RISC and CISC

CO 2206 Computer Organization

Topics

- RISC vs CISC
- Pipelining
- RISC Implementation
  - Modifications
  - ISA
  - Organization
- Data Hazard
- Data Hazard Solutions
- Control Hazard
- Control Hazard Solutions

RISC vs CISC: RISC

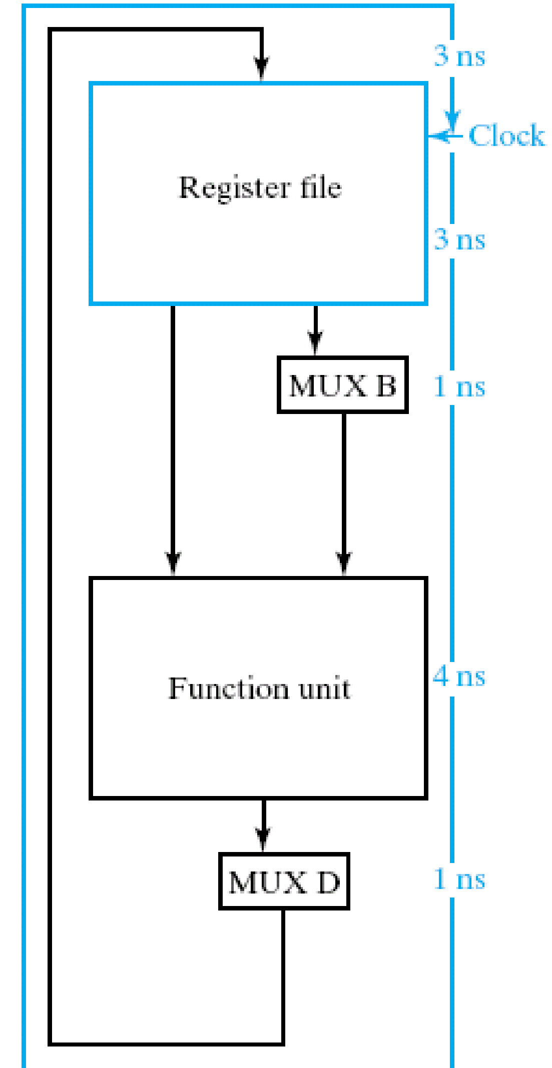
- Architecture paradigms - not so contrast nowadays
- RISC = Reduced Instruction Set Computer
  - Minimal instruction set
    - direct decoding of instruction set (without microcodes)
  - Relatively few addressing modes
    - memory access limited to load and store instructions
    - all operations are done within the registers of the CPU
  - Reduced hardware complexity for performance
  - Require more programming effort
    - more instructions to perform a task, hence larger program
    - use more storage resource
  - Usually fixed instruction length and time of execution
    - suitable for pipelining
    - single-cycle instruction execution
  - E.g. PowerPC, MIPS, SPARC

RISC vs CISC: CISC

- CISC = Complex Instruction Set Computer
  - Has many and complex instructions – usually more than 200 instructions
    - many microcodes to implement each instruction
    - some instructions that perform specialized tasks and are used infrequently
  - A large variety of addressing modes - typically 5 to 20 different modes
    - instructions that manipulate operands in memory
  - Increased hardware complexity, hence increased hardware delay
  - Less programming effort
    - smaller program, less storage required
  - Variable instruction length and time of execution
    - not easy for pipelining
  - E.g. Motorola 68000, x86 based

Single-Cycle CPU Datapath

- A max. of 12ns is required to perform a single microop
  - max. rate at which the microoperations can be performed is 83.3MHz
  - this is max. frequency at which clock can be operated since 12ns is the smallest clock period that will allow completion with certainty



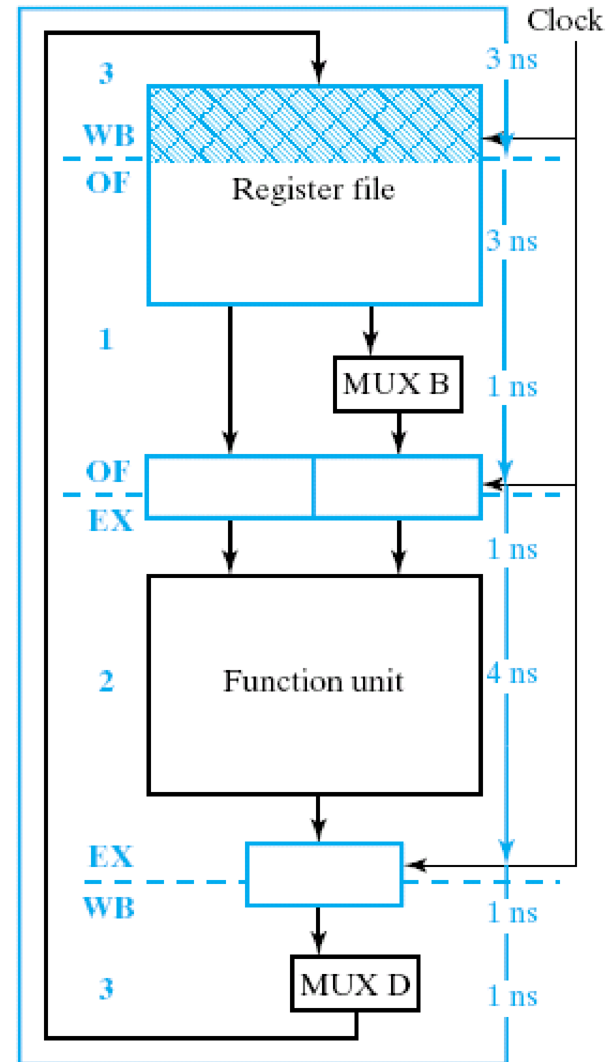
(a) Conventional

Pipelining

- In a single-cycle computer, the rate of execution of instructions equals to clock frequency
- Possible to reduce clock period and increase clock frequency through *pipelining*
  - the resulting datapath is known as *pipelined datapath*
- Pipelining
  - an implementation technique in which multiple instructions are overlapped in execution
  - a key to making processors fast
  - exploits parallelism among instructions in a sequential instruction stream

Pipelined Datapath: Example

- Original datapath is divided into 3 independent parts or stages
- Place registers wherever there are dividing lines between stages, otherwise the information is lost as the next instruction enters that pipeline stage
- A separate pipeline register for write-back stage is redundant



