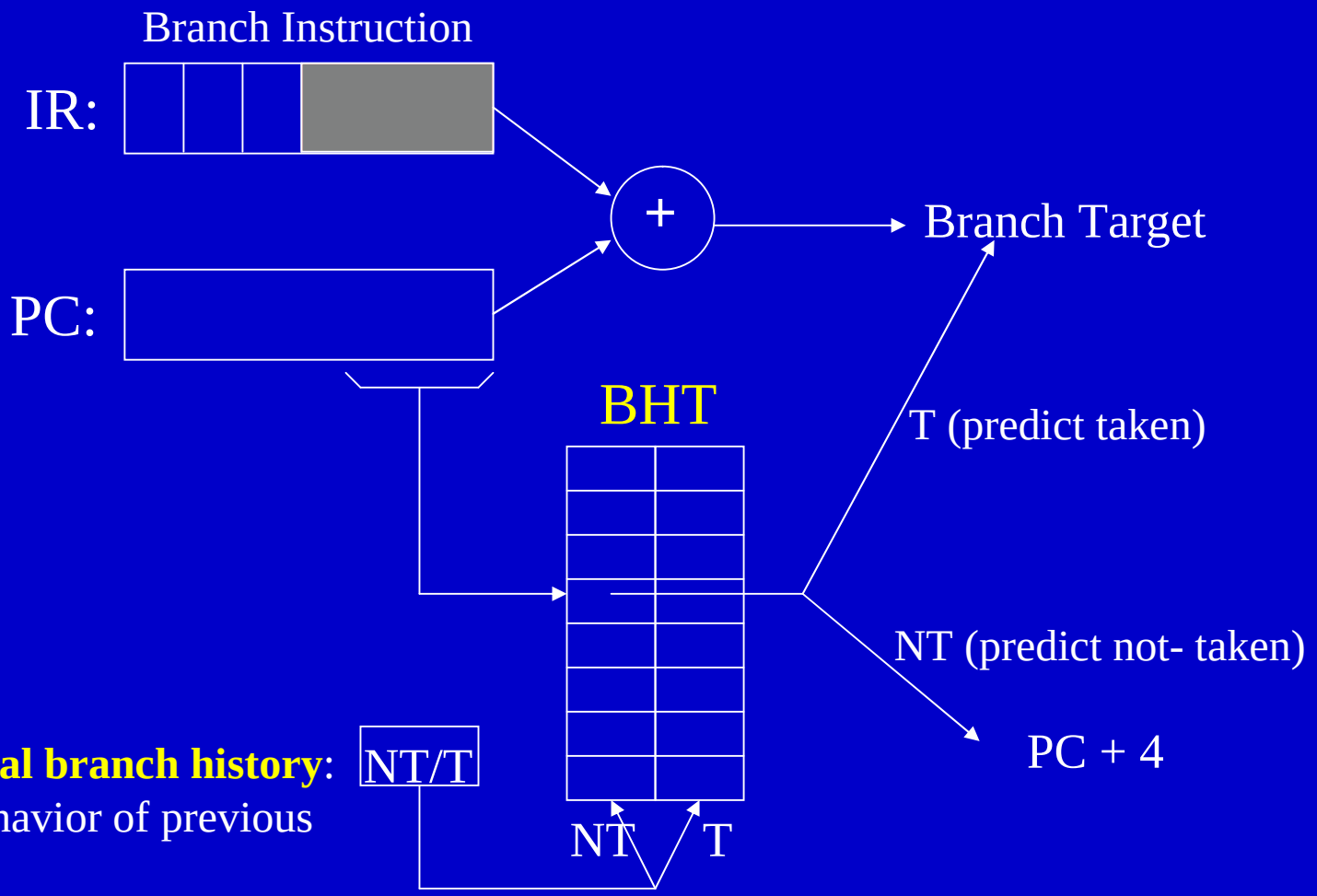
The integer programs have higher branch frequencies.
We should focus on the accurancy of each predicator.

Correlating Branch Predictors

- The previous schemes use only the recent behavior of a signal branch to predict the future behavior of the branch.
- Correlating Branch Predicator
  - Branch predictors that use the behavior of other branches to make a predication.

Correlating Branch Predictors

a.k.a. Two-level Predictors – Use recent behavior of other (previous) branches, 1 bit of correction.



1-bit global branch history: (stores behavior of previous branch)

Example

```
        BNEZ        R1, L1        ; branch b1 (d!=0)
        DADDIU      R1, R0, #1
L1:     DADDIU      R3, R1, #-1
        BNEZ        R3, L2 . . .  ; branch b2
L2:
```

Basic one-bit predictor

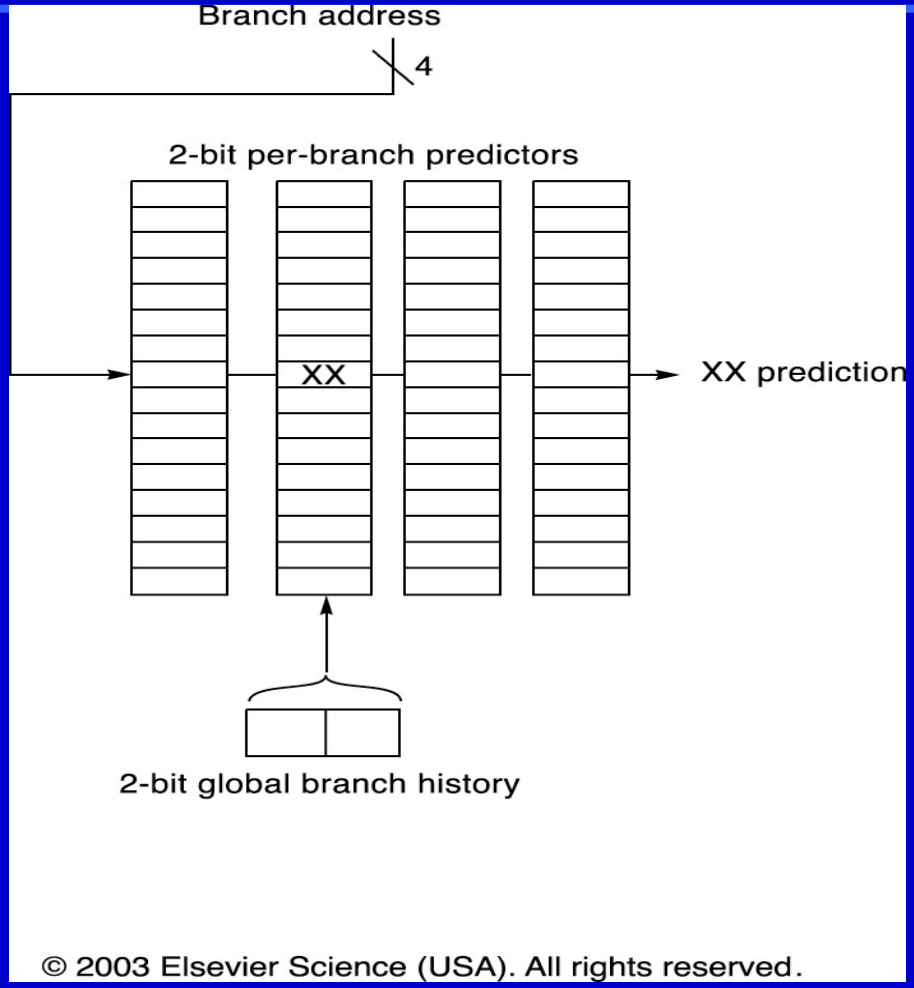
d=?	b1 pred	b1 action	new b1 pred	b2 pred	b2 action	new b2 pred
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

One-bit predictor with one-bit correlation

d=?	b1 pred	b1 action	new b1 pred	b2 pred	b2 action	new b2 pred
2	**NT**/NT	T	T/NT	NT/**NT**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T
2	**T**/NT	T	T/NT	NT/**T**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T

(2,2) Branch Prediction Buffer

$2^4 * 2^2 = 64$



The indexing is done by concatenating the global history bits and the number of required bits from the branch address.

