Microprocessor History

- Mid 1970
  - Microprocessor introduced.

- Mid 1970 – 1980s
  - Two factors influenced architecture:
    - Microprogramming and Complex Instructions
      - Ferrite memory core had a long access time.
      - The slow main store held complex instructions.
      - Fetching and executing microprograms from the much faster microprogram memory within the CPU was advantageous.
      - The complex instructions were seen to help programmers.
- Today, most of the advantages of microprogramming have evaporated due to the low access time of system memory and cache systems.

The RISC Revolution

- 1980s
  - Initial reaction against the trend towards complex instructions:
    - IBM’s 801 architecture
      - In 1974 John Cocke (IBM) starting working on RISC like architectures.
    - Berkeley: David Patterson and David Ditzel
      - Coined RISC

RISCy Research

- Research conducted in the late 1970s by Fairclough demonstrated that the relative frequency with which different classes of instructions are executed is not uniform and some types of instructions are executed much more frequently than others.
- Conclusion: optimize the highly frequent instructions.

Instruction Type	Frequency
Data Movement	45.28
Instruction Flow Modification (Branch, Call, Return)	28.73
Arithmetic	10.75
Compare	5.92
Logical	3.91
Shift	2.93
Bit Manipulation	2.05
IO and miscellaneous	0.44

RISCy Research

- Tanenbaum reported that 56% of all constant values lie in the range from -15 to +15, and 98% of all constant values lie in the range from -511 to +511.
  - Optimum size of an instruction.
- Other research showed that 12 words of storage are sufficient for parameter passing to and from subroutines:
  - 12 internal registers dedicated for subroutine parameter passing.

Characteristics of the RISC Architecture

1. Have sufficient on-chip registers to overcome processor-memory bottleneck.

2. Three address, register to register instruction set architecture.

```
OPERATION Ra, Rb, Rc
```

1. Facilitate passing of parameters to and from subroutines
   - For example, internal registers.

1. Program flow instructions are implemented efficiently.

Characteristics of the RISC Architecture

5. Don’t implement infrequently used instructions.
    - Complex instructions waste hardware real estate.
    - Complex instructions increase design, fabrication, and test times.

6. Execute an instruction in a single clock cycle, through
    - Regularity in the instruction set.
        - Fixed size instruction.
        - All instructions take the same number of clock cycles to execute.
    - Pipelining.

Characteristics of the RISC Architecture

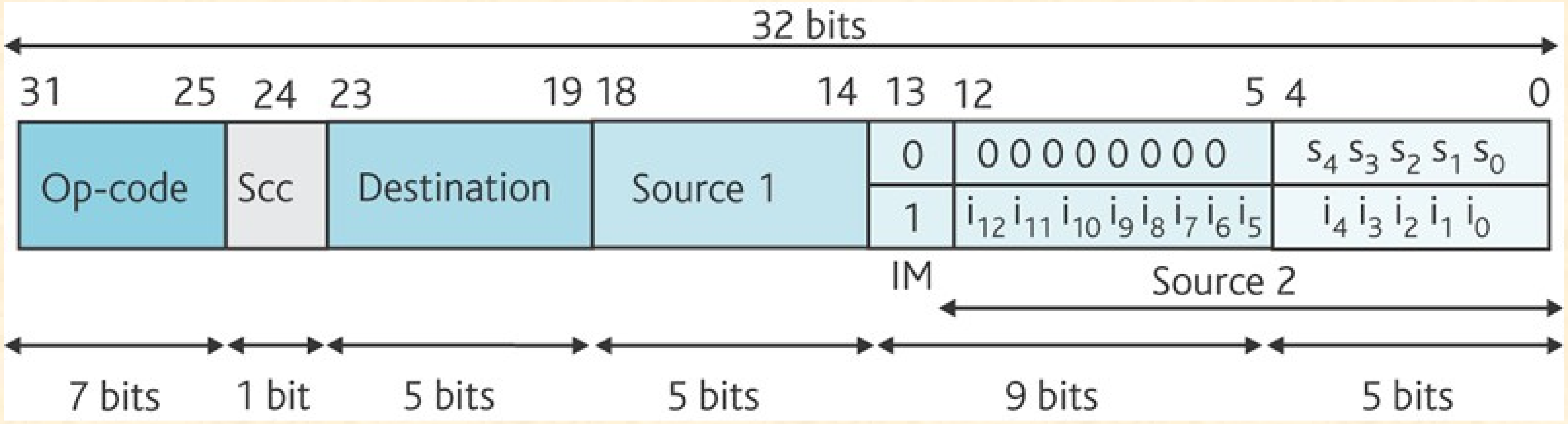
7. Because of (6) the RISC does not implement a microprogrammed architecture.
   - The distinction between machine cycle (number of clock cycles to complete an instruction) and microcycle (one clock cycle to complete a micro-operation) has vanished.

8. Single instruction format:
   - Decoding logic is simpler than for variable length instructions.
   - Memory usage may not be as efficient as variable length instructions.

The RISC Revolution

- Mid 1990s - Today
  - Distinction between RISC and CISC is blurred
    - Many RISC processors have become more complex than the CISC processors they were said to replace.
    - Many so called CISC processors have RISC like features.
    - Some RISC processors have more instructions that CISC processors.
  - RISC might be better referred to as Regular Instruction Set.