(b) Pipelined

Pipelined Datapath: Stages

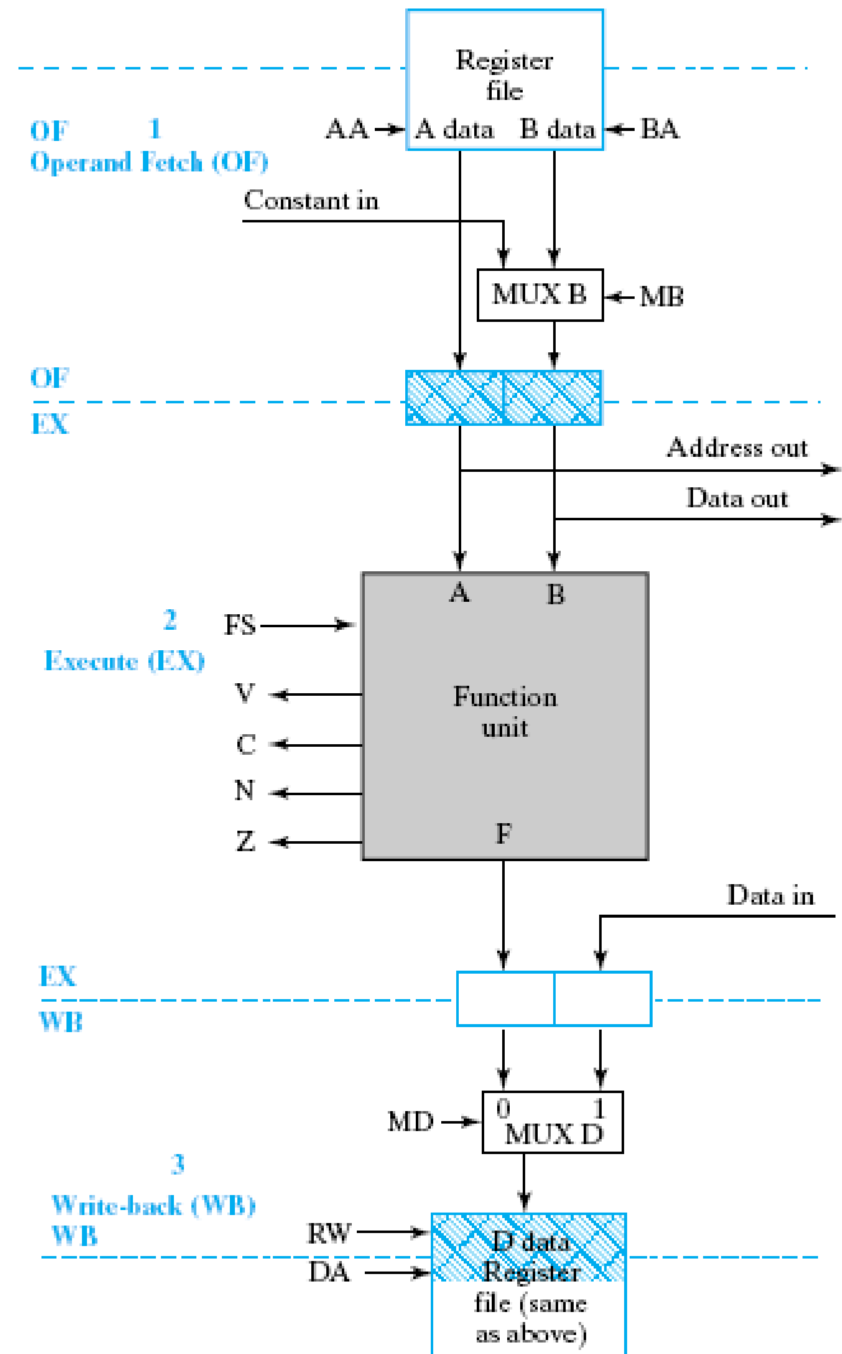
- Stage 1
  - Operand Fetch (OF)
- Stage 2
  - Execution (EX)
- Stage 3
  - Write-back (WB)

Pipelined Datapath: Performance

- Length of time or the clock period in each stages is the same for all stages
- Length of time required to process an instruction is called *latency or execution time*
- Pipelining improves instruction *throughput* rather than individual instruction execution time
- Min. clock period = max. stage delay
  - E.g. 5ns
    - max. clock frequency = 200MHz
    - 2.4 times better than non-pipelined datapath (83.3)
    - time to fill pipeline initially is negligible

Pipelined Datapath

- Register file is shown twice
  - In OF stage, it is read
  - In WB stage, it is written
- Pipeline registers store the operand(s) for use in next stage during the next clock cycle



Microoperations through Pipeline

- Each horizontal position represents a clock cycle
- Time required for pipeline execution = 9x5 = 45ns, compared to 7x12 = 84ns

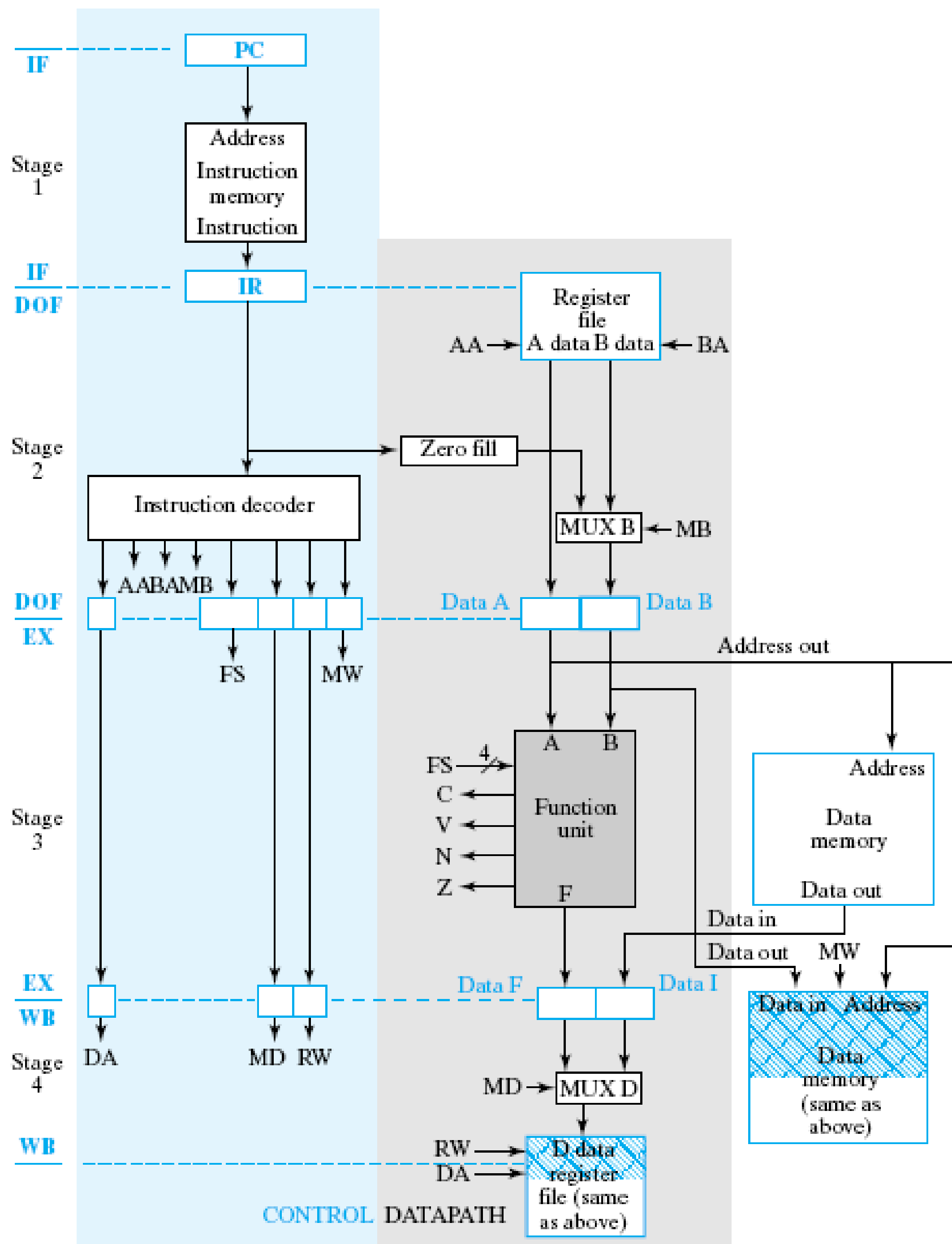
Clock cycle

Microoperation		1	2	3	4	5	6	7	8	9
$R1 \leftarrow R2 - R3$	1	OF	EX	WB						
$R4 \leftarrow$ sl $R6$	2		OF	EX	WB					
$R7 \leftarrow R7 + 1$	3			OF	EX	WB				
$R1 \leftarrow R0 + 2$	4				OF	EX	WB			
Data out $\leftarrow R3$	5					OF	EX	WB		
$R4 \leftarrow$ Data in	6						OF	EX	WB	
$R5 \leftarrow 0$	7							OF	EX	WB

Filling and Emptying

- *Pipeline filling*
  - first 2 clock cycle when not all pipeline stages are active
- *Pipeline emptying*
  - last 2 clock cycle when not all pipeline stages are active

Pipeline Control



Pipeline Control: Stages - 1

- Instruction fetch (IF)
  - added stage
  - includes PC and instruction memory
- Decode and operand fetch (DOF)
  - decoding of *instruction register* (IR) into control signals
  - AA, BA and MB are used here, the rest passed on (to execution and writeback stage)

Pipeline Control: Stages - 2

- Execution (EX)
  - ALU, shift or memory operation
  - FS, MW used
  - read part of data memory considered
    - for a memory read, value of memory word addressed is read from *Data out* of data memory
- Write-back (WB)
  - write part of data memory considered
    - memory write may occur (note register is sequential, hence MW available a clock pulse earlier)
  - DA, MD, RW are used

