Instruction Level Parallelism

Introduction

- All processors since about 1985 use pipelining to overlap the execution of instructions and improve performance.

- This potential overlap among instructions is called instruction-level parallelism (ILP), since the instructions can be evaluated in parallel.

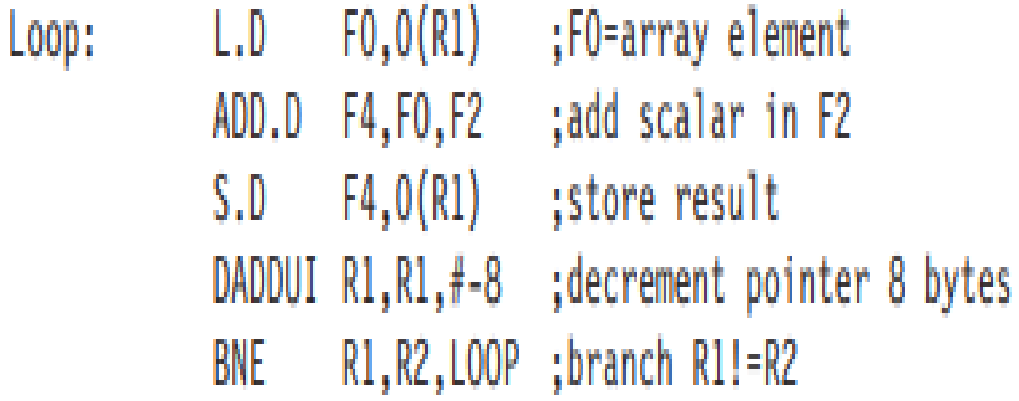
- There are two largely separable approaches to exploiting ILP:
  1. an approach that relies on hardware to help discover and exploit the parallelism dynamically, and
  2. an approach that relies on software technology to find parallelism statically at compile time

Data Dependences and Hazards

- There are three different types of dependences:
  - data dependences (also called true data dependences),
  - name dependences, and
  - control dependences.
- An instruction j is data dependent on instruction i if either of the following holds:
  1. Instruction i produces a result that may be used by instruction j.
  2. Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.

Data Dependence

- consider the following MIPS code sequence that increments a vector of values in memory (starting at 0(R1) and with the last element at 8(R2)) by a scalar in register F2.



```
Loop:    L.D    F0,0(R1)    ;F0=array element
         ADD.D  F4,F0,F2    ;add scalar in F2
         S.D    F4,0(R1)    ;store result
         DADDUI R1,R1,#-8   ;decrement pointer 8 bytes
         BNE    R1,R2,LOOP  ;branch R1!=R2
```

The data dependences in this code sequence involve both floating-point data:

```
Loop:     L.D     F0,0(R1)      ;F0=array element
          ADD.D   F4,F0,F2      ;add scalar in F2
          S.D     F4,0(R1)      ;store result
```

and integer data:

```
          DADDIU  R1,R1,#-8   ;decrement pointer
                              ;8 bytes (per DW)
          BNE     R1,R2,Loop  ;branch R1!=R2
```

- Since a data dependence can limit the amount of instruction-level parallelism
- A dependence can be overcome in two different ways:
  - (1) maintaining the dependence but avoiding a hazard, and
  - (2) eliminating a dependence by transforming the code.
- Scheduling the code is the primary method used to avoid a hazard without altering a dependence, and such scheduling can be done both by the compiler and by the hardware

Name Dependence

- A name dependence occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between the instructions associated with that name.
- There are two types of name dependences between an instruction i that precedes instruction j in program order:

  1. An antidependence between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. The original ordering must be preserved to ensure that i reads the correct value. In the example there is an antidependence between S.D and DADDIU on register R1.

  2. An output dependence occurs when instruction i and instruction j write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j.

Data Hazard

- A hazard exists whenever there is a name or data dependence between instructions, and they are close enough that the overlap during execution would change the order of access to the operand involved in the dependence.

Different Data Hazards

- **RAW (read after write)—**j tries to read a source before i writes it, so j incorrectly gets the old value. This hazard is the most common type and corresponds to a true data dependence. Program order must be preserved to ensure that j receives the value from i.
- **WAW (write after write)—**j tries to write an operand before it is written by i. The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination. This hazard corresponds to an output dependence. WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.
- **WAR (write after read)—**j tries to write a destination before it is read by i, so i incorrectly gets the new value. This hazard arises from an antidependence (or name dependence). WAR hazards cannot occur in most static issue pipelines— even deeper pipelines or floating-point pipelines