The Berkeley RISC: Instruction Format
(Led to the Commercial SPARC)

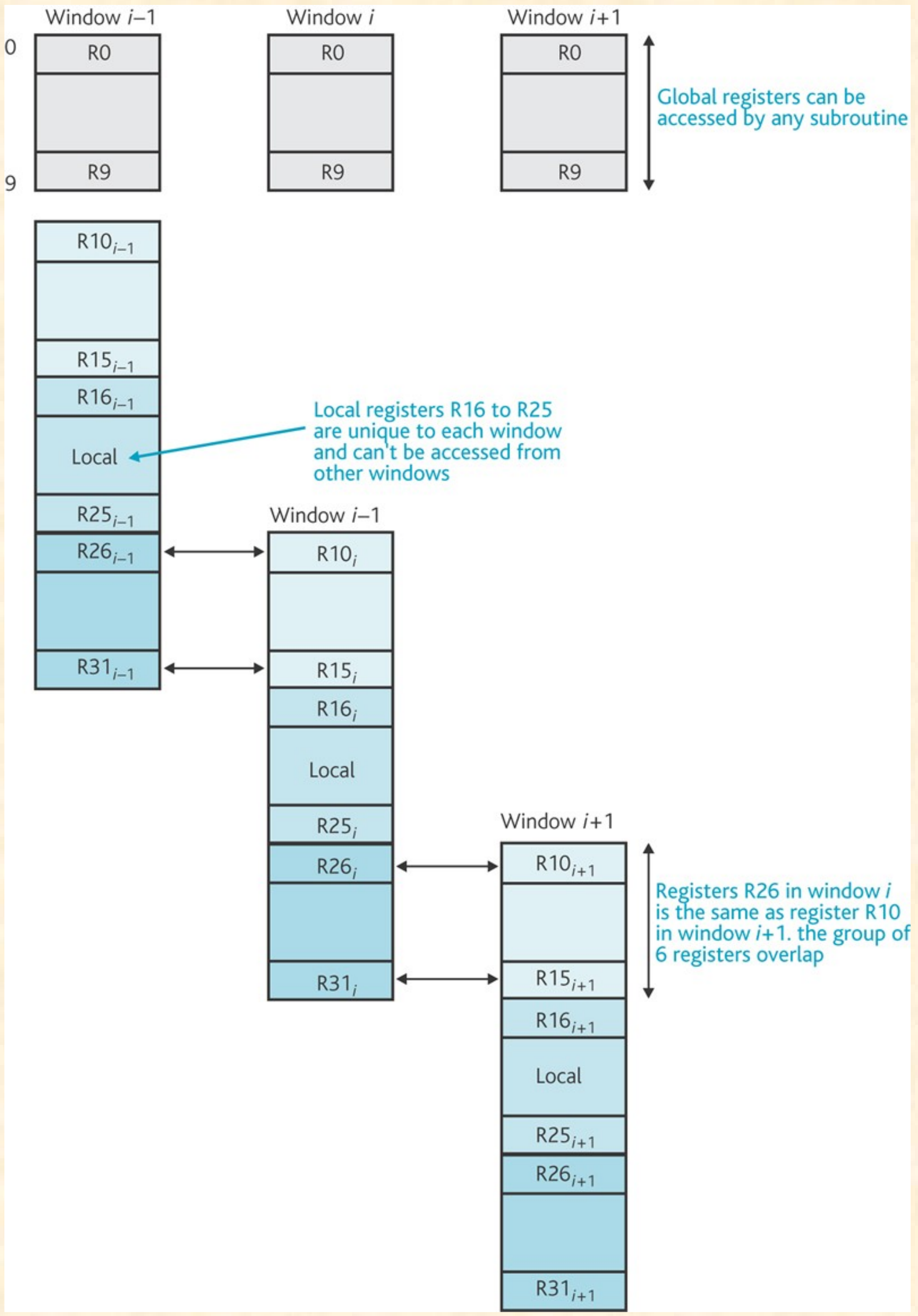


Scc:	Specifies whether the condition code bits will be written to a the end of the instruction.
Oper:	Each of five bit operands specify one of 32 internal registers.
IM:	If IM=0, [4..0] specify 2nd operand.
	If IM=1, [12..0] specify a 13-bit constant, immediate value.
R1:	Hardwired to 0. Constant and ADD R0,R1,R2 → MOVE R1,R2