Correlating Branches

- Hypothesis: recent branches are correlated; that is, behavior of recently executed branches affects prediction of current branch
- Idea: record m most recently executed branches as taken or not taken, and use that pattern to select the proper branch history table
- In general, (m,n) predictor means record last m branches to select between 2^m history talbes each with n-bit counters
  - Old 2-bit BHT is then a (0,2) predictor

Correlating Branches

- Often the behavior of one branch is correlated with the behavior of other branches.
- For example

C CODE		DLX CODE
if (aa == 2)		SUBI R3, R1, #2; BNEZ R3, L1
aa = 0;		ADD R1, R0, R0
if (bb == 2)	L1:	SUBI R3, R2, #2; BNEZ R3, L2
bb = 0;		ADD R2, R0, R0
if (aa != bb)	L2:	SUBI R3, R1, R2; BEQZ R3, L3
cc = 4;		ADD, R4, R0, #4
	L3:	

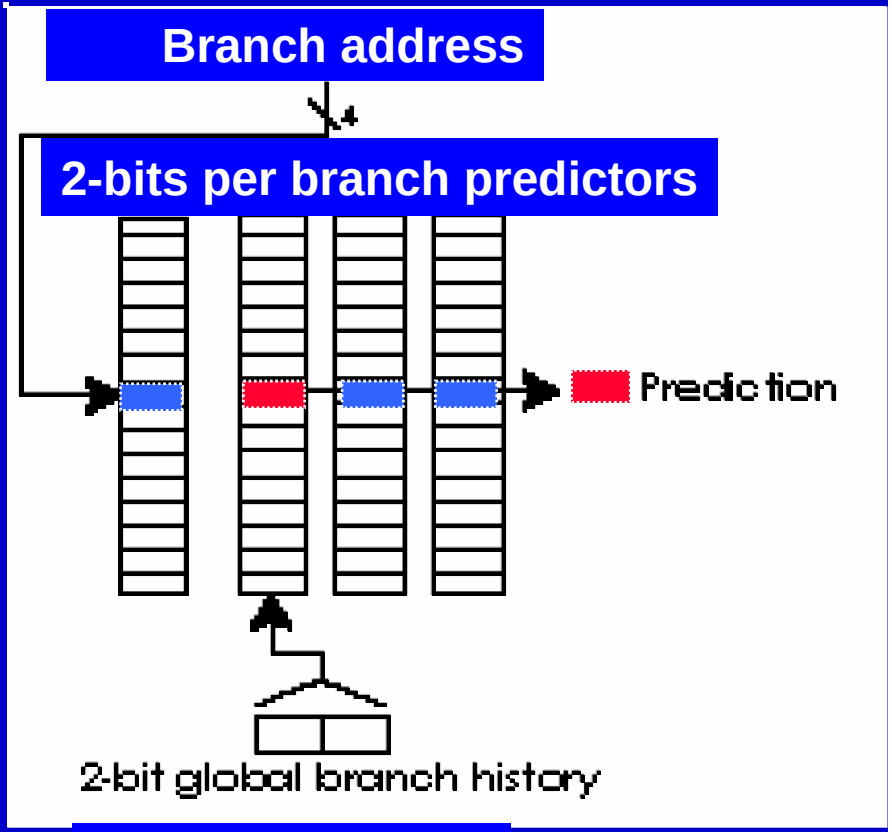
- If the first two branches are not taken, the third one will be.

Correlating Predicators

- Correlating predicators or two-level predictors use the behavior of other branches to predict if the branch is taken.
  - An (m, n) predictor uses the behavior of the last m branches to chose from (2^m) n-bit predictors.
  - The branch predictor is accessed using the low order k bits of the branch address and the m-bit global history.
  - The number of bits needed to implement an (m, n) predictor, which uses k bits of the branch address is

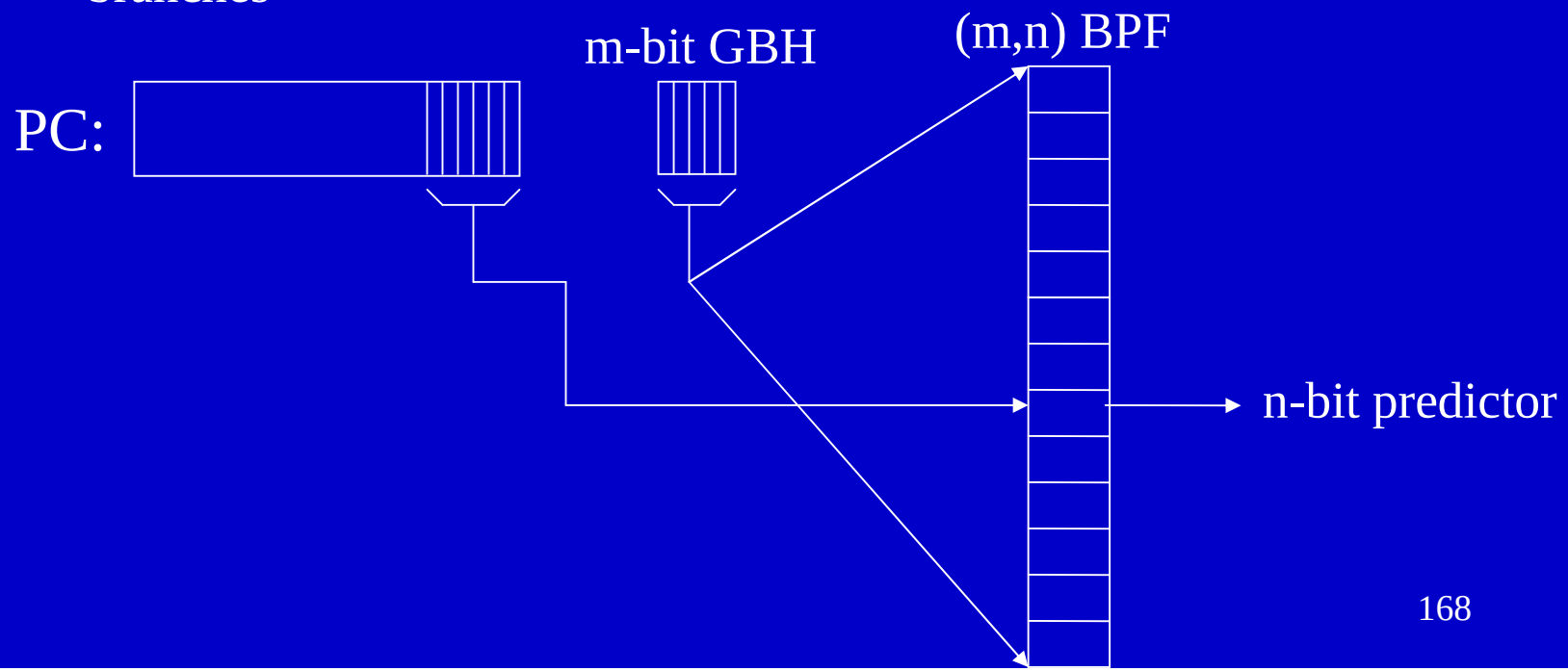
    $2^m \times n \times 2^k$
  - In the figure, we have m = 2, n = 2, k=4

