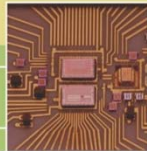


COMPUTER ARCHITECTURE

From Microprocessors



To Supercomputers



BEHROOZ PARHAMI

Part IV

Data Path and Control

Parts	Chapters
I. Background and Motivation	1. Combinational Digital Circuits 2. Digital Circuits with Memory 3. Computer System Technology 4. Computer Performance
II. Instruction-Set Architecture	5. Instructions and Addressing 6. Procedures and Data 7. Assembly Language Programs 8. Instruction-Set Variations
III. The Arithmetic/Logic Unit	9. Number Representation 10. Adders and Simple ALUs 11. Multipliers and Dividers 12. Floating-Point Arithmetic
IV. Data Path and Control	13. Instruction Execution Steps 14. Control Unit Synthesis 15. Pipelined Data Paths 16. Pipeline Performance Limits
V. Memory System Design	17. Main Memory Concepts 18. Cache Memory Organization 19. Mass Memory Concepts 20. Virtual Memory and Paging
VI. Input/Output and Interfacing	21. Input/Output Devices 22. Input/Output Programming 23. Buses, Links, and Interfacing 24. Context Switching and Interrupts
VII. Advanced Architectures	25. Road to Higher Performance 26. Vector and Array Processing 27. Shared-Memory Multiprocessing 28. Distributed Multicomputing

About This Presentation

This presentation is intended to support the use of the textbook *Computer Architecture: From Microprocessors to Supercomputers*, Oxford University Press, 2005, ISBN 0-19-515455-X. It is updated regularly by the author as part of his teaching of the upper-division course ECE 154, Introduction to Computer Architecture, at the University of California, Santa Barbara. Instructors can use these slides freely in classroom teaching and for other educational purposes. Any other use is strictly prohibited. © Behrooz Parhami


Edition	Released	Revised	Revised	Revised	Revised
First	July 2003	July 2004	July 2005	Mar. 2006	Feb. 2007
		Feb. 2008	Feb. 2009	Feb. 2011	Nov. 2014

A Few Words About Where We Are Headed

Performance = 1 / Execution time *simplified to* 1 / CPU execution time


CPU execution time = Instructions × CPI / (Clock rate)

Performance = Clock rate / (Instructions × CPI)





Try to achieve CPI = 1
with clock that is as
high as that for CPI > 1
designs; is CPI < 1
feasible? (Chap 15-16)

Design memory & I/O
structures to support
ultrahigh-speed CPUs
(chap 17-24)



Define an instruction set;
make it simple enough
to require a small number
of cycles and allow high
clock rate, but not so
simple that we need many
instructions, even for very
simple tasks (Chap 5-8)



Design hardware
for CPI = 1; seek
improvements with
CPI > 1 (Chap 13-14)



Design ALU for
arithmetic & logic
ops (Chap 9-12)

IV Data Path and Control

Design a simple computer (MicroMIPS) to learn about:

- Data path – part of the CPU where data signals flow
- Control unit – guides data signals through data path
- Pipelining – a way of achieving greater performance

Topics in This Part

Chapter 13 Instruction Execution Steps

Chapter 14 Control Unit Synthesis

Chapter 15 Pipelined Data Paths

Chapter 16 Pipeline Performance Limits

13 Instruction Execution Steps

A simple computer executes instructions one at a time

- Fetches an instruction from the loc pointed to by PC
- Interprets and executes the instruction, then repeats

Topics in This Chapter

13.1 A Small Set of Instructions

13.2 The Instruction Execution Unit

13.3 A Single-Cycle Data Path

13.4 Branching and Jumping

13.5 Deriving the Control Signals

13.6 Performance of the Single-Cycle Design

13.1 A Small Set of Instructions

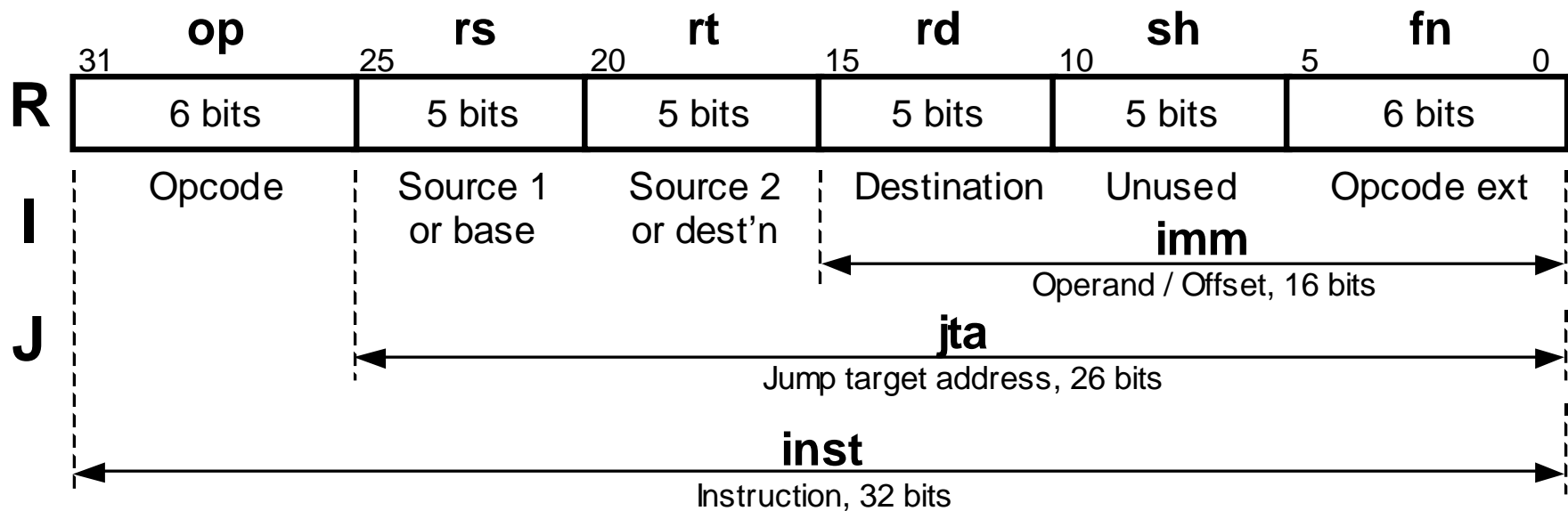


Fig. 13.1 MicroMIPS instruction formats and naming of the various fields.

We will refer to this diagram later

- Seven R-format ALU instructions (add, sub, slt, and, or, xor, nor)
- Six I-format ALU instructions (lui, addi, slti, andi, ori, xori)
- Two I-format memory access instructions (lw, sw)
- Three I-format conditional branch instructions (bltz, beq, bne)
- Four unconditional jump instructions (j, jr, jal, syscall)

Including
la rt,imm(rs)
(load address)
makes it easier
to write useful
programs

Arithmetic

Logic

Memory access

Control transfer

Table 13.1

Instruction	Usage	op	fn
Load upper immediate	lui rt,imm	15	
Add	add rd,rs,rt	0	32
Subtract	sub rd,rs,rt	0	34
Set less than	slt rd,rs,rt	0	42
Add immediate	addi rt,rs,imm	8	
Set less than immediate	slti rd,rs,imm	10	
AND	and rd,rs,rt	0	36
OR	or rd,rs,rt	0	37
XOR	xor rd,rs,rt	0	38
NOR	nor rd,rs,rt	0	39
AND immediate	andi rt,rs,imm	12	
OR immediate	ori rt,rs,imm	13	
XOR immediate	xori rt,rs,imm	14	
Load word	lw rt,imm(rs)	35	
Store word	sw rt,imm(rs)	43	
Jump	j L	2	
Jump register	jr rs	0	8
Branch less than 0	bltz rs,L	1	
Branch equal	beq rs,rt,L	4	
Branch not equal	bne rs,rt,L	5	
Jump and link	jal L	3	
System call	syscall	0	12

13.2 The Instruction Execution Unit

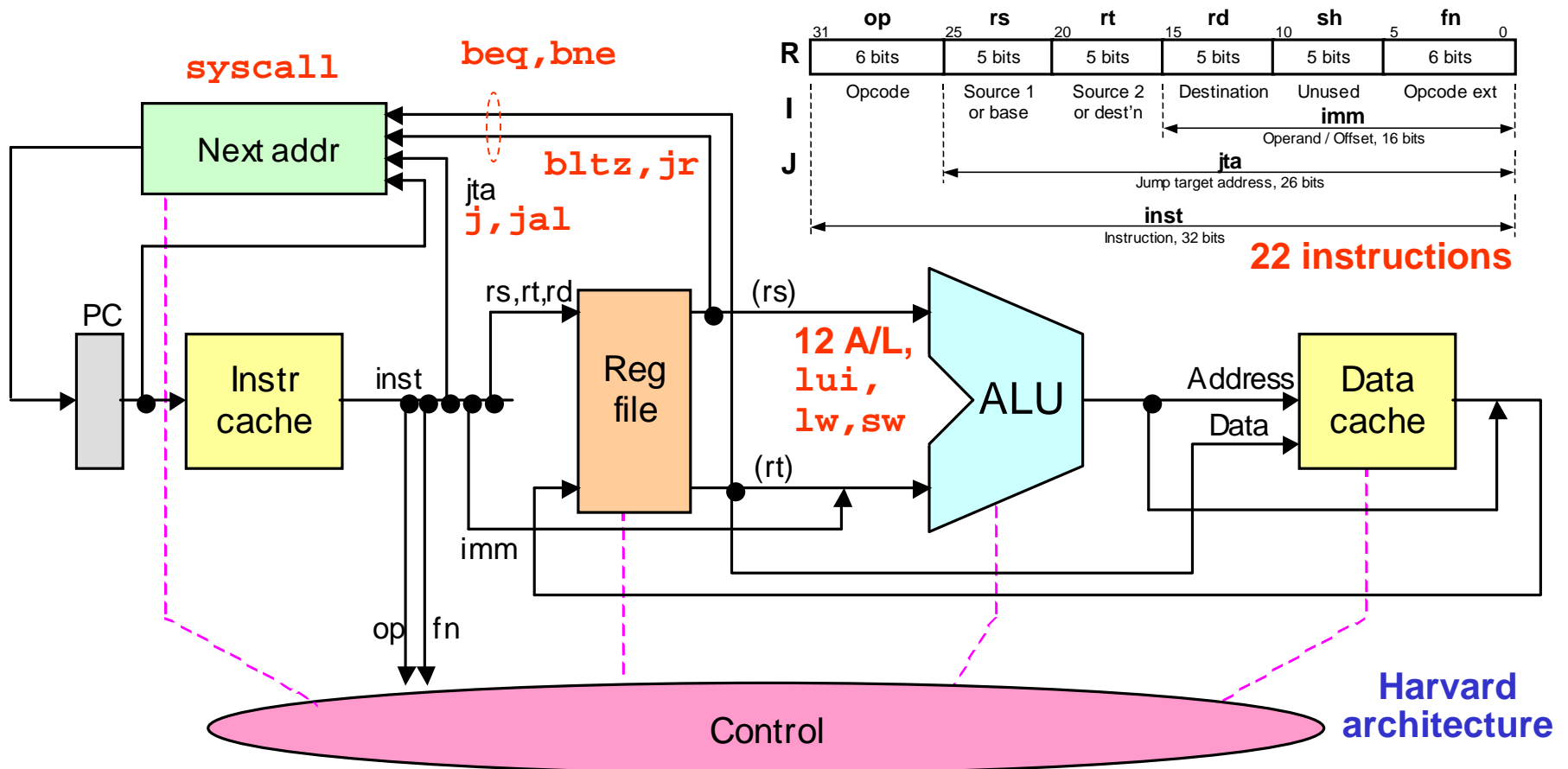


Fig. 13.2 Abstract view of the instruction execution unit for MicroMIPS. For naming of instruction fields, see Fig. 13.1.

13.3 A Single-Cycle Data Path

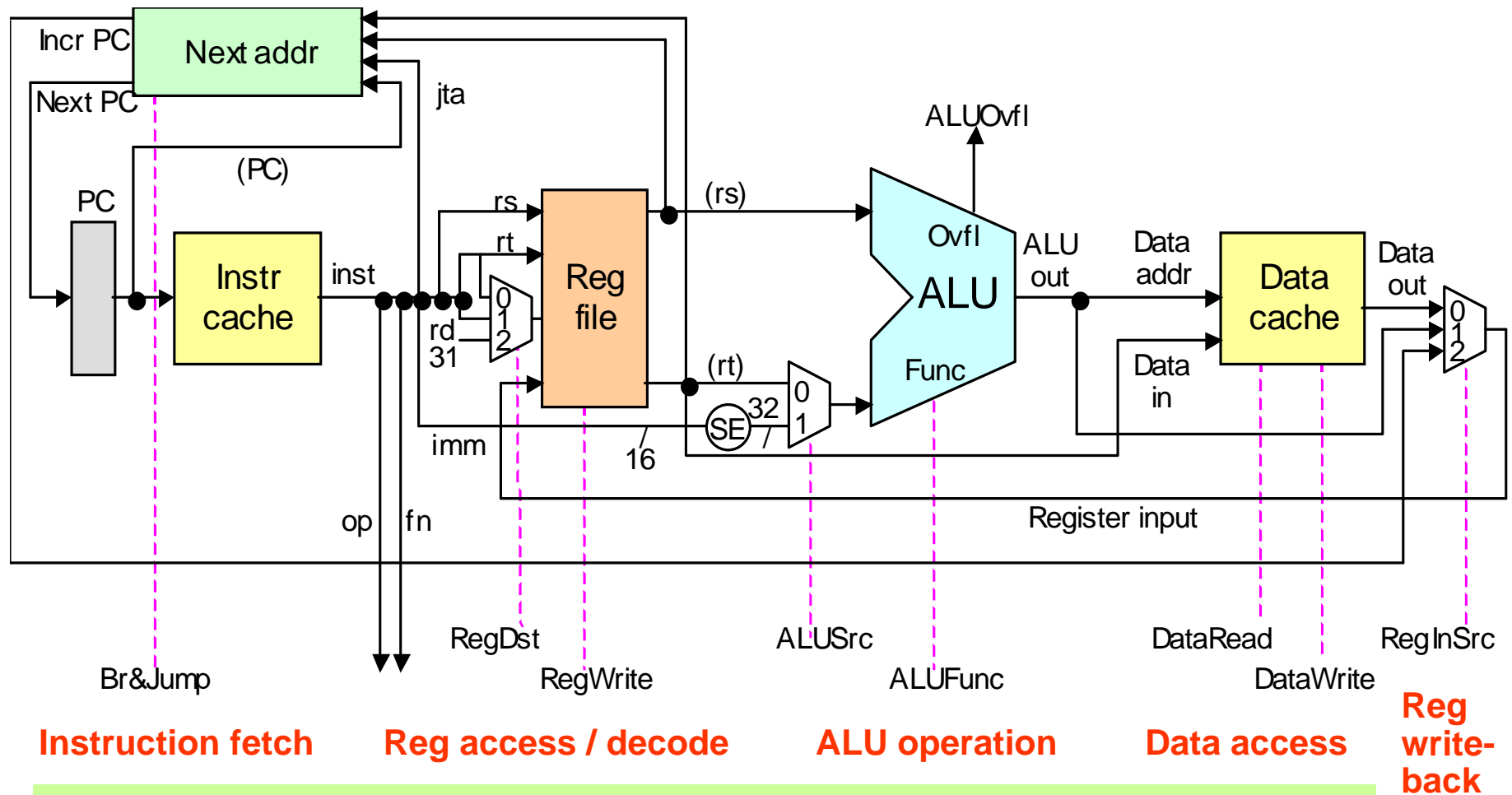


Fig. 13.3 Key elements of the single-cycle MicroMIPS data path.

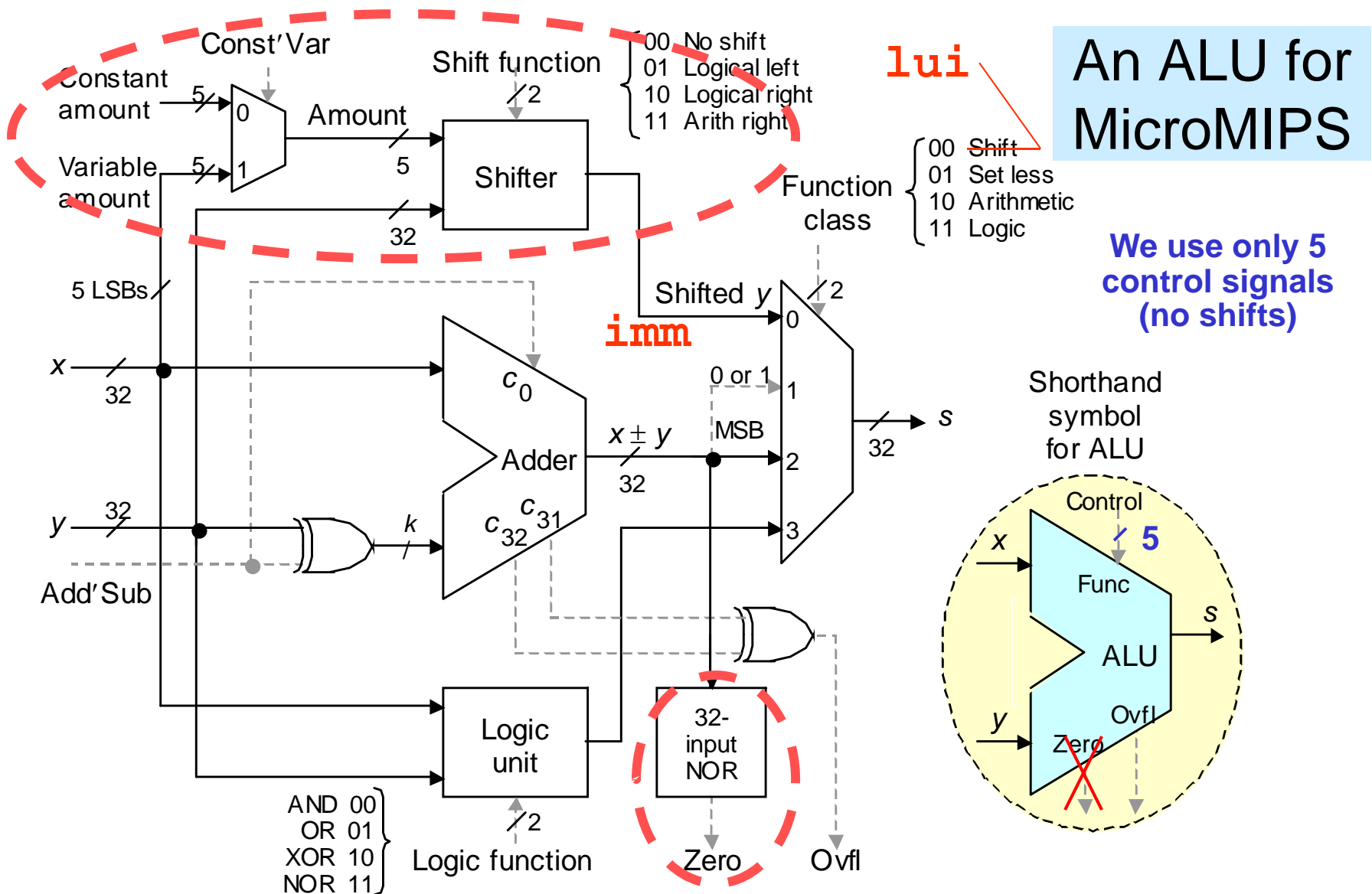


Fig. 10.19 A multifunction ALU with 8 control signals (2 for function class, 1 arithmetic, 3 shift, 2 logic) specifying the operation.

13.4 Branching and Jumping

Update
options
for PC

$(PC)_{31:2} + 1$

$(PC)_{31:2} + 1 + \text{imm}$

$(PC)_{31:28} \mid \text{jta}$

$(rs)_{31:2}$

SysCallAddr

Default option

When instruction is branch and condition is met

When instruction is j or jal

When the instruction is jr

Start address of an operating system routine

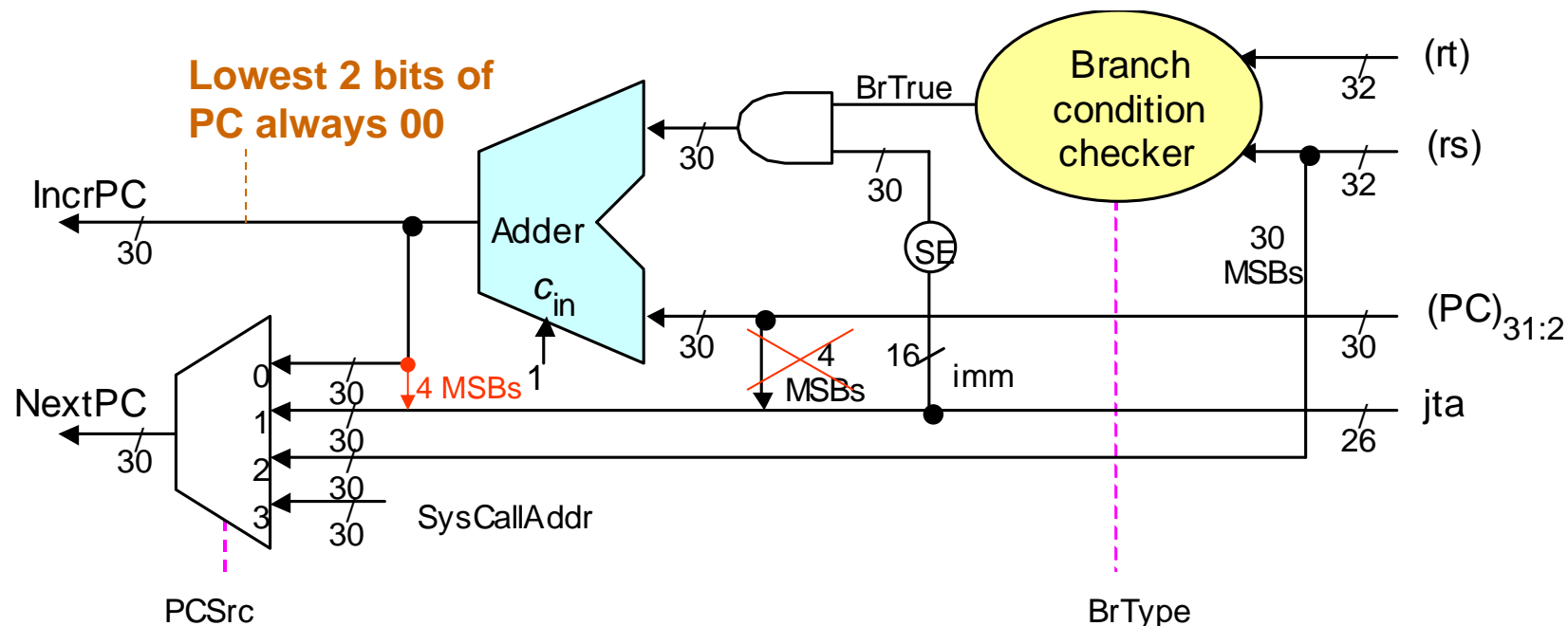


Fig. 13.4 Next-address logic for MicroMIPS (see top part of Fig. 13.3).

13.5 Deriving the Control Signals

Table 13.2 Control signals for the single-cycle MicroMIPS implementation.

Control signal		0	1	2	3
Reg file	RegWrite	Don't write	Write		
	RegDst ₁ , RegDst ₀	rt	rd	\$31	
	RegInSrc ₁ , RegInSrc ₀	Data out	ALU out	IncrPC	
ALU	ALUSrc	(rt)	imm		
	Add'Sub	Add	Subtract		
	LogicFn ₁ , LogicFn ₀	AND	OR	XOR	NOR
	FnClass ₁ , FnClass ₀	lui	Set less	Arithmetic	Logic
Data cache	DataRead	Don't read	Read		
	DataWrite	Don't write	Write		
Next addr	BrType ₁ , BrType ₀	No branch	beq	bne	bltz
	PCSrc ₁ , PCSrc ₀	IncrPC	jta	(rs)	SysCallAddr

Single-Cycle Data Path, Repeated for Reference

Outcome of an executed instruction:
A new value loaded into PC
Possible new value in a reg or memory loc

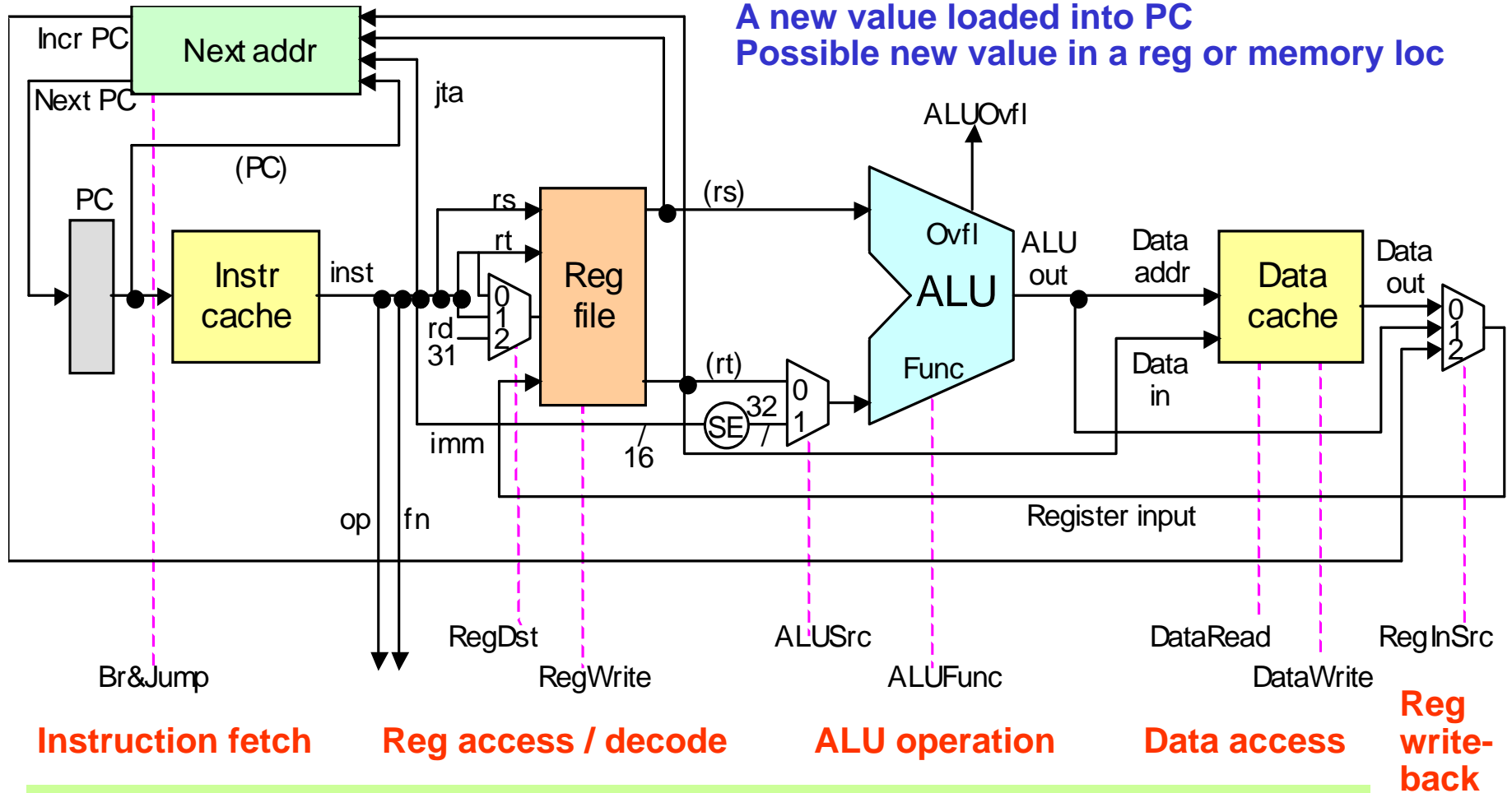


Fig. 13.3 Key elements of the single-cycle MicroMIPS data path.