Pipeline Control: Performance

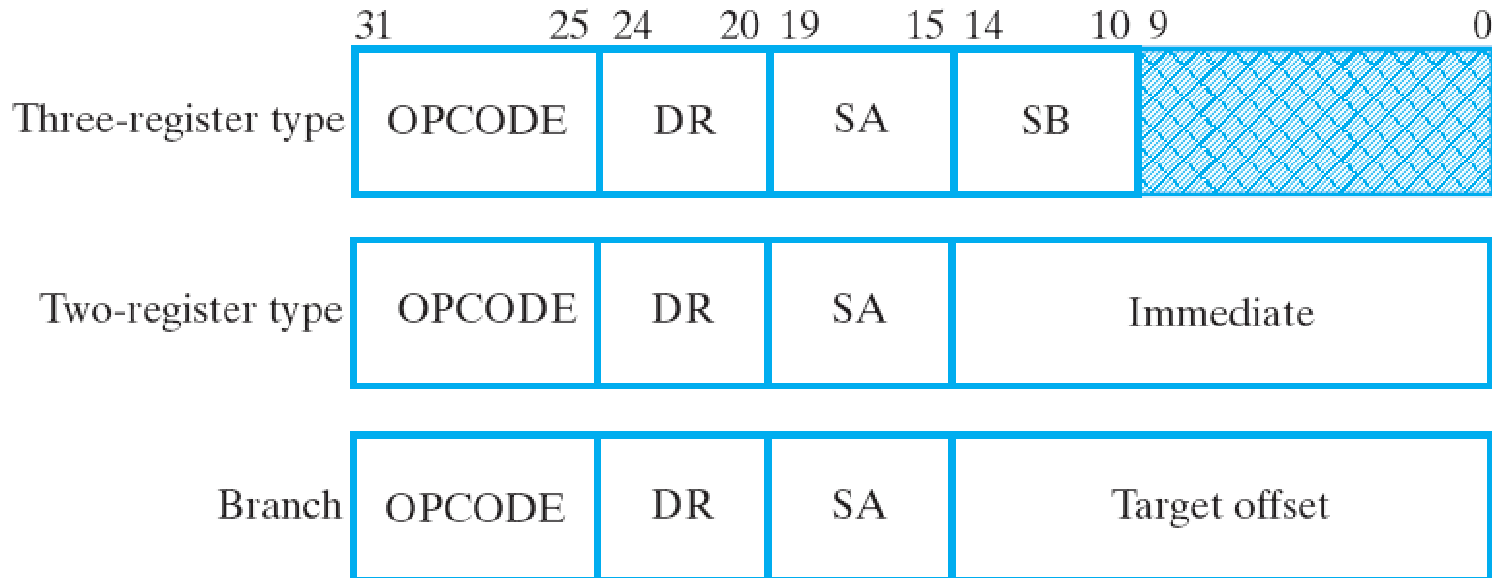
- Stage delays are balanced
- Delays per stage no more than 5ns
  - Max. clock frequency = 200MHz
  - However, each instruction takes 4x5 = 20ns to execute
    - Compared to 17ns in single-cycle computer
- Pipelining improves *instruction throughput* but not *latency* or *execution time*

Reduced Instruction Set Computer

- RISC pipelining
  - a pipelined datapath with a hardwired pipelined control unit
  - analogous to single-cycle computer
- In order to implement RISC, modifications are made to register file, function unit, bus structure
  - Modification represent effects of
    - longer instruction word length
    - increased register file capacity
    - to include multiple position shifts among elementary operations
    - also in response to control and data hazards associated with pipelined designs

RISC ISA: Instruction Format - 1

- Larger register (32-bit) to hold addresses within single instruction (single-cycle computer)
  - 5-bit register fields SA, SB, DR
  - 7-bit OPCODE, max. 128 operations
  - Immediate and Target offset, 15-bit constant



RISC ISA: Instruction Format - 2

- *Target address* – effective address
  - Formed by target offset + content of PC
  - Branch taken if source register (specified in SA) contains 0
  - Destination register (specified in DR) is used to store the return address for the branch (no stack)
- Rightmost 5 bits of 15-bit constant used as shift amount SH for multiple bit shift
- Leftmost 17 bits filled to form a 32-bit operand
  - Zero fill for logical operations
  - Sign extension – sign bit 14 is copied to 17 bits

RISC Instruction Set

RISC Instruction Operations

Operation	Symbolic Notation	Opcode	Action
No Operation	NOP	0000000	None
Move A	MOVA	1000000	$R[DR] \leftarrow R[SA]$
Add	ADD	0000010	$R[DR] \leftarrow R[SA] + R[SB]$
Subtract	SUB	0000101	$R[DR] \leftarrow R[SA] + \overline{R[SB]} + 1$
AND	AND	0001000	$R[DR] \leftarrow R[SA] \wedge R[SB]$
OR	OR	0001001	$R[DR] \leftarrow R[SA] \vee R[SB]$
Exclusive-OR	XOR	0001010	$R[DR] \leftarrow R[SA] \oplus \mathrm{R[SB]}$
Complement	NOT	0001011	$R[DR] \leftarrow \overline{R[SA]}$
Add Immediate	ADI	0100010	$R[DR] \leftarrow R[SA] + \text{se } IM$
Subtract Immediate	SBI	0100101	$R[DR] \leftarrow R[SA] + \overline{\text{se } IM} + 1$
AND Immediate	ANI	0101000	$R[DR] \leftarrow R[SA] \wedge (0 \,\|\, IM)$
OR Immediate	ORI	0101001	$R[DR] \leftarrow R[SA] \vee (0 \,\|\, IM)$
Exclusive-OR Immediate	XRI	0101010	$R[DR] \leftarrow R[SA] \oplus (0 \,\|\, IM)$
Add Immediate Unsigned	AIU	1000010	$R[DR] \leftarrow R[SA] + (0 \,\|\, IM)$
Subtract Immediate Unsigned	SIU	1000101	$R[DR] \leftarrow R[SA] + \overline{(0 \,\|\, IM)} + 1$
Move B	MOVB	0001100	$R[DR] \leftarrow R[SB]$
Logical Right Shift by SH Bits	LSR	0001101	$R[DR] \leftarrow \text{lsr } R[SA] \text{ by } SH$
Logical Left Shift by SH Bits	LSL	0001110	$R[DR] \leftarrow \text{lsl } R[SA] \text{ by } SH$
Load	LD	0010000	$R[DR] \leftarrow M[R[SA]]$
Store	ST	0100000	$M[R[SA]] \leftarrow R[SB]$
Jump Register	JMR	1110000	$PC \leftarrow R[SA]$
Set if Less Than	SLT	1100101	If $R[SA] < R[SB]$, then $R[DR] + 1$
Branch on Zero	BZ	1100000	If $R[SA] = 0$, then $PC \leftarrow PC + 1 + \text{se } IM$
Branch on Nonzero	BNZ	1010000	If $R[SA] \neq 0$, then $PC \leftarrow PC + 1 + \text{se } IM$
Jump	JMP	1101000	$PC \leftarrow PC + 1 + \text{se } IM$
Jump and Link	JML	0110000	$PC \leftarrow PC + 1 + \text{se } IM, R[DR] \leftarrow PC + 1$