    $2^2 \times 2 \times 2^4 = 128$ bits



(m, n) Predictors

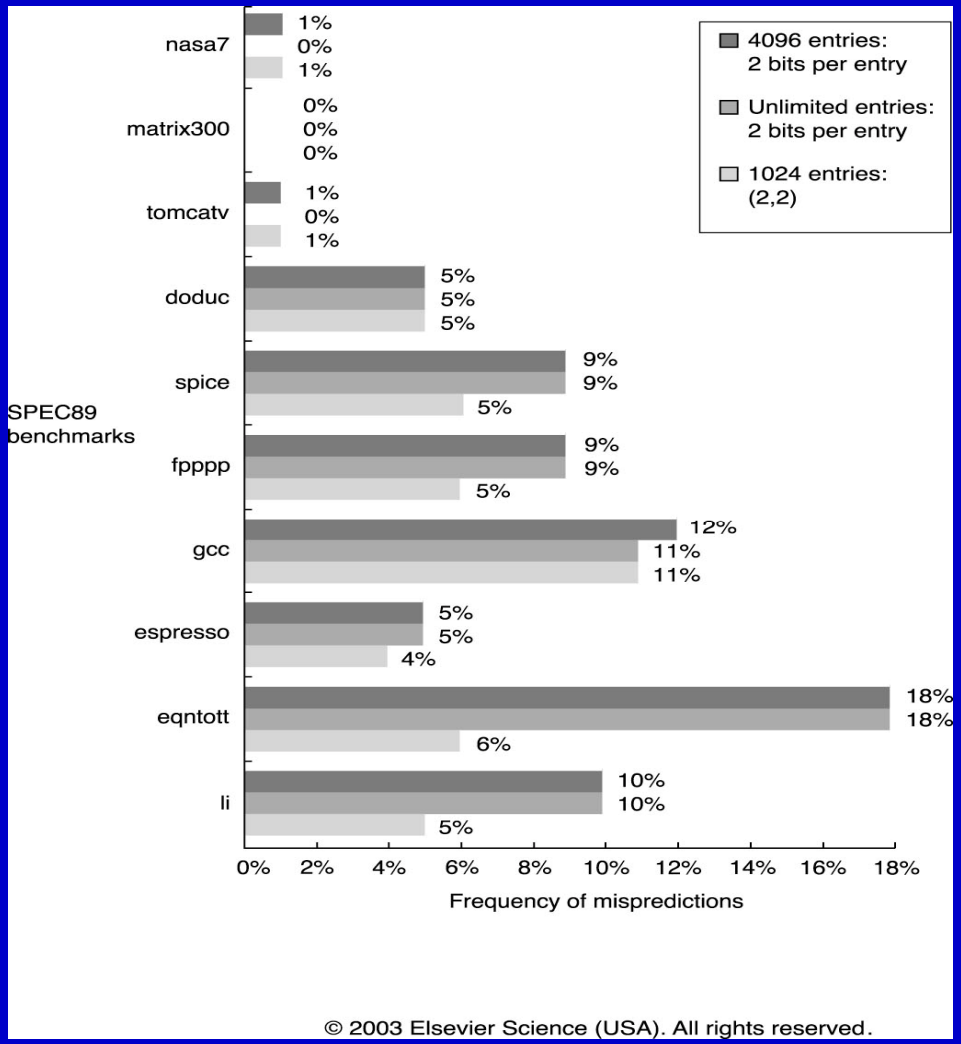
- Use behavior of the last m branches
  - Use last m branches = global branch history
  - Use n bit predictor
- 2^m n-bit predictors for each branch
- Simple implementation
  - Use m-bit shift register to record the behavior of the last m branches



Size of the Buffers

- Number of bits in a (m,n) predictor
  - $2^m \times n \times$ Number of entries in the table
    - ***x***: the nubmer of predication entries selected by the branch address
- Example – assume 8K bits in the BHT
  - (0,1): 8K entries
  - (0,2): 4K entries
  - (2,2): 1K entries
  - (12,2): 1 entry!
    - Does not use the branch address
    - Relies only on the global branch history

Performance Comparison of 2-bit Predictors

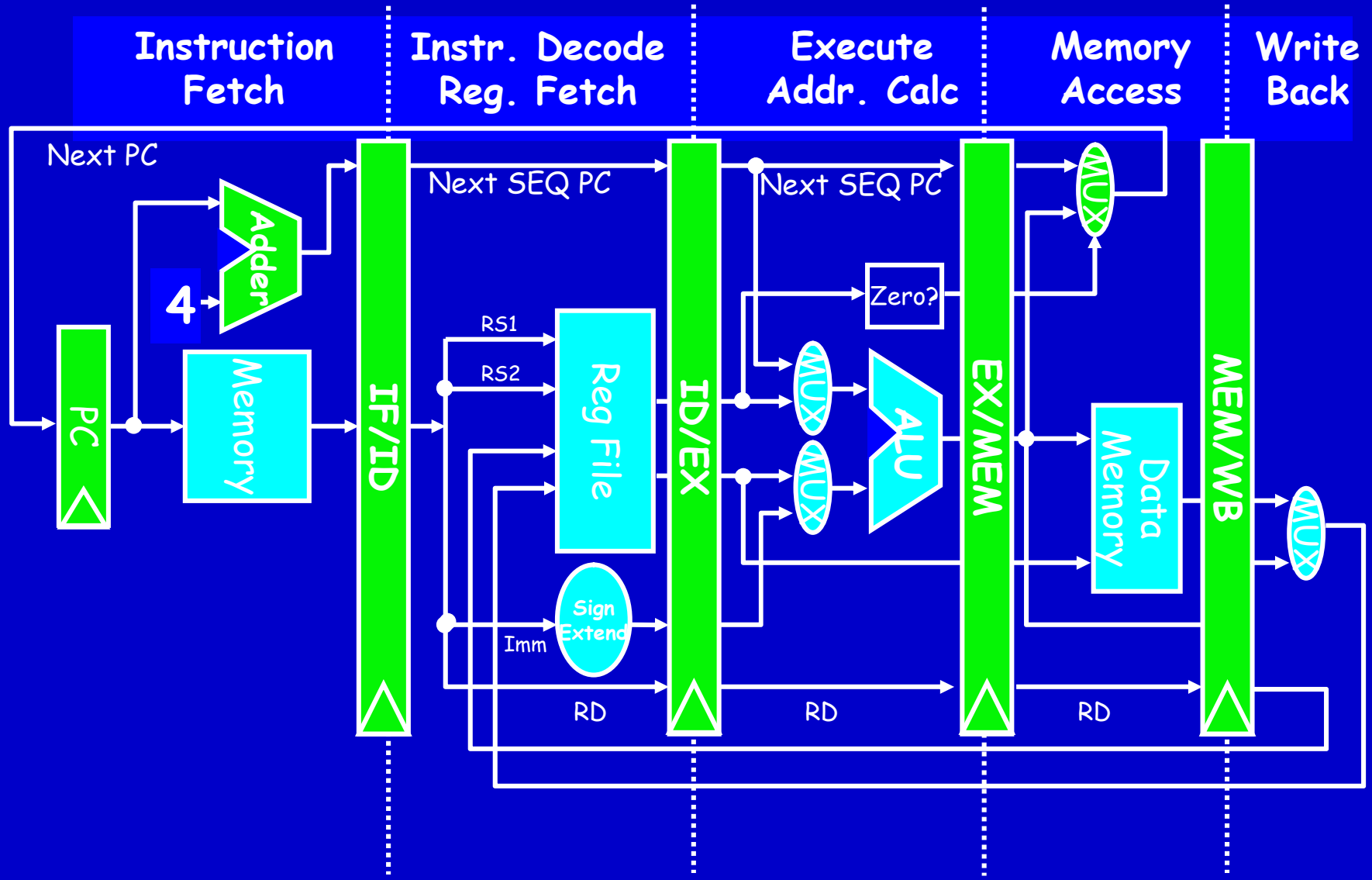


3.5 High-Performance Instruction Delivery

Motivation

- To reduce the branch penalty for the five-stage pipeline, we need to know from what address to fetch by the end of IF.
- Branch-Target Buffers (BTB)
  - Store the predicated address for the next instruction after a branch
- A branch-*predication* buffer is accessed during the ID cycle, so that at the end of ID we know the branch–target address.
- For a branch-*target* buffer, we access the buffer during the IF stage using instruction address of the fetched instruction.
  - If it is a branch instruction, then we know the predicted instruction address at the end of the IF cycle.
  - It is one cycle earlier than for a branch-prediction buffer.