Control Signal Settings

Table 13.3

Instruction	op	fn	RegWrite	RegDst	RegInSrc	ALUSrc	Add' Sub	LogicFn	FnClass	DataRead	DataWrite	BrType	PCSrc
Load upper immediate	001111		1	00	01	1			00	0	0	00	00
Add	000000	100000	1	01	01	0	0		10	0	0	00	00
Subtract	000000	100010	1	01	01	0	1		10	0	0	00	00
Set less than	000000	101010	1	01	01	0	1		01	0	0	00	00
Add immediate	001000		1	00	01	1	0		10	0	0	00	00
Set less than immediate	001010		1	00	01	1	1		01	0	0	00	00
AND	000000	100100	1	01	01	0		00	11	0	0	00	00
OR	000000	100101	1	01	01	0		01	11	0	0	00	00
XOR	000000	100110	1	01	01	0		10	11	0	0	00	00
NOR	000000	100111	1	01	01	0		11	11	0	0	00	00
AND immediate	001100		1	00	01	1		00	11	0	0	00	00
OR immediate	001101		1	00	01	1		01	11	0	0	00	00
XOR immediate	001110		1	00	01	1		10	11	0	0	00	00
Load word	100011		1	00	00	1	0		10	1	0	00	00
Store word	101011		0			1	0		10	0	1	00	00
Jump	000010		0							0	0		01
Jump register	000000	001000	0							0	0		10
Branch on less than 0	000001		0							0	0	11	00
Branch on equal	000100		0							0	0	01	00
Branch on not equal	000101		0							0	0	10	00
Jump and link	000011		1	10	10					0	0	00	01
System call	000000	001100	0							0	0		11

Control Signals in the Single-Cycle Data Path

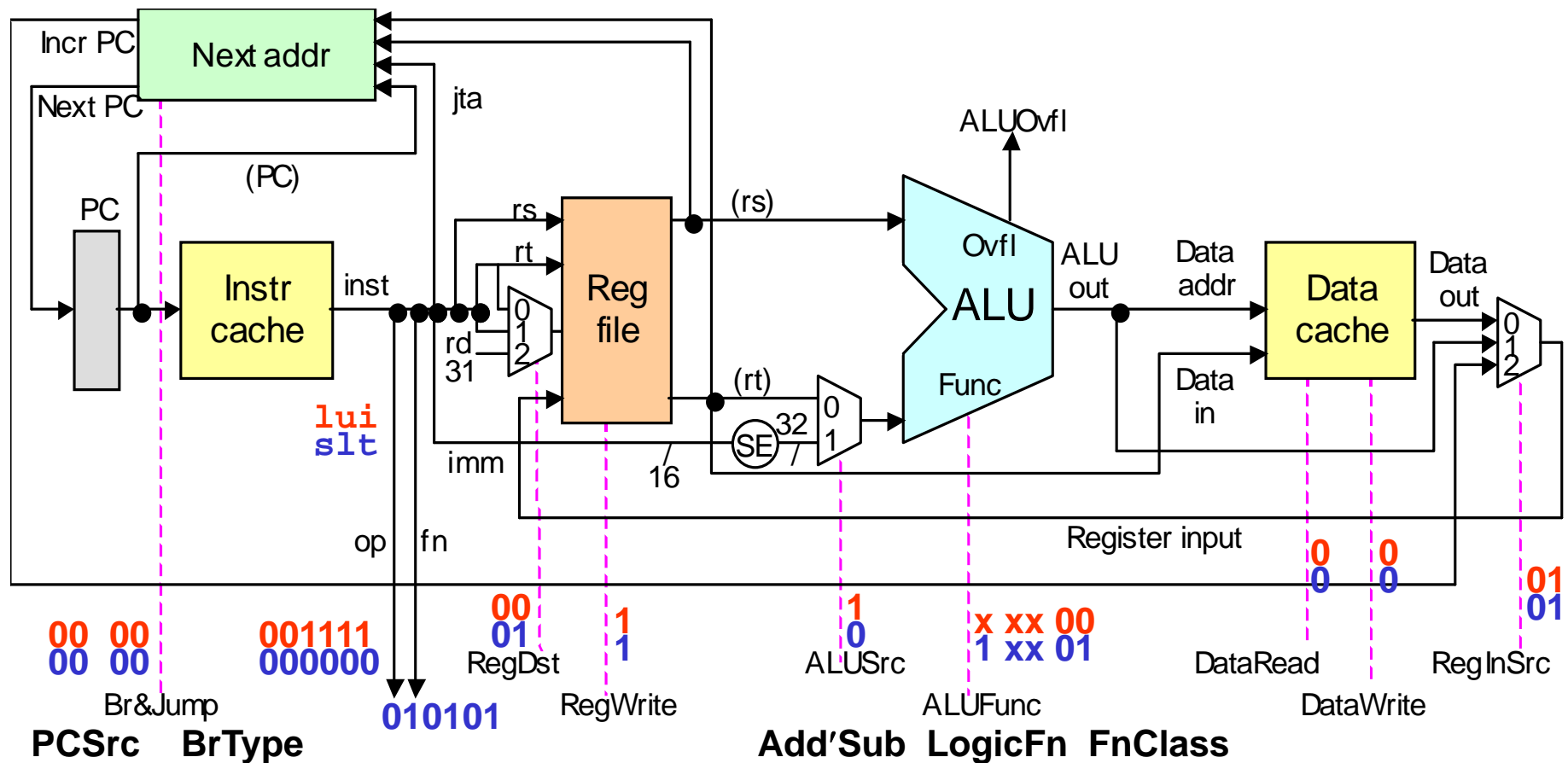


Fig. 13.3 Key elements of the single-cycle MicroMIPS data path.

Instruction Decoding

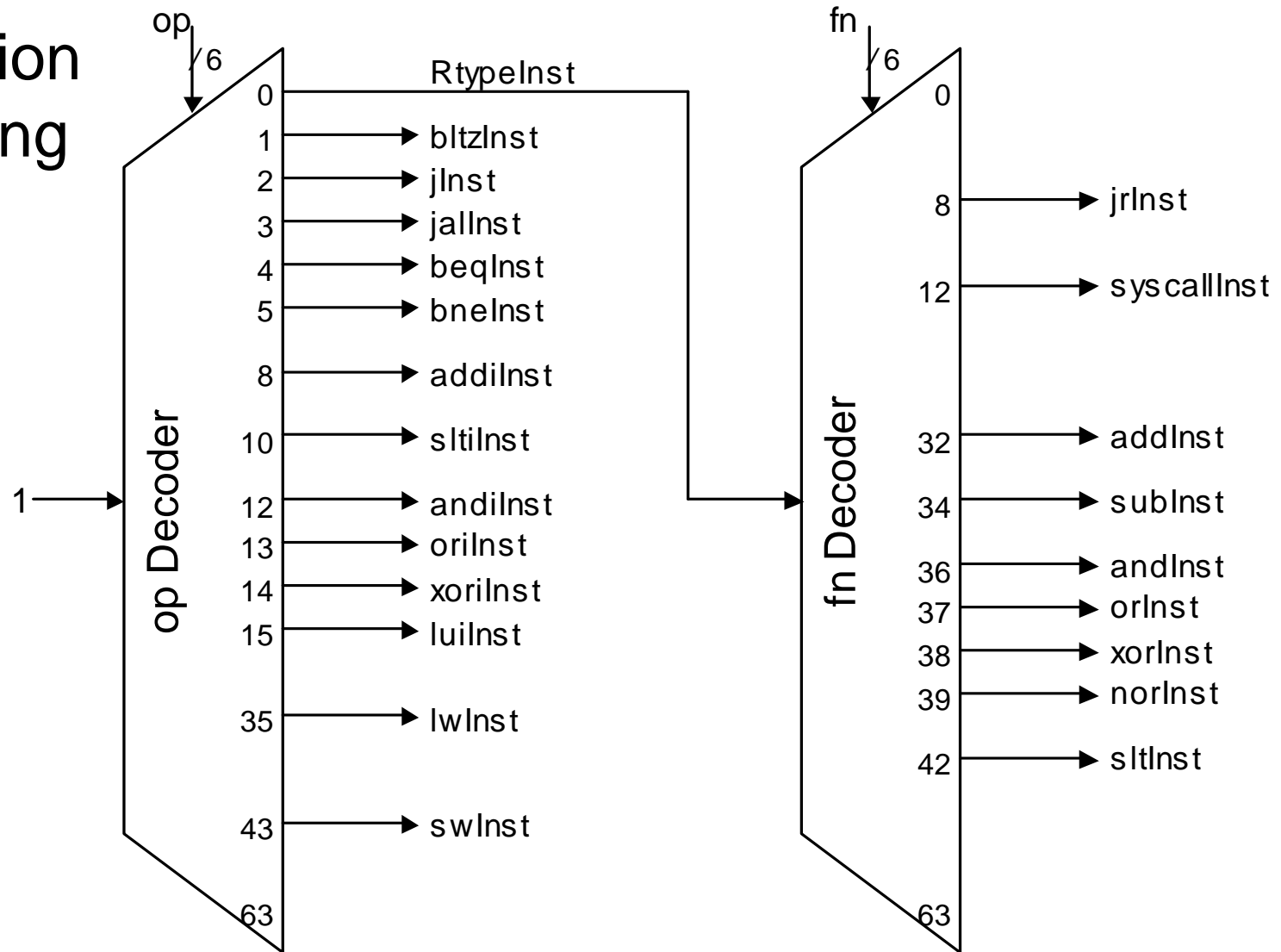


Fig. 13.5 Instruction decoder for MicroMIPS built of two 6-to-64 decoders.

Control
Signal
Settings:
Repeated
for
Reference

Table 13.3

Instruction	op	fn	RegWrite	RegDst	RegInSrc	ALUSrc	Add' Sub	LogicFn	FnClass	DataRead	DataWrite	BrType	PCSrc
Load upper immediate	001111		1	00	01	1			00	0	0	00	00
Add	000000	100000	1	01	01	0	0		10	0	0	00	00
Subtract	000000	100010	1	01	01	0	1		10	0	0	00	00
Set less than	000000	101010	1	01	01	0	1		01	0	0	00	00
Add immediate	001000		1	00	01	1	0		10	0	0	00	00
Set less than immediate	001010		1	00	01	1	1		01	0	0	00	00
AND	000000	100100	1	01	01	0		00	11	0	0	00	00
OR	000000	100101	1	01	01	0		01	11	0	0	00	00
XOR	000000	100110	1	01	01	0		10	11	0	0	00	00
NOR	000000	100111	1	01	01	0		11	11	0	0	00	00
AND immediate	001100		1	00	01	1		00	11	0	0	00	00
OR immediate	001101		1	00	01	1		01	11	0	0	00	00
XOR immediate	001110		1	00	01	1		10	11	0	0	00	00
Load word	100011		1	00	00	1	0		10	1	0	00	00
Store word	101011		0			1	0		10	0	1	00	00
Jump	000010		0							0	0		01
Jump register	000000	001000	0							0	0		10
Branch on less than 0	000001		0							0	0	11	00
Branch on equal	000100		0							0	0	01	00
Branch on not equal	000101		0							0	0	10	00
Jump and link	000011		1	10	10					0	0	00	01
System call	000000	001100	0							0	0		11

Control Signal Generation

Auxiliary signals identifying instruction classes

$\text{arithInst} = \text{addInst} \vee \text{subInst} \vee \text{sltInst} \vee \text{addiInst} \vee \text{sltiInst}$

$\text{logicInst} = \text{andInst} \vee \text{orInst} \vee \text{xorInst} \vee \text{norInst} \vee \text{andiInst} \vee \text{oriInst} \vee \text{xoriInst}$

$\text{immInst} = \text{luiInst} \vee \text{addiInst} \vee \text{sltiInst} \vee \text{andiInst} \vee \text{oriInst} \vee \text{xoriInst}$

Example logic expressions for control signals

$\text{RegWrite} = \text{luiInst} \vee \text{arithInst} \vee \text{logicInst} \vee \text{lwInst} \vee \text{jallInst}$

$\text{ALUSrc} = \text{immInst} \vee \text{lwInst} \vee \text{swInst}$

$\text{Add'Sub} = \text{subInst} \vee \text{sltInst} \vee \text{sltiInst}$

$\text{DataRead} = \text{lwInst}$

$\text{PCSrc}_0 = \text{jInst} \vee \text{jallInst} \vee \text{syscallInst}$

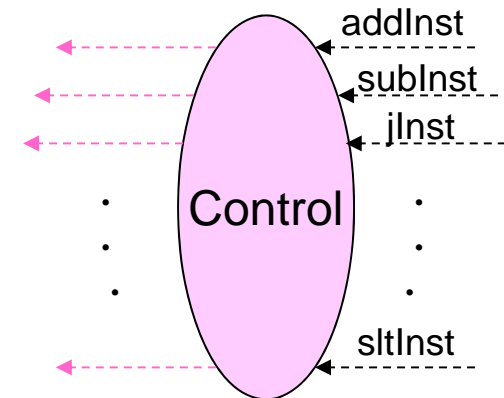
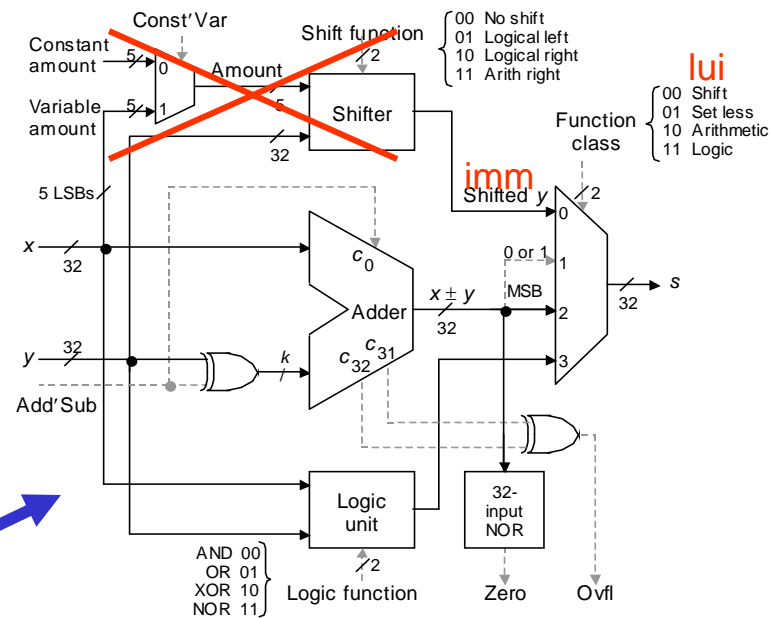
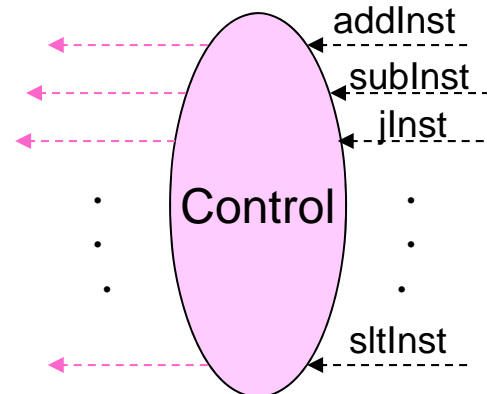
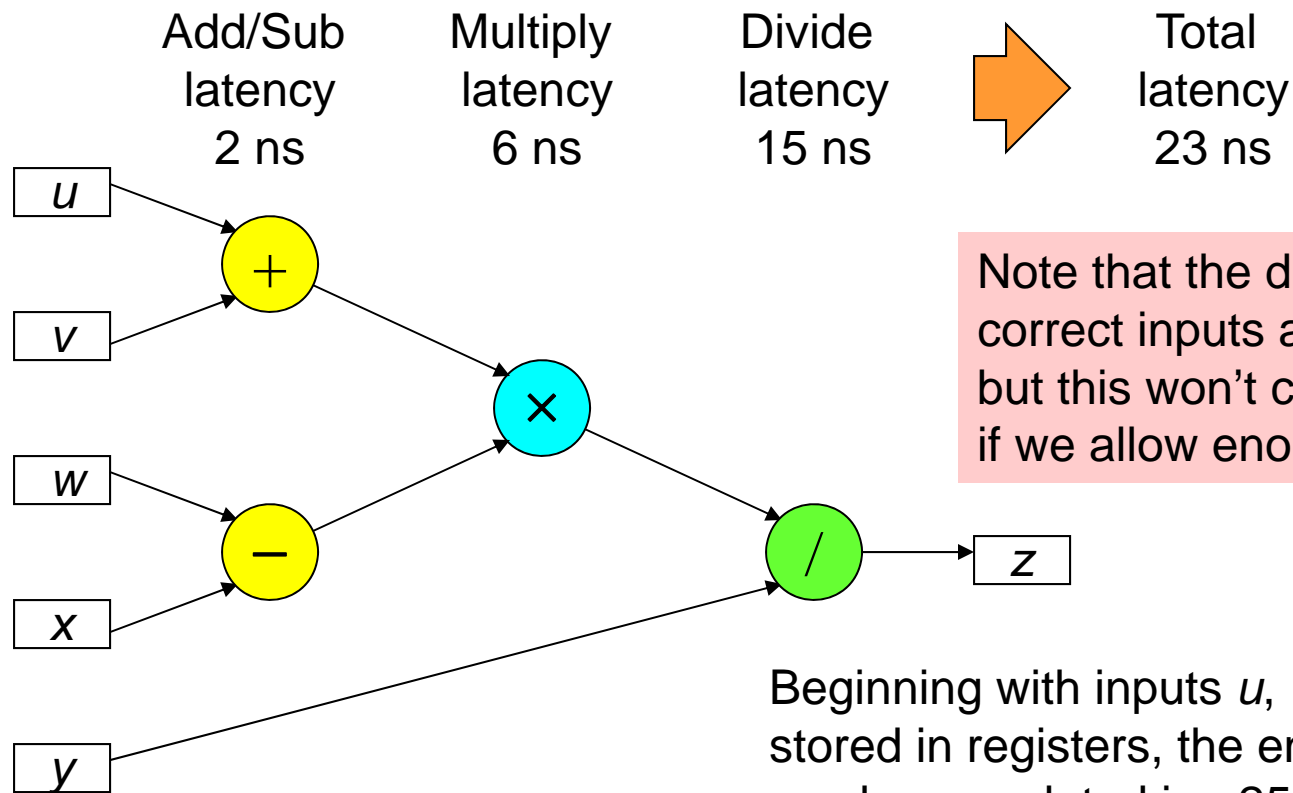


Fig. 10.19

[illegible][illegible]

13.6 Performance of the Single-Cycle Design

An example combinational-logic data path to compute $z := (u + v)(w - x) / y$



Note that the divider gets its correct inputs after ≈ 9 ns, but this won't cause a problem if we allow enough total time

Beginning with inputs u , v , w , x , and y stored in registers, the entire computation can be completed in ≈ 25 ns, allowing 1 ns each for register readout and write

Performance Estimation for Single-Cycle MicroMIPS

Instruction access	2 ns
Register read	1 ns
ALU operation	2 ns
Data cache access	2 ns
Register write	<u>1 ns</u>
Total	8 ns

Single-cycle clock = 125 MHz

R-type	44%	6 ns
Load	24%	8 ns
Store	12%	7 ns
Branch	18%	5 ns
Jump	2%	<u>3 ns</u>

Weighted mean \cong 6.36 ns

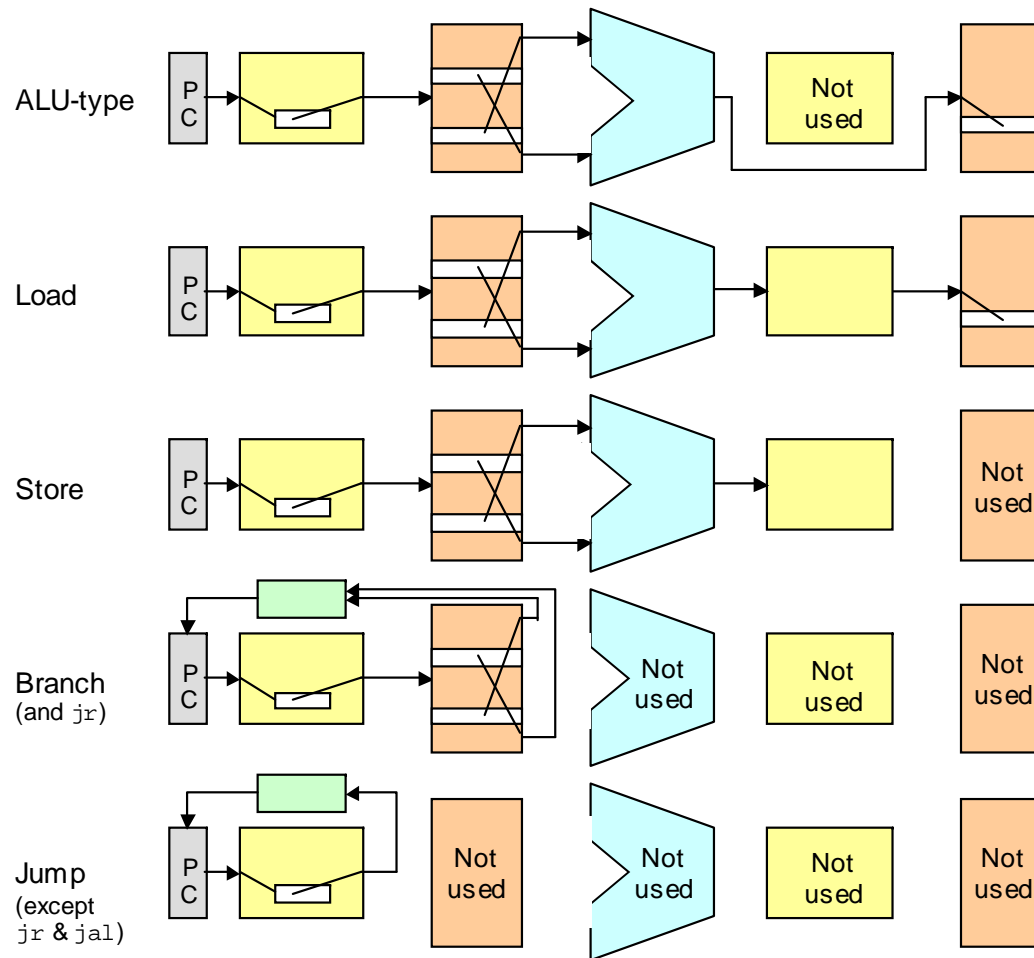


Fig. 13.6 The MicroMIPS data path unfolded (by depicting the register write step as a separate block) so as to better visualize the critical-path latencies.

How Good is Our Single-Cycle Design?

Clock rate of 125 MHz not impressive

How does this compare with
current processors on the market?

Not bad, where latency is concerned

Instruction access	2 ns
Register read	1 ns
ALU operation	2 ns
Data cache access	2 ns
Register write	<u>1 ns</u>
Total	8 ns

Single-cycle clock = 125 MHz

A 2.5 GHz processor with 20 or so pipeline stages has a latency of about

$$0.4 \text{ ns/cycle} \times 20 \text{ cycles} = 8 \text{ ns}$$

Throughput, however, is much better for the pipelined processor:

Up to 20 times better with single issue

Perhaps up to 100 times better with multiple issue

14 Control Unit Synthesis

The control unit for the single-cycle design is memoryless

- Problematic when instructions vary greatly in complexity
- Multiple cycles needed when resources must be reused

Topics in This Chapter

14.1 A Multicycle Implementation

14.2 Choosing the Clock Cycle

14.3 The Control State Machine

14.4 Performance of the Multicycle Design

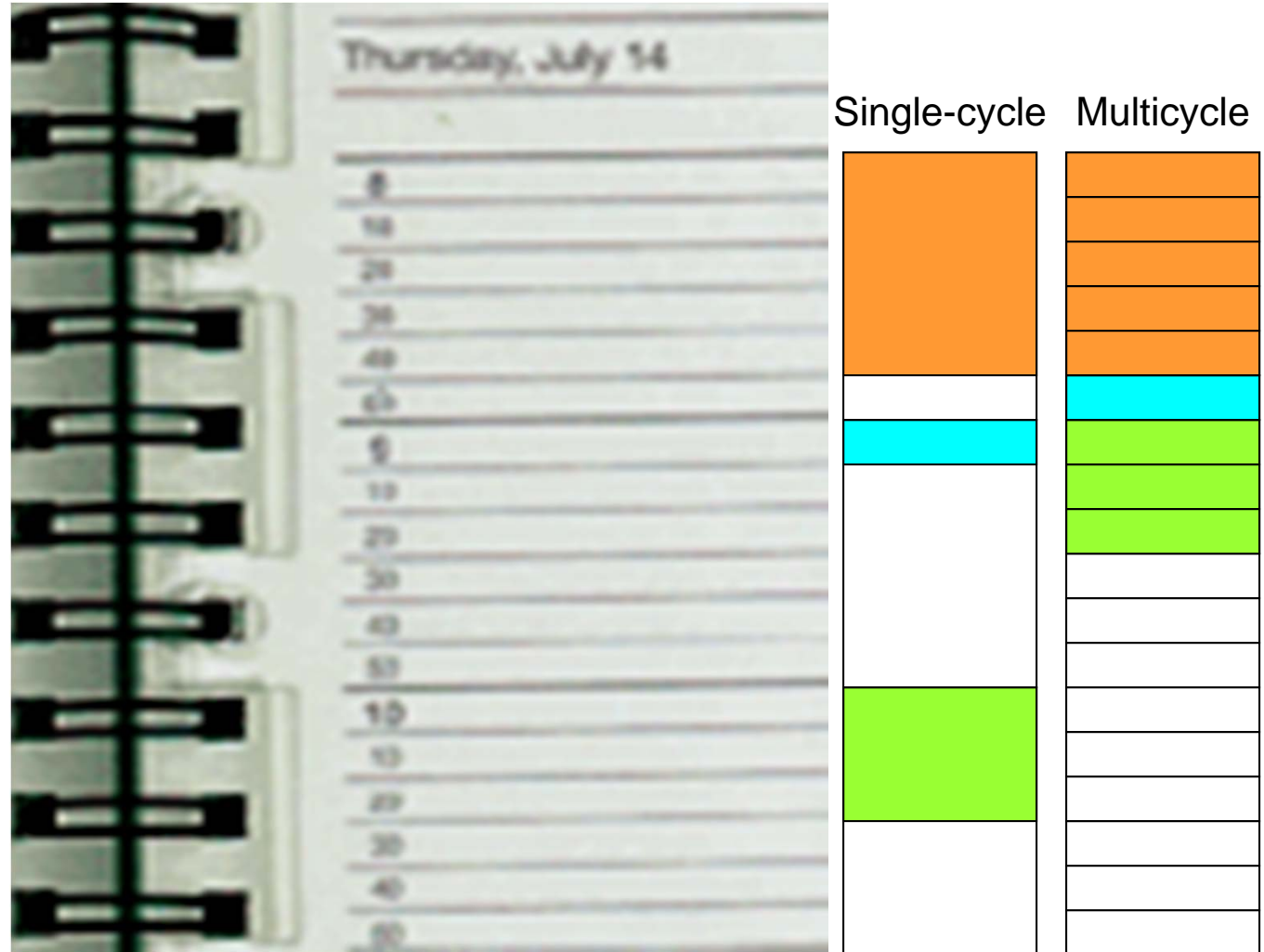
14.5 Microprogramming

14.6 Exception Handling

14.1 A Multicycle Implementation

Appointment book for a dentist

Assume longest
treatment takes
one hour



Single-Cycle vs. Multicycle MicroMIPS

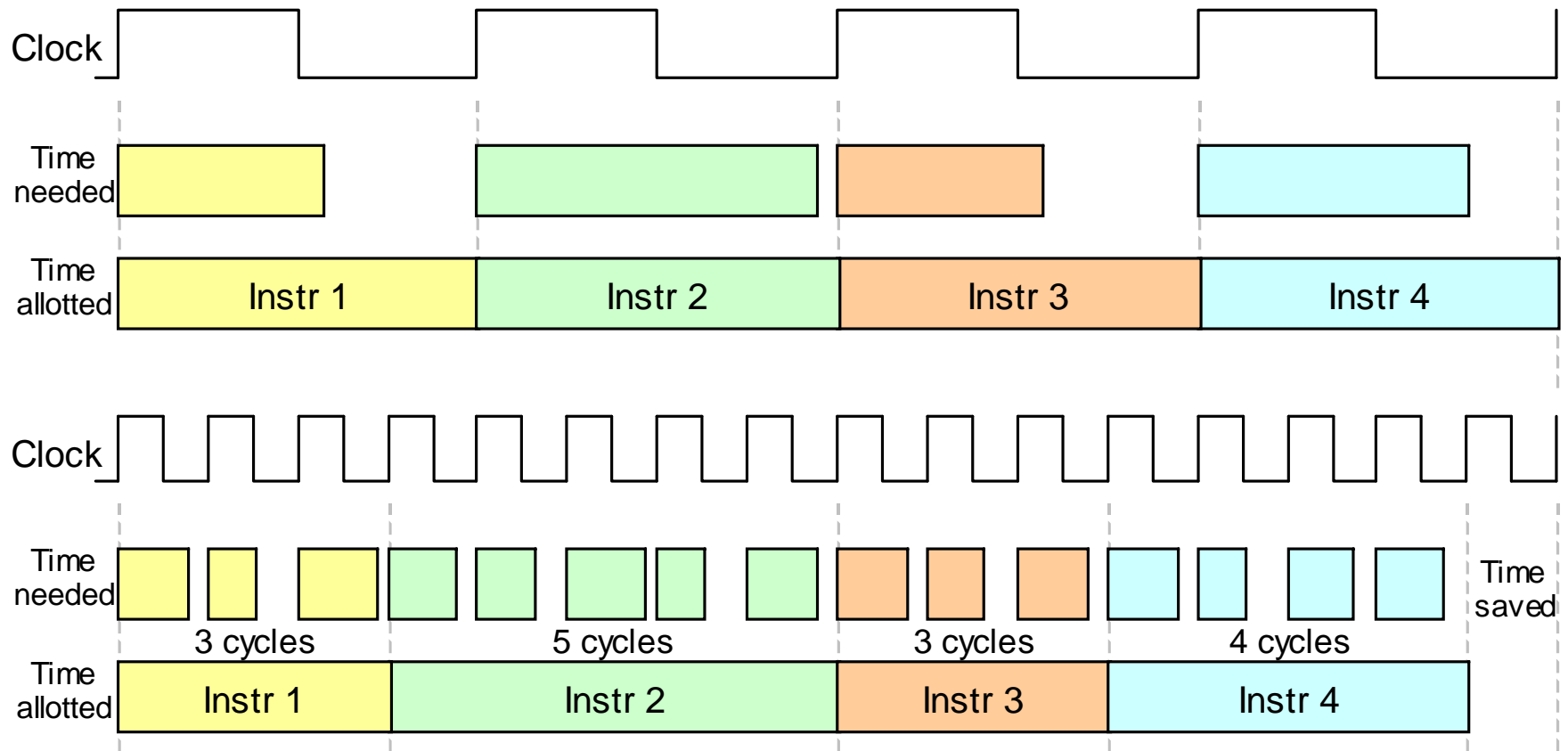


Fig. 14.1 Single-cycle versus multicycle instruction execution.