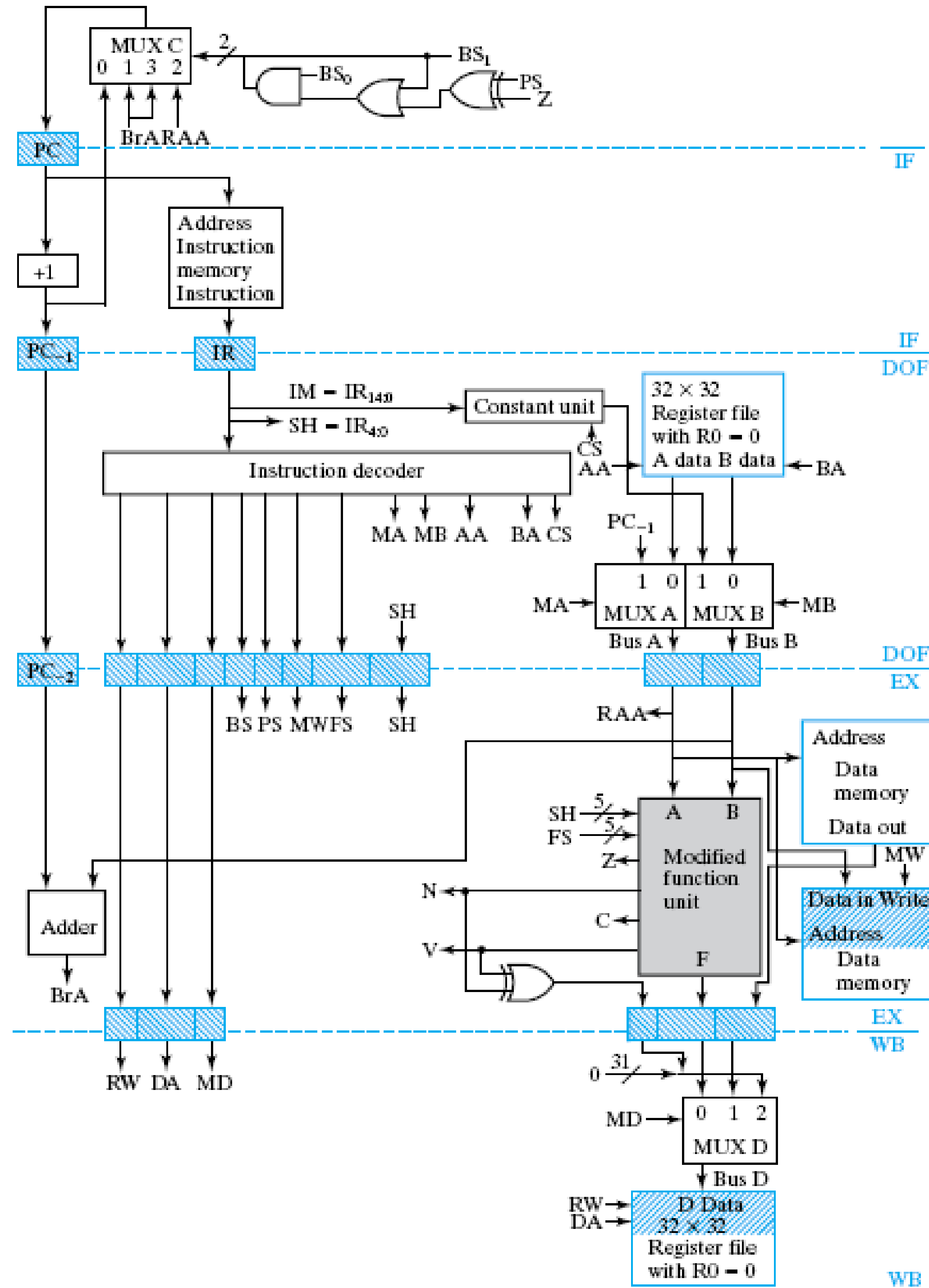
RISC Instruction Set

- Refer to table in previous slide
- All operations are elementary
  - can be described by single register transfer statement
- Only Load and Store operations access memory
  - all other operations only access registers
  - minimize memory access to improve performance
- JML provides a mechanism for implementing procedures.
  - return from a called procedure can use JMR with SA equal to DR
- No status word register
  - use BZ, BNZ and SLT for testing, comparison

RISC Addressing Modes

- Four addressing modes:
  - Register
    - three register type
  - Register Indirect
    - only Load & Store access memory
  - Immediate
    - two register type
  - Relative
    - branch type (branch and jump instructions)
- Other addressing modes implemented by multiple RISC instructions

RISC Organization



RISC Datapath Organization

- Only modifications highlighted
- Register file:
  - 32 32-bit registers
    - to fit operands/addresses for most operations
  - Special reading register R0
    - gives 0 when read, discard value when written
  - Register file is no longer edge-triggered. Uses latch instead and special timing signals are provided that permit the register file to be written in the first half and to be read in the last half of the cycle.
    - in particular, in the 2nd half of the clock cycle , it is possible to read data written into the register file during the first half of the same clock cycle.
    - this is called a read-after-write register file to avoid added complexity for handling hazards and reduce cost of register file

RISC Datapath Organization

- Function Unit:
  - ALU is 32-bit
  - replacement of 1-bit position shifter with barrel shifter
    - SH is 5-bit input – 1 to 31-bit position shift
- Other parts:
  - constant unit – performs zero fill for CS=0 and sign extension for CS=1
  - MUXA – to provide a path for the updated PC, PC-1 to the register file for the implementation of Jump and Link (JML) instruction.
  - additional input to MUX D helps implement Set if Less Than (SLT) if R[SA] < R[SB] set R[DR] = 1

RISC Control Organization

- Modifications include:
  - modified instruction decoder to deal with new instructions
  - SH field added in IR
  - CS bit added to the instruction decoder
  - new control logics for PC
    - MUX C to select updated PC
    - Adder to compute target address

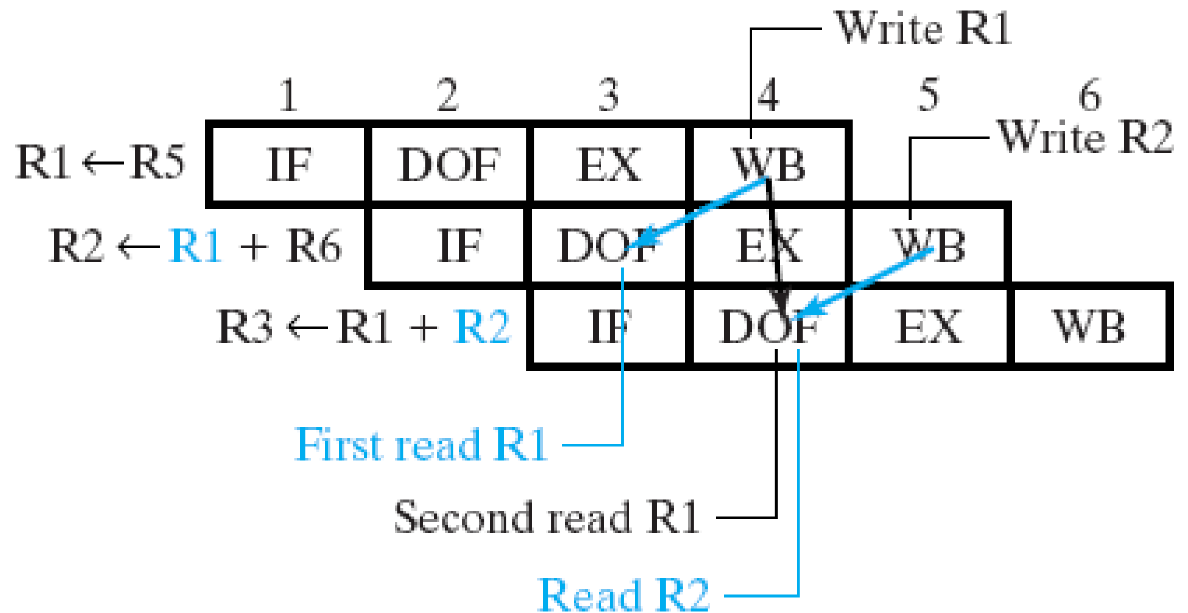
Data Hazards

- *Hazards* are timing problems that arise because
  - the execution of an operation in a pipeline is delayed by one or more clock cycles
- Hazard reduces throughput
- *Data hazards* occur
  - if a subsequent instruction tries to use the result of an operation as an operand before the result is available

Data Hazards: Illustration

- e.g.