5 Steps of DLX Datapath

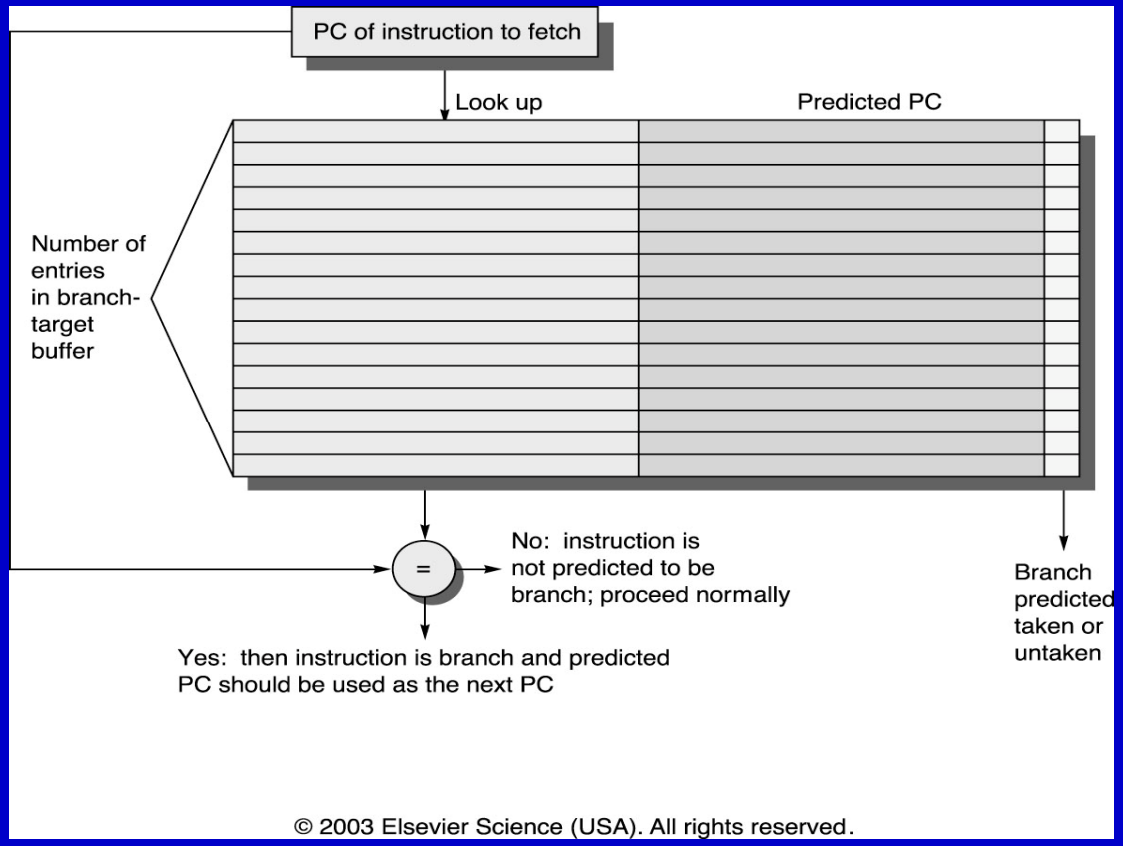


Branch-Target Buffers

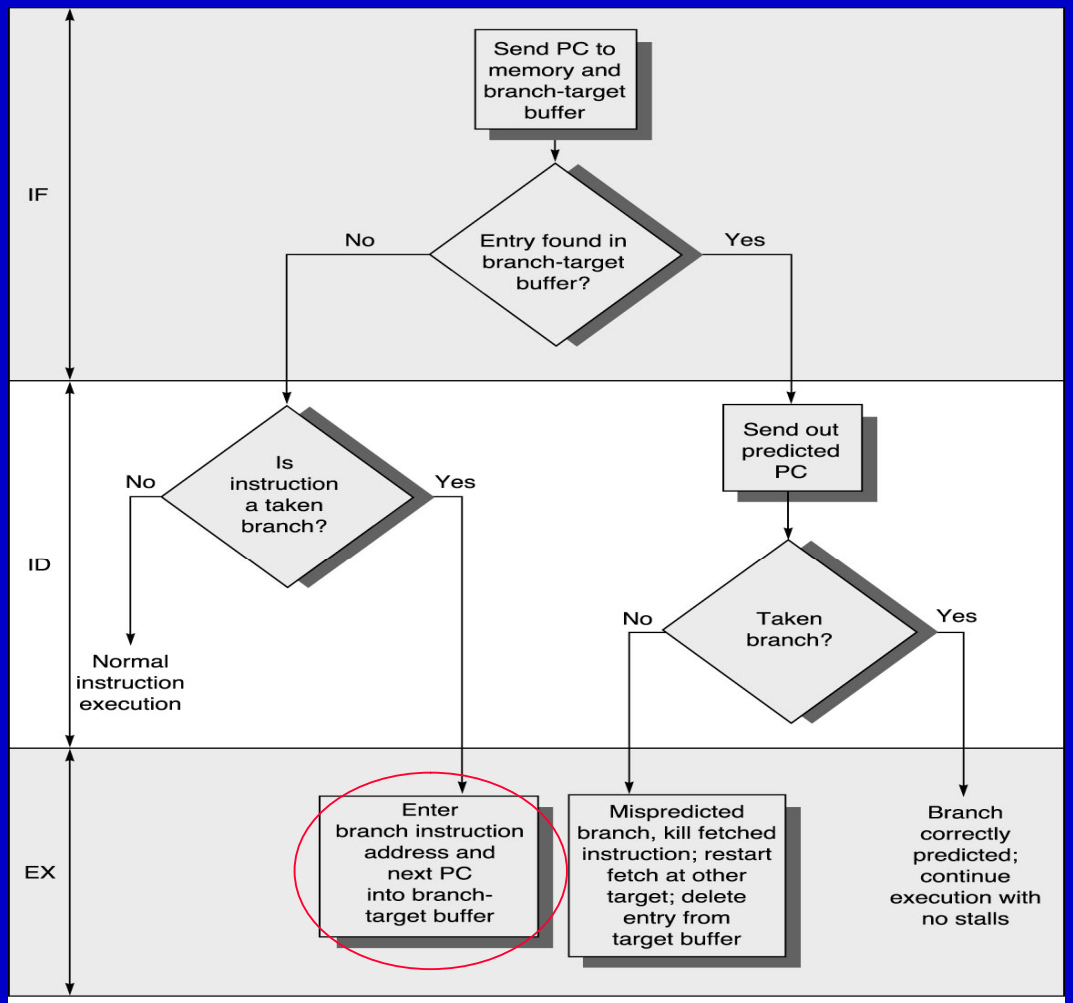
- DLX computes the branch target in the ID stage, which leads to a one cycle stall when the branch is taken.
- A branch-target buffer or branch-target cache stores the predicted address of branches that are predicted to be taken.
- Values not in the buffer are predicted to be not taken.
- The branch-target buffer is accessed during the IF stage, based on the k low-order bits of the branch address.
- If the branch target is in the buffer and is predicted correctly, the one cycle stall is eliminated.

Branch-Target Buffers

- Further reduce control stalls (hopefully to 0)
- Store the predicted address in the buffer
- Access the buffer during IF



Prediction with BTB



Target Instruction Buffers

- Store target instructions instead of the predicated target address
- Advantages
  - BTB access can take longer than time between IFs and BTB can be larger
  - Branch folding
    - Buffering the actual target instructions allows us to perform an optimization
    - Zero-cycle unconditional branches
      - Replace branch with target instruction

Performance Issues

- Limitations of branch prediction schemes
  - Prediction accuracy (80% - 95%)
    - Type of program
    - Size of buffer
  - Penalty of misprediction
- Fetch from both directions to reduce penalty
  - Memory system should:
    - Dual-ported
    - Have an interleaved cache
    - Fetch from one path and then from the other

3.6 Taking Advantage of More ILP with Multiple Issue

Issuing Multiple Instructions/Cycle

- Two variations
- Superscalar: varying no. instructions/cycle (1 to 8), scheduled by compiler or by HW (Tomasulo)
  - IBM PowerPC, Sun UltraSparc, DEC Alpha, HP 8000
- (Very) Long Instruction Words (V)LIW: fixed number of instructions (4-16) scheduled by the compiler; put ops into wide templates
- Anticipated success lead to use of Instructions Per Clock cycle (IPC) vs. CPI