A Multicycle Data Path

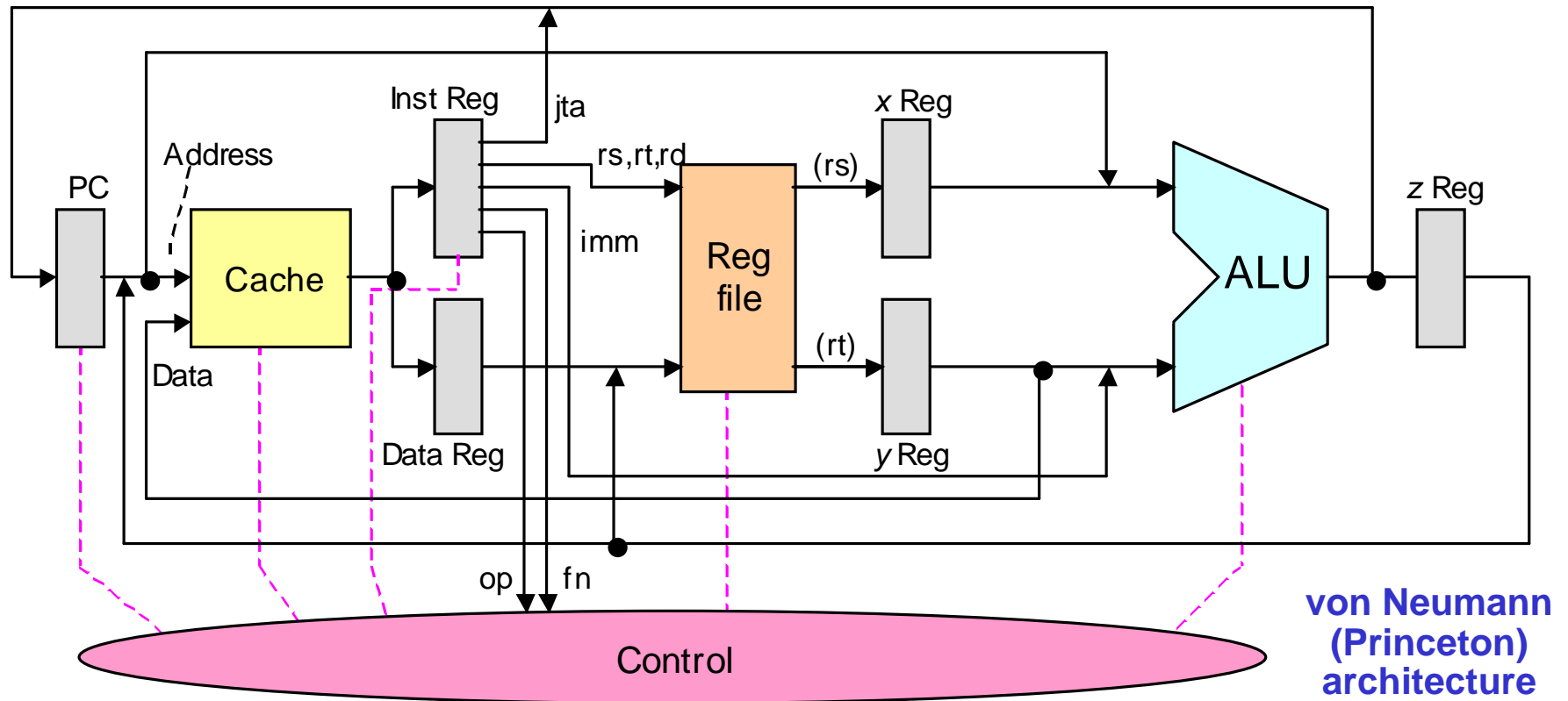


Fig. 14.2 Abstract view of a multicycle instruction execution unit for MicroMIPS. For naming of instruction fields, see Fig. 13.1.

Multicycle Data Path with Control Signals Shown

Three major changes relative to the single-cycle data path:

2. ALU performs double duty for address calculation

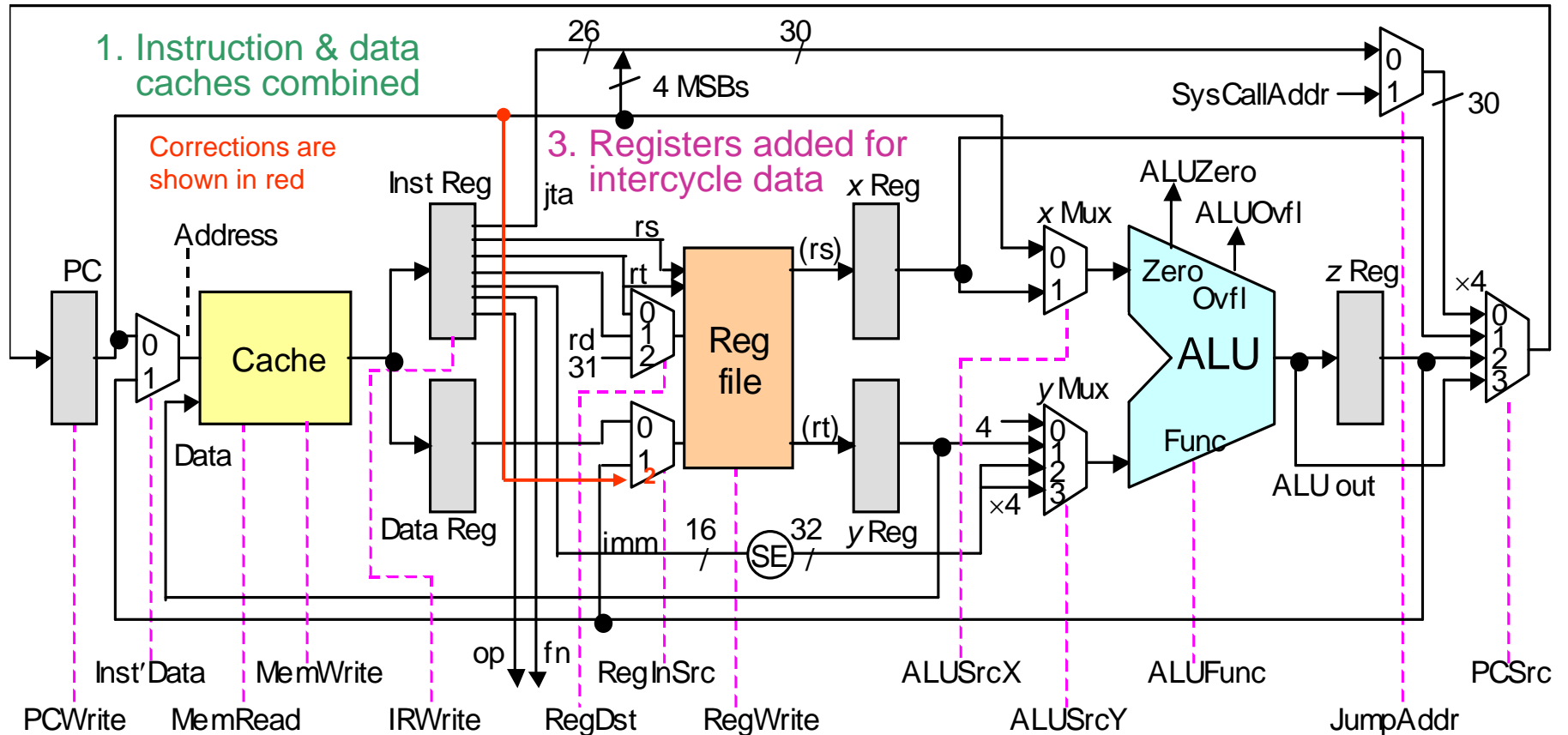


Fig. 14.3 Key elements of the multicycle MicroMIPS data path.

14.2 Clock Cycle and Control Signals

Table 14.1

Control signal		0	1	2	3
Program counter	JumpAddr	jta	SysCallAddr		
	PCSrc ₁ , PCSrc ₀	Jump addr	x reg	z reg	ALU out
	PCWrite	Don't write	Write		
Cache	Inst'Data	PC	z reg		
	MemRead	Don't read	Read		
	MemWrite	Don't write	Write		
Register file	IRWrite	Don't write	Write		
	RegWrite	Don't write	Write		
	RegDst ₁ , RegDst ₀	rt	rd	\$31	
ALU	RegInSrc ₁ , RegInSrc ₀	Data reg	z reg	PC	
	ALUSrcX	PC	x reg		
	ALUSrcY ₁ , ALUSrcY ₀	4	y reg	imm	4 × imm
	Add'Sub	Add	Subtract		
	LogicFn ₁ , LogicFn ₀	AND	OR	XOR	NOR
	FnClass ₁ , FnClass ₀	lui	Set less	Arithmetic	Logic

Multicycle Data Path, Repeated for Reference

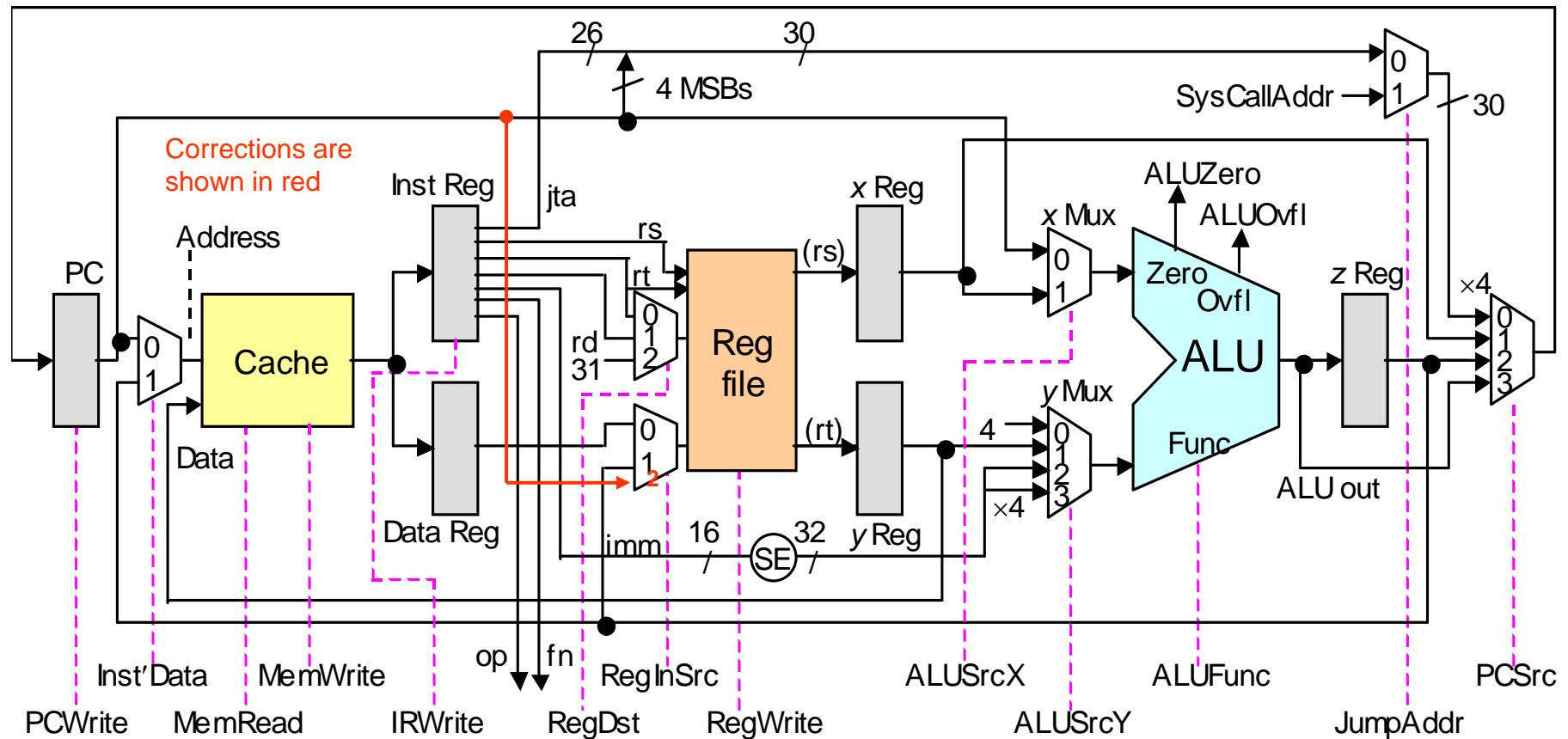


Fig. 14.3 Key elements of the multicycle MicroMIPS data path.

Execution Cycles

Table 14.2 Execution cycles for multicycle MicroMIPS

		Instruction	Operations	Signal settings
Fetch & PC incr	1	Any	Read out the instruction and write it into instruction register, increment PC	Inst'Data = 0, MemRead = 1 IRWrite = 1, ALUSrcX = 0 ALUSrcY = 0, ALUFunc = '+' PCSrc = 3, PCWrite = 1
Decode & reg read	2	Any	Read out <i>rs</i> & <i>rt</i> into <i>x</i> & <i>y</i> registers, compute branch address and save in <i>z</i> register	ALUSrcX = 0, ALUSrcY = 3 ALUFunc = '+'
ALU oper & PC update	3	ALU type	Perform ALU operation and save the result in <i>z</i> register	ALUSrcX = 1, ALUSrcY = 1 or 2 ALUFunc: Varies
		Load/Store	Add base and offset values, save in <i>z</i> register	ALUSrcX = 1, ALUSrcY = 2 ALUFunc = '+'
		Branch	If (<i>x</i> reg) = \neq < (<i>y</i> reg), set PC to branch target address	ALUSrcX = 1, ALUSrcY = 1 ALUFunc = '-', PCSrc = 2 PCWrite = ALUZero or ALUZero' or ALUOut ₃₁
		Jump	Set PC to the target address <i>jta</i> , SysCallAddr, or (<i>rs</i>)	JumpAddr = 0 or 1, PCSrc = 0 or 1, PCWrite = 1
Reg write or mem access	4	ALU type	Write back <i>z</i> reg into <i>rd</i>	RegDst = 1, RegInSrc = 1 RegWrite = 1
		Load	Read memory into data reg	Inst'Data = 1, MemRead = 1
		Store	Copy <i>y</i> reg into memory	Inst'Data = 1, MemWrite = 1
Reg write for lw	5	Load	Copy data register into <i>rt</i>	RegDst = 0, RegInSrc = 0 RegWrite = 1

14.3 The Control State Machine

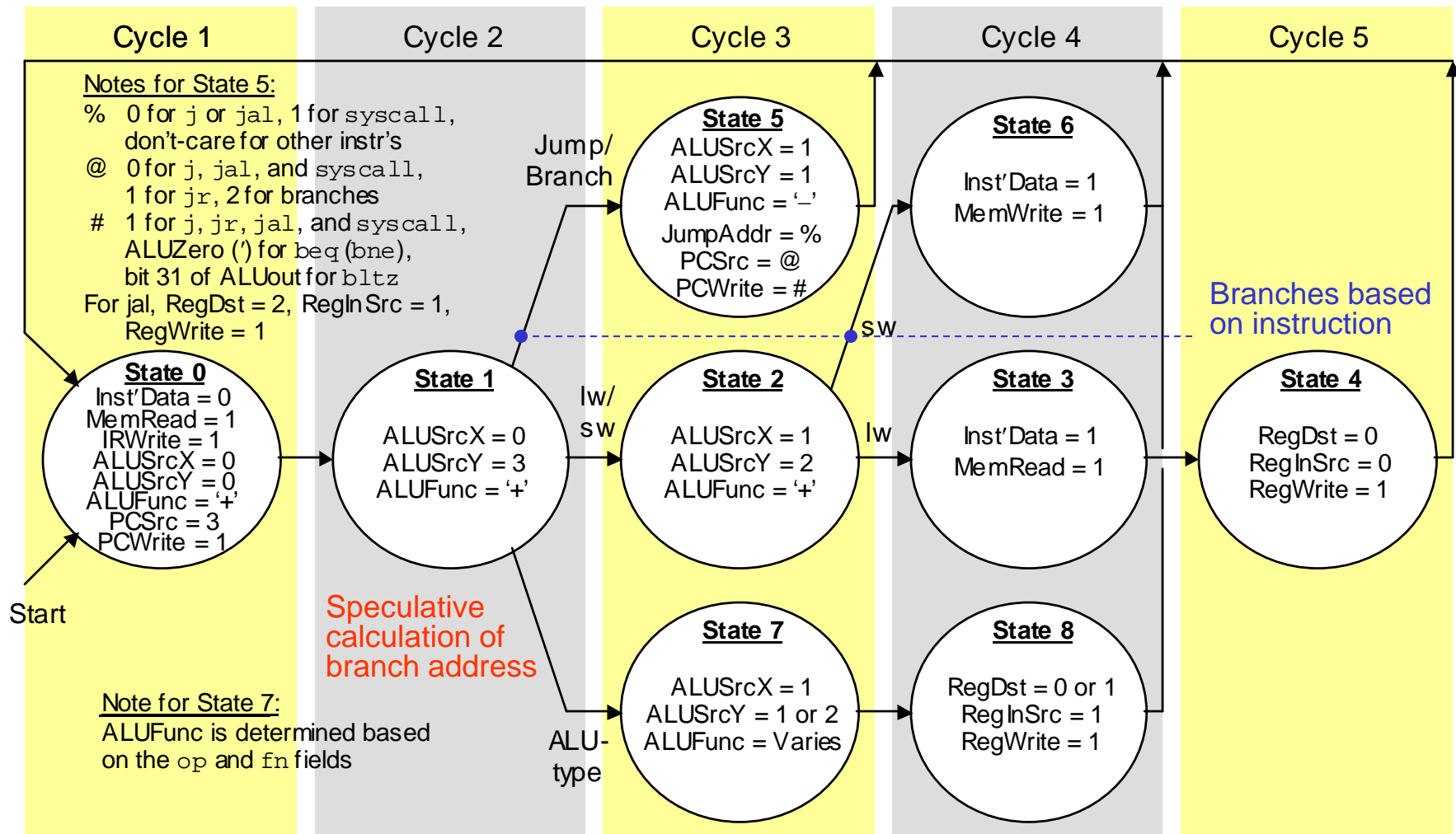


Fig. 14.4 The control state machine for multicycle MicroMIPS.

State and Instruction Decoding

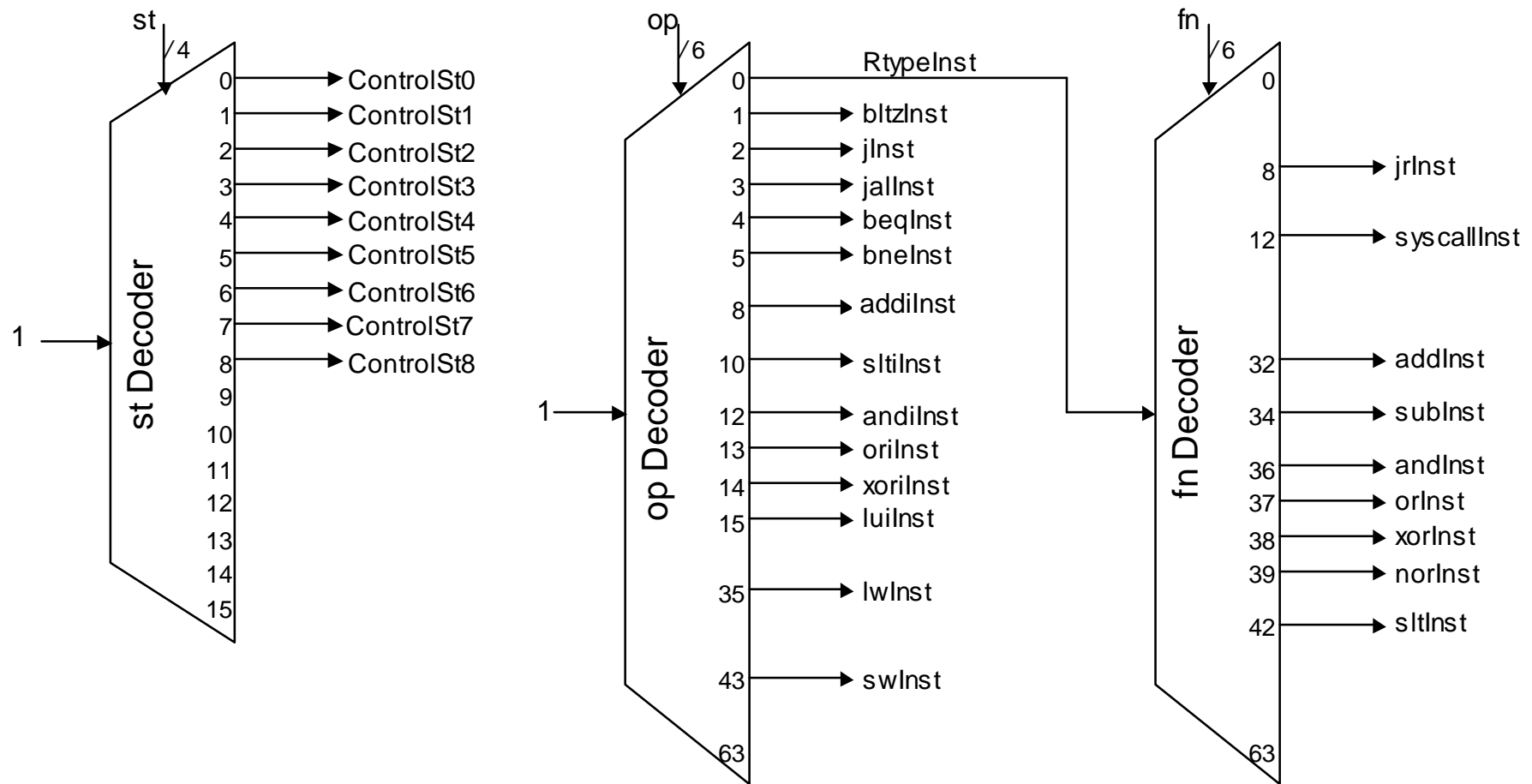


Fig. 14.5 State and instruction decoders for multicycle MicroMIPS.

Control Signal Generation

Certain control signals depend only on the control state

$\text{ALUSrcX} = \text{ControlSt2} \vee \text{ControlSt5} \vee \text{ControlSt7}$

$\text{RegWrite} = \text{ControlSt4} \vee \text{ControlSt8}$

Auxiliary signals identifying instruction classes

$\text{addsubInst} = \text{addInst} \vee \text{subInst} \vee \text{addiInst}$

$\text{logicInst} = \text{andInst} \vee \text{orInst} \vee \text{xorInst} \vee \text{norInst} \vee \text{andiInst} \vee \text{oriInst} \vee \text{xoriInst}$

Logic expressions for ALU control signals

$\text{Add'Sub} = \text{ControlSt5} \vee (\text{ControlSt7} \wedge \text{subInst})$

$\text{FnClass}_1 = \text{ControlSt7}' \vee \text{addsubInst} \vee \text{logicInst}$

$\text{FnClass}_0 = \text{ControlSt7} \wedge (\text{logicInst} \vee \text{sllInst} \vee \text{slltInst})$

$\text{LogicFn}_1 = \text{ControlSt7} \wedge (\text{xorInst} \vee \text{xoriInst} \vee \text{norInst})$

$\text{LogicFn}_0 = \text{ControlSt7} \wedge (\text{orInst} \vee \text{oriInst} \vee \text{norInst})$

14.4 Performance of the Multicycle Design

R-type	44%	4 cycles
Load	24%	5 cycles
Store	12%	4 cycles
Branch	18%	3 cycles
Jump	2%	3 cycles

	Contribution to CPI
R-type	$0.44 \times 4 = 1.76$
Load	$0.24 \times 5 = 1.20$
Store	$0.12 \times 4 = 0.48$
Branch	$0.18 \times 3 = 0.54$
Jump	$0.02 \times 3 = 0.06$

Average CPI $\cong 4.04$

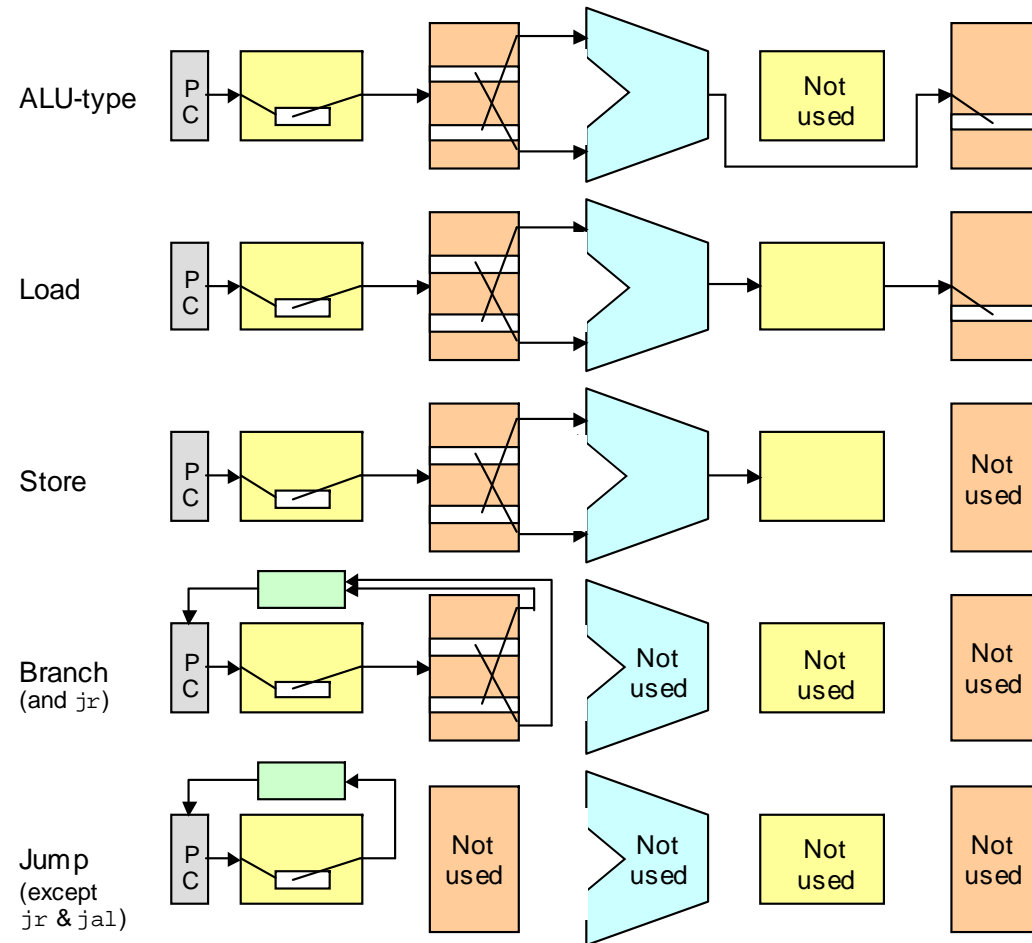


Fig. 13.6 The MicroMIPS data path unfolded (by depicting the register write step as a separate block) so as to better visualize the critical-path latencies.

How Good is Our Multicycle Design?

Clock rate of 500 MHz better than 125 MHz of single-cycle design, but still unimpressive

Cycle time = 2 ns
Clock rate = 500 MHz

How does the performance compare with current processors on the market?