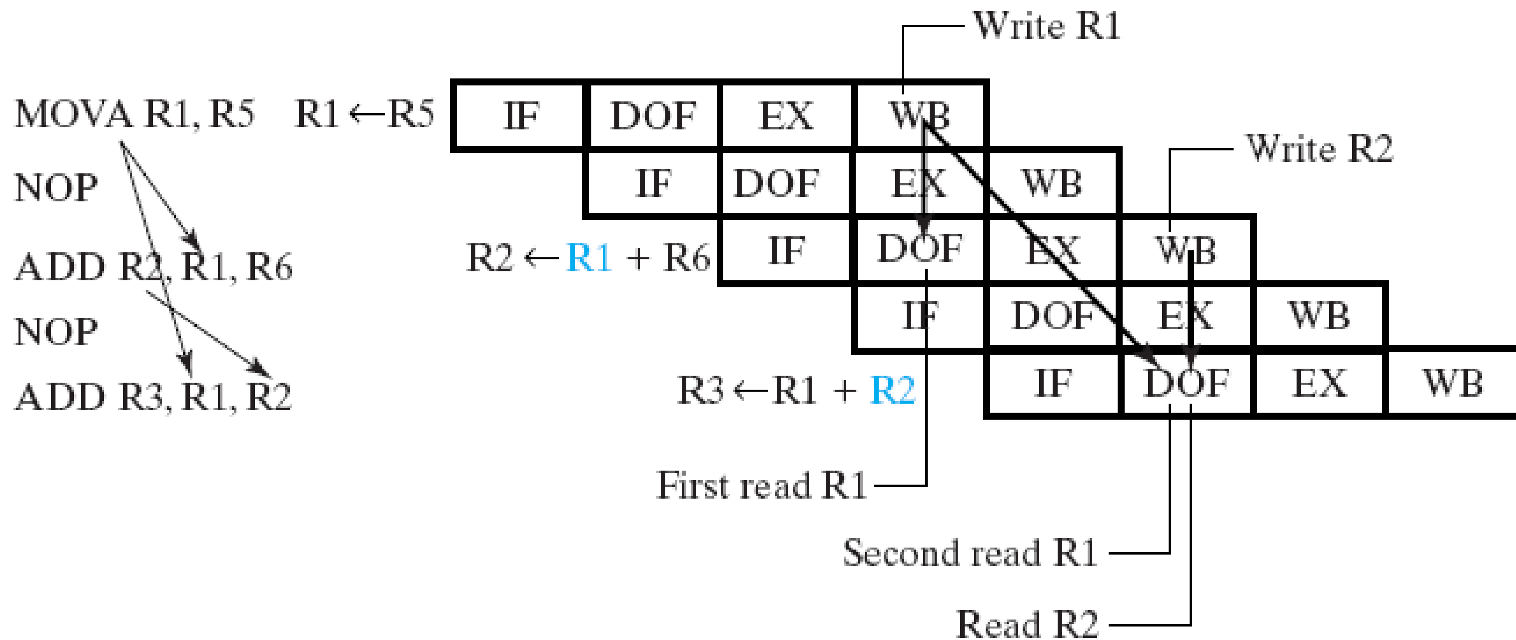
1	MOVA	R1, R5
2	ADD	R2, R1, R6
3	ADD	R3, R1, R2



2 data hazards

Data Hazards Solution: Software

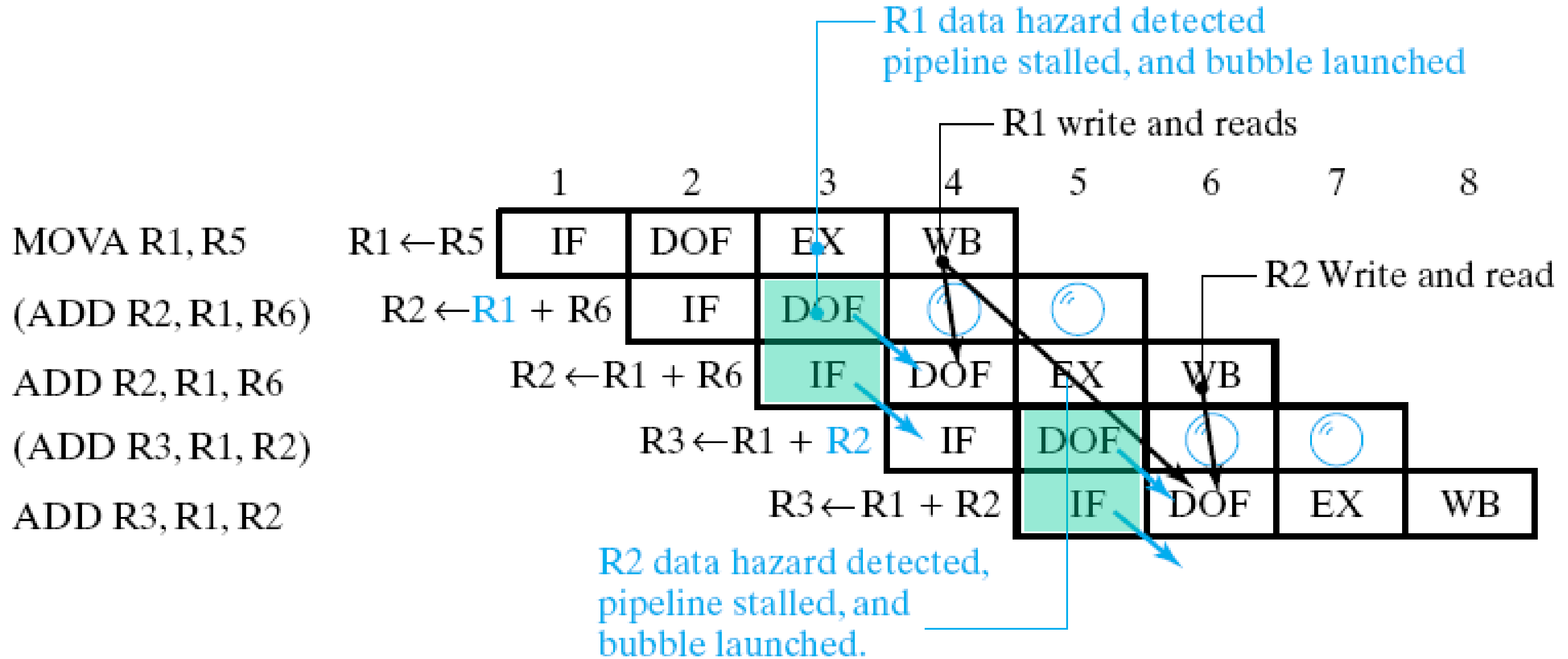
- Software solution: ***NOPs***
  - Have compiler or programmer generate machine code to delay instructions using *NOPs*
    - needs to have detailed info on how pipeline operates



Data Hazards Solution: Stalling

- Hardware solution 1: ***Stalling***
  - When operand is found at the DOF stage that has not been written back yet,
    - associated EX and WB are delayed by *stalling* the pipeline for 1 clock cycle
  - The pipeline is said to contain a *bubble* in subsequent clock cycles and stages

Stalling: Illustration



- Data hazard will occur if there is a destination register at EX that is to be written back and that is to be read at the current DOF

Stalling: Implementation - 1

- Conditions for stalling

$$HA = \overline{MA_{DOF}} \cdot (DA_{EX} = AA_{DOF}) \cdot RW_{EX} \cdot \sum_{i=0}^{4} (DA_{EX})_i$$

$$HB = \overline{MB_{DOF}} \cdot (DA_{EX} = BA_{DOF}) \cdot RW_{EX} \cdot \sum_{i=0}^{4} (DA_{EX})_i$$

$$DHS = HA + HB$$

- MA in DOF must be 0
  - A operand is coming from the register file
- AA in DOF equals DA in EX