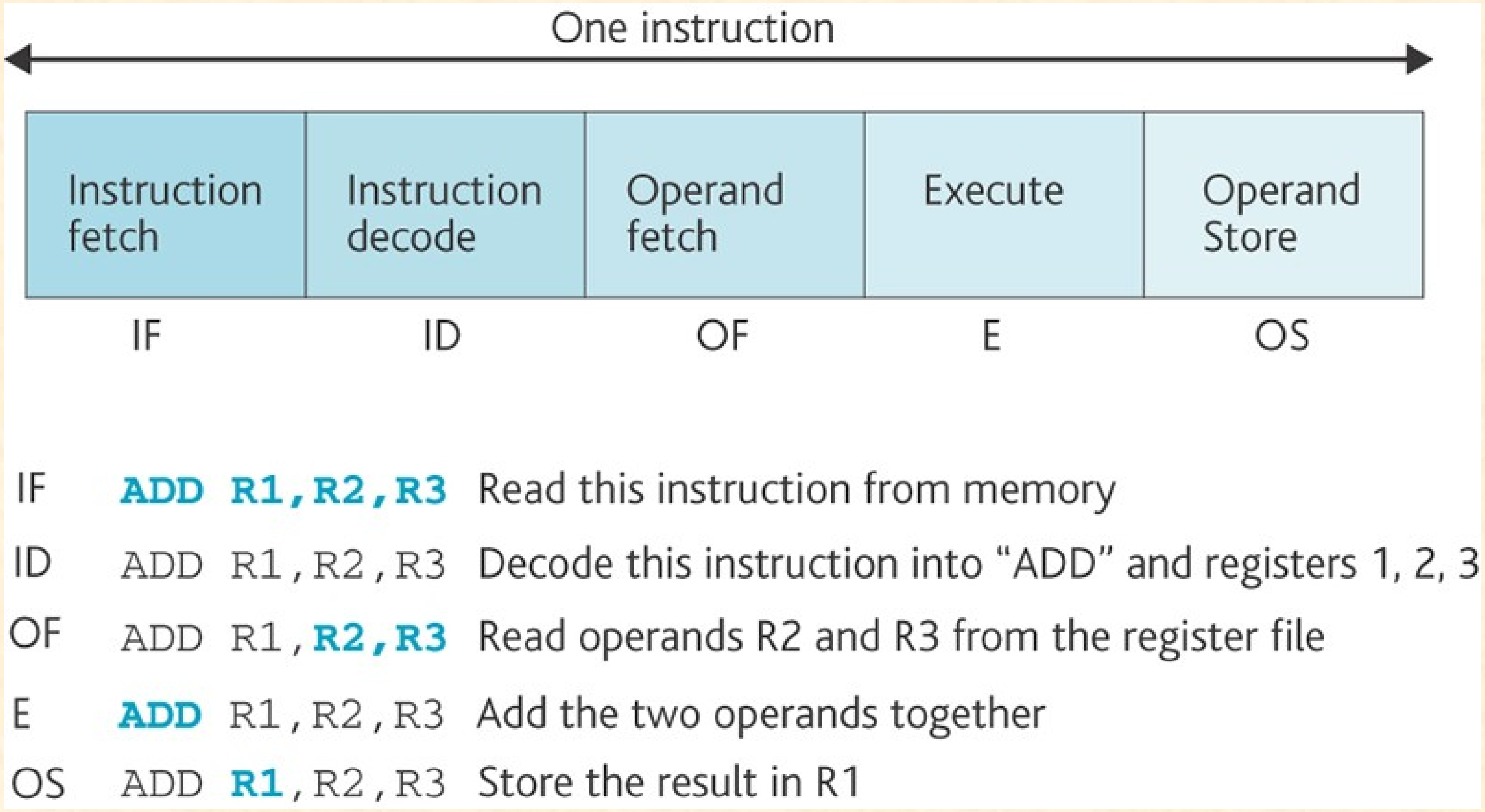
The Berkeley RISC: Register Windows

- When a subroutine is called, the window pointer is incremented by 1.
- Program counter saved in Rd.
- The subroutine sees a different set of registers.
- 10 global + 8 x 10 local + 8 x 6 parameter transfer registers = 138 registers.
- Supports up to 8 nested subroutines.

- Main store is used if greater than 8.
- Context switching is expensive.


Window i–1
Window i
Window i+1
0
9
R0
R9
Global registers can be accessed by any subroutine
R10i–1
R15i–1
R16i–1
Local
R25i–1
R26i–1
R31i–1
Local registers R16 to R25 are unique to each window and can't be accessed from other windows
Window i–1
R10i
R15i
R16i
Local
R25i
R26i
R31i
Window i+1
R10i+1
R15i+1
R16i+1
Local
R25i+1
R26i+1
R31i+1
Registers R26 in window i is the same as register R10 in window i+1. the group of 6 registers overlap

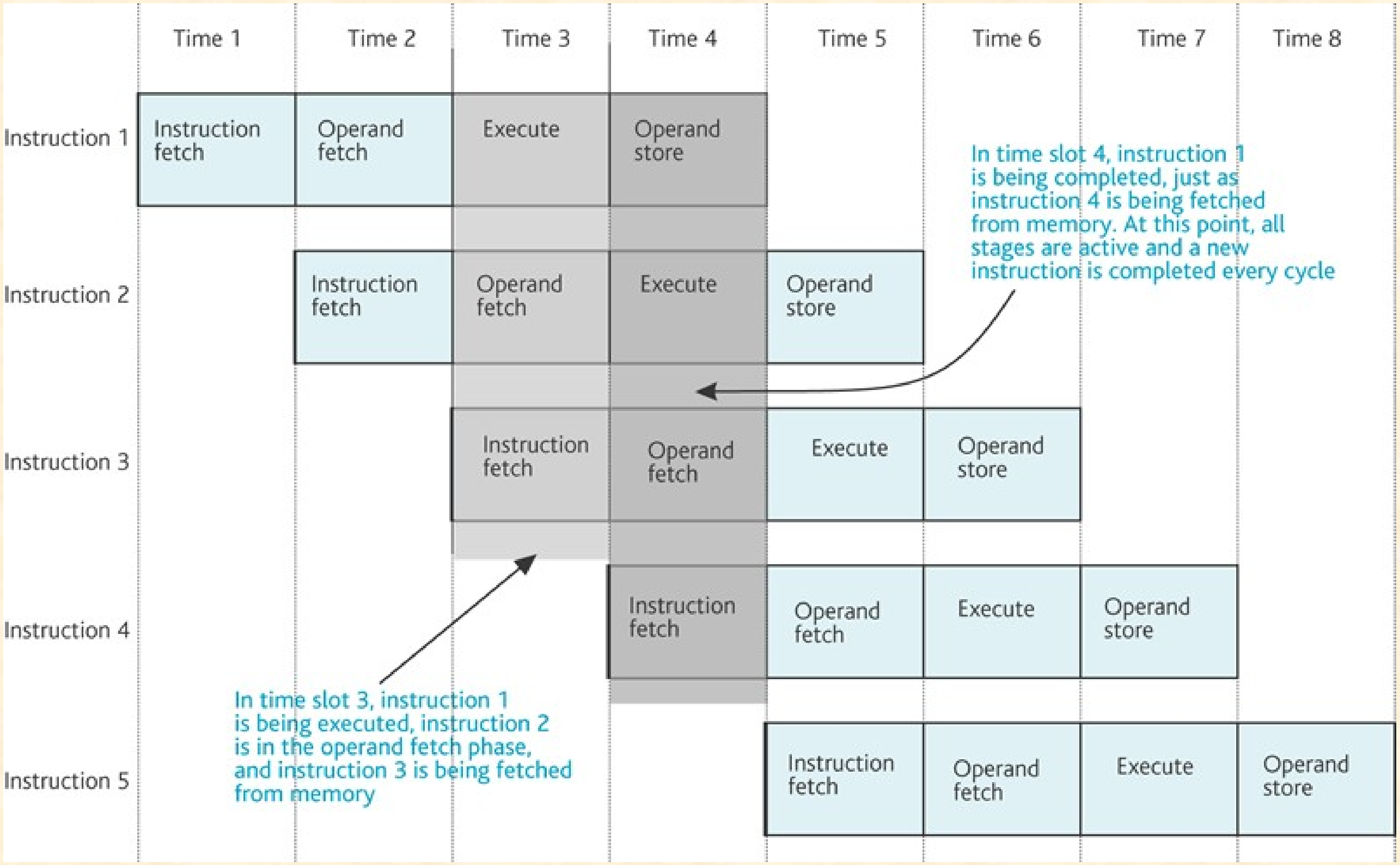
Instruction Execution Phases



IF	**ADD R1,R2,R3**	Read this instruction from memory
ID	ADD R1,R2,R3	Decode this instruction into "ADD" and registers 1, 2, 3
OF	ADD R1,**R2,R3**	Read operands R2 and R3 from the register file
E	**ADD** R1,R2,R3	Add the two operands together
OS	ADD **R1**,R2,R3	Store the result in R1

- RISC processors don't need an Instruction Decode phase because their encoding is so simple.