R-type	44%	4 cycles
Load	24%	5 cycles
Store	12%	4 cycles
Branch	18%	3 cycles
Jump	2%	3 cycles

Not bad, where latency is concerned

A 2.5 GHz processor with 20 or so pipeline stages has a latency of about $0.4 \times 20 = 8$ ns

Contribution to CPI

R-type	$0.44 \times 4 = 1.76$
Load	$0.24 \times 5 = 1.20$
Store	$0.12 \times 4 = 0.48$
Branch	$0.18 \times 3 = 0.54$
Jump	$0.02 \times 3 = 0.06$

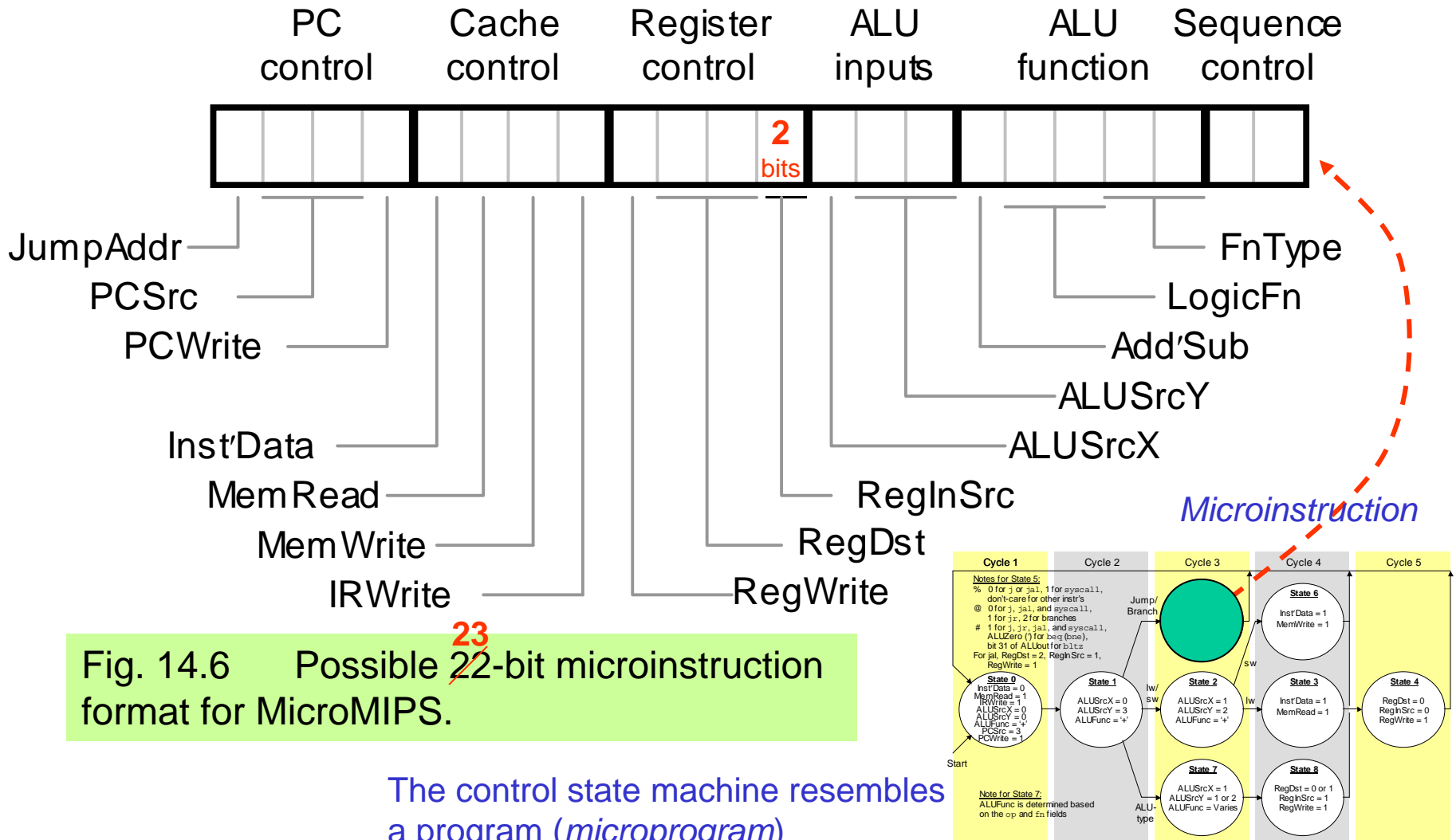
Throughput, however, is much better for the pipelined processor:

Up to 20 times better with single issue

Perhaps up to 100× with multiple issue

Average CPI $\cong 4.04$

14.5 Microprogramming



The Control State Machine as a Microprogram

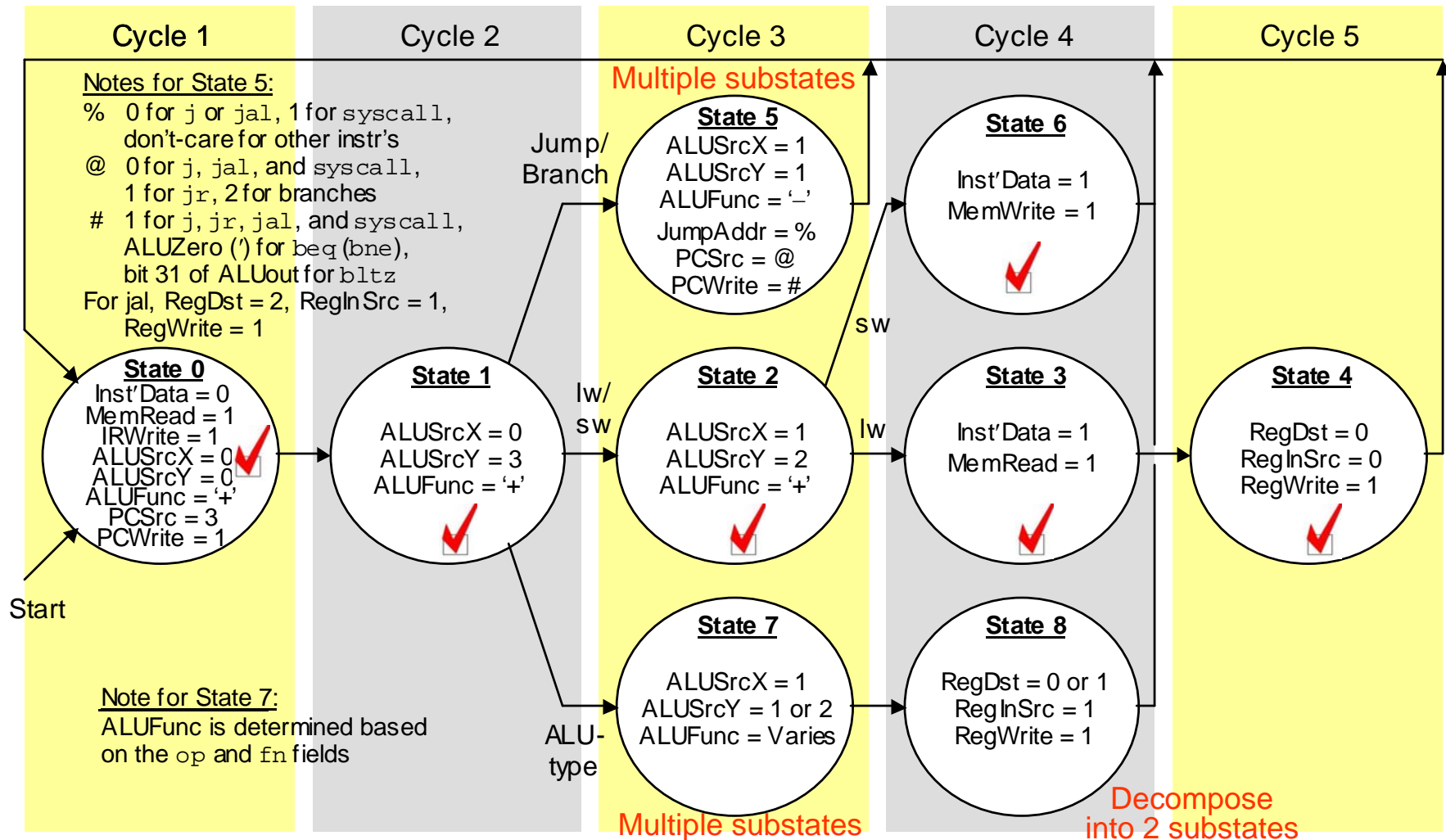


Fig. 14.4 The control state machine for multicycle MicroMIPS.

Symbolic Names for Microinstruction Field Values

Table 14.3 Microinstruction field values and their symbolic names. The default value for each unspecified field is the all 0s bit pattern.

Field name	Possible field values and their symbolic names				
PC control	0001	1001	x011	x101	x111
	PCjump	PCsyscall	PCjreg	PCbranch	PCnext
Cache control	0101	1010	1100		
	CacheFetch	CacheStore	CacheLoad		
Register control	1000 10000	1001 10001	1011 10101	1101 11010	
	rt ← Data	rt ← z	rd ← z	\$31 ← PC	
ALU inputs*	000	011	101	110	x10
	PC ⊗ 4	PC ⊗ 4imm	x ⊗ y	x ⊗ imm	(imm)
ALU function*	0xx10	1xx01	1xx10	x0011	x0111
	+	<	−	^	∨
	x1011	x1111	xxx00		
	⊕	~∨	lui		
Seq. control	01	10	11		
	μPCdisp1	μPCdisp2	μPCfetch		

* The operator symbol ⊗ stands for any of the ALU functions defined above (except for “lui”).

Control Unit for Microprogramming

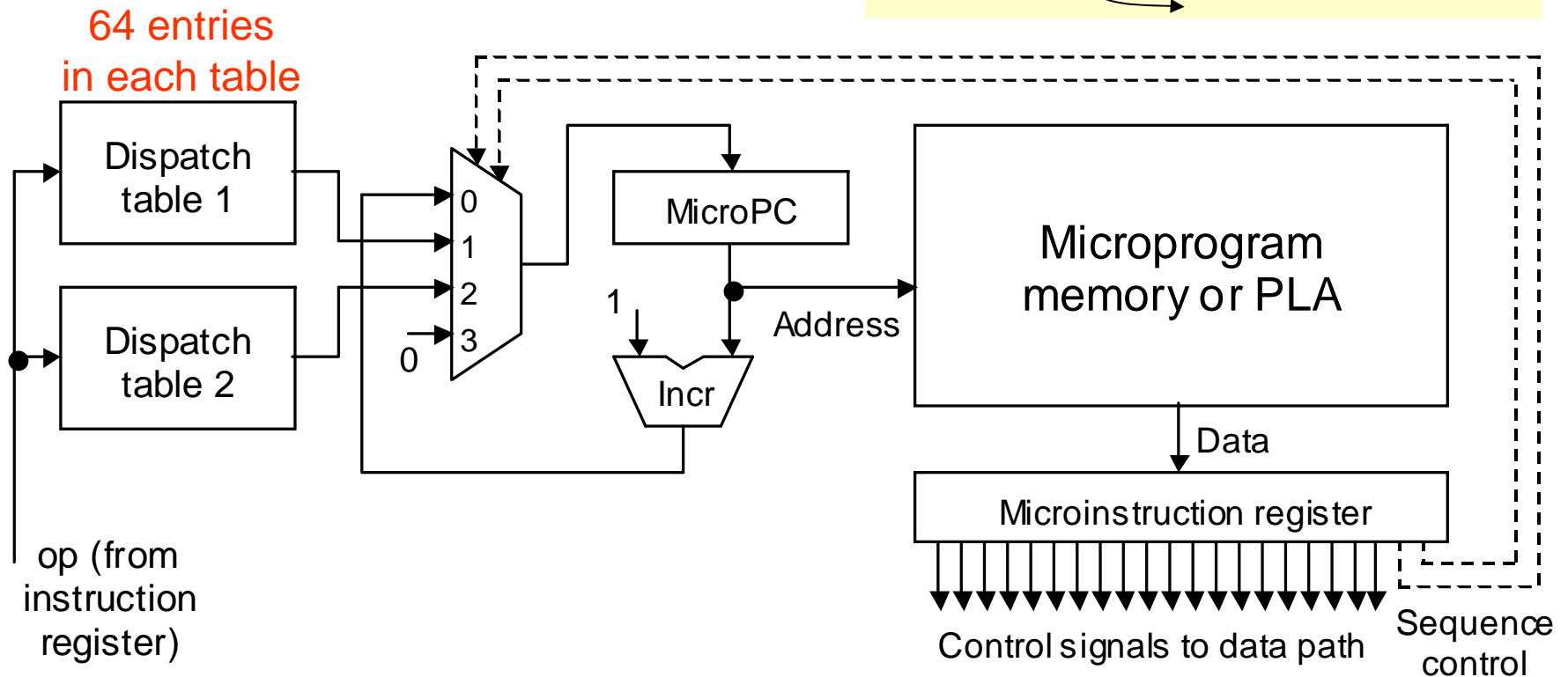
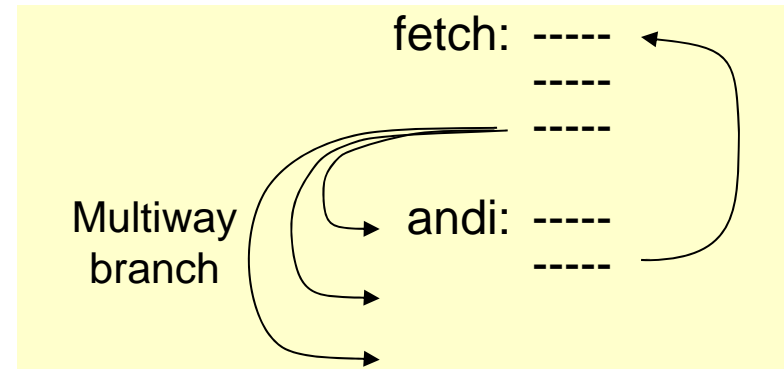


Fig. 14.7 Microprogrammed control unit for MicroMIPS .

Microprogram for MicroMIPS

37 microinstructions

Fig. 14.8
The complete
MicroMIPS
microprogram.

fetch:	PCnext, CacheFetch	# State 0 (start)
	PC + 4imm, μ PCdisp1	# State 1
lui1:	lui(imm)	# State 7lui
	rt \leftarrow z, μ PCfetch	# State 8lui
add1:	x + y	# State 7add
	rd \leftarrow z, μ PCfetch	# State 8add
sub1:	x - y	# State 7sub
	rd \leftarrow z, μ PCfetch	# State 8sub
slt1:	x - y	# State 7slt
	rd \leftarrow z, μ PCfetch	# State 8slt
addi1:	x + imm	# State 7addi
	rt \leftarrow z, μ PCfetch	# State 8addi
slti1:	x - imm	# State 7slti
	rt \leftarrow z, μ PCfetch	# State 8slti
and1:	x \wedge y	# State 7and
	rd \leftarrow z, μ PCfetch	# State 8and
or1:	x \vee y	# State 7or
	rd \leftarrow z, μ PCfetch	# State 8or
xor1:	x \oplus y	# State 7xor
	rd \leftarrow z, μ PCfetch	# State 8xor
nor1:	x $\sim\vee$ y	# State 7nor
	rd \leftarrow z, μ PCfetch	# State 8nor
andi1:	x \wedge imm	# State 7andi
	rt \leftarrow z, μ PCfetch	# State 8andi
ori1:	x \vee imm	# State 7ori
	rt \leftarrow z, μ PCfetch	# State 8ori
xori:	x \oplus imm	# State 7xori
	rt \leftarrow z, μ PCfetch	# State 8xori
lwsw1:	x + imm, μ PCdisp2	# State 2
lw2:	CacheLoad	# State 3
	rt \leftarrow Data, μ PCfetch	# State 4
sw2:	CacheStore, μ PCfetch	# State 6
j1:	PCjump, μ PCfetch	# State 5j
jr1:	PCjreg, μ PCfetch	# State 5jr
branch1:	PCbranch, μ PCfetch	# State 5branch
jall:	PCjump, \$31 \leftarrow PC, μ PCfetch	# State 5jal
syscall1:	PCsyscall, μ PCfetch	# State 5syscall

14.6 Exception Handling

Exceptions and interrupts alter the normal program flow

Examples of exceptions (things that can go wrong):

- ALU operation leads to overflow (incorrect result is obtained)
- Opcode field holds a pattern not representing a legal operation
- Cache error-code checker deems an accessed word invalid
- Sensor signals a hazardous condition (e.g., overheating)

Exception handler is an OS program that takes care of the problem

- Derives correct result of overflowing computation, if possible
- Invalid operation may be a software-implemented instruction

Interrupts are similar, but usually have external causes (e.g., I/O)

Exception Control States

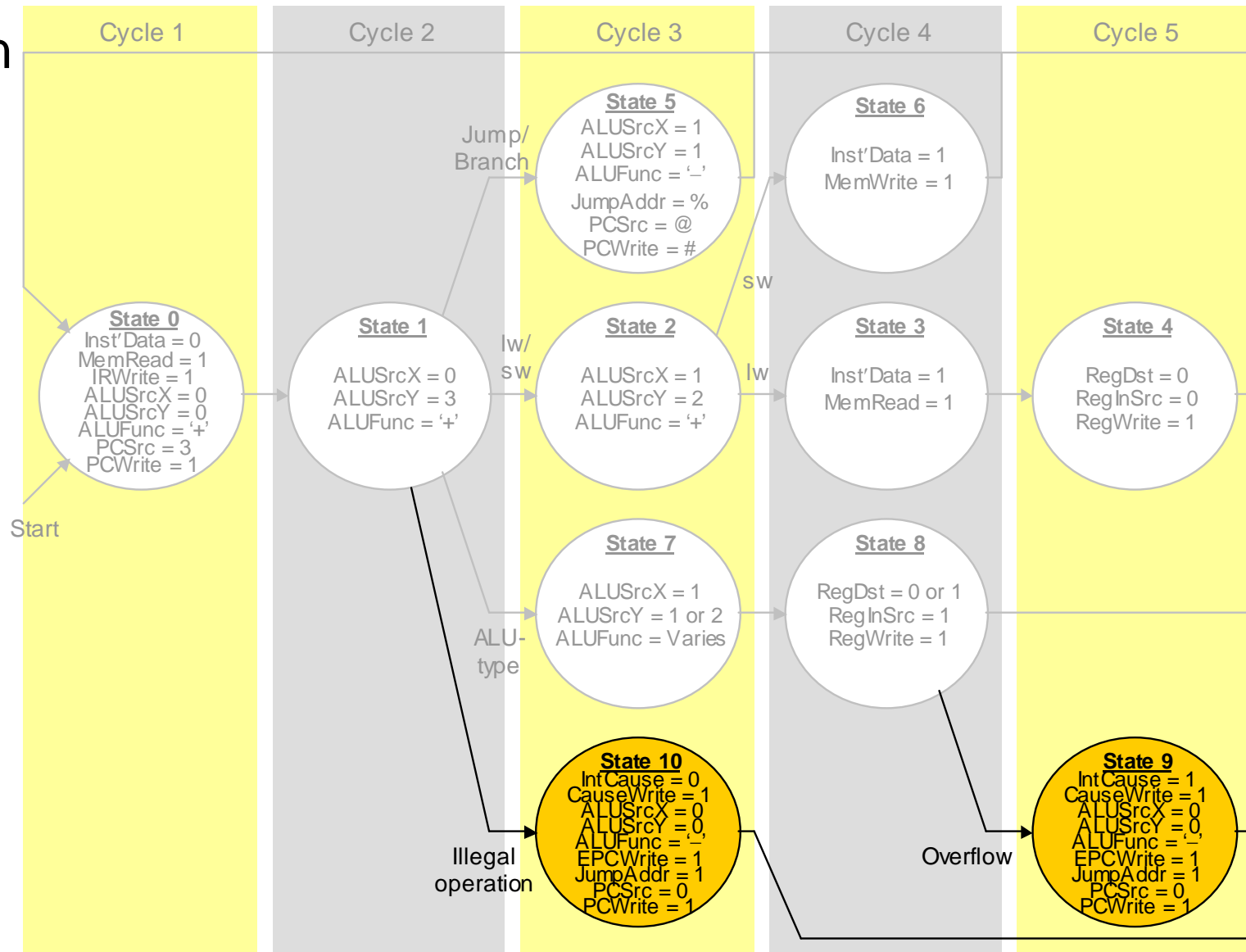


Fig. 14.10 Exception states 9 and 10 added to the control state machine.

15 Pipelined Data Paths

Pipelining is now used in even the simplest of processors

- Same principles as assembly lines in manufacturing
- Unlike in assembly lines, instructions not independent

Topics in This Chapter

15.1 Pipelining Concepts

15.2 Pipeline Stalls or Bubbles

15.3 Pipeline Timing and Performance

15.4 Pipelined Data Path Design

15.5 Pipelined Control

15.6 Optimal Pipelining



Single-Cycle Data Path of Chapter 13

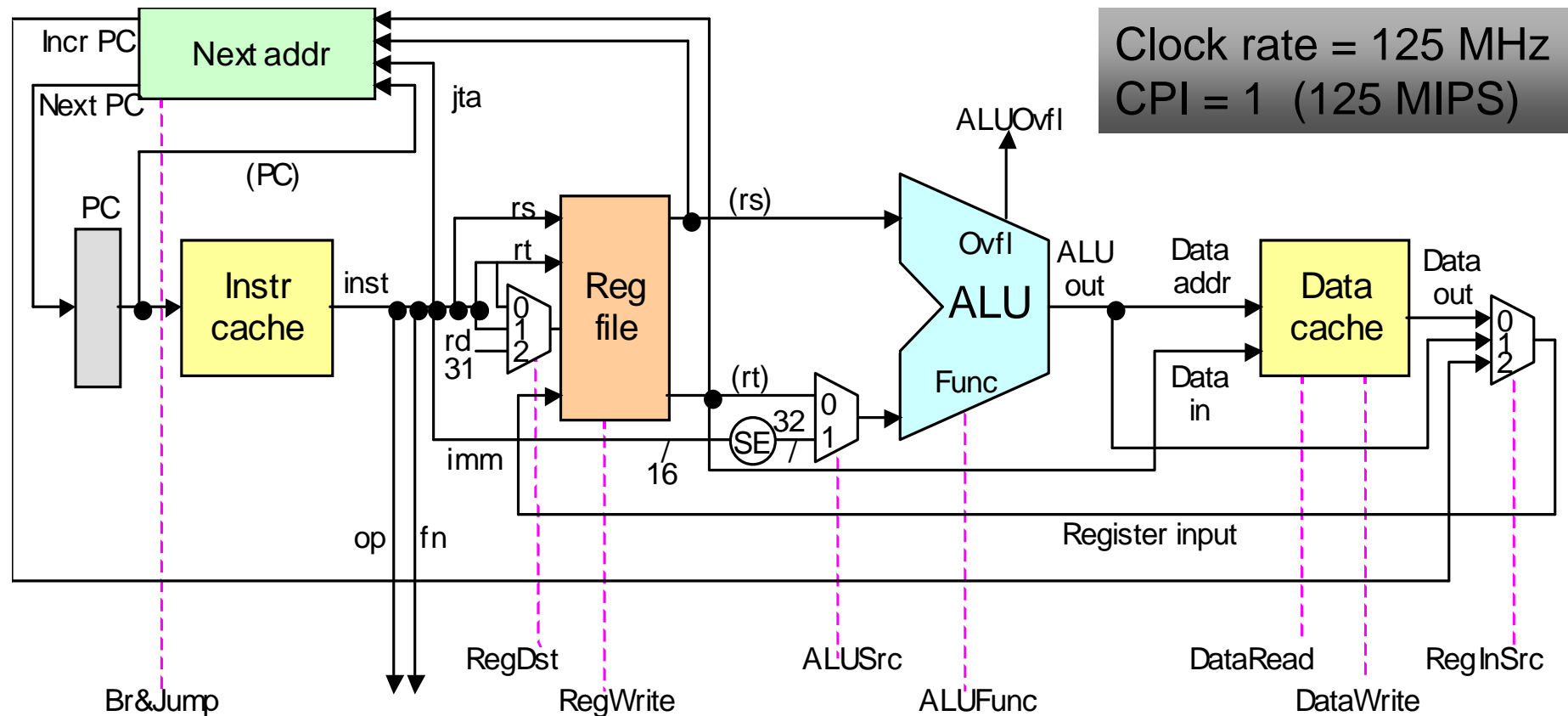


Fig. 13.3 Key elements of the single-cycle MicroMIPS data path.

Multicycle Data Path of Chapter 14

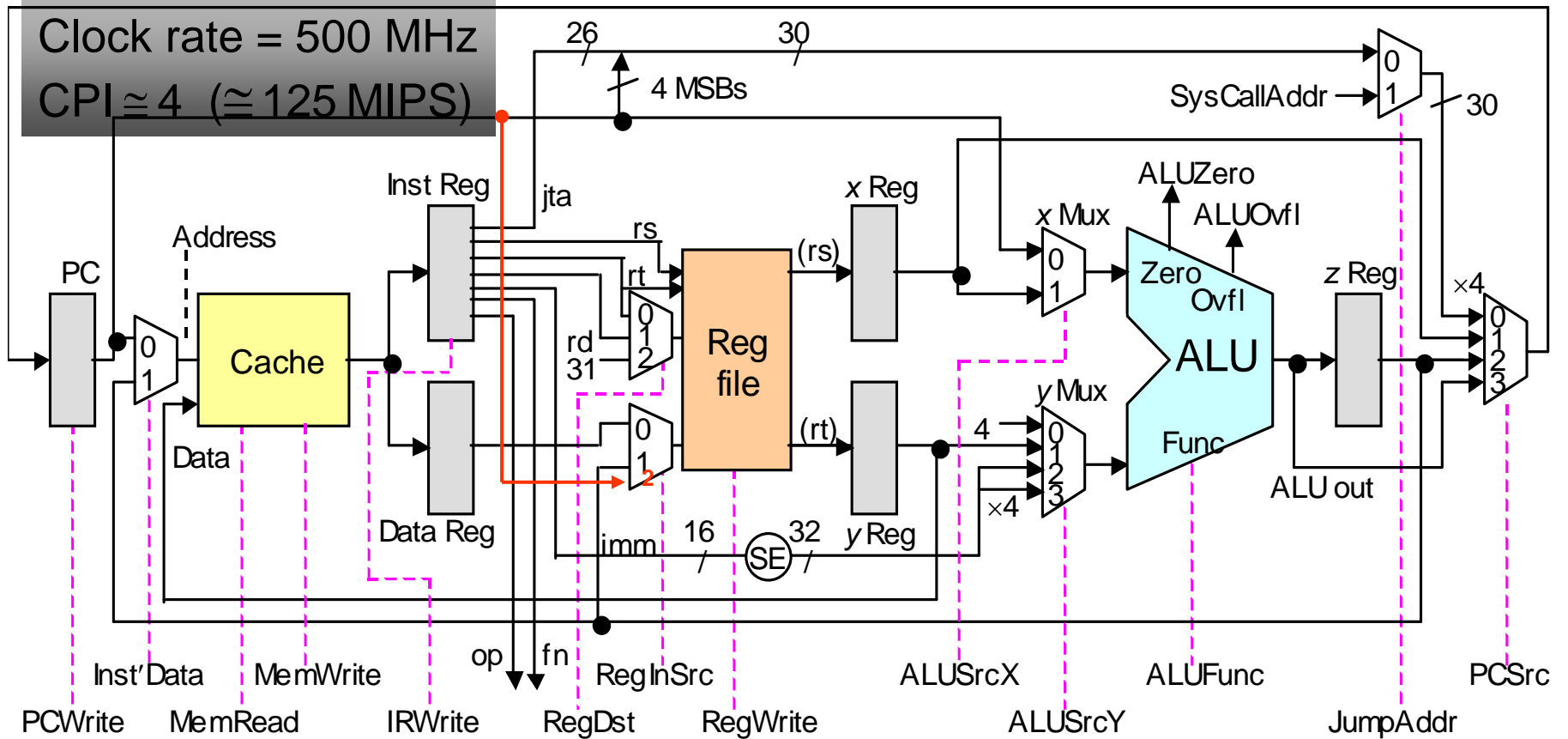
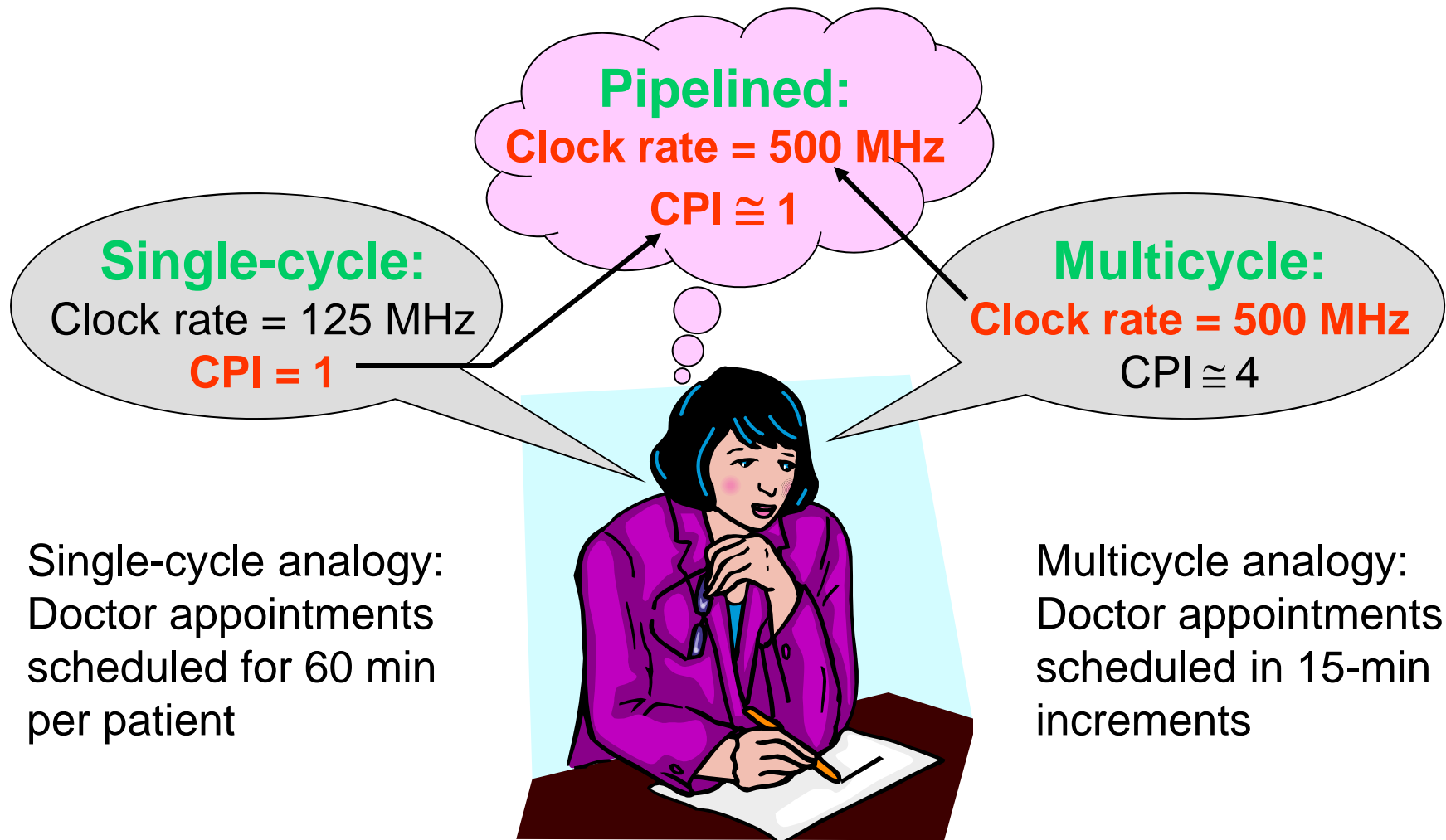


Fig. 14.3 Key elements of the multicycle MicroMIPS data path.