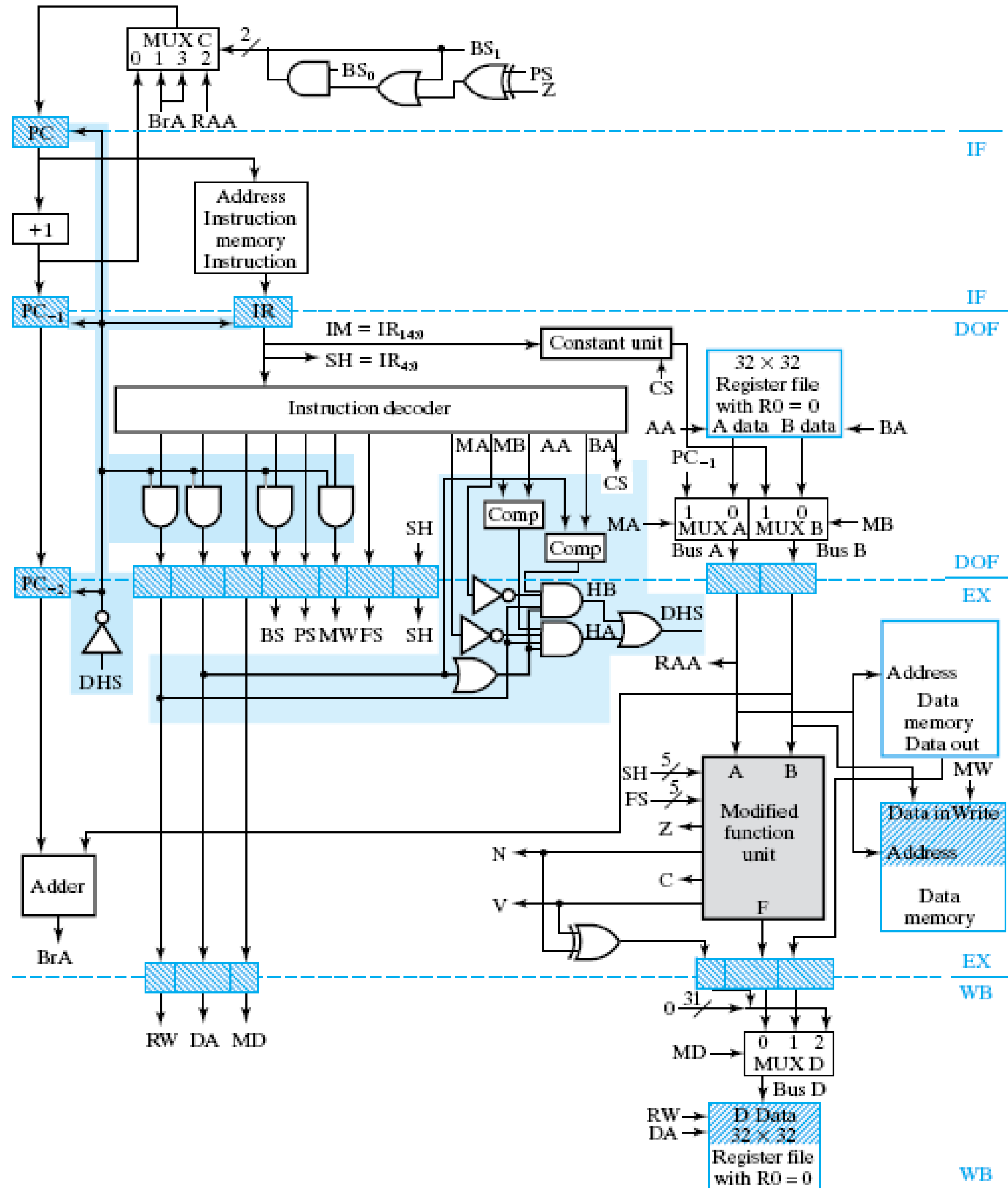
Stalling: Implementation - 2

- RW in EX is 1
  - DA in EX will definitely be written in WB
- OR (Σ) of all bits of DA is 1
  - Register to be written is not R0
- If either HA or HB is 1, data hazard
  - DHS = 1, stall is required
- Inverted DHS signal is used to initiate a bubble
  - IR, PC, PC_{-1}, PC_{-2} stopped from changing because their load signals become 0

Stalling: Implementation - 3

- Bubble produced by using AND gates to force RW and MW to 0
  - Force DA to become 0 to prevent stalling in the next and subsequent clock cycles because Σ will then return 0
- Thus, in clock cycle 3 when data hazard for R1 is detected
  - A bubble is launched into EX for ADD
  - At clock cycle 3, IF and DOF are stalled, thus info. is retained
  - At clock cycle 4, since DA_{EX}=0, no stall, so execution of stalled ADD proceeds
- Data hazard stall has the same throughput penalty as NOPs

Stalling Implementation



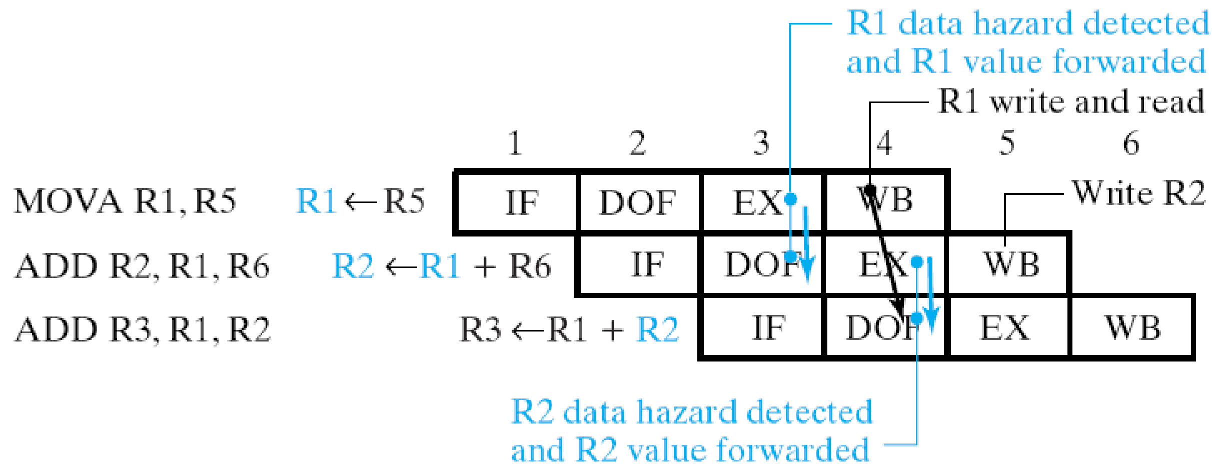
owh@ieee.org

Data Hazards Solution: Data Forwarding

- Hardware solution 2: *Data forwarding*
  - Does not have the penalty mentioned
  - Based on the fact that results are available in the pipeline before it is written back to the destination register
    - Thus, mux is needed to select data from register file or the pipeline
    - Logic is similar to stalling except now HA and HB are used