Superscalar DLX

- Superscalar DLX: 2 instructions; 1 FP op, 1 other
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - 2 more ports for FP registers to do FP load or FP store and FP op

Type	*Pipe Stages*						
Int. instruction	IF	ID	EX	MEM	WB		
FP instruction	IF	ID	EX	MEM	WB		
Int. instruction		IF	ID	EX	MEM	WB	
FP instruction		IF	ID	EX	MEM	WB	
Int. instruction			IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB

- 1 cycle load delay expands to 3 cycles in 2-way SS
  - instruction in right half can't use it, nor instructions in next slot
- Branches also have a delay of 3 cycles

Unrolled Loop that Minimizes Stalls for Scalar

```
1 Loop: LD     F0,0(R1)
2       LD     F6,-8(R1)
3       LD     F10,-16(R1)
4       LD     F14,-24(R1)
5       ADDD   F4,F0,F2
6       ADDD   F8,F6,F2
7       ADDD   F12,F10,F2
8       ADDD   F16,F14,F2
9       SD     0(R1),F4
10      SD     -8(R1),F8
12      SUBI   R1,R1,#32
11      SD     16(R1),F12
13      BNEZ   R1,LOOP
14      SD     8(R1),F16
```

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

14 clock cycles, or 3.5 per iteration

Loop Unrolling in Superscalar

	Integer instruction	*FP instruction*	*Clock cycle*
Loop:	LD F0,0(R1)		1
	LD F6,-8(R1)		2
	LD F10,-16(R1)	ADDD F4,F0,F2	3
	LD F14,-24(R1)	ADDD F8,F6,F2	4
	LD F18,-32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD -8(R1),F8	ADDD F20,F18,F2	7
	SD -16(R1),F12		8
	SUBI R1,R1,#40		10
	SD 16(R1),F16		9
	BNEZ R1,LOOP		11
	SD 8(R1),F20		12

- Unrolled 5 times to avoid delays (+1 due to SS)
- 12 clocks, or 2.4 clocks per iteration

Dynamic Scheduling in Superscalar

- How to issue two instructions and keep in-order instruction issue for Tomasulo?
  - Assume 1 integer + 1 floating point
  - 1 Tomasulo control for integer, 1 for floating point
- Issue 2X clock rate, so that issue remains in order
- Only FP loads might cause dependency between integer and FP issue:
  - Replace load reservation station with a load queue; operands must be read in the order they are fetched
  - Load checks addresses in Store Queue to avoid RAW violation
  - Store checks addresses in Load & Store Queues to avoid WAR,WAW
  - Called "decoupled architecture"

Limits of Superscalar

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - No hazards
- If more instructions issue at same time, greater difficulty of decode and issue
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue
- Issue rates of modern processors vary between 2 and 8 instructions per cycle.

VLIW Processors

- Very Long Instruction Word (VLIW) processors
  - Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word can execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - 16 to 24 bits per field => 7*16 (112) to 7*24 (168) bits wide
  - Need compiling technique that schedules across branches

Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2			6
SD -16(R1),F12	SD -24(R1),F16				7
SD -32(R1),F20	SD -40(R1),F24			SUBI R1,R1,#48	8
SD -0(R1),F28				BNEZ R1,LOOP	9

- Unroll loop 7 times to avoid delays
- 7 results in 9 clocks, or 1.3 clocks per iteration
- Need more registers in VLIW

Limits to Multi-Issue Machines

- Limitations specific to either SS or VLIW implementation
  - Decode/issue in SS
  - VLIW code size: unroll loops + wasted fields in VLIW
  - VLIW lock step => 1 hazard & all instructions stall
- Inherent limitations of ILP
  - 1 branch in 5 instructions => how to keep a 5-way VLIW busy?
  - Latencies of units => many operations must be scheduled
  - Need about Pipeline Depth x No. Functional Units of independent instructions to keep all busy

Example

- Can issue two arbitrary operations per clock
- One integer FU for ALU operation and EA-calculation
- A separate pipelined FP FU
- One memory unit, 2CDB
- no delayed branch with perfect branch prediction
  - Fetch and issue as if the branch predictions are always correct
- Latency between a source instruction and an instruction consuming the result – presence of Write Result stage
  - 1 CC for integer ALU operations
  - 2 CC for loads
  - 3 CC for FP add

Note

- WR stages does not apply to either stores or branches
- For L.D and S.D, the execution cycle is EA calculation
- For branches, the execution cycle shows when the branch condition can be evaluated and the prediction checked
- Any instruction following a branch cannot start execution until after the branch condition has been evaluated
- If two instructions could use the same FU at the same point (structural hazard), priority is given to the older instruction

Consider adding a scalar *s* to a vector (p.221)

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

```
Loop:  L.D      F0,0( R1 )     ;F0=vector element
       ADD.D    F4,F0,F2       ;add scalar from F2
       S.D      F4, 0(R1)      ;store result
       DAADIU   R1,R1,#-8      ;decrement pointer 8B (DW)
       BNE      R1, R2, Loop   ;branch R1!=R2
```

Execution Timing

Perfect branch predicting

Iteration number	Instructions		Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D	F0,0(R1)	1	2	3	4	First issue
1	ADD.D	F4,F0,F2	1	5		8	Wait for L.D
1	S.D	F4,0(R1)	2	3	9		Wait for ADD.D
1	DADDIU	R1,R1,#-8	2	4		5	Wait for ALU
1	BNE	R1,R2,Loop	3	6			Wait for DADDIU
2	L.D	F0,0(R1)	4	7	8	9	Wait for BNE complete
2	ADD.D	F4,F0,F2	4	10		13	Wait for L.D
2	S.D	F4,0(R1)	5	8	14		Wait for ADD.D
2	DADDIU	R1,R1,#-8	5	9		10	Wait for ALU
2	BNE	R1,R2,Loop	6	11			Wait for DADDIU
3	L.D	F0,0(R1)	7	12	13	14	Wait for BNE complete
3	ADD.D	F4,F0,F2	7	15		18	Wait for L.D
3	S.D	F4,0(R1)	8	13	19		Wait for ADD.D
3	DAADIU	R1,R1,#-8	8	14		15	Wait for ALU
3	BNE	R1,R2,Loop	9	16			Wait for DADDIU

Execution Timing (Cont.)

Clock number	Integer ALU	FP ALU	Data cache	CDB
2	1/ L.D			
3	1 / S.D		1/ L.D	
4	1 / DADDIU			1/ L.D
5		1 / ADD.D		1 / DADDIU
6				
7	2 / L.D			
8	2 / S.D		2 / L.D	1 / ADD.D
9	2 / DADDIU		1 / S.D	2 / L.D
10		2 / ADD.D		2 / DADDIU
11				
12	3 / L.D			
13	3 / S.D		3 / L.D	2 / ADD.D
14	3 / DADDIU		2 / S.D	3 / L.D
15		3 / ADD.D		3 / DADDIU
16				
17				
18				3 / ADD.D
19			3 / S.D	
20				

Example Result

- Result
  - IPC issued = 5/3 = 1.67; Instruction execution rate = 15/16 = 0.94
    - Only one load, store, and Integer ALU operation can execute
  - Load of the next iteration performs its memory address before the store of the current iteration
  - A single CDB is actually required
  - Integer operations become the bottleneck
    - Many integer operations, but only one integer ALU
  - One stall cycle each loop iteration due to a branch hazard

Another Example – Execution Timing (p.223)

Iteration number	Instructions		Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D	F0,0(R1)	1	2	3	4	First issue
1	ADD.D	F4,F0,F2	1	5		8	Wait for L.D
1	S.D	F4,0(R1)	2	3	9		Wait for ADD.D
1	DADDIU	R1,R1,#-8	2	3		4	Executes earlier
1	BNE	R1,R2,Loop	3	5			Wait for DADDIU
2	L.D	F0,0(R1)	4	6	7	8	Wait for BNE complete
2	ADD.D	F4,F0,F2	4	9		12	Wait for L.D
2	S.D	F4,0(R1)	5	7	13		Wait for ADD.D
2	DADDIU	R1,R1,#-8	5	6		7	Executes earlier
2	BNE	R1,R2,Loop	6	8			Wait for DADDIU
3	L.D	F0,0(R1)	7	9	10	11	Wait for BNE complete
3	ADD.D	F4,F0,F2	7	12		15	Wait for L.D
3	S.D	F4,0(R1)	8	10	16		Wait for ADD.D
3	DADDIU	R1,R1,#-8	8	9		10	Executes earlier
3	BNE	R1,R2,Loop	9	11			Wait for DADDIU

Separate integer FU for EA calculation and ALU operations

Execution Timing (Cont.)