Getting the Best of Both Worlds



15.1 Pipelining Concepts

Strategies for improving performance

- 1 – Use multiple independent data paths accepting several instructions that are read out at once: *multiple-instruction-issue* or *superscalar*
- 2 – Overlap execution of several instructions, starting the next instruction before the previous one has run to completion: *(super)pipelined*

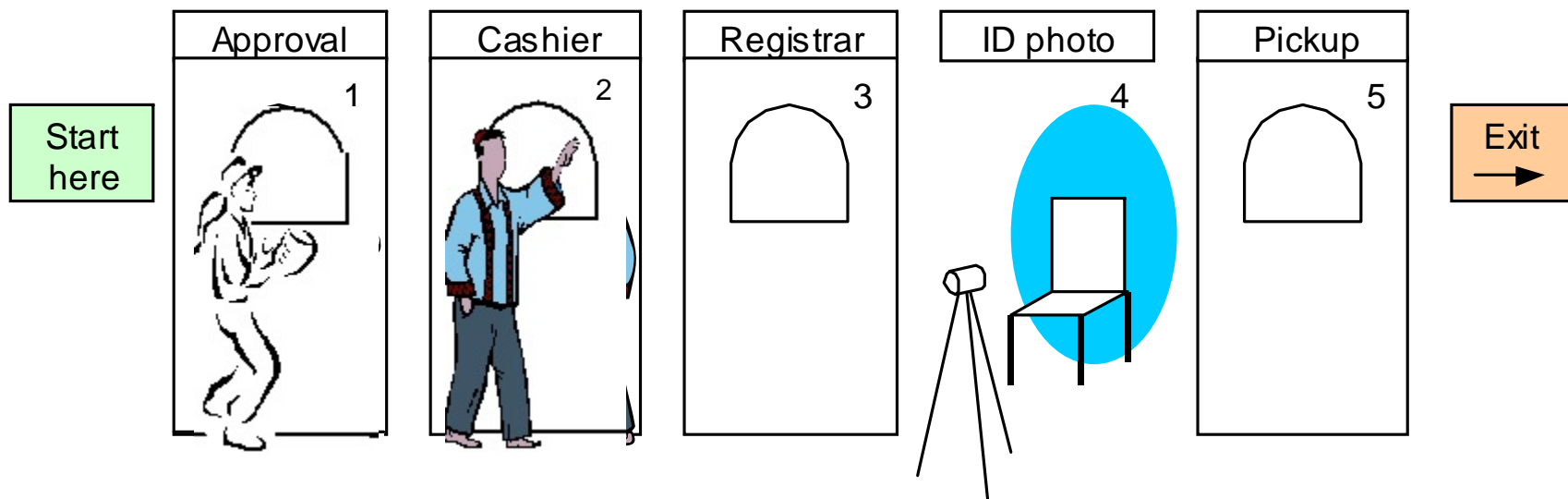


Fig. 15.1 Pipelining in the student registration process.

Pipelined Instruction Execution

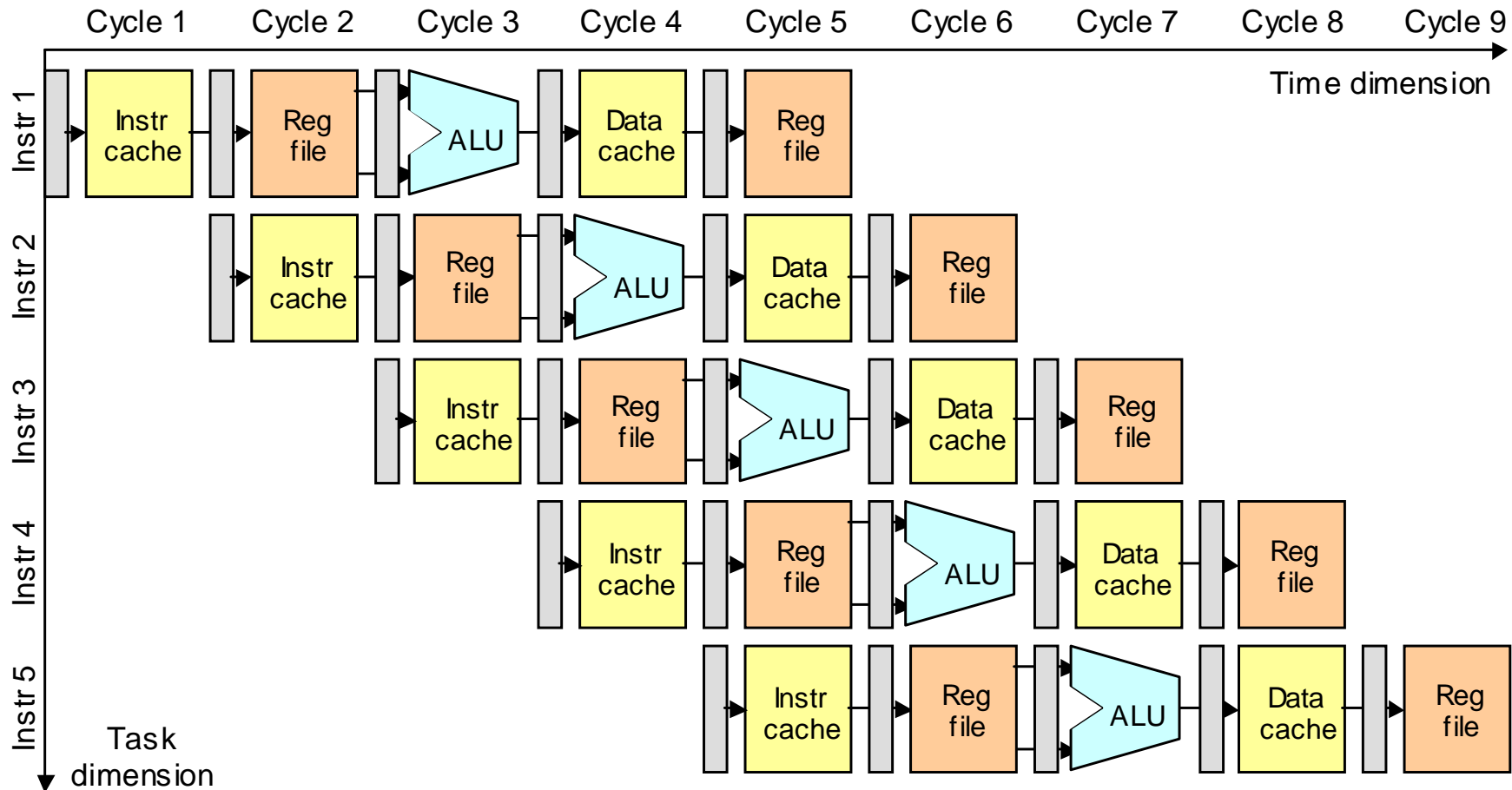
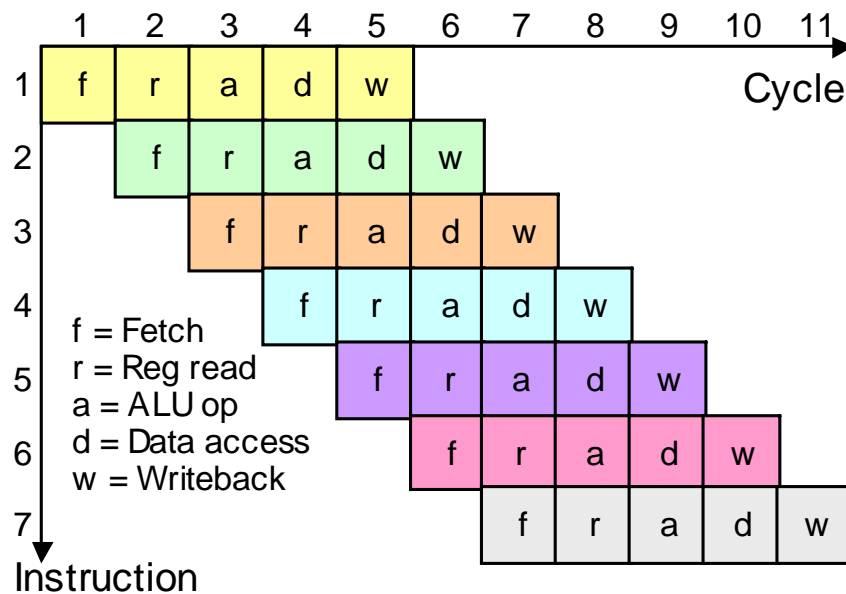


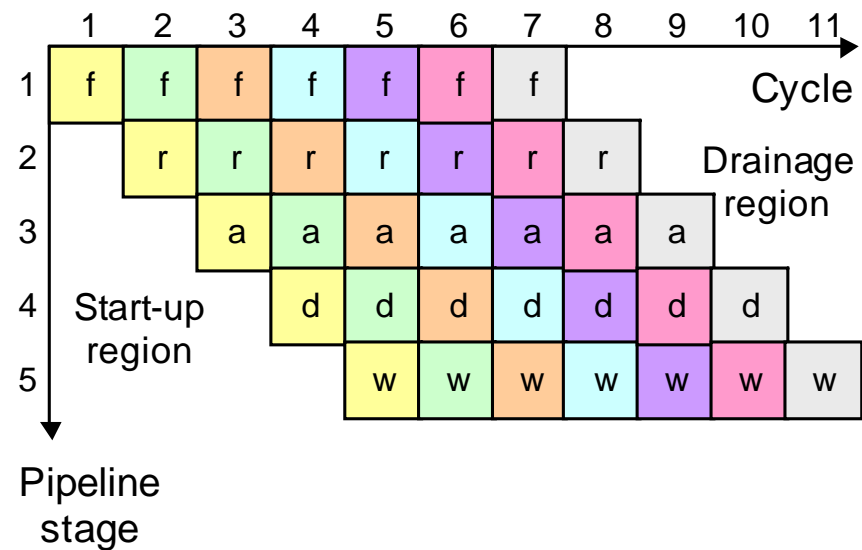
Fig. 15.2 Pipelining in the MicroMIPS instruction execution process.

Alternate Representations of a Pipeline

Except for start-up and drainage overheads, a pipeline can execute one instruction per clock tick; IPS is dictated by the clock frequency



(a) Task-time diagram



(b) Space-time diagram

Fig. 15.3 Two abstract graphical representations of a 5-stage pipeline executing 7 tasks (instructions).

Pipelining Example in a Photocopier

Example 15.1

A photocopier with an x-sheet document feeder copies the first sheet in 4 s and each subsequent sheet in 1 s. The copier's paper path is a 4-stage pipeline with each stage having a latency of 1s. The first sheet goes through all 4 pipeline stages and emerges after 4 s. Each subsequent sheet emerges 1s after the previous sheet. How does the throughput of this photocopier vary with x , assuming that loading the document feeder and removing the copies takes 15 s.

Solution

Each batch of x sheets is copied in $15 + 4 + (x - 1) = 18 + x$ seconds. A nonpipelined copier would require $4x$ seconds to copy x sheets. For $x > 6$, the pipelined version has a performance edge. When $x = 50$, the pipelining speedup is $(4 \times 50) / (18 + 50) = 2.94$.

15.2 Pipeline Stalls or Bubbles

First type of data dependency

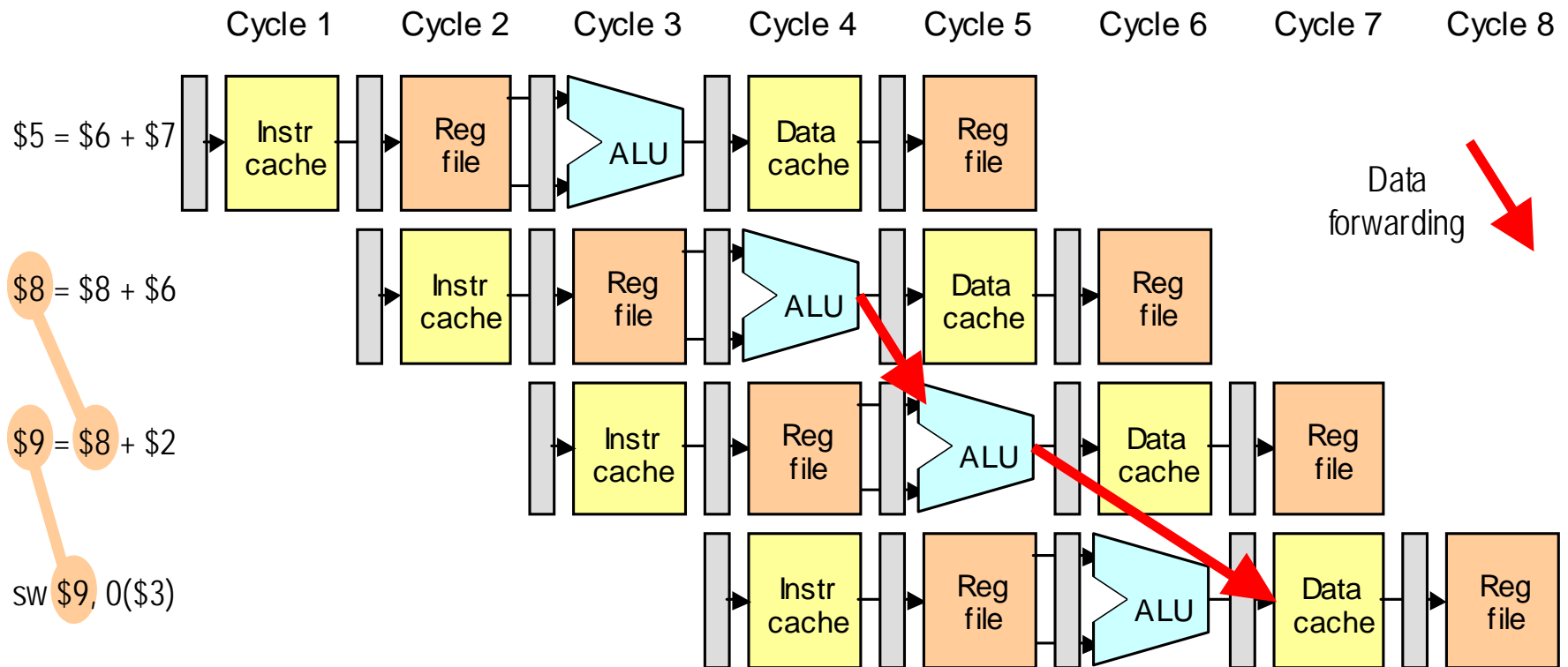
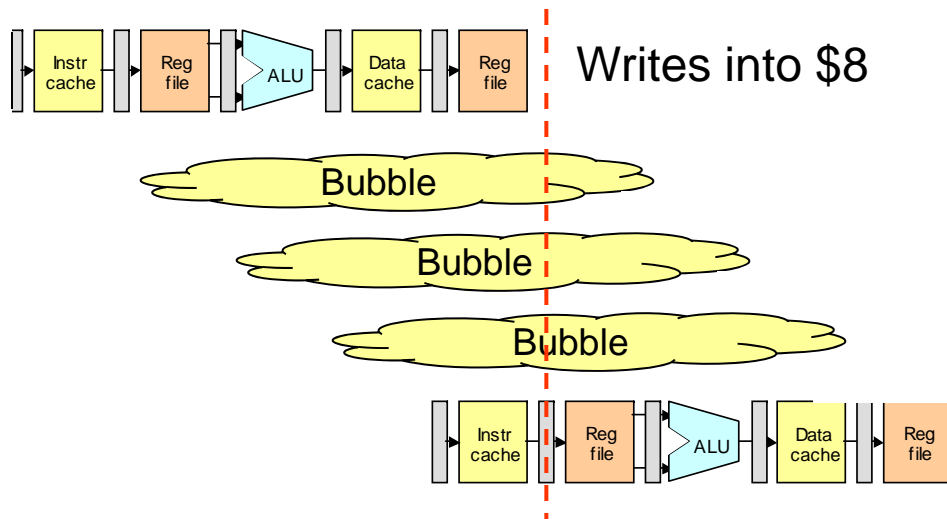
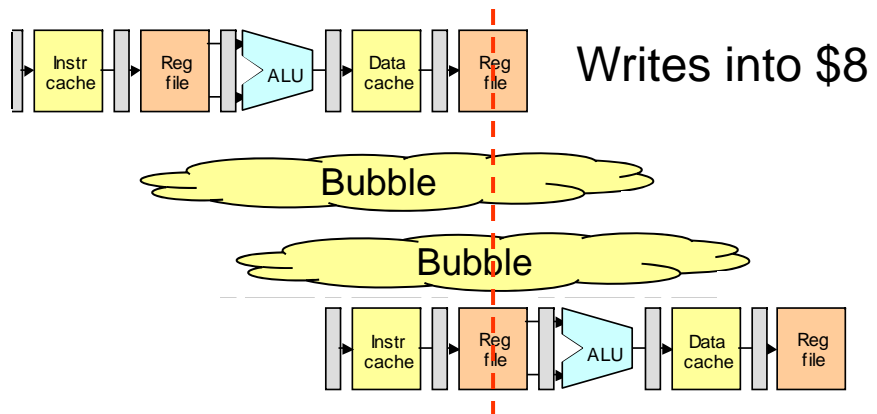


Fig. 15.4 Read-after-write data dependency and its possible resolution through data forwarding .

Inserting Bubbles in a Pipeline

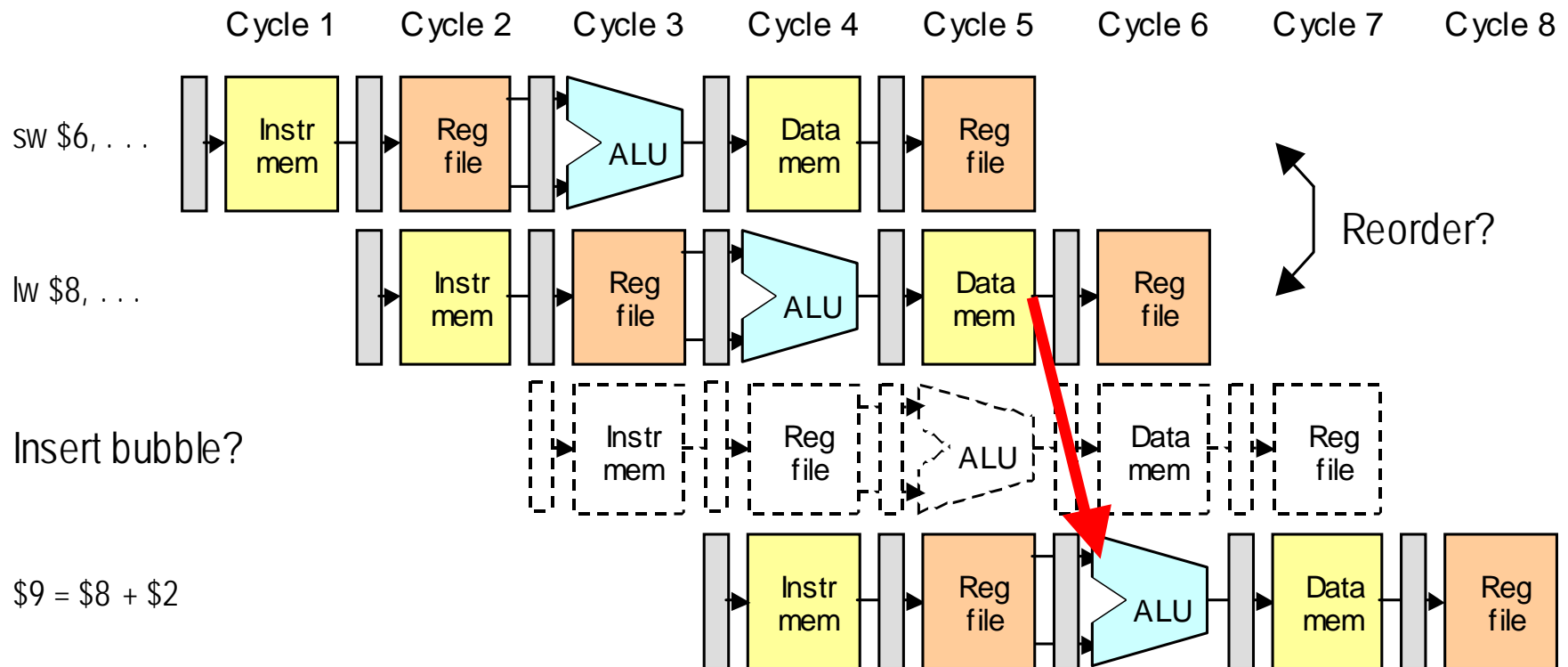


Without data forwarding, three bubbles are needed to resolve a read-after-write data dependency



Two bubbles, if we assume that a register can be updated and read from in one cycle

Second Type of Data Dependency



Without data forwarding, three (two) bubbles are needed to resolve a read-after-load data dependency

Fig. 15.5 Read-after-load data dependency and its possible resolution through bubble insertion and data forwarding.

Control Dependency in a Pipeline

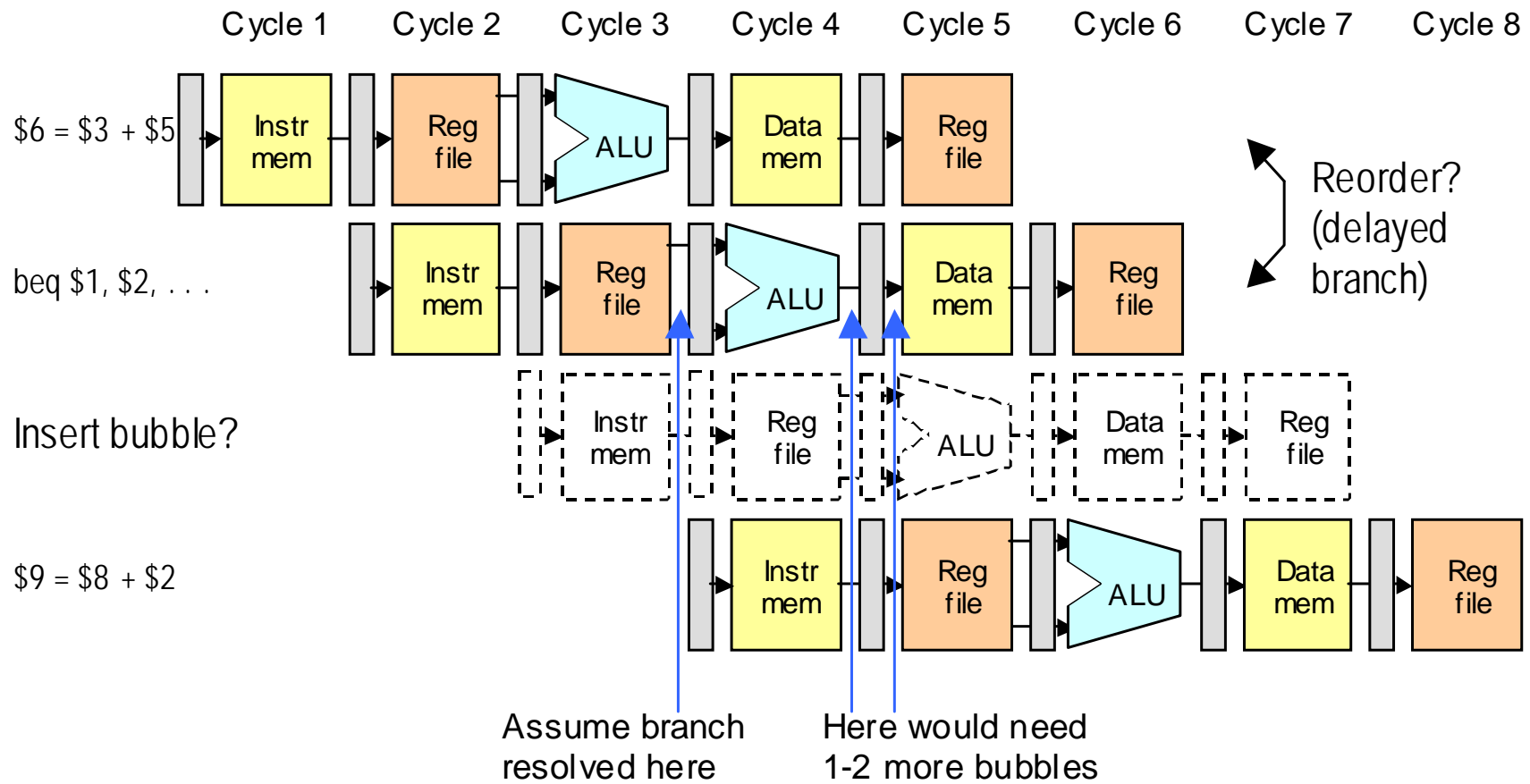


Fig. 15.6 Control dependency due to conditional branch.