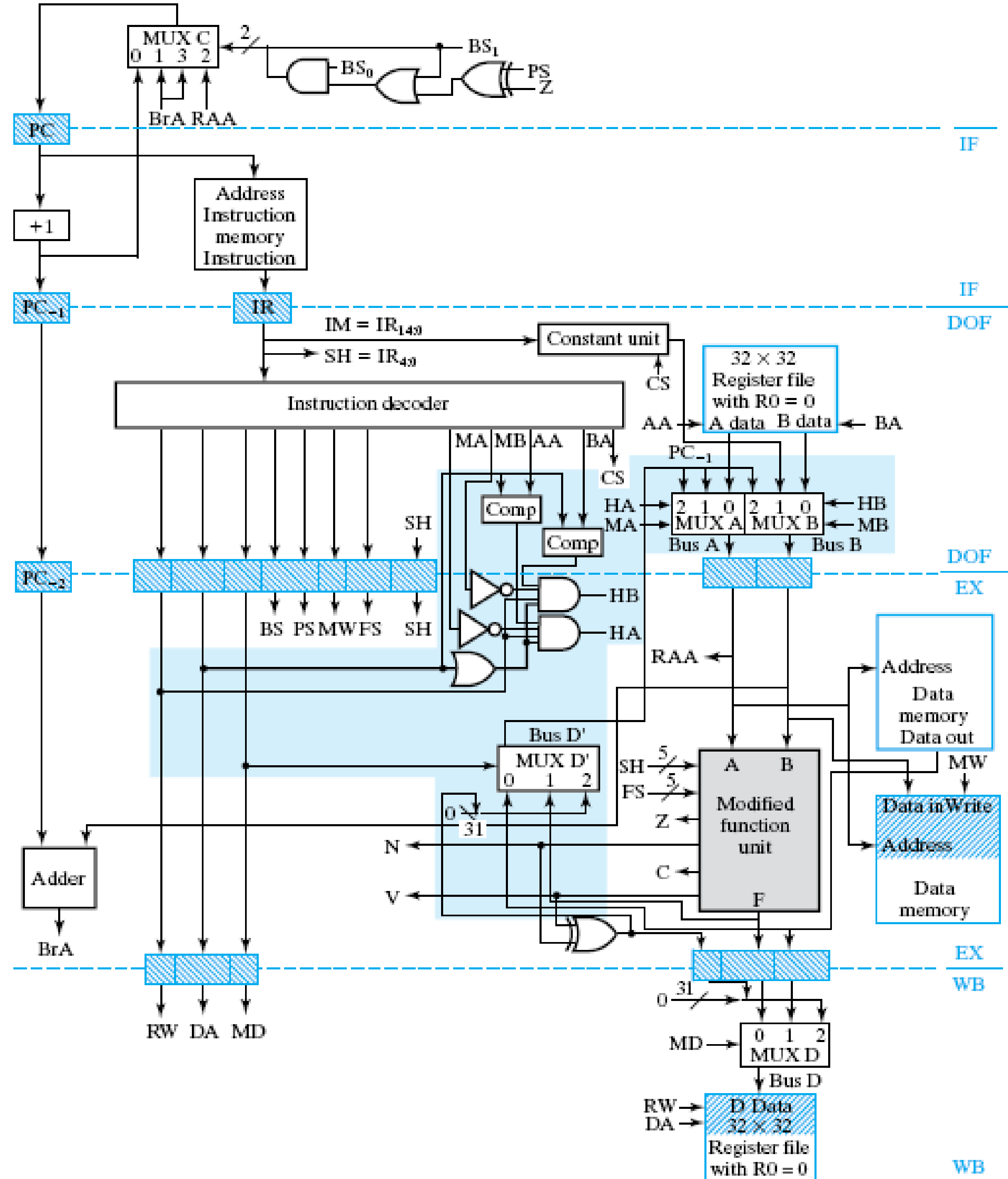
Data Forwarding: Illustration

– Data forwarding may adds delay, causing clock period to be somewhat longer



- Data hazards can also occur with memory access as in ST and LD instructions

Data Forwarding Implementation



owh@ieee.org

Control Hazard - 1

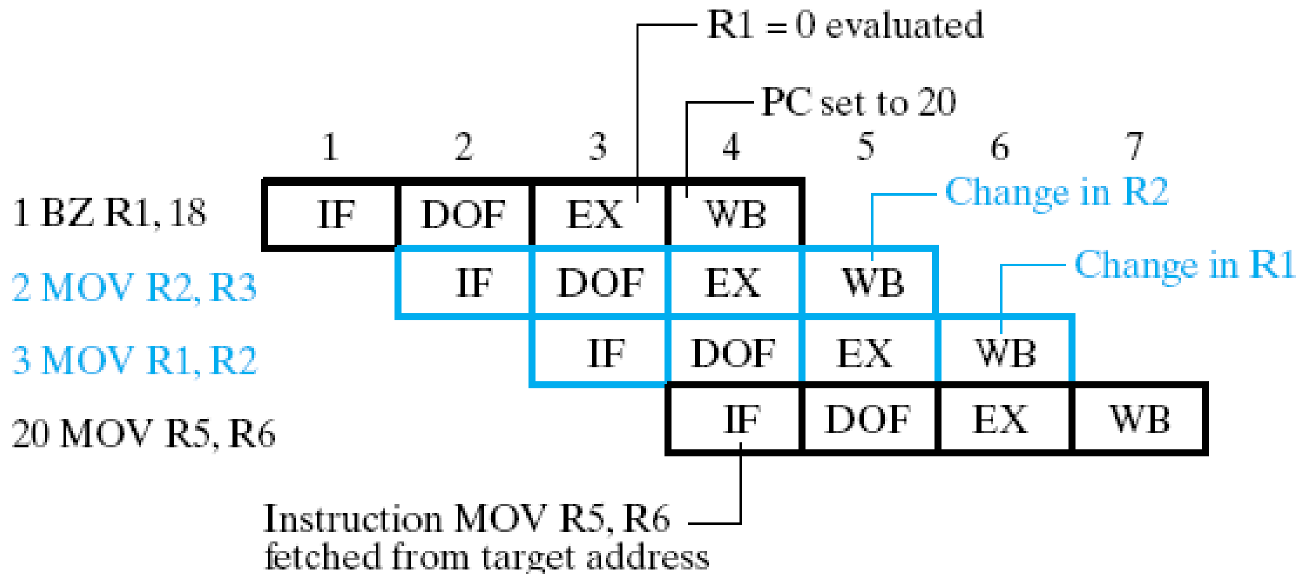
- Control hazards are associated with branches in the control flow of a program
- e.g.

```
1     BZ        R1, 18
2     MOVA      R2, R3
3     MOVA      R1, R2
4     MOVA      R4, R2
20    MOVA      R5, R6
```

  - If R1=0, branch to location 20 (addressing is PC relative)

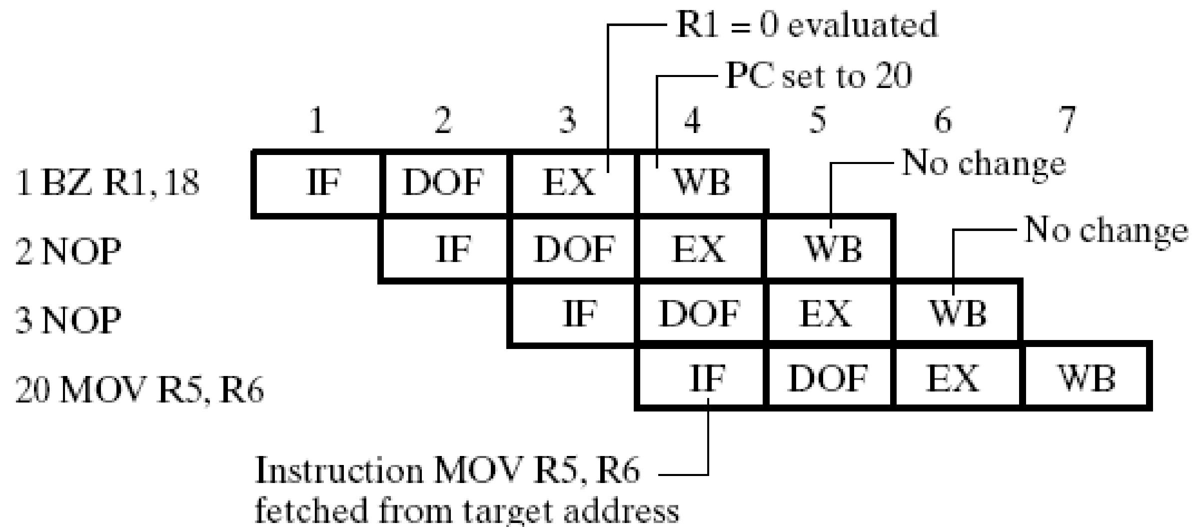
Control Hazard - 2

– But R1=0 is not detected until EX, so PC is set to 20 on the clock edge at end of clock cycle
– Thus, 2 instructions (not supposed to be executed) already in the pipeline in EX and DOF respectively



Control Hazard Solution: Delayed Branch

- *Delayed branch*:
  - 2 NOPs are inserted and performed regardless of whether branch is taken
  - increase processing by 2 clock cycles
  - wasted cycles can be avoided by rearranging the order of instructions