Clock number	Integer ALU	Address adder	FP ALU	Data cache	CDB #1	CDB #2
2		1/ L.D				
3	1 / DADDIU	1 / S.D		1/ L.D		
4					1/L.D	1 / DADDIU
5			1 / ADD.D			
6	2 / DADDIU	2 / L.D				
7		2 / S.D		2 / L.D	2 / DADDIU	
8					1 / ADD.D	2 / L.D
9	3 / DADDIU	3 / L.D	2 / ADD.D	1 / S.D		
10		3 / S.D		3 / L.D	3 / DADDIU	
11					3 / L.D	
12			3 / ADD.D		2 / ADD.D	
13				2 / S.D		
14						
15						3 / ADD.D
16				3 / S.D		

Note

- Result
  - IPC issued = 5/3 = 1.67; Instruction execution rate = 15/11 = 1.36
  - A second CDB is needed
  - This example has a higher instruction execution rate but lower efficiency as measured by the utilization of FU

Summary

- Branch Prediction
  - Branch History Table: 2 bits for loop accuracy
  - Correlation: Recently executed branches correlated with next branch
  - Branch Target Buffer: include branch address if predicted taken
- Superscalar and VLIW
  - CPI < 1
  - Superscalar is more hardware dependent (dynamic)
  - VLIW is more compiler dependent (static)
  - More instructions issue at same time => larger penalties for hazards

3.7 Hardware-based Speculation

Hardware-Based Speculation

- Overcome control dependence by speculating on the outcome of branches and executing the program as if our guesses were correct
  - Fetch, issue, and execute instructions
  - Need mechanisms to handle the situation when the speculation is incorrect
- Dynamic scheduling: only fetch and issue such instructions

Key Ideas

- Dynamic branch prediction to choose which instructions to execute
- Speculation to allow the speculated blocks to execution before the control dependences are resolved
  - And undo the effects of an incorrectly speculated sequence
- Dynamic scheduling to deal with the scheduling of different combinations of basic blocks (Tomasulo style approach)

HW support for More ILP

- *Speculative execution*: allow an instruction to issue that is dependent on branch predicted to be taken *without* any consequences (including exceptions) if branch is not actually taken.
- Hardware needs to undo the instruction - hard to do if there are exceptions
- Simple solution - don't do speculative moves if the instruction can cause an exception
  - Eliminates benefits for memory and floating point operations
- Need to consider two types of exceptions
  - Exception due to a program error and causes termination (e.g., memory violation). Result of program is undefined. Don't want a speculative instruction to cause this.
  - Exception that can be handled normally and resume (e.g. , page fault). Result of program is defined. O.K. if a speculative instruction causes this.

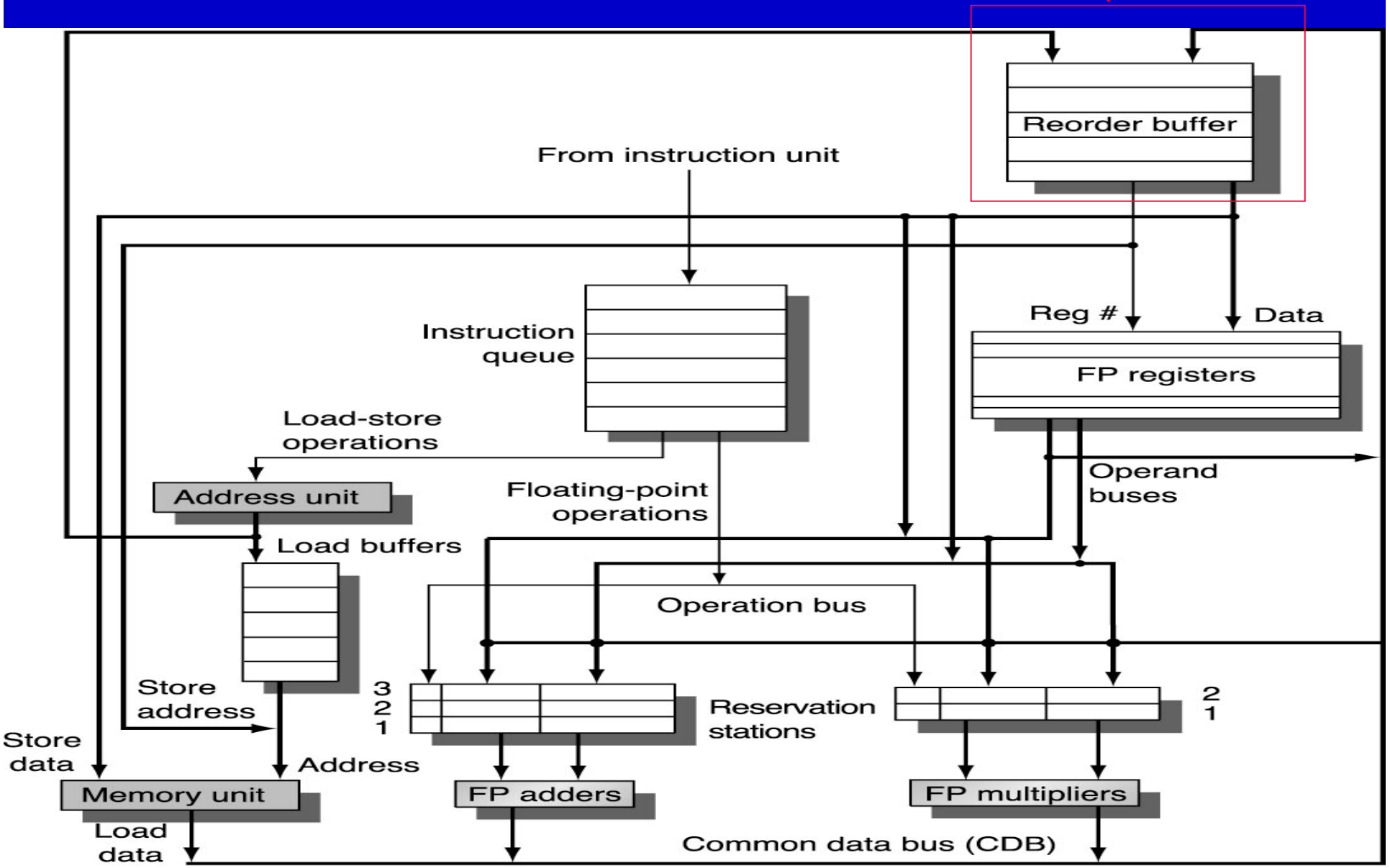
Speculative Execution

- There are three main methods for supporting speculation without introducing incorrect exception behavior.
  - The hardware and O.S. work together to ignore exceptions for speculative instructions. Programs that would terminate, may give incorrect results.
  - A set of status bits called poison bits are attached to result registers written by speculated instructions when the instructions cause exceptions. The poison bits cause a fault when a normal instruction uses the register.
  - Provide a mechanism to indicate that an instruction is speculative and hardware buffer the result until the instruction is no longer speculative (I.e., the result of the branch is known).

HW Speculation Approach

- Issue → execution → write result → commit
  - Commit is the point where the operation is no longer speculative
- Allow out of order execution
  - Require in-order commit
  - Prevent speculative instructions from performing destructive state changes (e.g. memory write or register write)
- Collect pre-commit instructions in a reorder buffer (ROB)
  - Holds completed but not committed instructions
  - Effectively contains a set of virtual registers to store the result of speculative instructions until they are no longer speculative
    - Similar to reservation station → And becomes a bypass source

The Speculative MIPS



The Speculative MIPS (Cont.)