Pipelining and Instruction Overlap


Time 1
Time 2
Time 3
Time 4
Time 5
Time 6
Time 7
Time 8
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Instruction fetch
Operand fetch
Execute
Operand store
In time slot 4, instruction 1 is being completed, just as instruction 4 is being fetched from memory. At this point, all stages are active and a new instruction is completed every cycle
In time slot 3, instruction 1 is being executed, instruction 2 is in the operand fetch phase, and instruction 3 is being fetched from memory

	Time 1	Time 2	Time 3	Time 4	Time 5	Time 6	Time 7	Time 8
Instruction 1	Instruction fetch	Operand fetch	Execute	Operand store				
Instruction 2		Instruction fetch	Operand fetch	Execute	Operand store			
Instruction 3			Instruction fetch	Operand fetch	Execute	Operand store		
Instruction 4				Instruction fetch	Operand fetch	Execute	Operand store	
Instruction 5					Instruction fetch	Operand fetch	Execute	Operand store

Pipelining Performance

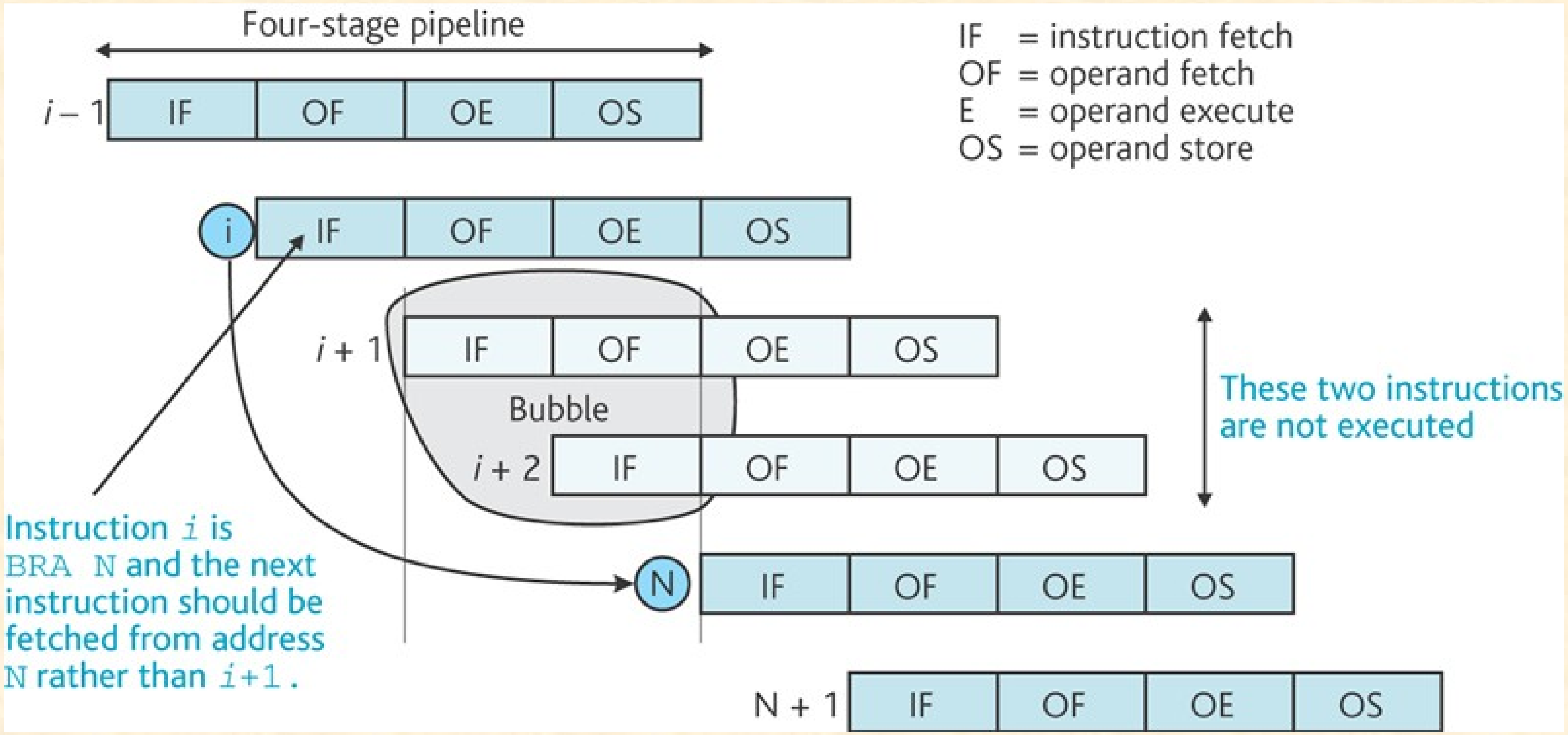
- Consider the execution of `n` instructions using an `m`-stage pipeline.
- It will take `m` clock cycles for the 1st instruction to complete.
- The remaining `n - 1` instructions execute at the rate of one clock cycle per instruction.
- The total time to execute the `n` instructions is:

```
m + (n - 1) cycles
```