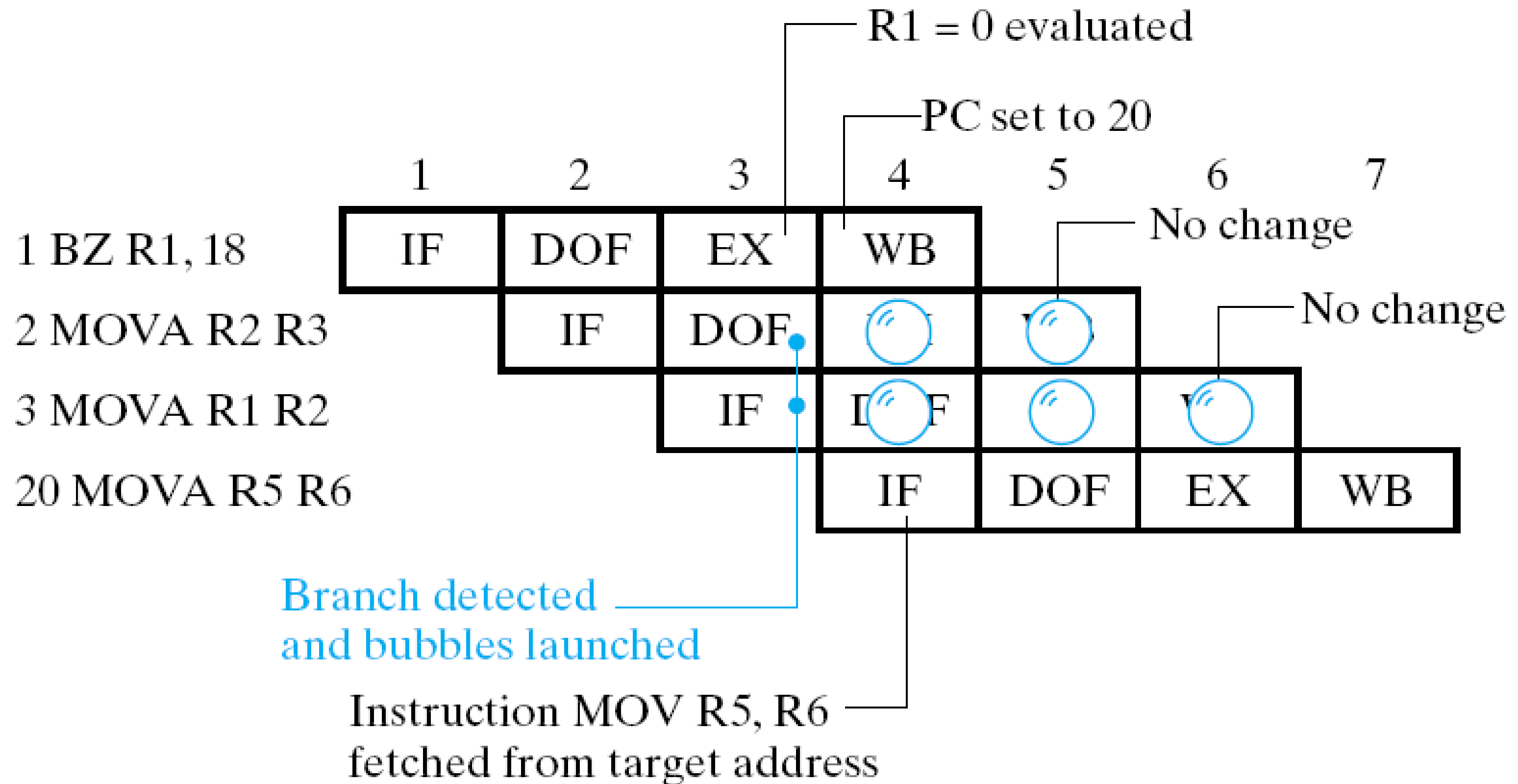


Control Hazard Solutions: Stalling

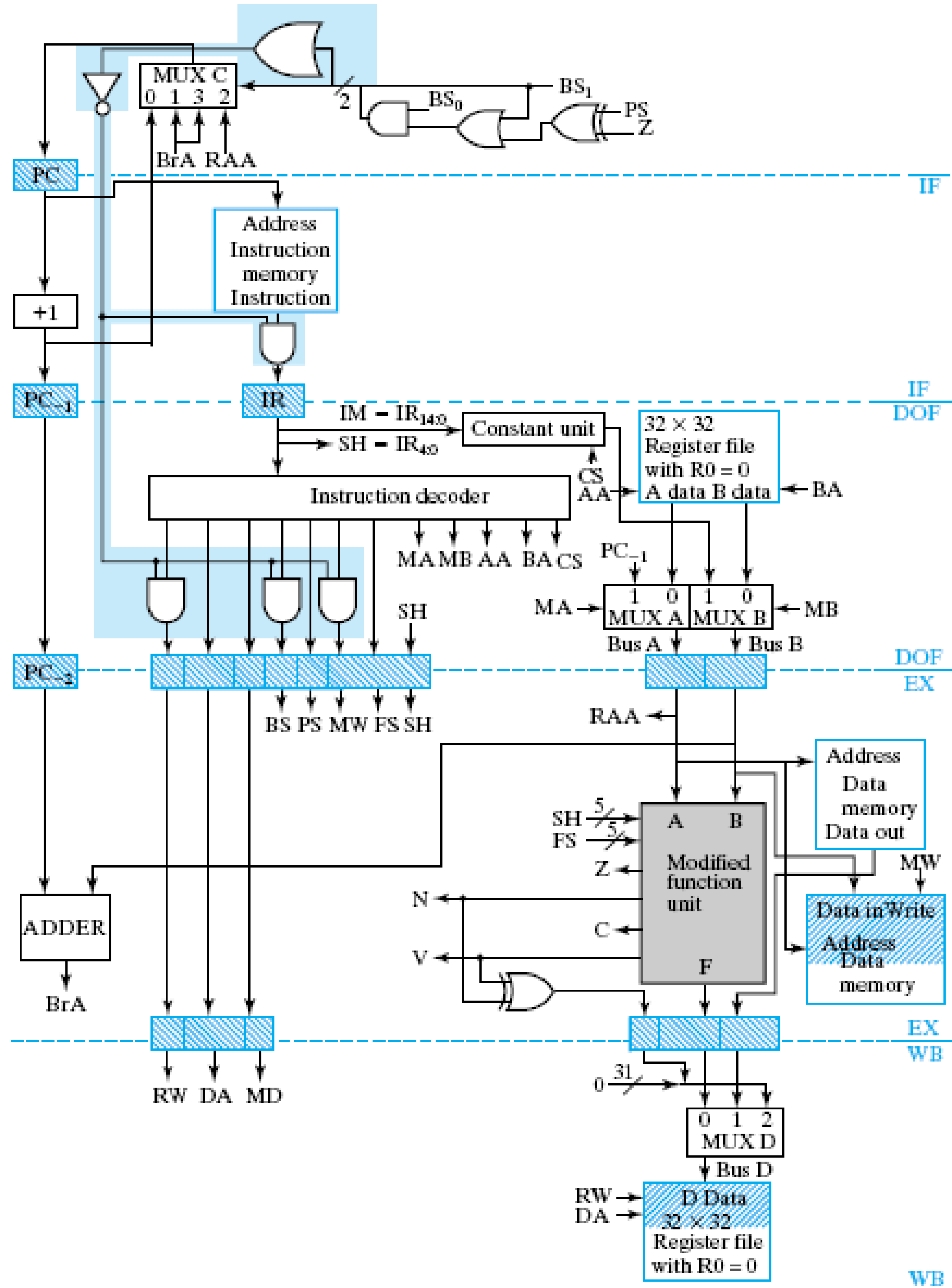
- *Branch hazard stall*:
  - Reduction in throughput
- *Branch prediction*:
  - Predicts the branch will never be taken
  - Fetch and decode next instruction in PC+1
  - If branch is not taken after all, no delay
  - If branch is taken, need to cancel instructions rendered invalid

Control Hazard Solution: Stalling Illustrated

- Cancellation is done by
  - Inserting bubbles into EX and WB



Branch Stalling Implementation



Branch Stalling Implementation

- Whether a branch is taken is determined by looking at selection input of MUX C
- If branch is taken
  - Cancel 1st instruction by:
    - Making RW, MW 0 to prevent writing of register file and data memory; and making BS 0 to prevent branch taking if it is a branch instruction
  - 2nd instruction is replaced with NOP by making IR all 0's (NOP OPCODE)
- Can also assume branch is always taken
  - Thus branch target address computed and used for fetching the target instruction

Summary

- RISC: simple ISA, simple hardware, considerable programming effort, effective for pipelining
- CISC: complex ISA, complex hardware, flexible to program
- Pipelining: CPU performance improvement technique
  - break datapath and control into sections – interfaced with registers
- RISC Implementation
  - Modifications: more registers
  - ISA: mostly deal with registers, only store/load for memory
  - Organization: pipelined
- Data Hazard: operand fetch stage requiring data modified in execute stage (before write back)
- Data Hazard Solutions: NOPs, Stalling, Data Forwarding
- Control Hazard: branch taken and unwanted instructions in pipeline
- Control Hazard Solutions: NOPS, Stalling