- Need HW buffer for results of uncommitted instructions: *reorder buffer (ROB)*
  - 4 fields: instruction type, destination field, value field, ready field
  - ROB is a source of operands → more registers like RS
    - ROB supplies operands in the interval between completion of instruction execution and instruction commit
    - Use ROB number instead of RS to indicate the source of operands when execution completes (but not committed)
  - Once instruction commits, result is put into register
  - As a result, its easy to undo speculated instructions on mispredicted branches or on exceptions

ROB Fields

- Instruction type – branch, store, register operations
- Destination field
  - Unused for branches
  - Memory address for stores
  - Register number for load and ALU operations (register operations)
- Value – hold the value of the instruction result until commit
- Ready – indicate if the instruction has completed execution

Steps in Speculative Execution

- Issue (or dispatch)
  - Get instruction from the instruction queue
  - In-order issue if available RS AND ROB slot; otherwise, stall
  - Send operands to RS if they are in register or ROB
  - Update Tomasulo DS and ROB
  - The ROB no. allocated for the result is sent to RS, so that the number can be used to tag the result when it is placed on CDB
- Execute
  - RS waits grabs results off the CDB if necessary
  - When all operands are there execution happens
- Write Result
  - Result posted to ROB via the CDB
  - Waiting reservation stations can grab it as well

Steps in Speculative Execution (Cont.)

- Commit (or graduate) – instruction reaches the ROB head
  - Normal commit – when instruction reaches the ROB head and its result is present in the buffer
    - Update the register and remove the instruction from ROB
  - Store – Update memory and remove the instruction from ROB
  - Branch with incorrect prediction – wrong speculation
    - Flush ROB and the related FP OP queue (RS)
    - Restart at the correct successor of the branch
    - Remove the instruction from ROB
  - Branch with correct prediction – finish the branch
    - Remove the instruction from ROB

Example (p. 229)

- The same example as Tomasulo without speculation. Show the status tables when MUL.D is ready to go to commit
  - 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
- Modified status tables
  - Qj and Qk fields, and register status fields use ROB (instead of RS)
  - Add Dest field to RS (ROB to put the operation result)

Example

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	no	MUL.D	Mem[45 + Regs[R3]]	Regs[F4]			#3	
Mult2	yes	DIV.D		Mem[34 + Regs[R2]]	#3		#5	

Reorder buffer

Entry	Busy	Instruction		State	Destination	Value
1	no	L.D	F6,34(R2)	Commit	F6	Mem[34 + Regs[R2]]
2	no	L.D	F2,45(R3)	Commit	F2	Mem[45 + Regs[R3]]
3	yes	MUL.D	F0,F2,F4	Write result	F0	#2 × Regs[F4]
4	yes	SUB.D	F8,F6,F2	Write result	F8	#1 – #2
5	yes	DIV.D	F10,F0,F6	Execute	F10	
6	yes	ADD.D	F6,F8,F2	Write result	F6	#4 + #2

FP register status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder #	3						6		4	5
Busy	yes	no	no	no	no	no	yes	. . .	yes	yes

Example Result

- Tomasulo without speculation
  - SUB.D and ADD.D have completed (clock cycle 16)
- Tomasulo with speculation
  - No instruction after the earliest uncompleted instruction (MUL.D) is allowed to complete
  - In-order commit
- Implication – ROB with in-order instruction commit provides precise exceptions
  - Precise exceptions – exceptions are handled in the instruction order

Speculative Execution Example

- if (A == 0) A = B; else A = A +4;

Old Code			Speculative Code		
LW	R1, 0(R3)		LW	R1, 0(R3)	
BNEZ	R1, L1		LW	R14, 0(R2)	/* spec. */
LW	R1, 0(R2)		BNEZ	R1, L3	
J	L2		ADD	R14, R1, 4	
L1: ADD	R1, R1, 4	L3:	SW	0(R3), R14	
SW	0(R3), R1				

- The load is executed speculatively, by moving it before the branch => need to handle exceptions.
- Extra register R14 is used to rename R1

Hardware-based Speculation

- Often speculative execution is combined with dynamic scheduling
- Tomasulo: separate *speculative* bypassing of results from real bypassing of results
  - When instruction no longer speculative, write results (*instruction commit*)
  - execute out-of-order but commit (write results) in order

Tomasulo's scheme offers 2 major advantages

the distribution of the hazard detection logic

- distributed reservation stations and the CDB
- If multiple instructions waiting on single result, & each instruction has other operand, then instructions can be released simultaneously by broadcast on CDB
- If a centralized register file were used, the units would have to read their results from the registers when register buses are available.

the elimination of stalls for WAW and WAR hazards

What about interrupts?

- We want the interrupt to happen as if the instructions would have been executed in order: precise interrupts
- State of machine looks as if no instruction beyond faulting instructions has issued
- Tomasulo had: in-order issue, out-of-order execution, and out-of-order completion

- Need to “fix” the out-of-order completion aspect so that we can find precise breakpoint in instruction stream.

Relationship between precise interrupts and speculation:

- Speculation: guess the outcome of the branches and execute as if our guesses were correct.
- Branch prediction is important:
  - Need to "take our best shot" at predicting branch direction.
- If we speculate and are wrong, need to back up and restart execution to the point at which we predicted incorrectly:
  - This is exactly same as precise exceptions!
- Technique for both precise interrupts/exceptions and speculation: *in-order completion or commit*

Hardware-based Speculation

- Need HW buffer for results of uncommitted instructions: *reorder buffer*
  - Reorder buffer can be operand source, if value not yet committed
  - Once operand commits, result is found in register file
  - 3 fields: instr. type, destination, value
  - Use reorder buffer number instead of reservation station to rename
  - Instructions commit in order
  - As a result, its easy to undo speculated instructions on mispredicted branches or on exceptions

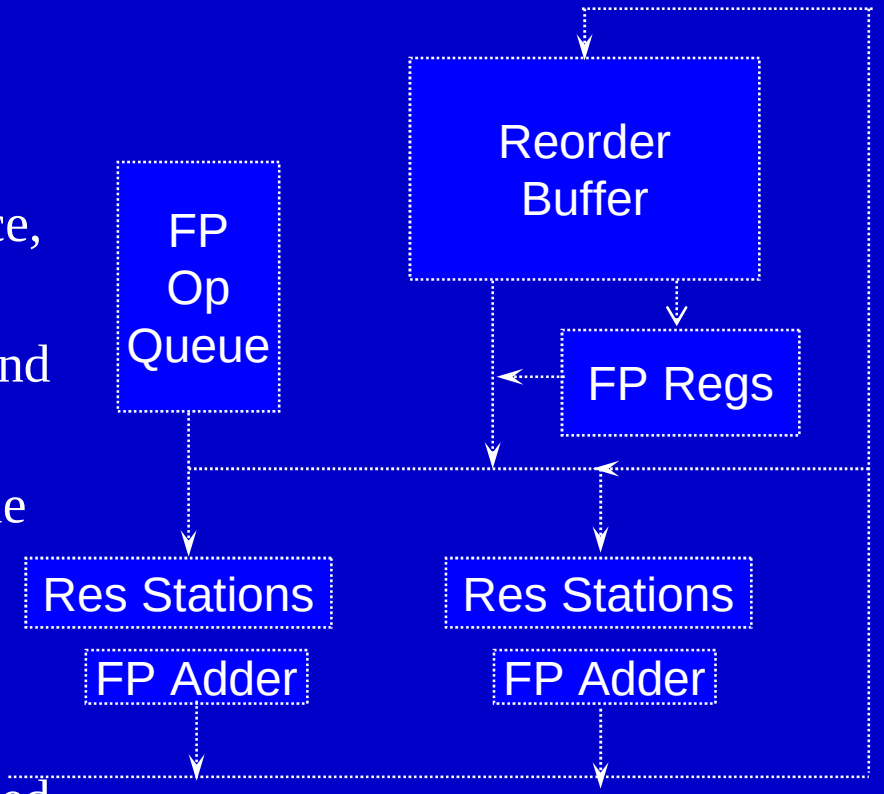


Figure 4.34, page 311

Hardware-based Speculation

- Hardware-based speculation offers many advantages
  - Allows memory references to be disambiguated
  - Can incorporate hardware-based branch prediction
  - Maintains a precise exception model even for speculative instructions, since results don't commit early
  - Does not require additional bookkeeping code
  - Does not depend on a good compiler
- Its main disadvantage is that it has substantial hardware requirements
- Implemented in the PowerPc 620, MIPS R10000, Intel P6, and AMD K5