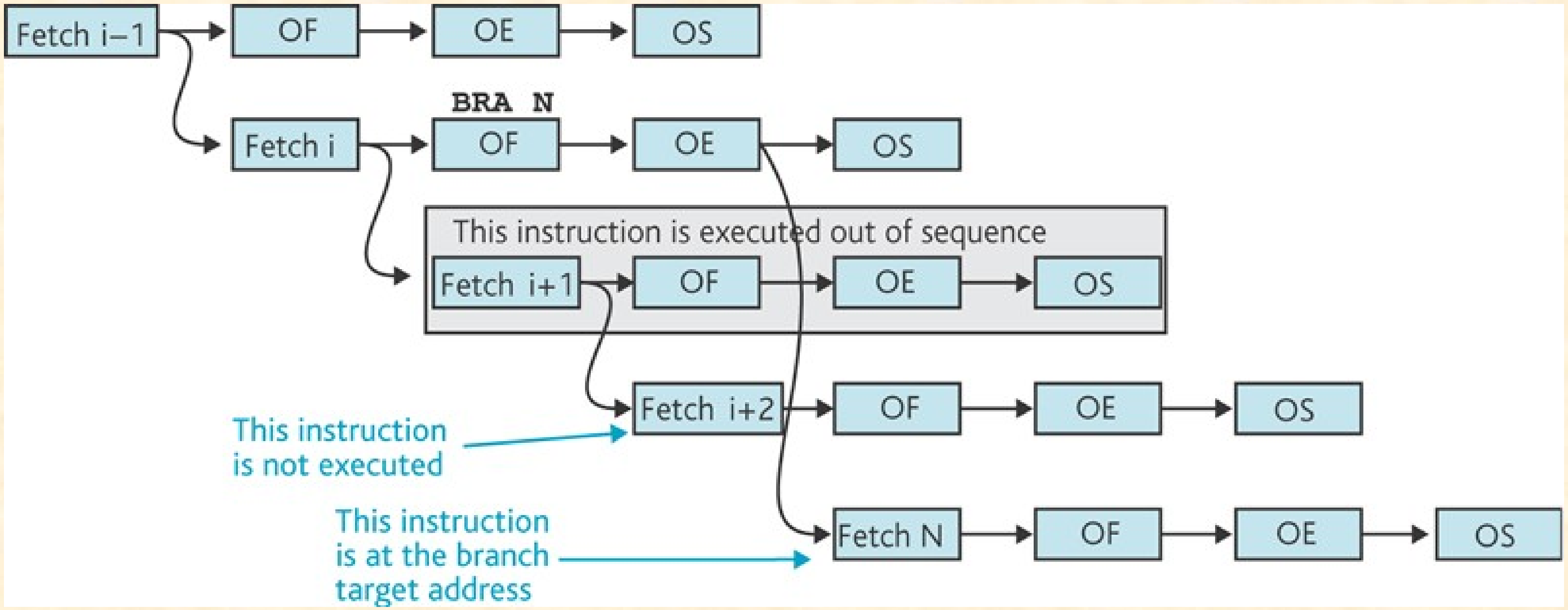
Pipelining Performance

Block Size	3-stage Pipeline	6-stage Pipeline	12-stage Pipeline
4	2.0000	2.6667	3.2000
8	2.4000	3.6923	5.0526
20	2.7272	4.8000	7.7419
100	2.9411	5.7143	10.8100
1000	2.9940	5.9701	11.8694
Infinite	3.0000	6.0000	12.0000

Pipeline Bubble

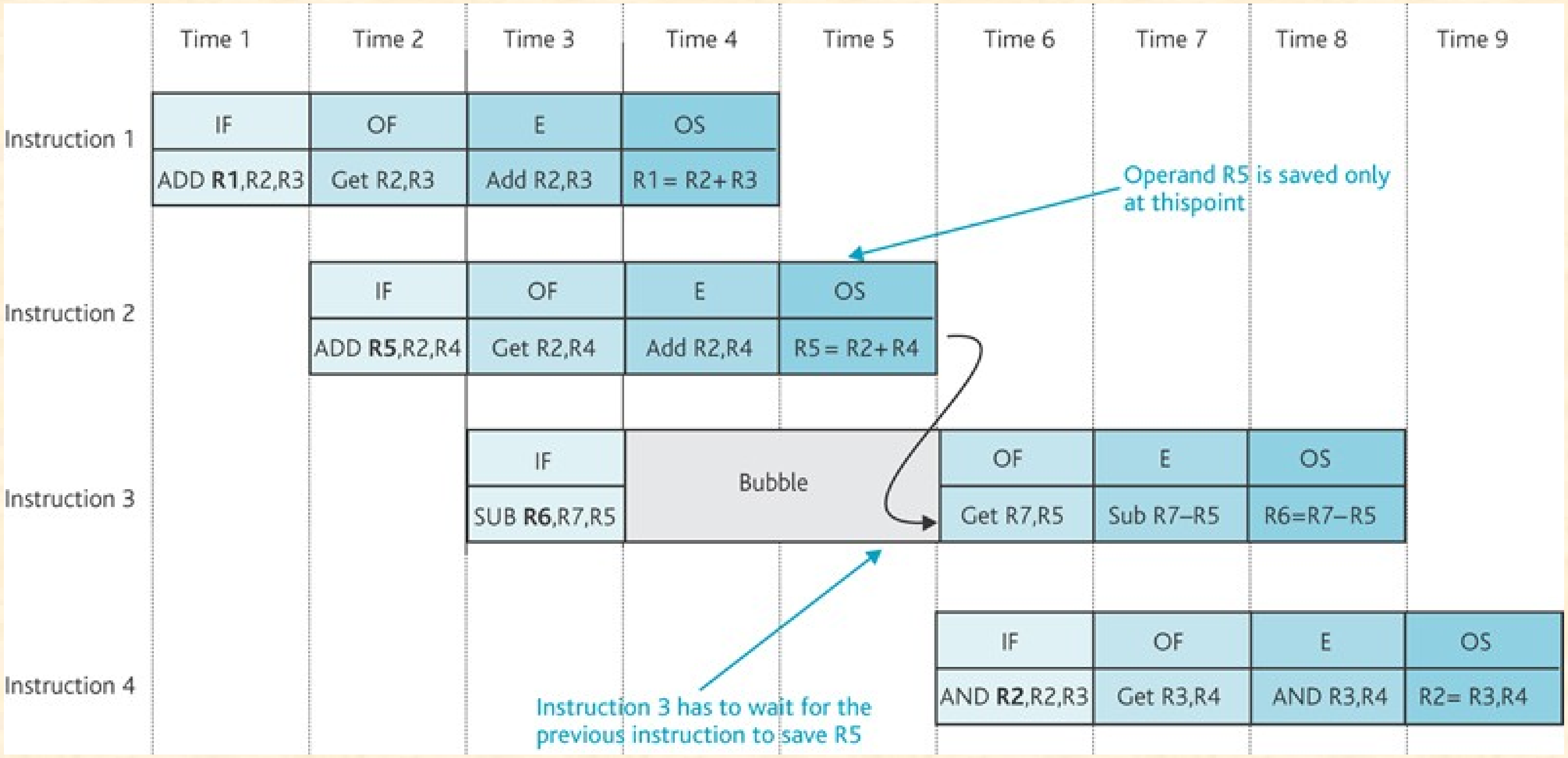

Four-stage pipeline
IF = instruction fetch
OF = operand fetch
E = operand execute
OS = operand store
i − 1
IF
OF
OE
OS
i
i + 1
Bubble
i + 2
N
N + 1
These two instructions are not executed
Instruction i is BRA N and the next instruction should be fetched from address N rather than i+1.