15.3 Pipeline Timing and Performance

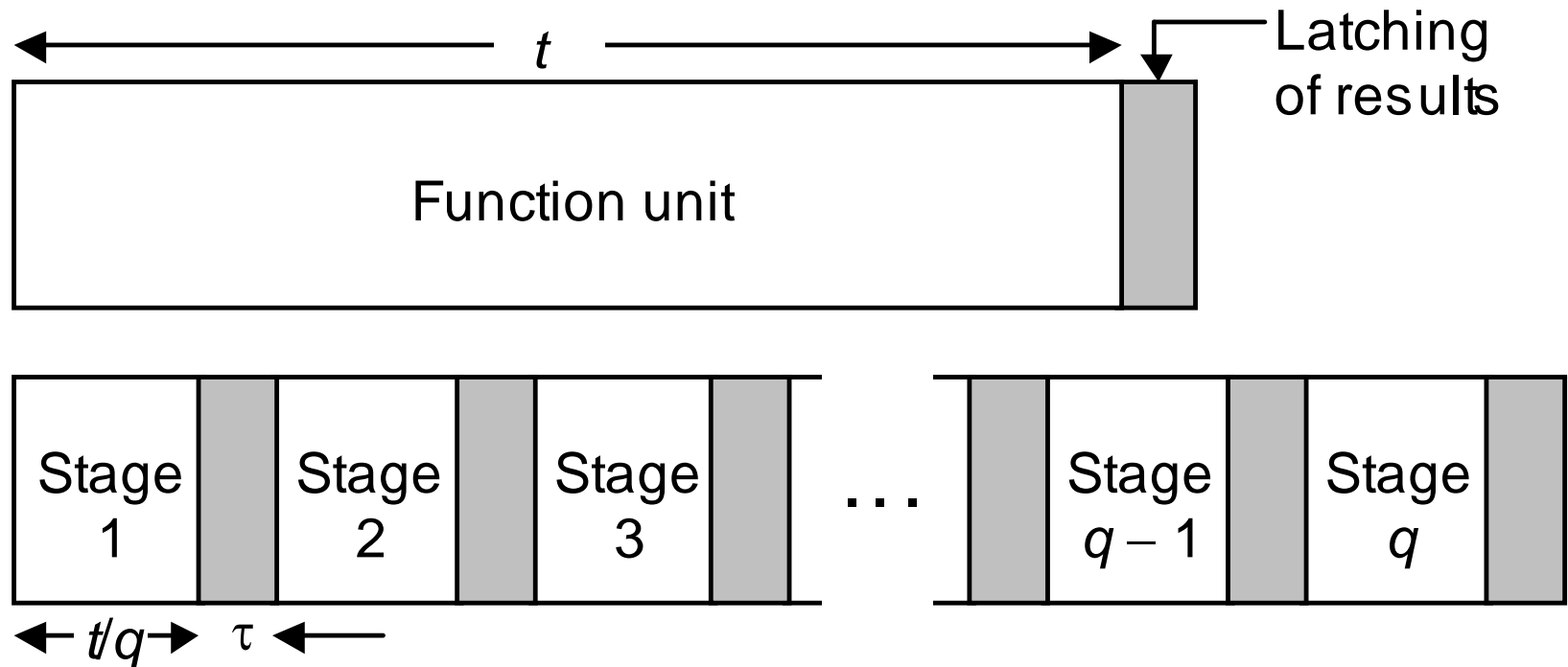
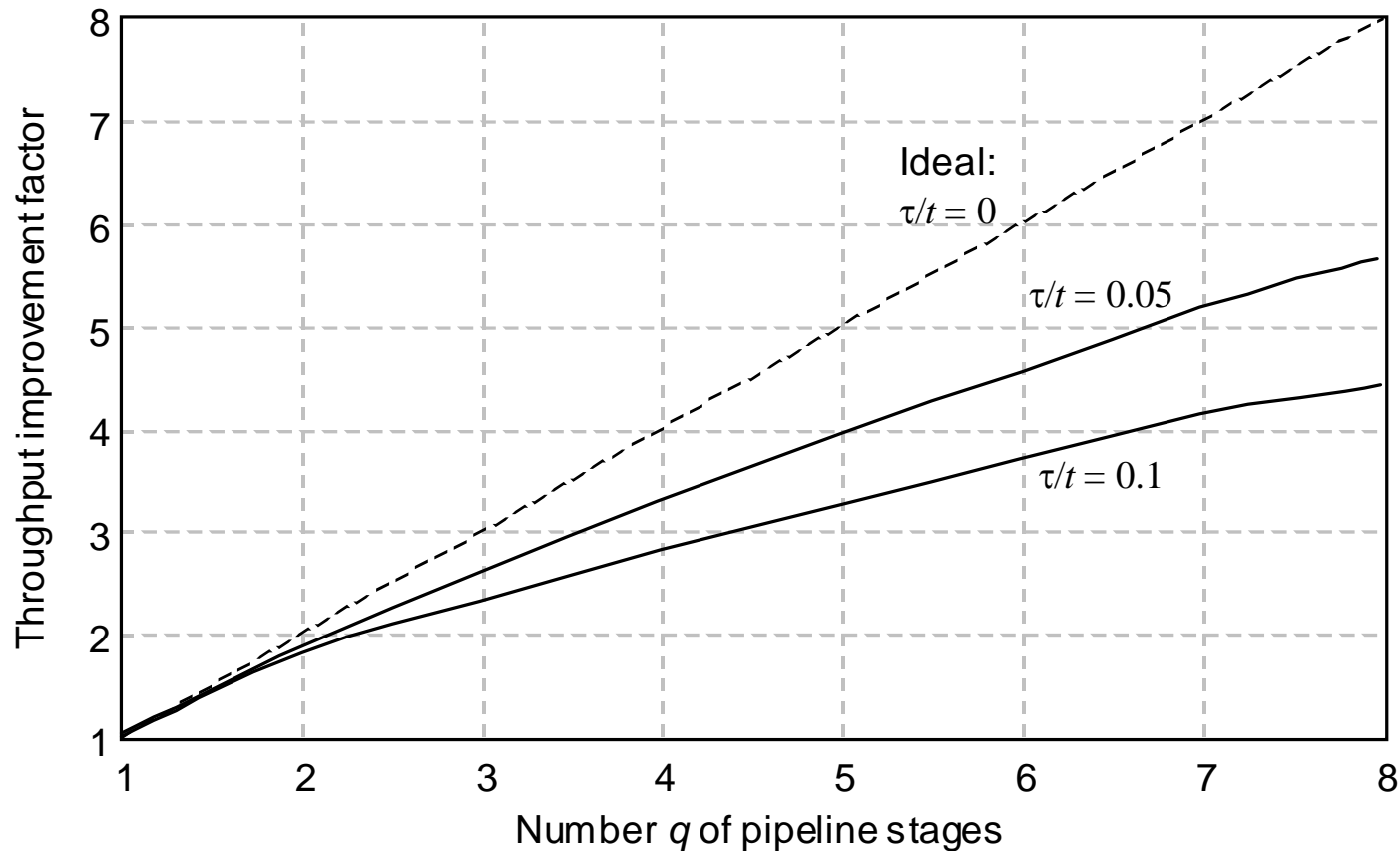


Fig. 15.7 Pipelined form of a function unit with latching overhead.

Throughput Increase in a q -Stage Pipeline



$$\frac{t}{t/q + \tau}$$

or

$$\frac{q}{1 + q\tau/t}$$

Fig. 15.8 Throughput improvement due to pipelining as a function of the number of pipeline stages for different pipelining overheads.

Pipeline Throughput with Dependencies

Assume that one bubble must be inserted due to read-after-load dependency and after a branch when its delay slot cannot be filled. Let β be the fraction of all instructions that are followed by a bubble.

$$\text{Pipeline speedup} = \frac{q}{(1 + q\tau/t)(1 + \beta)}$$

Effective
CPI

R-type	44%
Load	24%
Store	12%
Branch	18%
Jump	2%

Example 15.3

Calculate the effective CPI for MicroMIPS, assuming that a quarter of branch and load instructions are followed by bubbles.

Solution

Fraction of bubbles $\beta = 0.25(0.24 + 0.18) = 0.105$

CPI = $1 + \beta = 1.105$ (which is very close to the ideal value of 1)

15.4 Pipelined Data Path Design

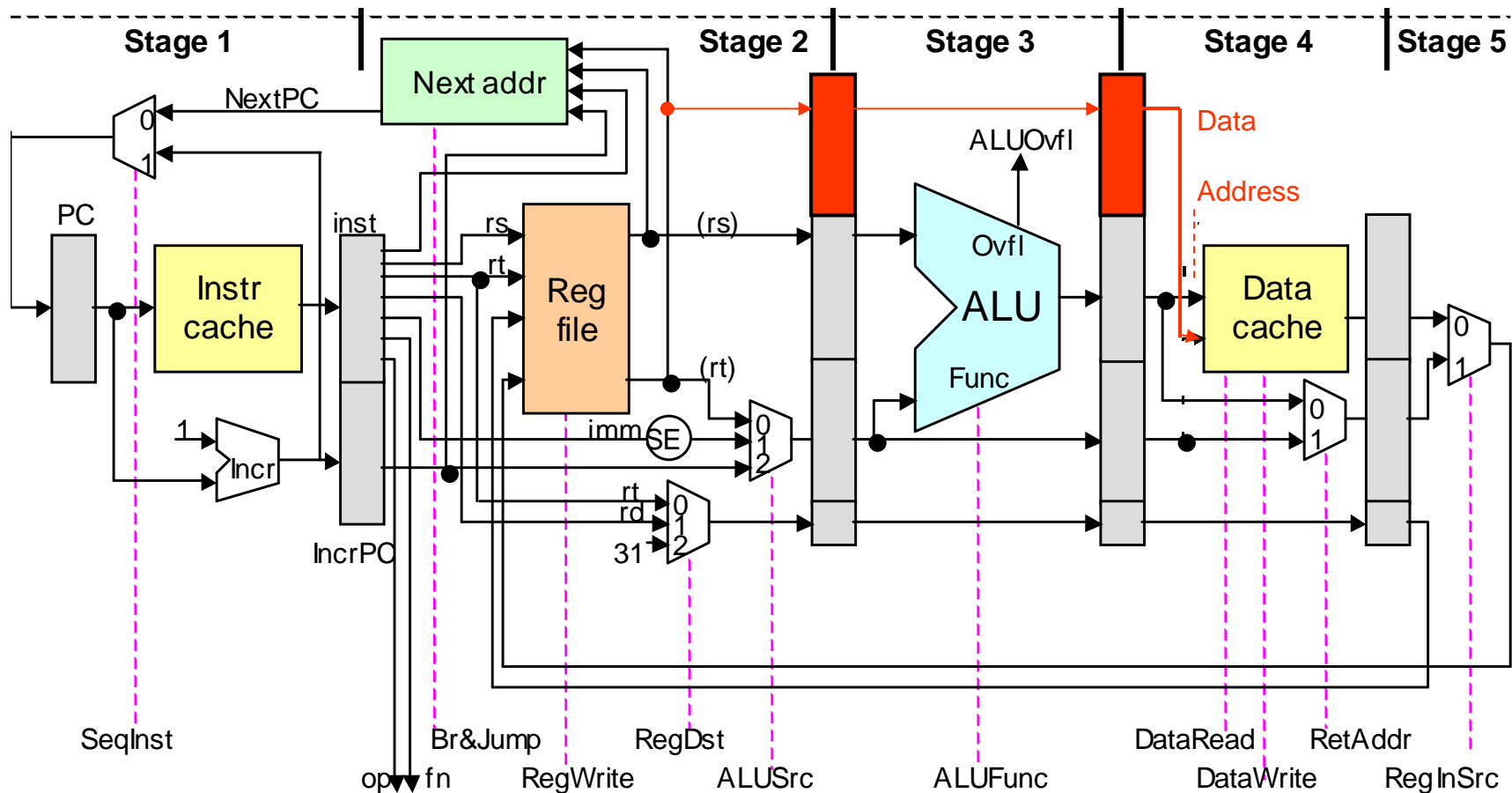


Fig. 15.9 Key elements of the pipelined MicroMIPS data path.

15.5 Pipelined Control

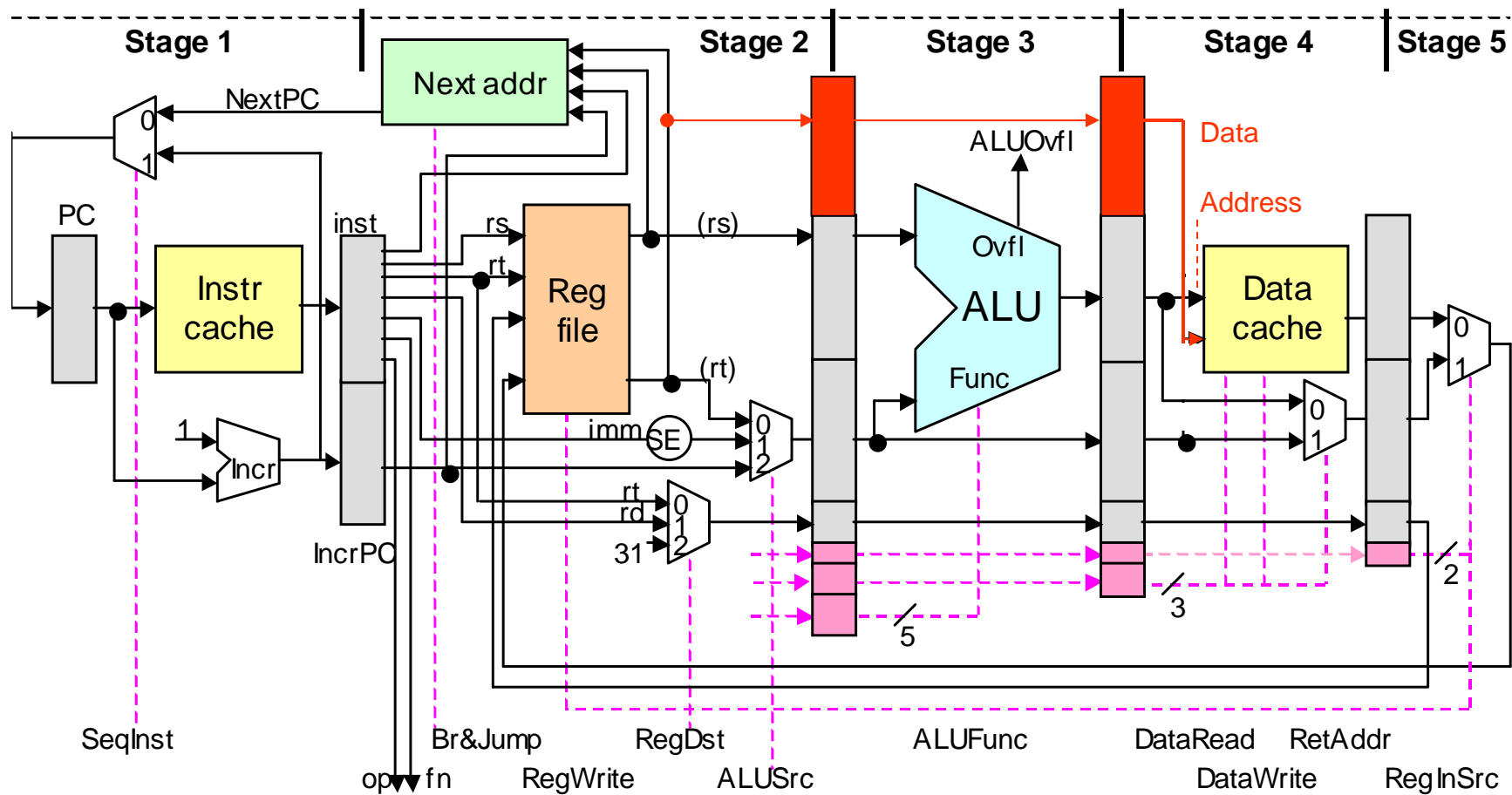


Fig. 15.10 Pipelined control signals.

15.6 Optimal Pipelining

MicroMIPS pipeline with more than four-fold improvement

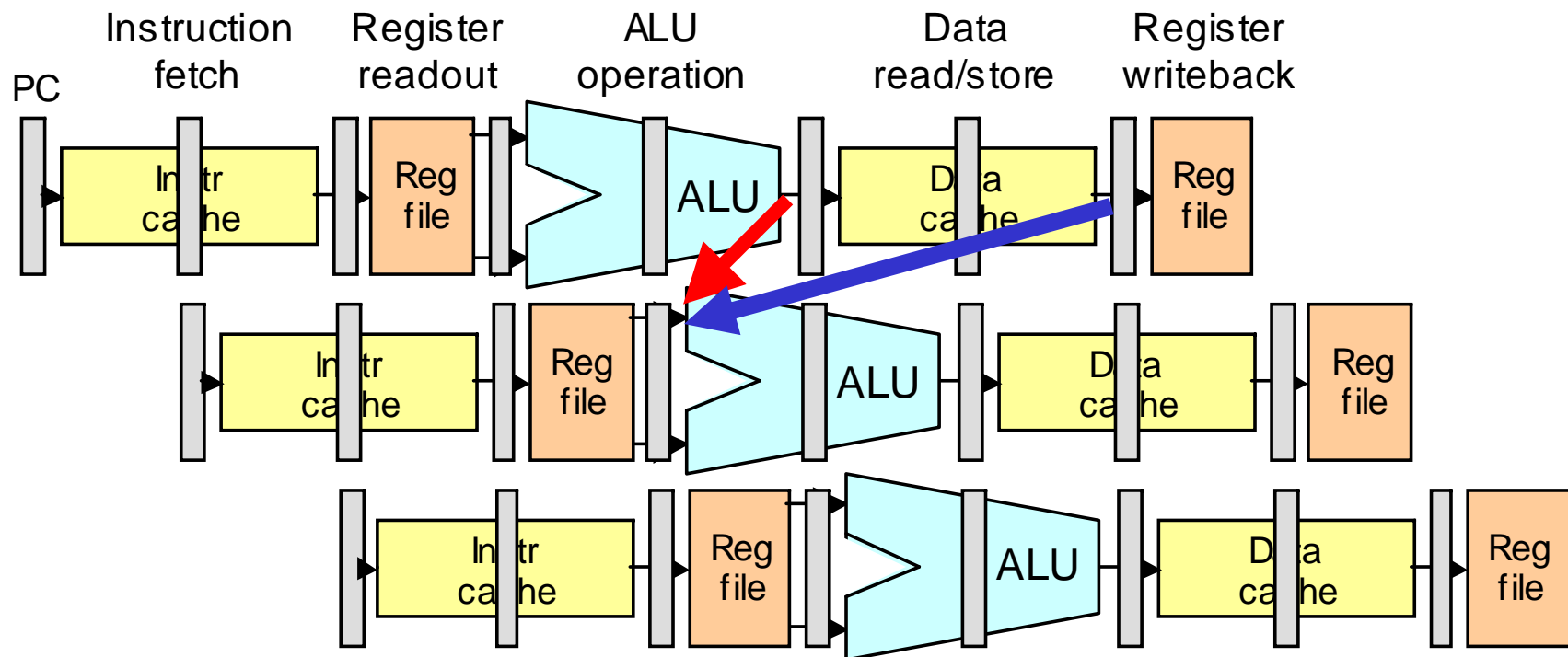


Fig. 15.11 Higher-throughput pipelined data path for MicroMIPS and the execution of consecutive instructions in it .

Optimal Number of Pipeline Stages

Assumptions:

Pipeline sliced into q stages
 Stage overhead is τ
 $q/2$ bubbles per branch
 (decision made midway)
 Fraction b of all instructions
 are taken branches

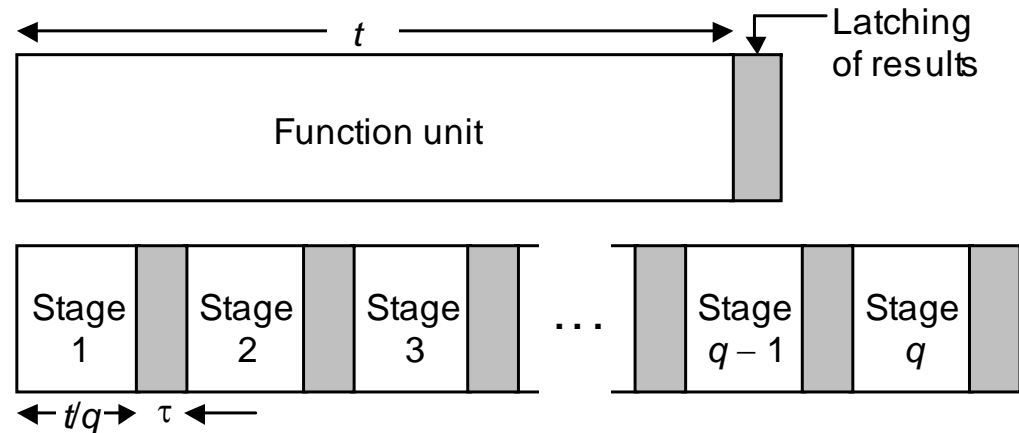


Fig. 15.7 Pipelined form of a function unit with latching overhead.

Derivation of q^{opt}

Average CPI = $1 + bq/2$

Throughput = Clock rate / CPI = $\frac{1}{(t/q + \tau)(1 + bq/2)}$

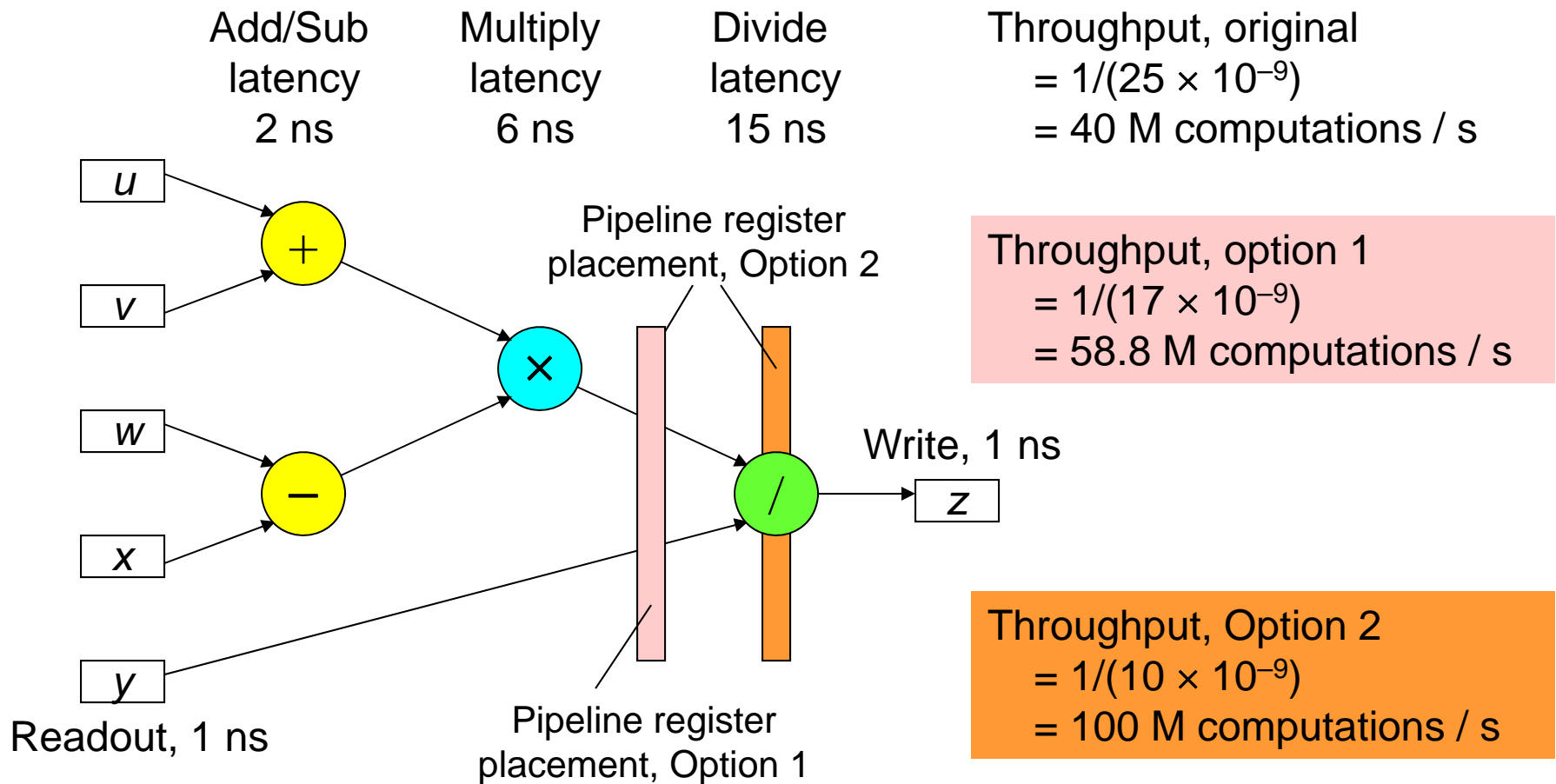
Differentiate throughput expression with respect to q and equate with 0

$$q^{\text{opt}} = \sqrt{\frac{2t/\tau}{b}}$$

Varies directly with t/τ and inversely with b

Pipelining Example

An example combinational-logic data path to compute $z := (u + v)(w - x) / y$



16 Pipeline Performance Limits

Pipeline performance limited by data & control dependencies

- Hardware provisions: data forwarding, branch prediction
- Software remedies: delayed branch, instruction reordering

Topics in This Chapter

16.1 Data Dependencies and Hazards

16.2 Data Forwarding

16.3 Pipeline Branch Hazards

16.4 Delayed Branch and Branch Prediction

16.5 Dealing with Exceptions

16.6 Advanced Pipelining

16.1 Data Dependencies and Hazards

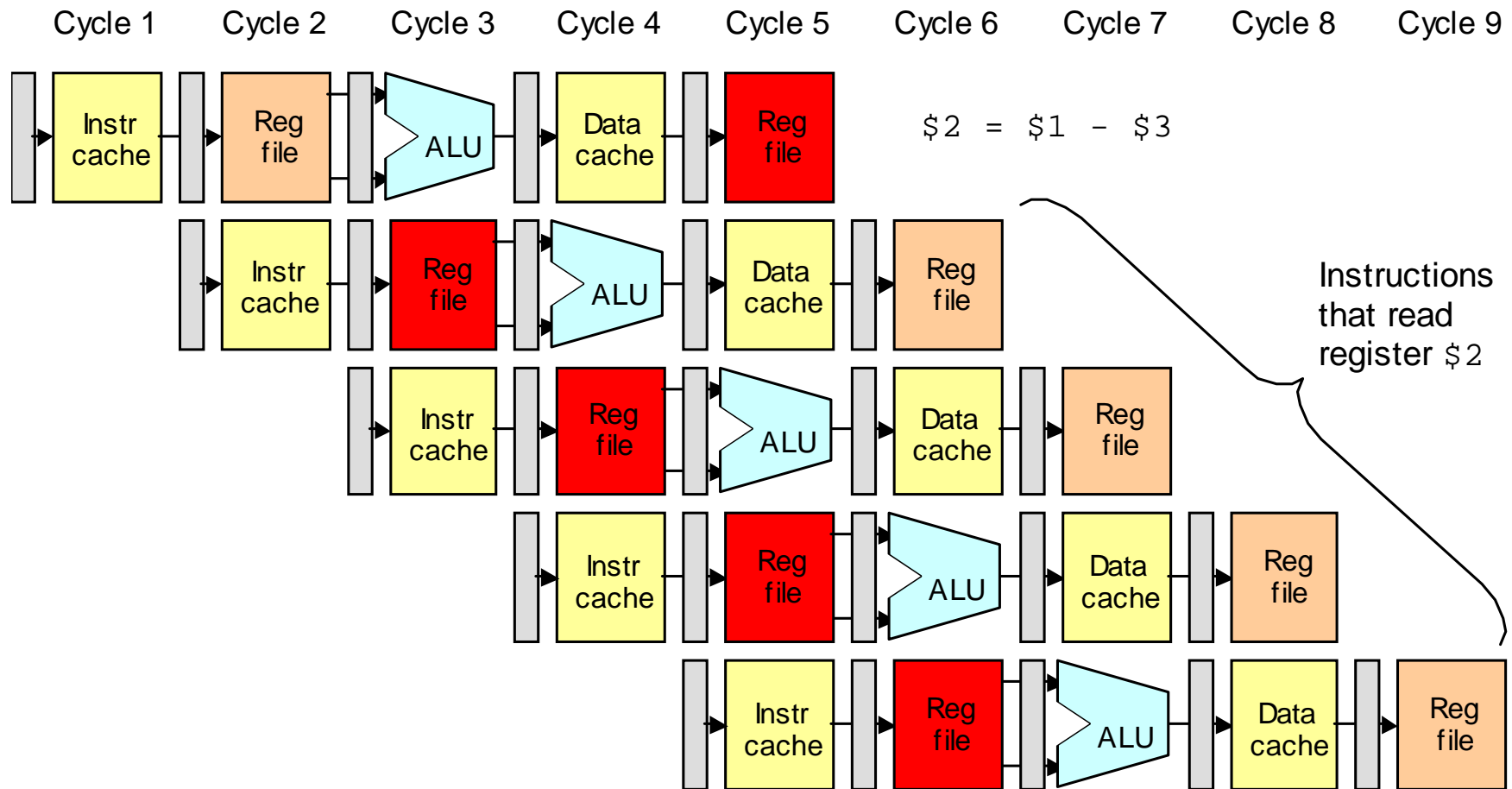


Fig. 16.1 Data dependency in a pipeline.

Resolving Data Dependencies via Forwarding

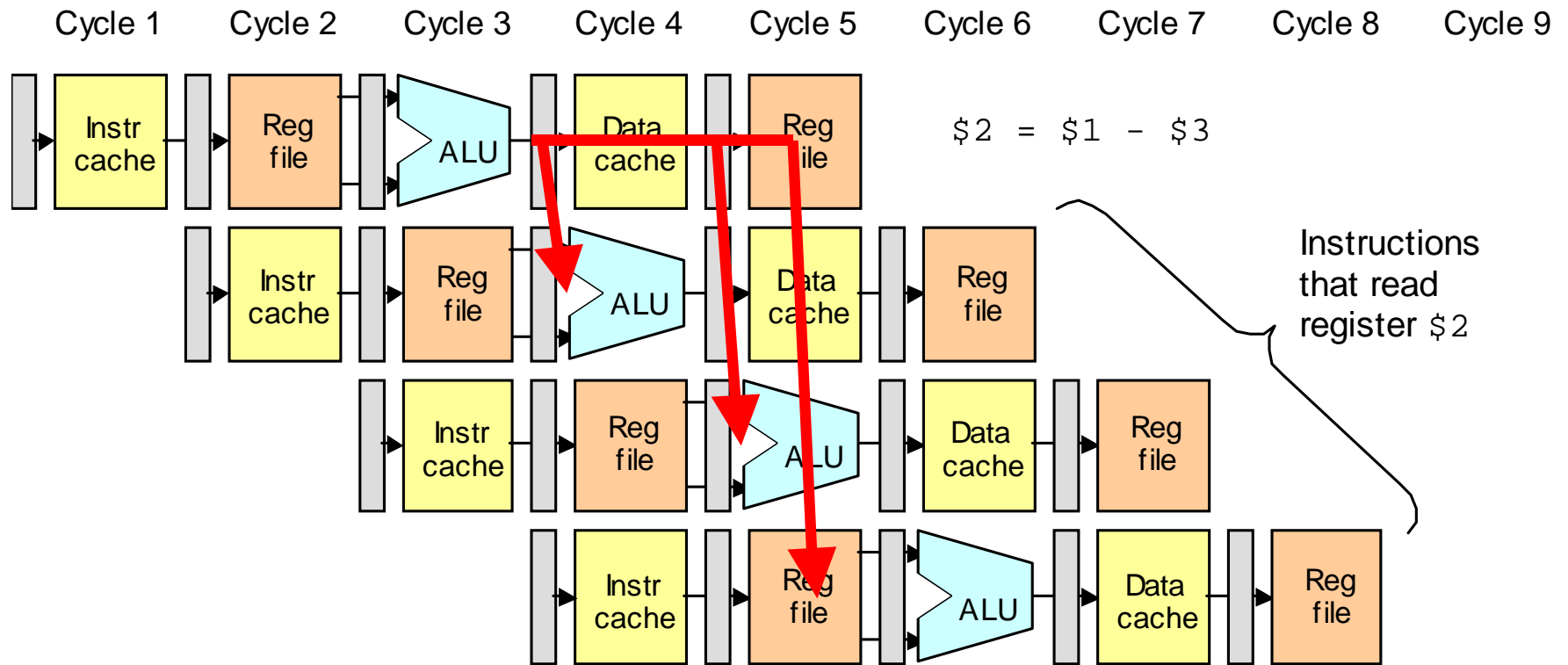


Fig. 16.2 When a previous instruction writes back a value computed by the ALU into a register, the data dependency can always be resolved through forwarding.