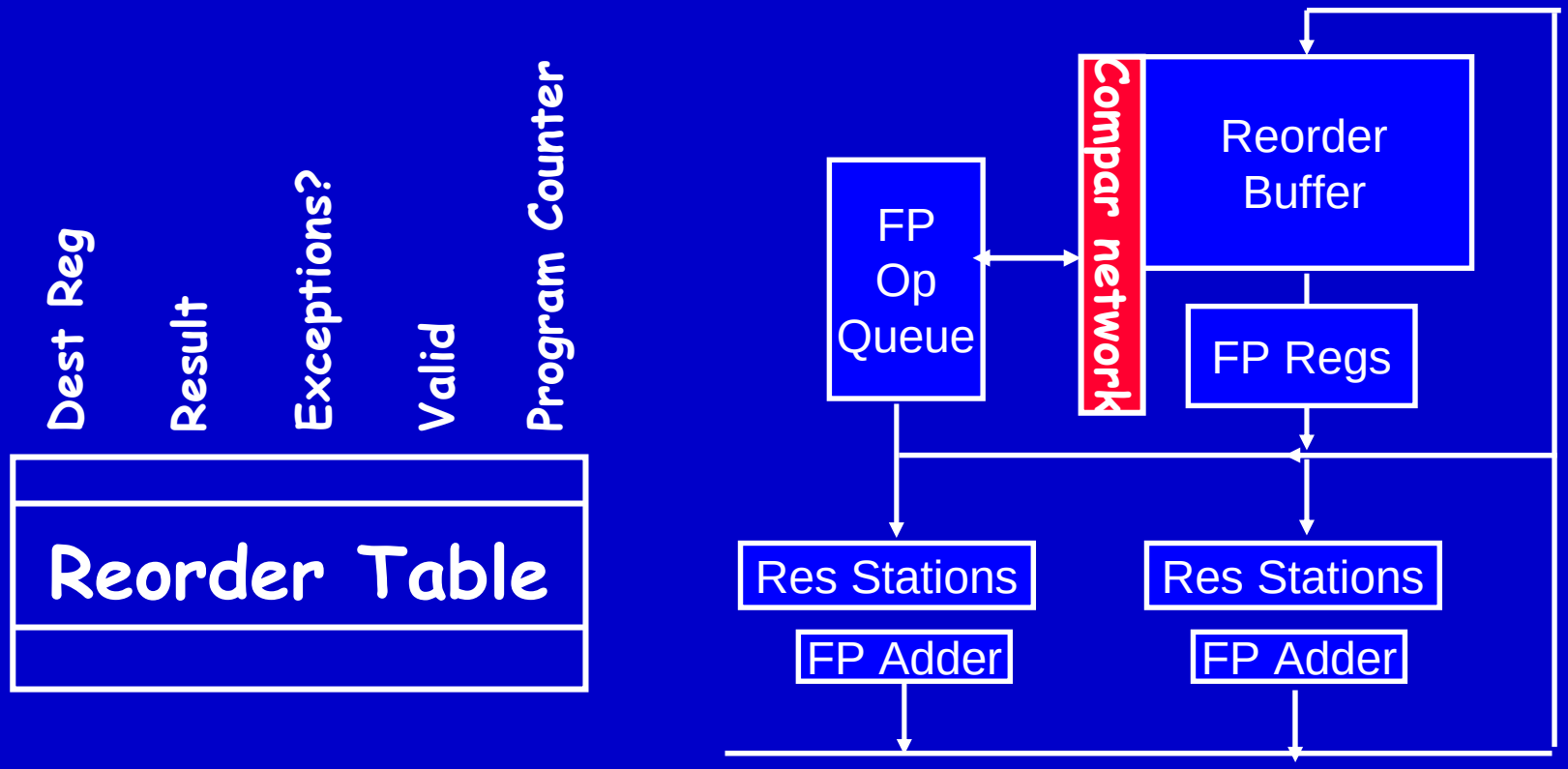
Assumptions for Ideal Processor

Initial HW Model here is an ideal processor

1. *Register renaming*–infinite virtual registers and all WAW & WAR hazards are avoided
2. *Branch prediction*–perfect; no mispredictions
3. *Jump prediction*–all jumps perfectly predicted => machine with perfect speculation & an unbounded buffer of instructions available (predicts address)
4. *Memory-address alias analysis*–addresses are known & a store can be moved before a load provided addresses not equal

1 cycle latency for all instructions

What are the hardware complexities with reorder buffer (ROB)?



- How do you find the latest version of a register?
  - (As specified by Smith paper) need associative comparison network
  - Could use future file or just use the register result status buffer to track which specific reorder buffer has received the value
- Need as many ports on ROB as register file

Loop Example (p. 231)

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

- Assume we have issued all the instructions in the loop twice
- Assume L.D and MUL.D from the first iteration have committed and all others have completed execution

Reorder buffer

Entry	Busy	Instruction	State	Destination	Value
1	no	L.D F0,0(R1)	Commit	F0	Mem[0 + Regs[R1]]
2	no	MUL.D F4,F0,F2	Commit	F4	#1 × Regs[F2]
3	yes	S.D F4,0(R1)	Write result	0 + Regs[R1]	#2
4	yes	DADDIU R1,R1,#-8	Write result	R1	Regs[R1] – 8
5	yes	BNE R1,R2,Loop	Write result		
6	yes	L.D F0,0(R1)	Write result	F0	Mem[#4]
7	yes	MUL.D F4,F0,F2	Write result	F4	#6 × Regs[F2]
8	yes	S.D F4,0(R1)	Write result	0 + #4	#7
9	yes	DADDIU R1,R1,#-8	Write result	R1	#4 – 8
10	yes	BNE R1,R2,Loop	Write result		

FP register status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8
Reorder #	6				7				
Busy	yes	no	no	no	yes	no	no	. . .	no

Loop Example Observation

- Suppose the first BNE is not taken → flush ROB and begins fetch instructions from the other path

Other Issues

- Performance is more sensitive to branch-prediction
  - Impact of a mis-prediction will be higher
  - Prediction accuracy, mis-prediction detection, and mis-prediction recovery increase in importance
- Precise exception
  - Handled by not recognizing the exception until it is ready to commit
  - If a speculation instruction raises an exception, the exception is recorded in ROB
    - Mis-prediction branch → exception are flushed as well
    - If the instruction reaches the ROB head → take the exception

Multiple Issue with Speculation

- Process multiple instructions per clock, assigning RS and ROB to the instructions
- To maintain throughput of greater than one instruction per cycle, must handle multiple instruction commits per clock
- Speculation helps significantly when a branch is a key potential performance limitation
- Speculation can be advantageous when there are data-dependent branches, which otherwise would limit performance
  - Depend on accurate branch prediction → incorrect speculation will typically harm performance

Example (p. 235)

- Assume separate integer FUs for ALU operations, effective address calculation, and branch condition evaluation
- Assume up to 2 instruction of any type can commit per clock
- Loop:

```
LD      R2, 0(R1)
DADDIU  R2, R2, #1
SD      R2, 0(R1)
DADDIU  R1, R1, #4
BNE     R2, R3, LOOP
```

No Speculation

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#4	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#4	8	14		15	Wait for BNE
3	BNZ R2,R3,LOOP	9	19			Wait for DADDIU

Speculation

Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LW
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LW
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#4	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#4	8	9		10	14	Executes earlier
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU

Example Result

- Without speculation
  - L.D following BNE cannot start execution earlier → wait until branch outcome is determined
  - Completion rate is falling behind the issue rate rapidly, stall when a few more iterations are issued
- With speculation
  - L.D following BNE can start execution early because it is speculative

END