Overcoming The Pipeline Bubble


Fetch i–1
OF
OE
OS
BRA N
Fetch i
OF
OE
OS
This instruction is executed out of sequence
Fetch i+1
OF
OE
OS
Fetch i+2
OF
OE
OS
This instruction is not executed
This instruction is at the branch target address
Fetch N
OF
OE
OS

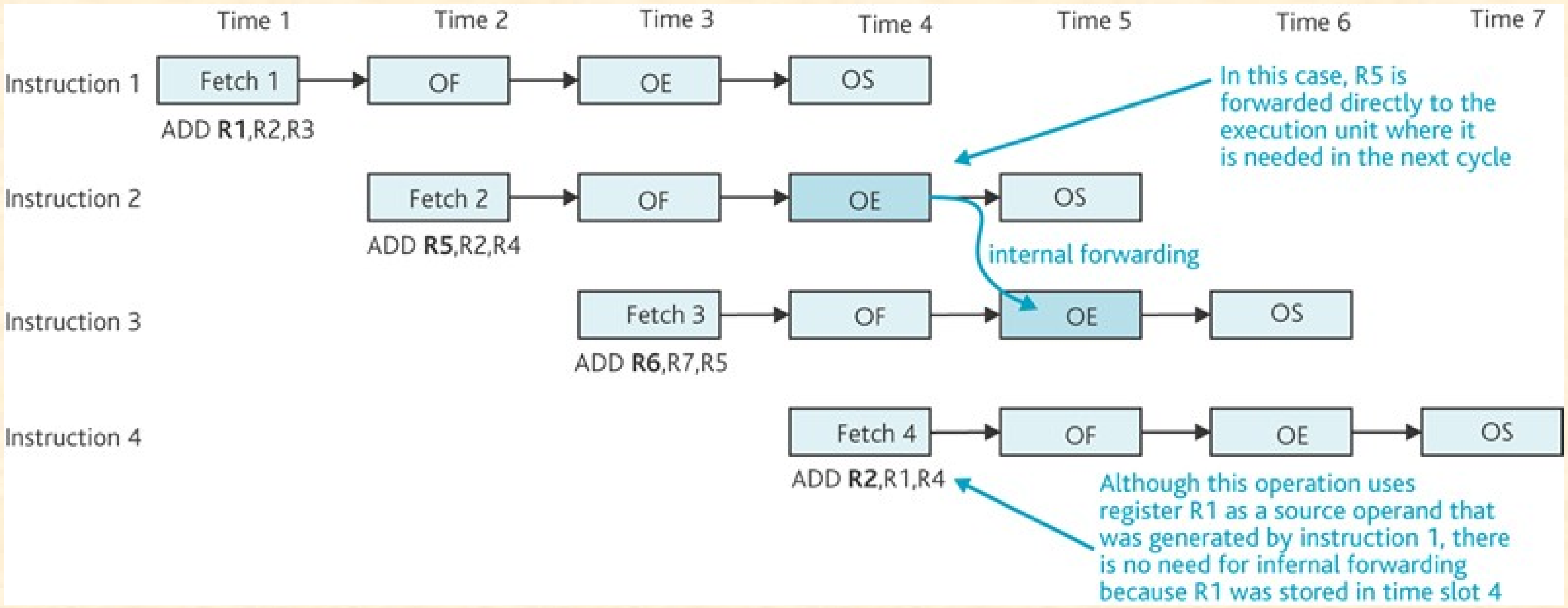
```
ADD R1,R2,R3       [R3] ← [R1] + [R2]
BRA N              GOTO ADDRESS N
ADD R2,R4,R5       [R5] ← [R2] + [R4]   This is executed.
ADD R7,R8,R9       Not executed because branch is taken.
```

Data Dependency



- The pipeline is stalled after the fetch phase of instruction 3 for two clocked cycles.

Overcoming Data Dependency:
Internal Forwarding



Time 1
Time 2
Time 3
Time 4
Time 5
Time 6
Time 7
Instruction 1
Fetch 1
OF
OE
OS
ADD R1,R2,R3
In this case, R5 is forwarded directly to the execution unit where it is needed in the next cycle
Instruction 2
Fetch 2
OF
OE
OS
ADD R5,R2,R4
internal forwarding
Instruction 3
Fetch 3
OF
OE
OS
ADD R6,R7,R5
Instruction 4
Fetch 4
OF
OE
OS
ADD R2,R1,R4
Although this operation uses register R1 as a source operand that was generated by instruction 1, there is no need for infernal forwarding because R1 was stored in time slot 4

```
1.     ADD R1,R2,R3        [R1] ← [R2] + [R3]
2.     ADD R5,R2,R4        [R5] ← [R2] + [R4]
3.     SUB R6,R7,R5        [R6] ← [R7] - [R5]
4.     ADD R1,R1,R4        [R2] ← [R1] + [R4]
```