Pipelined MicroMIPS – Repeated for Reference

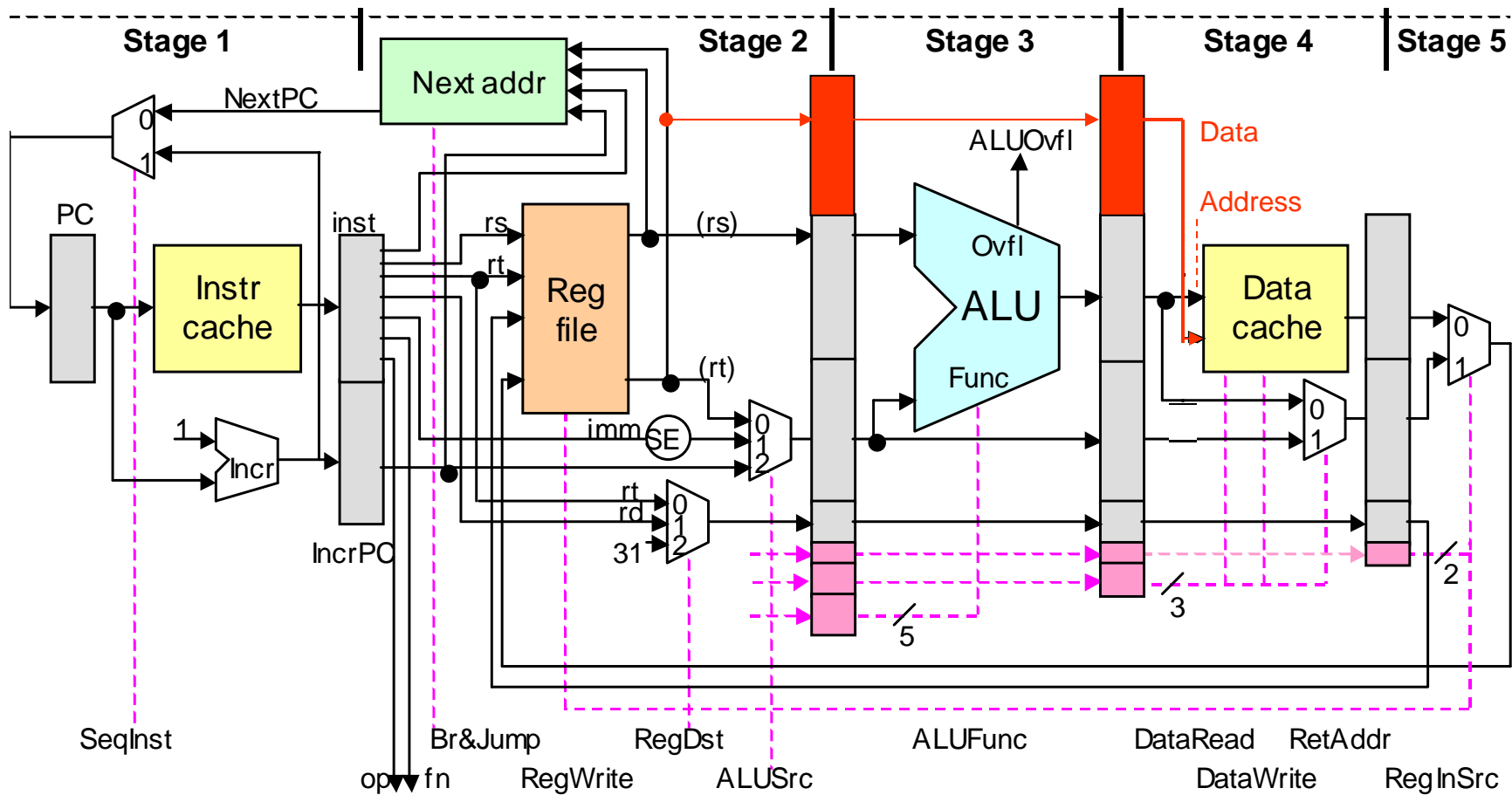


Fig. 15.10 Pipelined control signals.

Certain Data Dependencies Lead to Bubbles

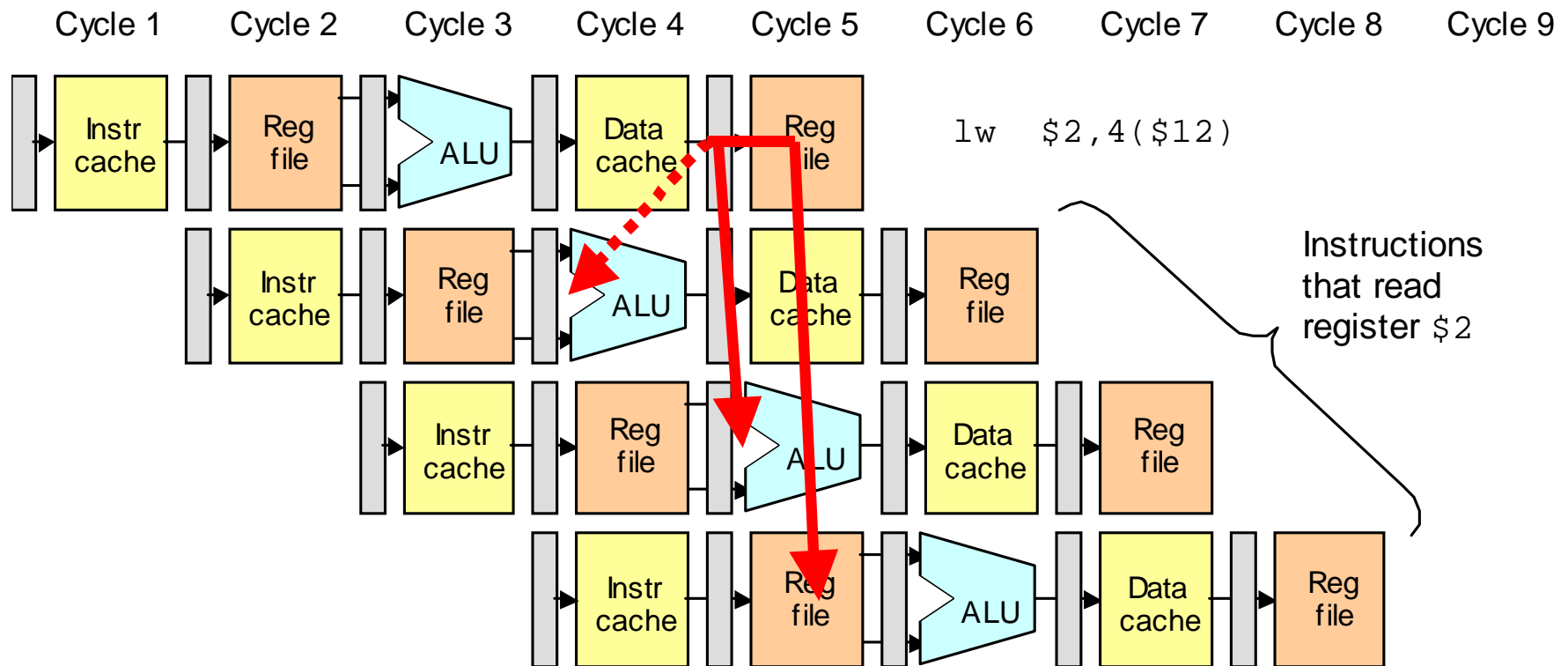


Fig. 16.3 When the immediately preceding instruction writes a value read out from the data memory into a register, the data dependency cannot be resolved through forwarding (i.e., we cannot go back in time) and a bubble must be inserted in the pipeline.

16.2 Data Forwarding

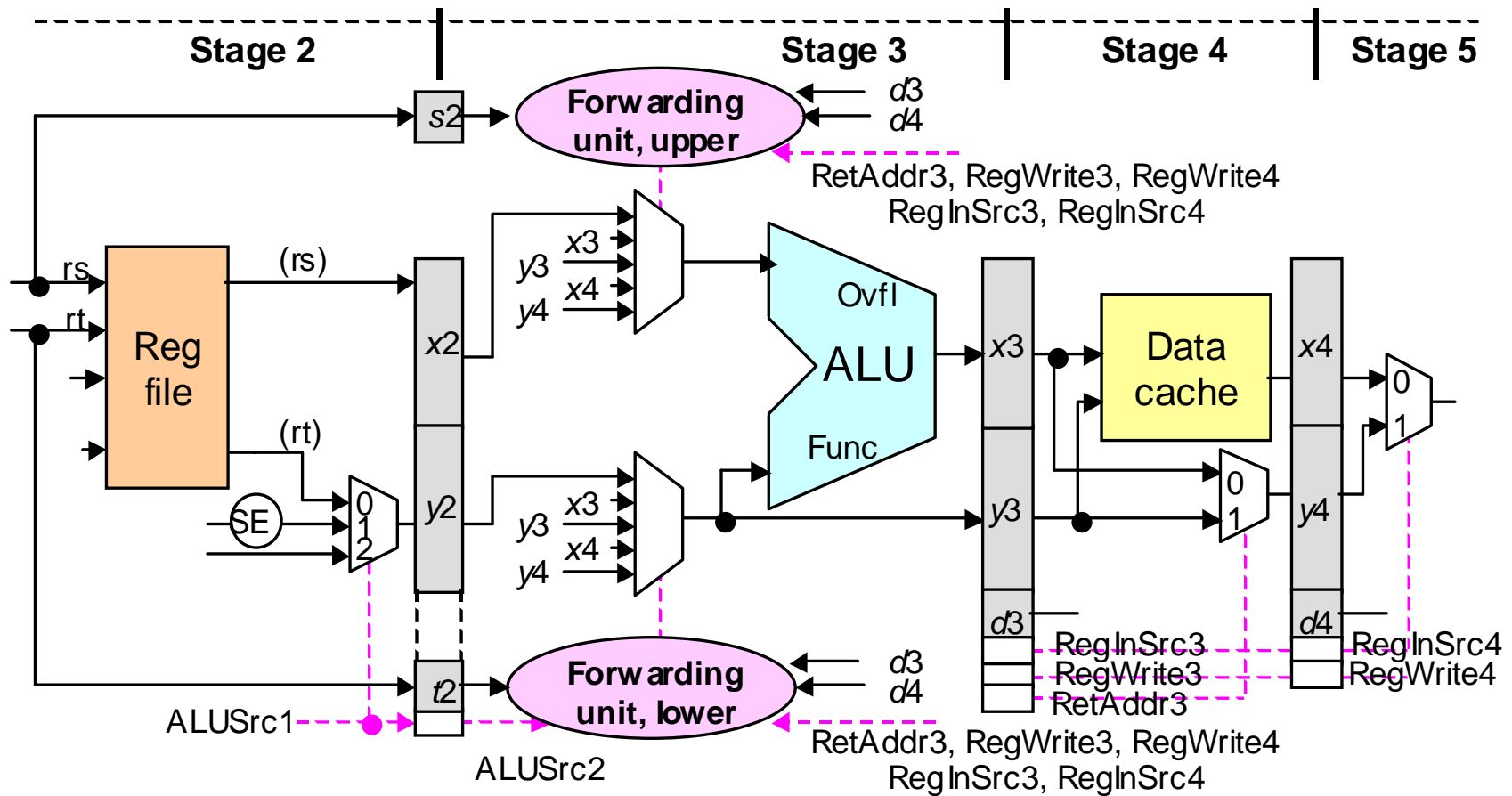


Fig. 16.4 Forwarding unit for the pipelined MicroMIPS data path.

Design of the Data Forwarding Units

Let's focus on designing the upper data forwarding unit

Table 16.1 Partial truth table for the upper forwarding unit in the pipelined MicroMIPS data path.

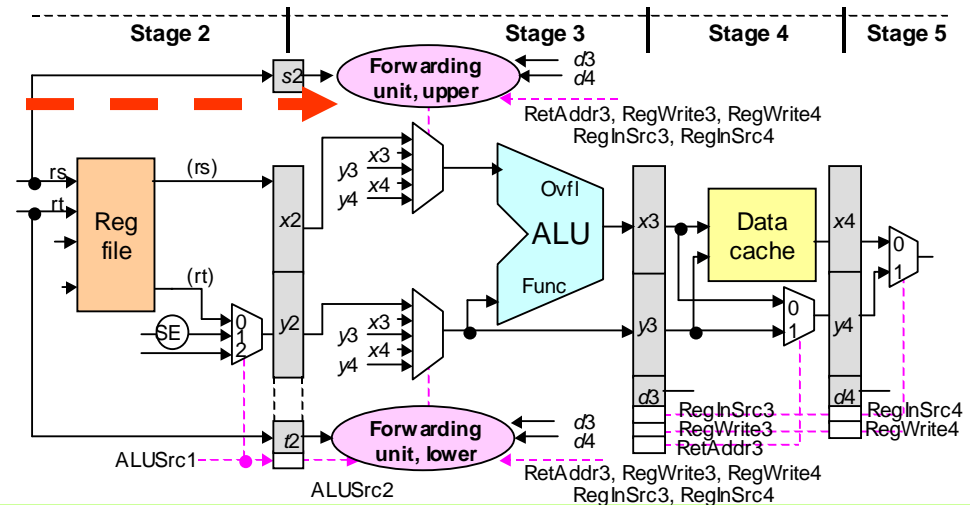


Fig. 16.4 Forwarding unit for the pipelined MicroMIPS data path.

RegWrite3	RegWrite4	s2matchesd3	s2matchesd4	RetAddr3	RegInSrc3	RegInSrc4	Choose
0	0	x	x	x	x	x	x2
0	1	x	0	x	x	x	x2
0	1	x	1	x	x	0	x4
0	1	x	1	x	x	1	y4
1	0	1	x	0	1	x	x3
1	0	1	x	1	1	x	y3
1	1	1	1	0	1	x	x3

Incorrect in textbook

Hardware for Inserting Bubbles

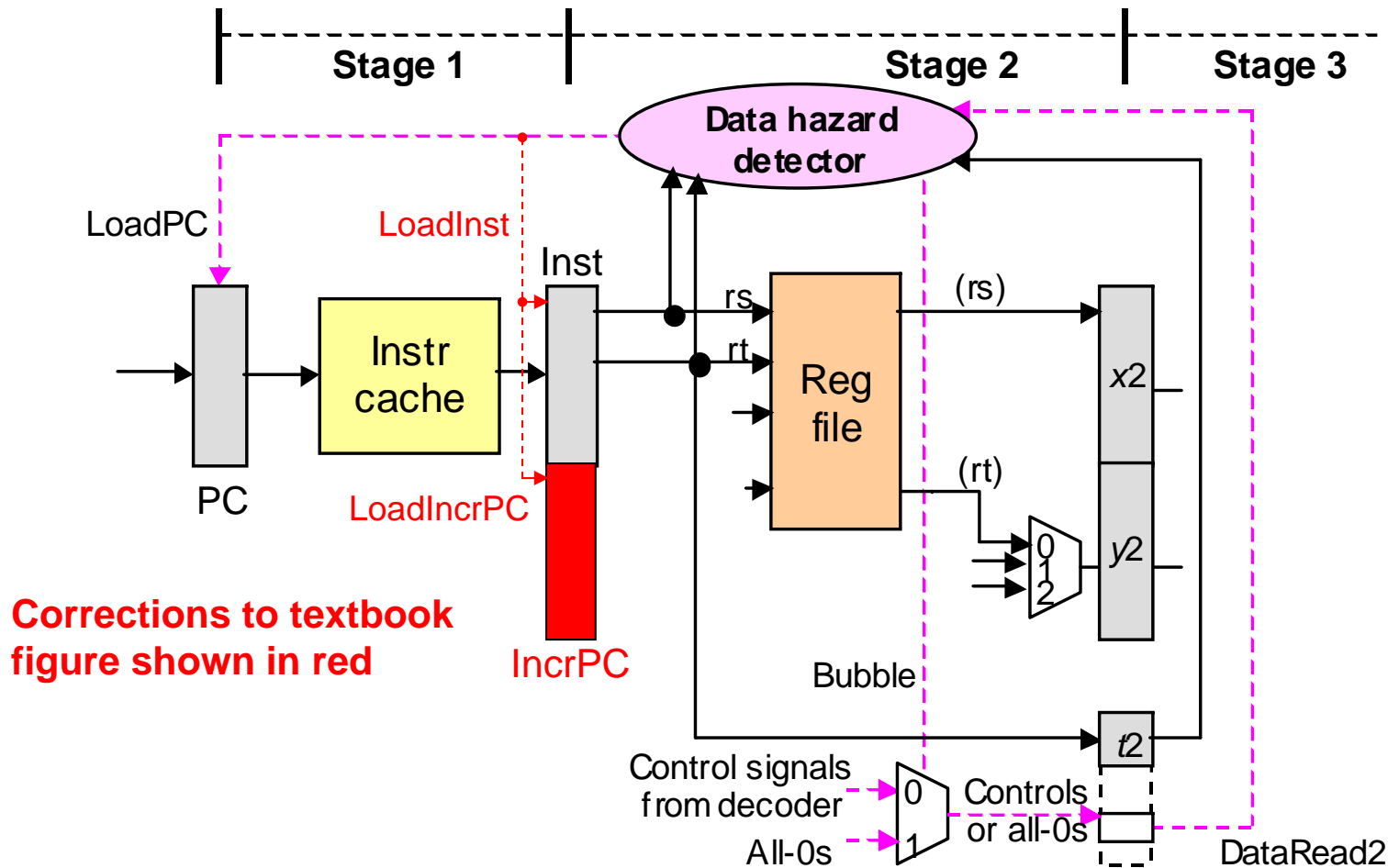


Fig. 16.5 Data hazard detector for the pipelined MicroMIPS data path.

Augmentations to Pipelined Data Path and Control

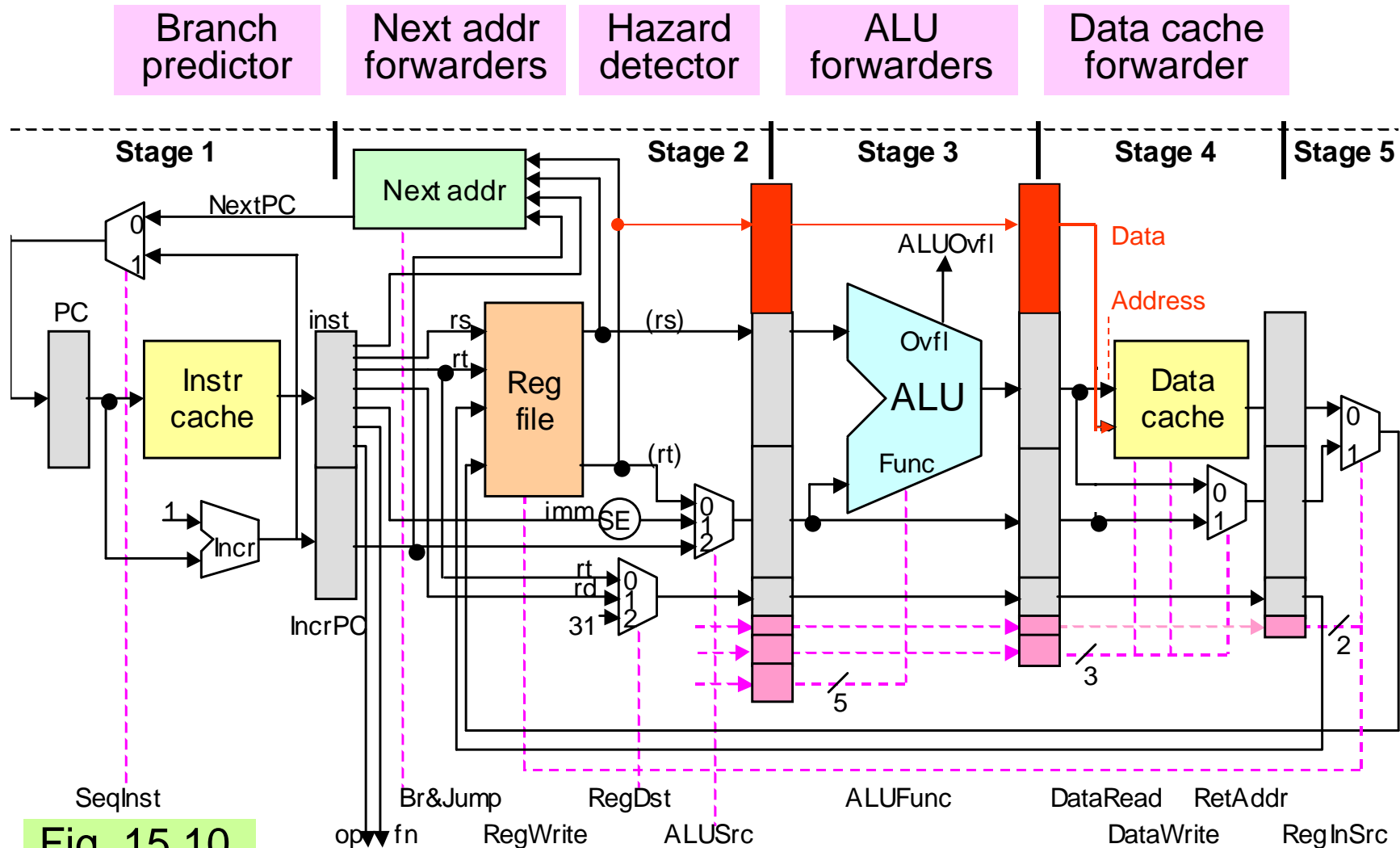


Fig. 15.10

16.3 Pipeline Branch Hazards

Software-based solutions

Compiler inserts a “no-op” after every branch (simple, but wasteful)

Branch is redefined to take effect after the instruction that follows it

Branch delay slot(s) are filled with useful instructions via reordering

Hardware-based solutions

Mechanism similar to data hazard detector to flush the pipeline

Constitutes a rudimentary form of branch prediction:

Always predict that the branch is not taken, flush if mistaken

More elaborate branch prediction strategies possible

16.4 Branch Prediction

Predicting whether a branch will be taken

- Always predict that the branch will not be taken
- Use program context to decide (backward branch is likely taken, forward branch is likely not taken)
- Allow programmer or compiler to supply clues
- Decide based on past history (maintain a small history table); to be discussed later
- Apply a combination of factors: modern processors use elaborate techniques due to deep pipelines

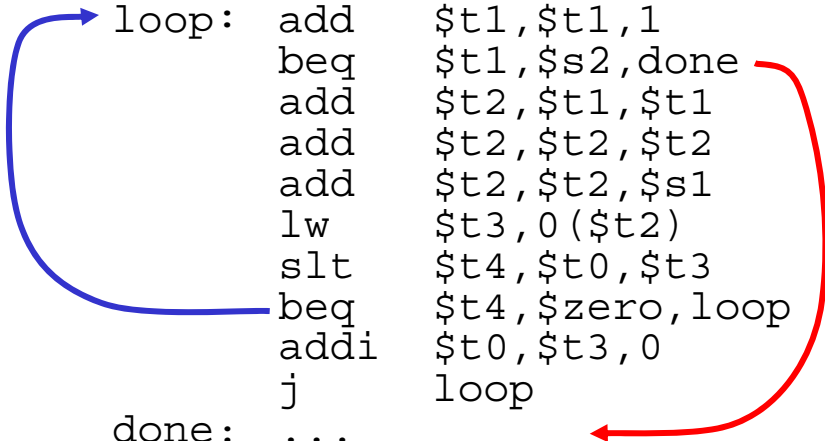
Forward and Backward Branches

Example 5.5

List A is stored in memory beginning at the address given in `$s1`.
List length is given in `$s2`.
Find the largest integer in the list and copy it into `$t0`.

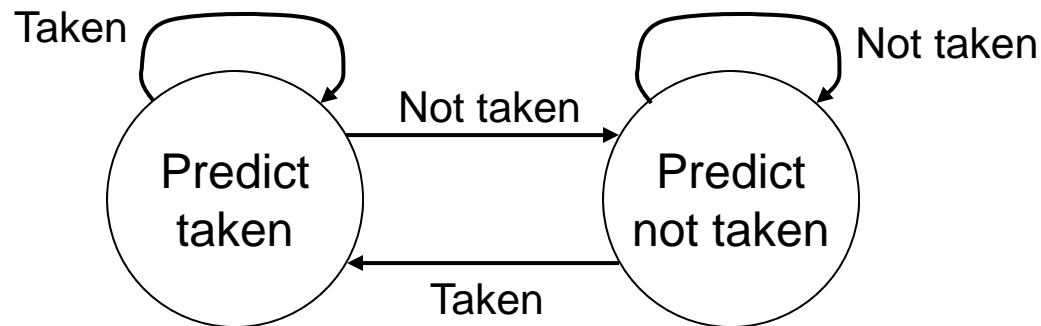
Solution

Scan the list, holding the largest element identified thus far in `$t0`.



```
loop:  lw      $t0,0($s1)      # initialize maximum to A[0]
      addi   $t1,$zero,0    # initialize index i to 0
      add    $t1,$t1,1      # increment index i by 1
      beq    $t1,$s2,done   # if all elements examined, quit
      add    $t2,$t1,$t1    # compute 2i in $t2
      add    $t2,$t2,$t2    # compute 4i in $t2
      add    $t2,$t2,$s1    # form address of A[i] in $t2
      lw     $t3,0($t2)     # load value of A[i] into $t3
      slt    $t4,$t0,$t3    # maximum < A[i]?
      beq    $t4,$zero,loop # if not, repeat with no change
      addi   $t0,$t3,0      # if so, A[i] is the new maximum
      j      loop          # change completed; now repeat
done:  ...                  # continuation of the program
```

Simple Branch Prediction: 1-Bit History



Two-state branch prediction scheme.

Problem with this approach:

Each branch in a loop entails two mispredictions:

Once in first iteration (loop is repeated, but the history indicates exit from loop)

Once in last iteration (when loop is terminated, but history indicates repetition)

Simple Branch Prediction: 2-Bit History

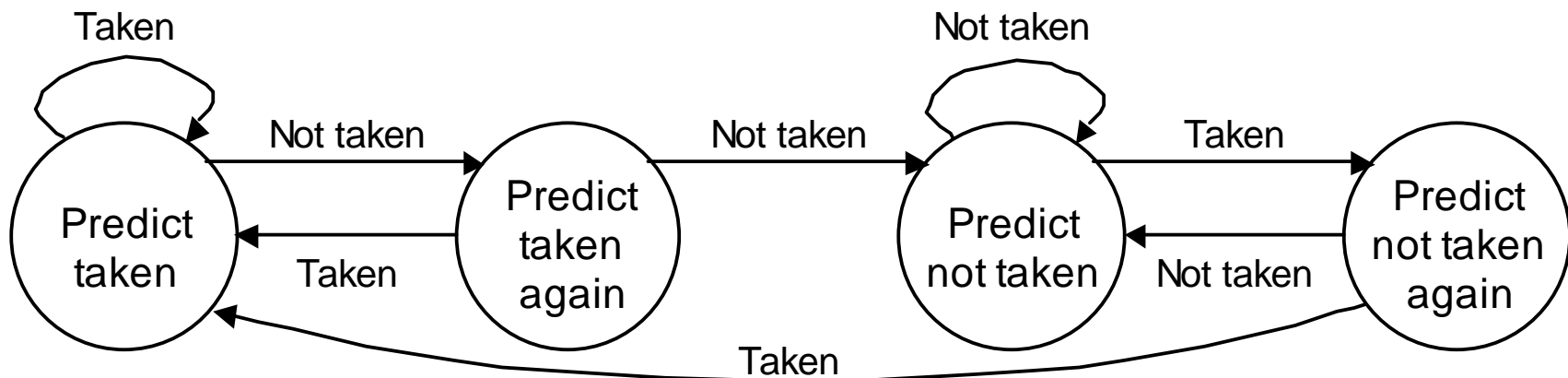
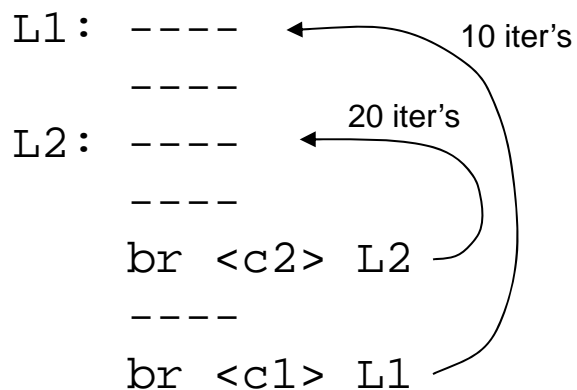


Fig. 16.6 Four-state branch prediction scheme.