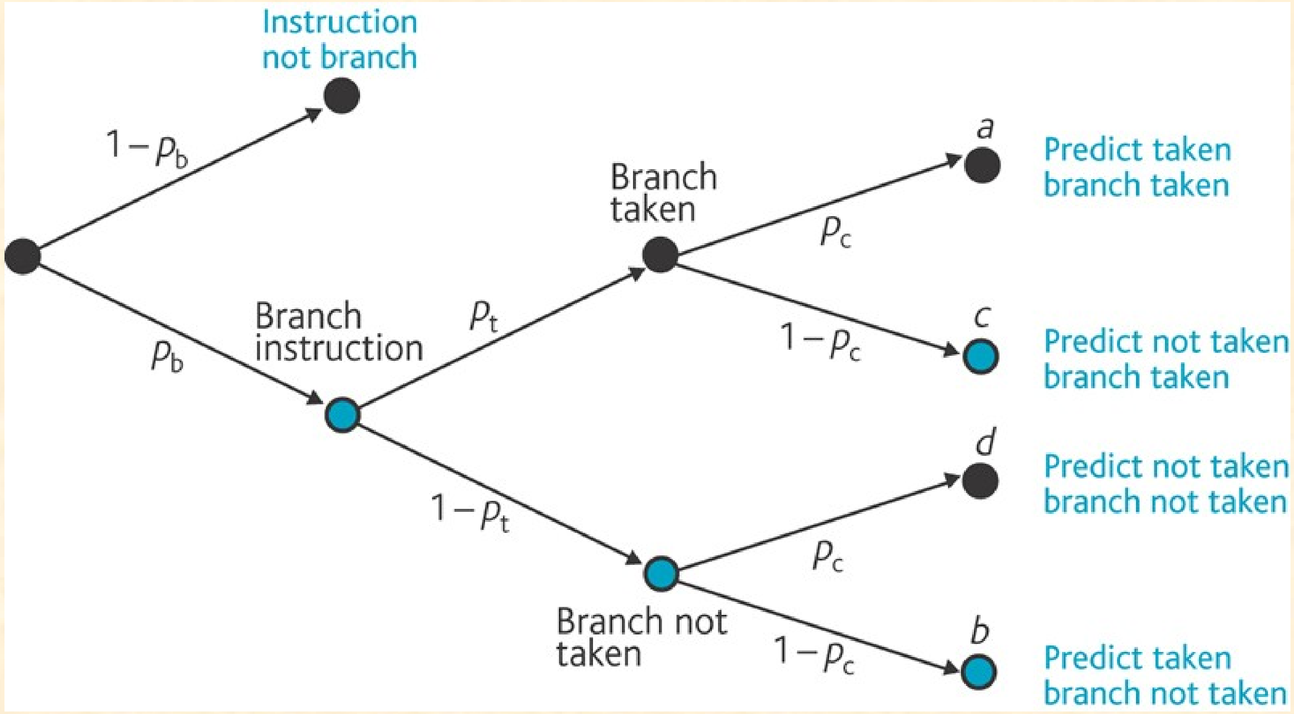
Probabilistic Model of Instruction Execution

1. Each non-branch instruction is executed in one cycle.
2. The probability that a given instruction is a branch is p_b.
3. The probability that a branch instruction will be taken is p_t.
4. If a branch is taken, the additional penalty is b cycles.
5. If a branch is not taken, there is no penalty.

- The average time an instruction takes to execute is:

$$
\begin{aligned}
Tave &= (1 - p_b)*1 + p_b p_t (1 + b) + p_b (1 - p_t)*1 \\
&= 1 + p_b p_t b
\end{aligned}
$$

Probabilistic Model of Branch Penalty


Instruction not branch
$1-p_b$
p_b
Branch instruction
p_t
$1-p_t$
Branch taken
Branch not taken
p_c
$1-p_c$
p_c
$1-p_c$
a
Predict taken branch taken
c
Predict not taken branch taken
d
Predict not taken branch not taken
b
Predict taken branch not taken

- The average number of cycles taken by a branch instruction is:

$$p_b(a(p_t p_c) + b(1-p_t)(1-p_c) + cp_t(1-p_c) + d(1-p_t)p_c)$$

Implementing Branch Prediction

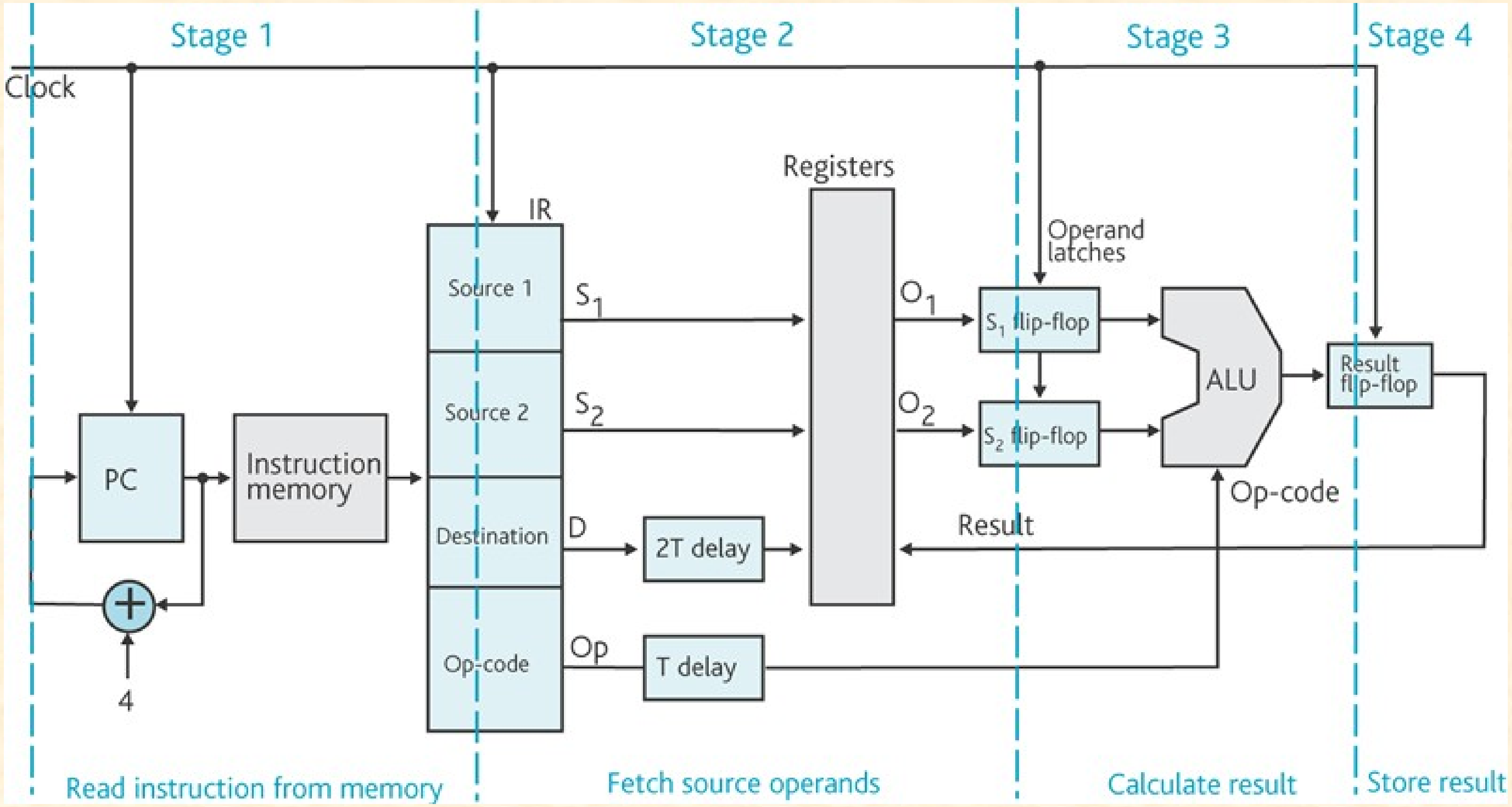
- Static Branch Prediction
  - Observation of real code has demonstrated > 50% chance that a branch will be taken
    - Fetch the next instruction at the target address.

  - Some branch instructions are taken more or less frequently than others.
    - Basing the prediction on the opcode can yield as much as 75% accuracy.
    - Devote a bit in the opcode of the branch instruction
      - This bit is set if the compiler estimates a branch will be taken.
      - 75 – 94 % accuracy.

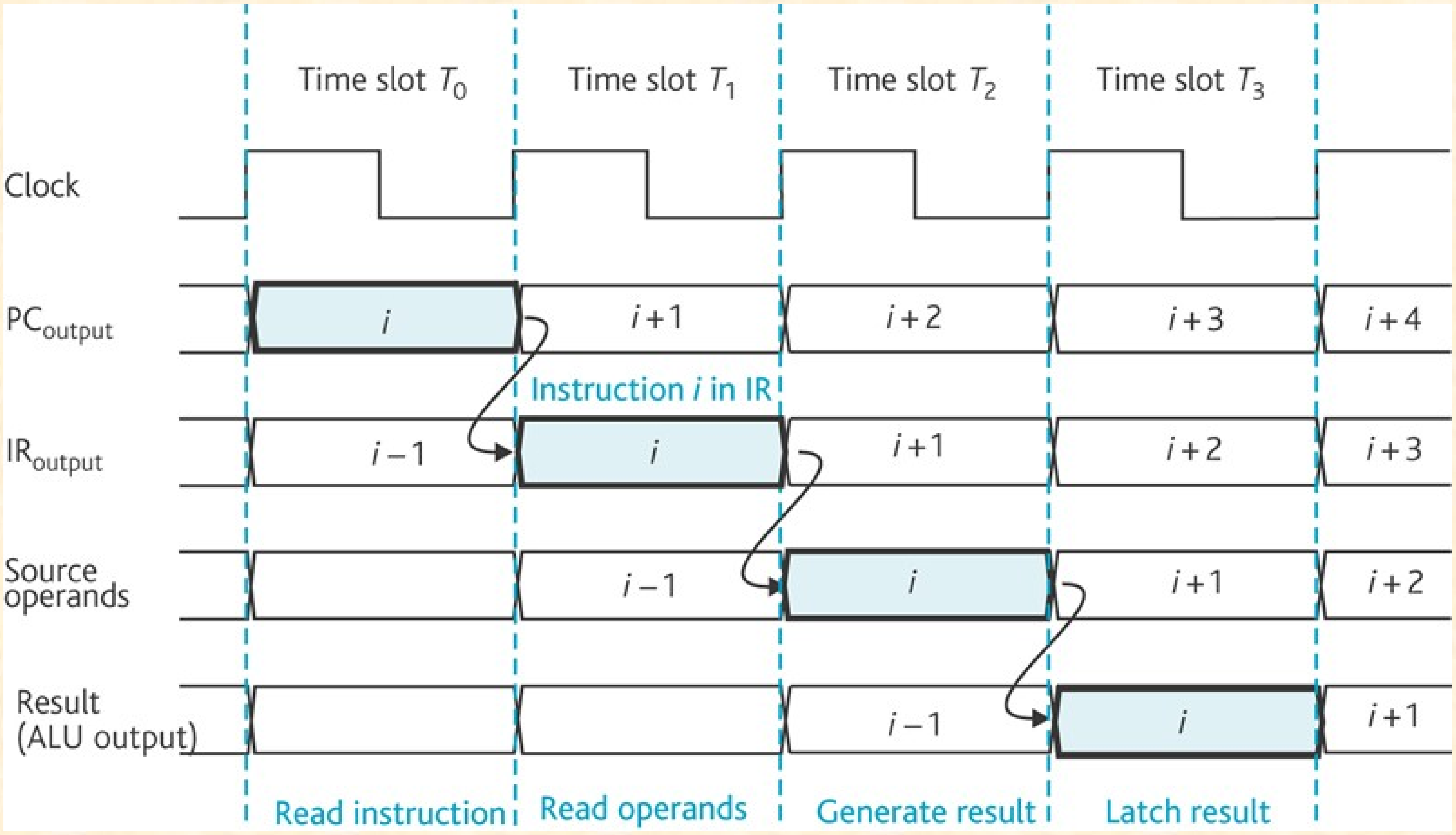
Implementing Branch Prediction

- Dynamic Branch Prediction
  - Prediction made at run time based on past behavior.
  - Processor uses a table that indicates the probability of each branch instruction.
  - The table is updated each time a branch instruction is executed.
  - Single bit branch prediction tables: 80%.
  - 5-bit bit branch prediction tables: 98%.

Using Latches to Implement Pipelining



Timing Diagram for a Pipelined Computer



Time slot T_0
Time slot T_1
Time slot T_2
Time slot T_3
Clock
PC_{output}
i
$i+1$
$i+2$
$i+3$
$i+4$
Instruction i in IR
IR_{output}
$i-1$
i
$i+1$
$i+2$
$i+3$
Source operands
$i-1$
i
$i+1$
$i+2$
Result (ALU output)
$i-1$
i
$i+1$
Read instruction
Read operands
Generate result
Latch result