Example 16.1



Impact of different branch prediction schemes

Solution

Always taken: 11 mispredictions, 94.8% accurate

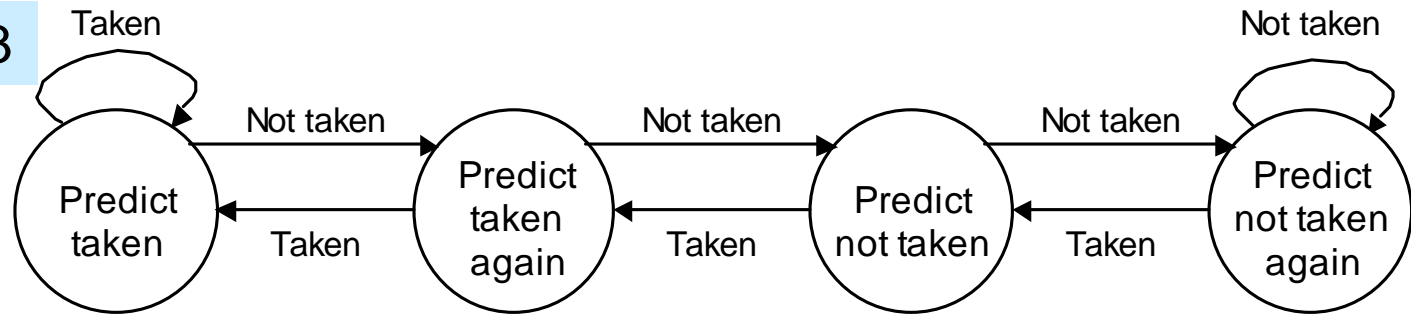
1-bit history: 20 mispredictions, 90.5% accurate

2-bit history: Same as always taken

Other Branch Prediction Algorithms

Problem 16.3

Part a



Part b

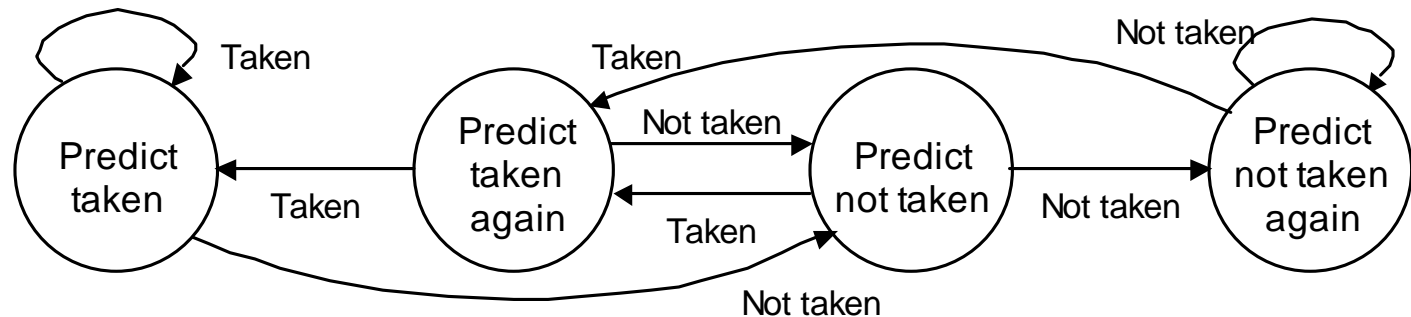
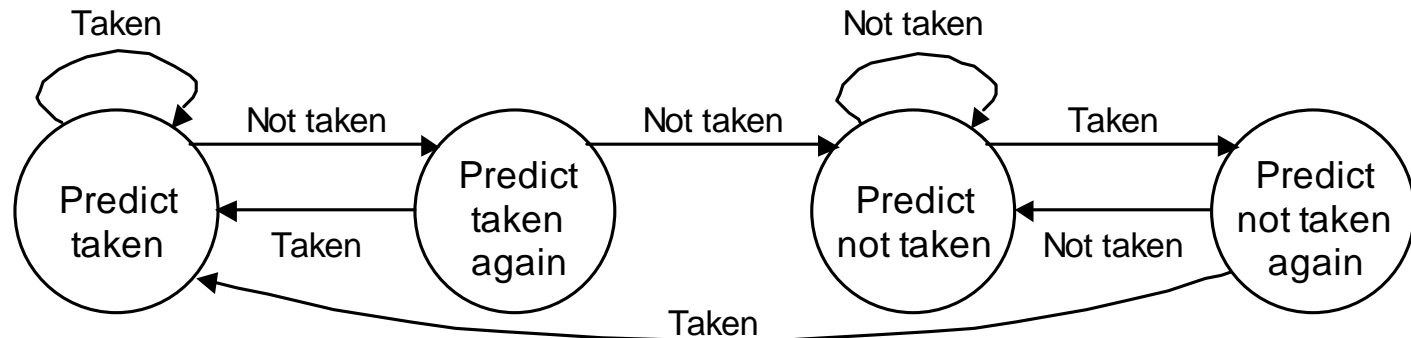


Fig. 16.6



Hardware Implementation of Branch Prediction

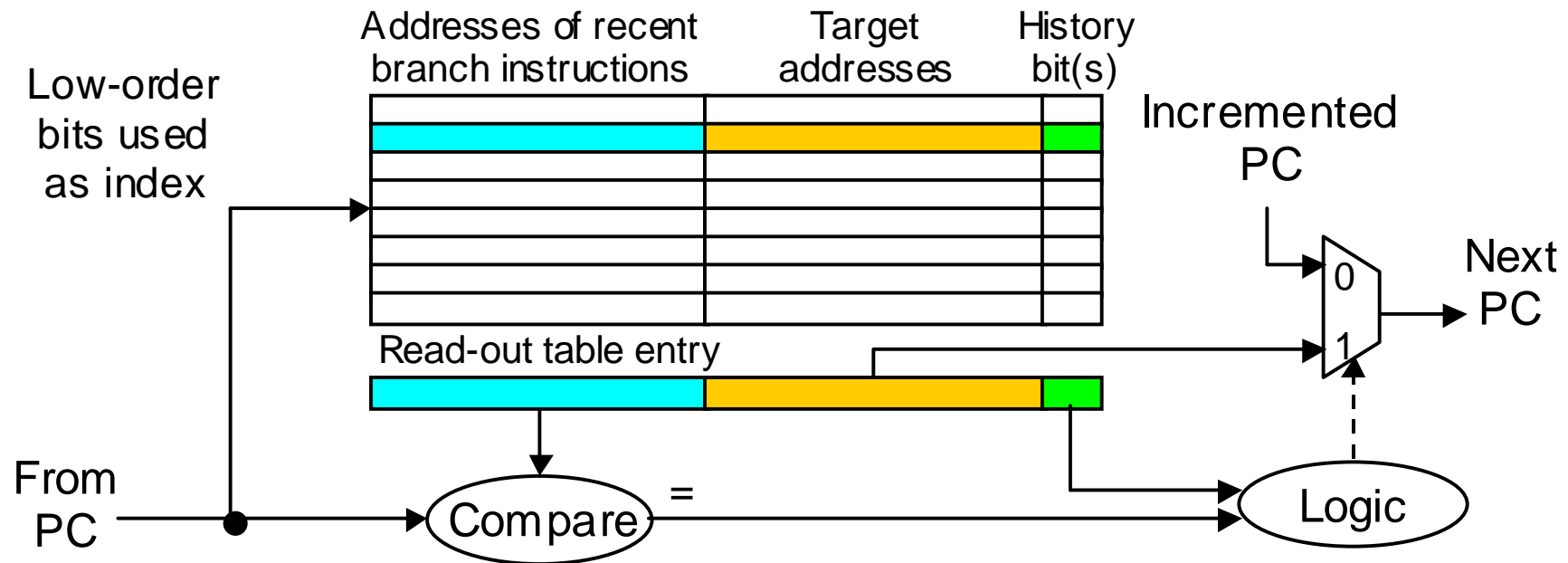
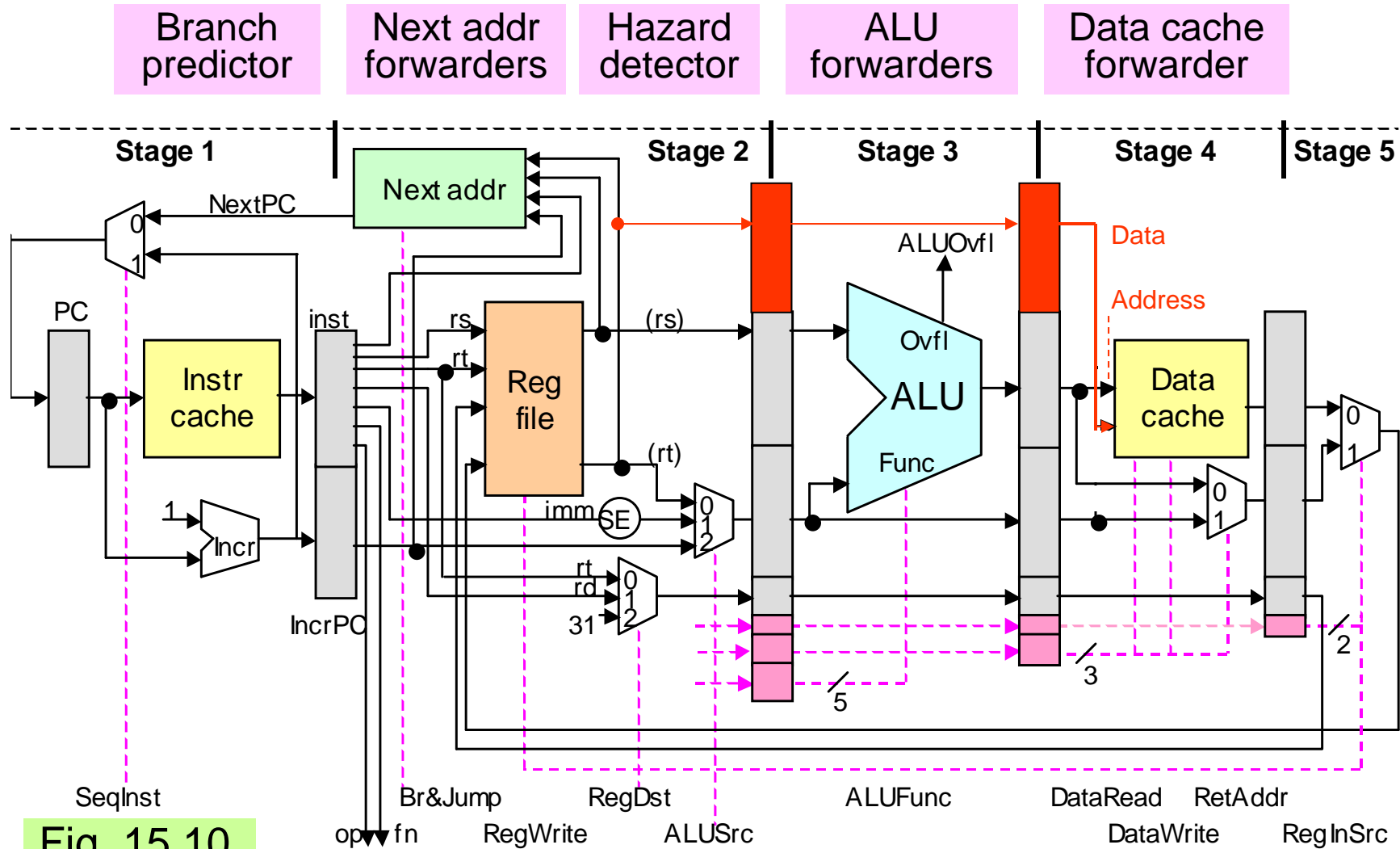


Fig. 16.7 Hardware elements for a branch prediction scheme.

The mapping scheme used to go from PC contents to a table entry is the same as that used in direct-mapped caches (Chapter 18)

Pipeline Augmentations – Repeated for Reference



16.5 Advanced Pipelining

Deep pipeline = *superpipeline*; also, *superpipelined*, *superpipelining*

Parallel instruction issue = *superscalar*, *j-way issue* (2-4 is typical)

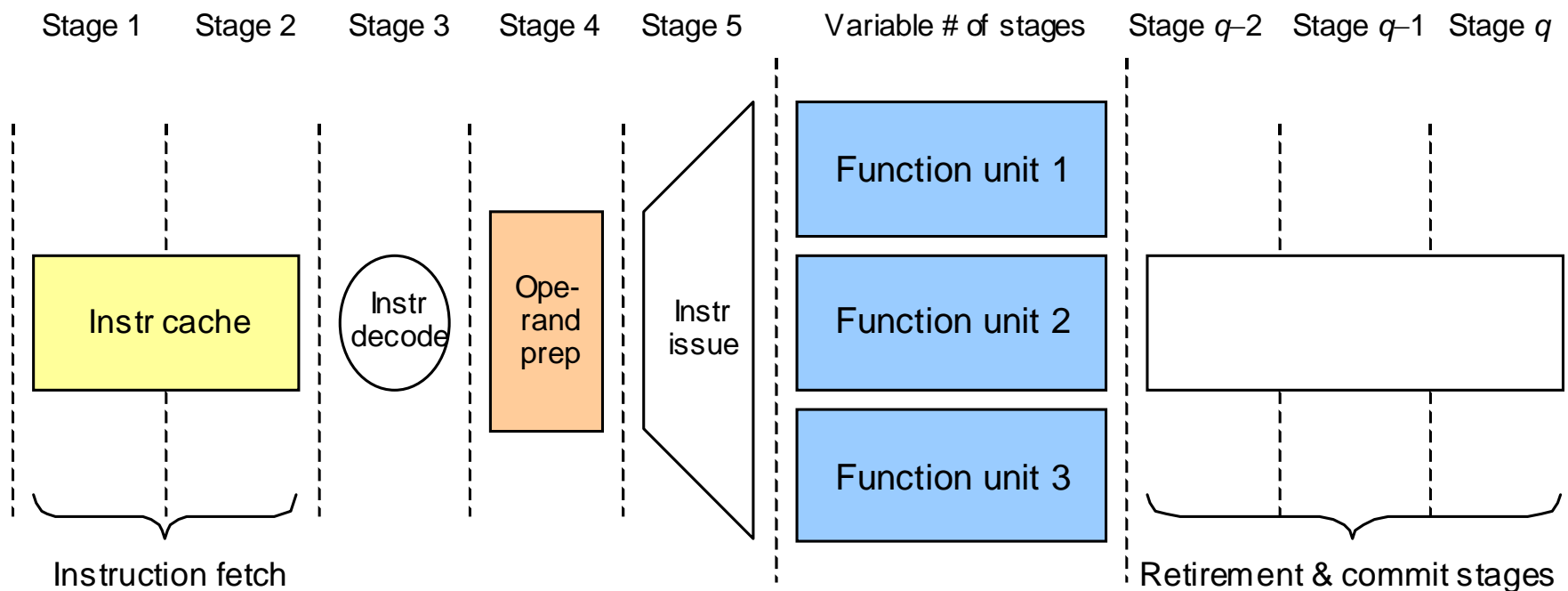


Fig. 16.8 Dynamic instruction pipeline with in-order issue, possible out-of-order completion, and in-order retirement.

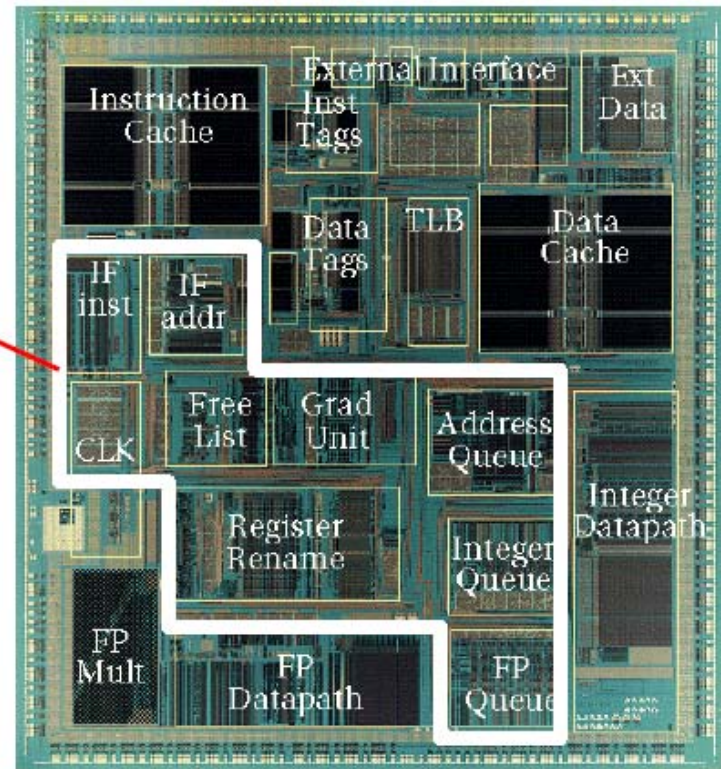
Design Space for Advanced Superscalar Pipelines

Front end:	In-order or out-of-order
Instr. issue:	In-order or out-of-order
Writeback:	In-order or out-of-order
Commit:	In-order or out-of-order

The more OoO stages,
the higher the complexity

Example of complexity due to
out-of-order processing:
MIPS R10000

Control
Logic



Source: Ahi, A. et al., "MIPS R10000
Superscalar Microprocessor,"
Proc. Hot Chips Conf., 1995.

Performance Improvement for Deep Pipelines

Hardware-based methods

Lookahead past an instruction that will/may stall in the pipeline
(out-of-order execution; requires in-order retirement)

Issue multiple instructions (requires more ports on register file)

Eliminate false data dependencies via register renaming

Predict branch outcomes more accurately, or speculate

Software-based method

Pipeline-aware compilation

Loop unrolling to reduce the number of branches

Loop: Compute with index i
Increment i by 1
Go to Loop if not done

Loop: Compute with index i
Compute with index $i + 1$
Increment i by 2
Go to Loop if not done

CPI Variations with Architectural Features

Table 16.2 Effect of processor architecture, branch prediction methods, and speculative execution on CPI.

Architecture	Methods used in practice	CPI
Nonpipelined, multicycle	Strict in-order instruction issue and exec	5-10
Nonpipelined, overlapped	In-order issue, with multiple function units	3-5
Pipelined, static	In-order exec, simple branch prediction	2-3
Superpipelined, dynamic	Out-of-order exec, adv branch prediction	1-2
Superscalar	2- to 4-way issue, interlock & speculation	0.5-1
Advanced superscalar	4- to 8-way issue, aggressive speculation	0.2-0.5

Need 100 for TIPS performance

Need 100,000 for 1 PIPS

3.3 inst / cycle \times 3 Gigacycles / s
 \cong 10 GIPS

Development of Intel's Desktop/Laptop Micros

In the beginning, there was the 8080; led to the 80x86 = IA32 ISA

Half a dozen or so pipeline stages

80286

80386

80486

Pentium (80586)

More advanced
technology

A dozen or so pipeline stages, with out-of-order instruction execution

Pentium Pro

Pentium II

Pentium III

Celeron

More advanced technology

Instructions are broken into micro-ops which are executed out-of-order but retired in-order

Two dozens or so pipeline stages

Pentium 4

Current State of Computer Performance

Multi-GIPS/GFLOPS desktops and laptops

Very few users need even greater computing power
Users unwilling to upgrade just to get a faster processor
Current emphasis on power reduction and ease of use

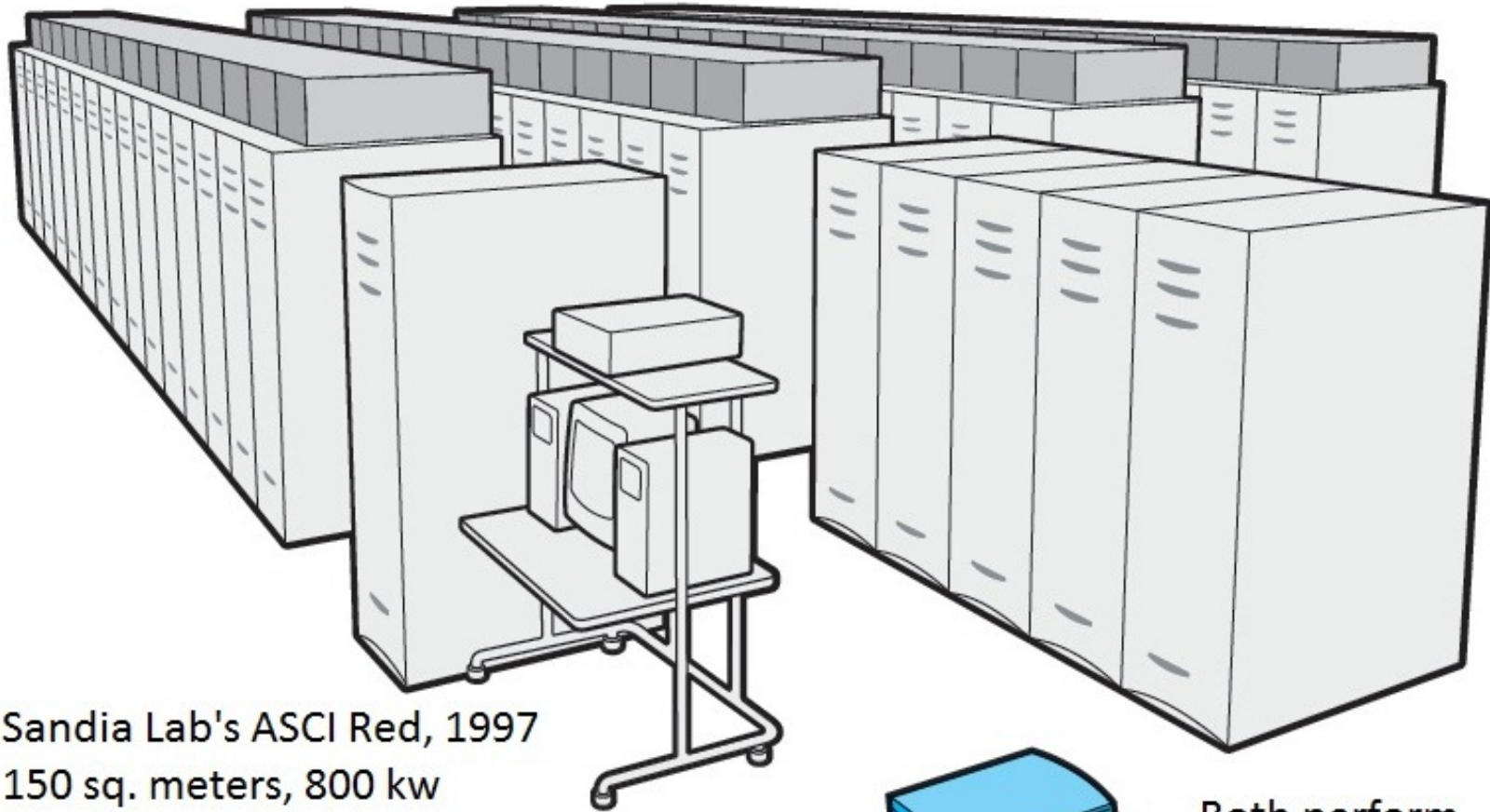
Multi-TIPS/TFLOPS in large computer centers

World's top 500 supercomputers, <http://www.top500.org>
Next list due in June 2009; as of Nov. 2008:
All 500 >> 10 TFLOPS, $\approx 30 > 100$ TFLOPS, 1 > PFLOPS

Multi-PIPS/PFLOPS supercomputers on the drawing board

IBM “smarter planet” TV commercial proclaims (in early 2009):
“We just broke the petaflop [sic] barrier.”
The technical term “petaflops” is now in the public sphere

The Shrinking Supercomputer



Sandia Lab's ASCI Red, 1997
150 sq. meters, 800 kw

Sony Playstation, 2006
0.08 sq. meter, < 0.2 kw

Both perform
at ~2 TFLOPS

16.6 Dealing with Exceptions

Exceptions present the same problems as branches

How to handle instructions that are ahead in the pipeline?
(let them run to completion and retirement of their results)

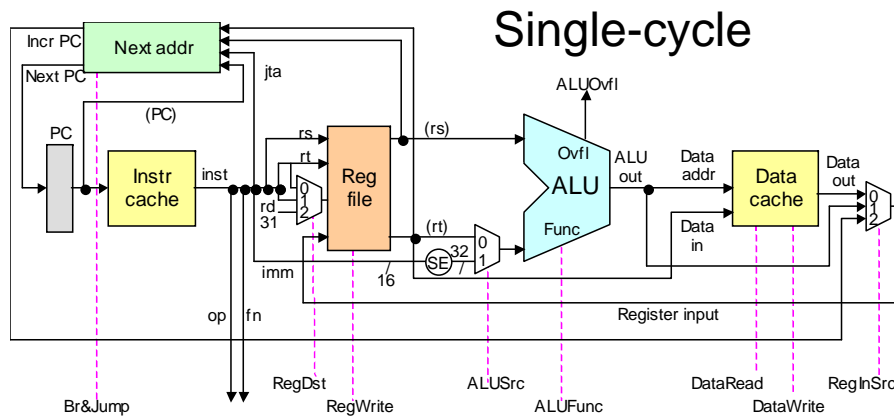
What to do with instructions after the exception point?
(flush them out so that they do not affect the state)

Precise versus imprecise exceptions

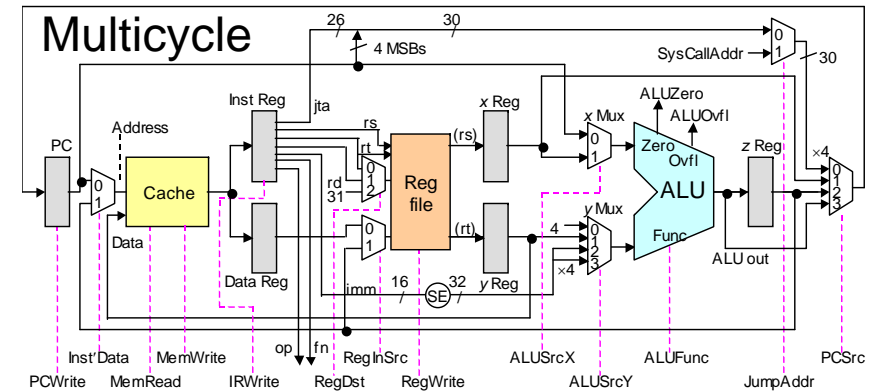
Precise exceptions hide the effects of pipelining and parallelism by forcing the same state as that of strict sequential execution
(desirable, because exception handling is not complicated)

Imprecise exceptions are messy, but lead to faster hardware
(interrupt handler can clean up to offer precise exception)

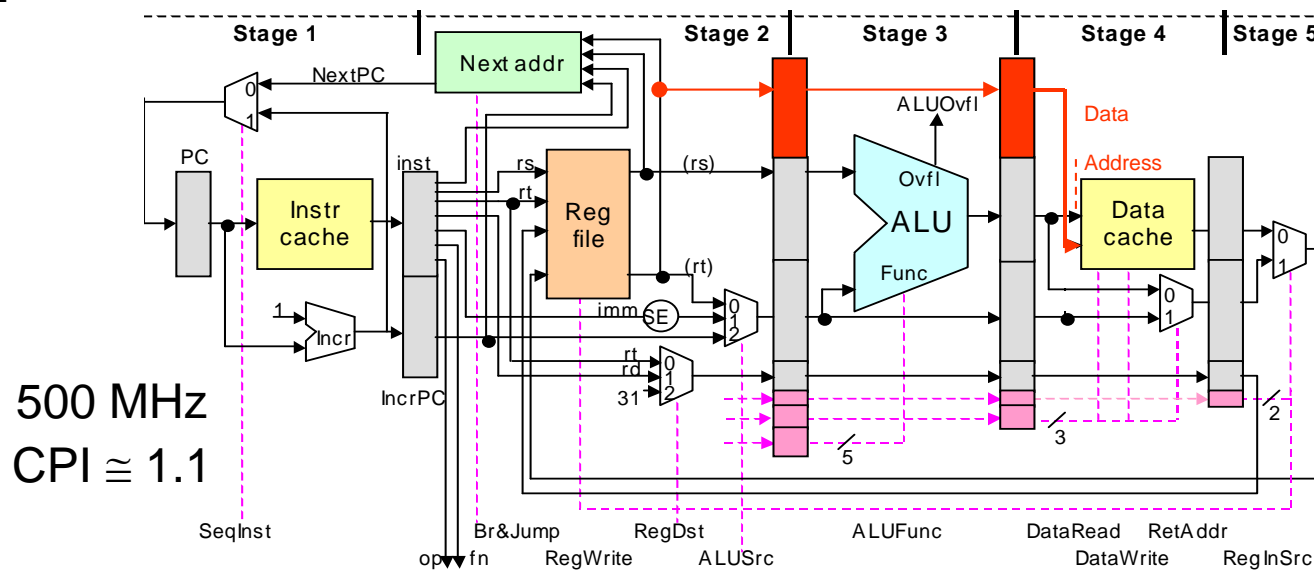
The Three Hardware Designs for MicroMIPS



125 MHz
CPI = 1



500 MHz
CPI \cong 4





Where Do We Go from Here?

Memory Design:

How to build a memory unit that responds in 1 clock

Input and Output:

Peripheral devices,
I/O programming,
interfacing, interrupts

Higher Performance:

Vector/array processing
Parallel processing