Control Dependence

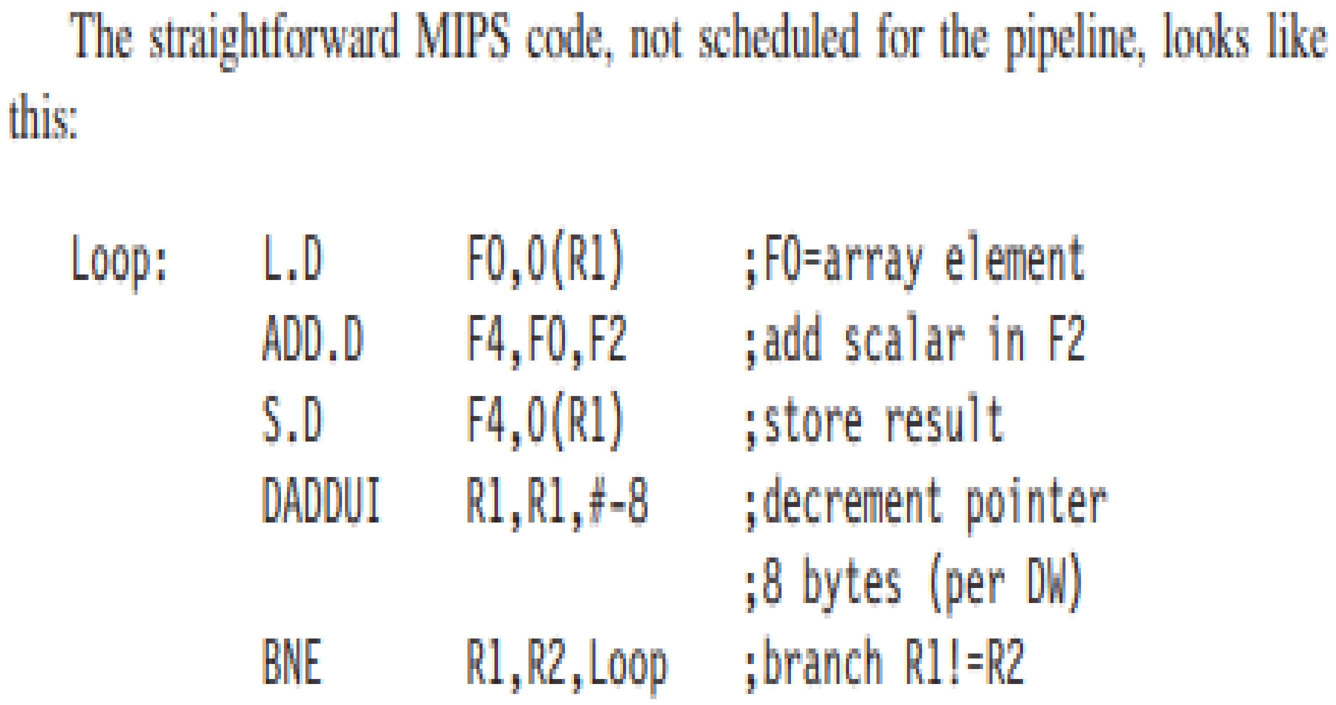
- A control dependence determines the ordering of an instruction, i, with respect to a branch instruction so that instruction i is executed in correct program order and only when it should be. Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches, and, in general, these control dependences must be preserved to preserve program order.

```
if p1 {
        S1;
};
if p2 {
        S2;
}
```

S1 is control dependent on `p1`, and S2 is control dependent on `p2` but not on `p1`. In general, two constraints are imposed by control dependences:

1. An instruction that is control dependent on a branch cannot be moved *before* the branch so that its execution *is no longer controlled* by the branch. For example, we cannot take an instruction from the then portion of an if statement and move it before the if statement.
2. An instruction that is not control dependent on a branch cannot be moved *after* the branch so that its execution *is controlled* by the branch. For example, we cannot take a statement before the if statement and move it into the then portion.

Basic Pipeline Scheduling and Loop Unrolling



The straightforward MIPS code, not scheduled for the pipeline, looks like this:

```
Loop:   L.D     F0,0(R1)     ;F0=array element
        ADD.D   F4,F0,F2     ;add scalar in F2
        S.D     F4,0(R1)     ;store result
        DADDUI  R1,R1,#-8    ;decrement pointer
                             ;8 bytes (per DW)
        BNE     R1,R2,Loop   ;branch R1!=R2
```

Without any scheduling, the loop will execute as follows, taking nine cycles:

```
                                              Clock cycle issued

Loop:   L.D        F0,0(R1)                        1
        stall                                      2
        ADD.D      F4,F0,F2                        3
        stall                                      4
        stall                                      5
        S.D        F4,0(R1)                        6
        DADDUI     R1,R1,#-8                       7
        stall                                      8
        BNE        R1,R2,Loop                      9
```

We can schedule the loop to obtain only two stalls and reduce the time to seven cycles:

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

The stalls after ADD.D are for use by the S.D.

Pipelining: Basic Introduction

- Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction.
- time per instruction on the pipelined processor—assuming ideal conditions—is equal to

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

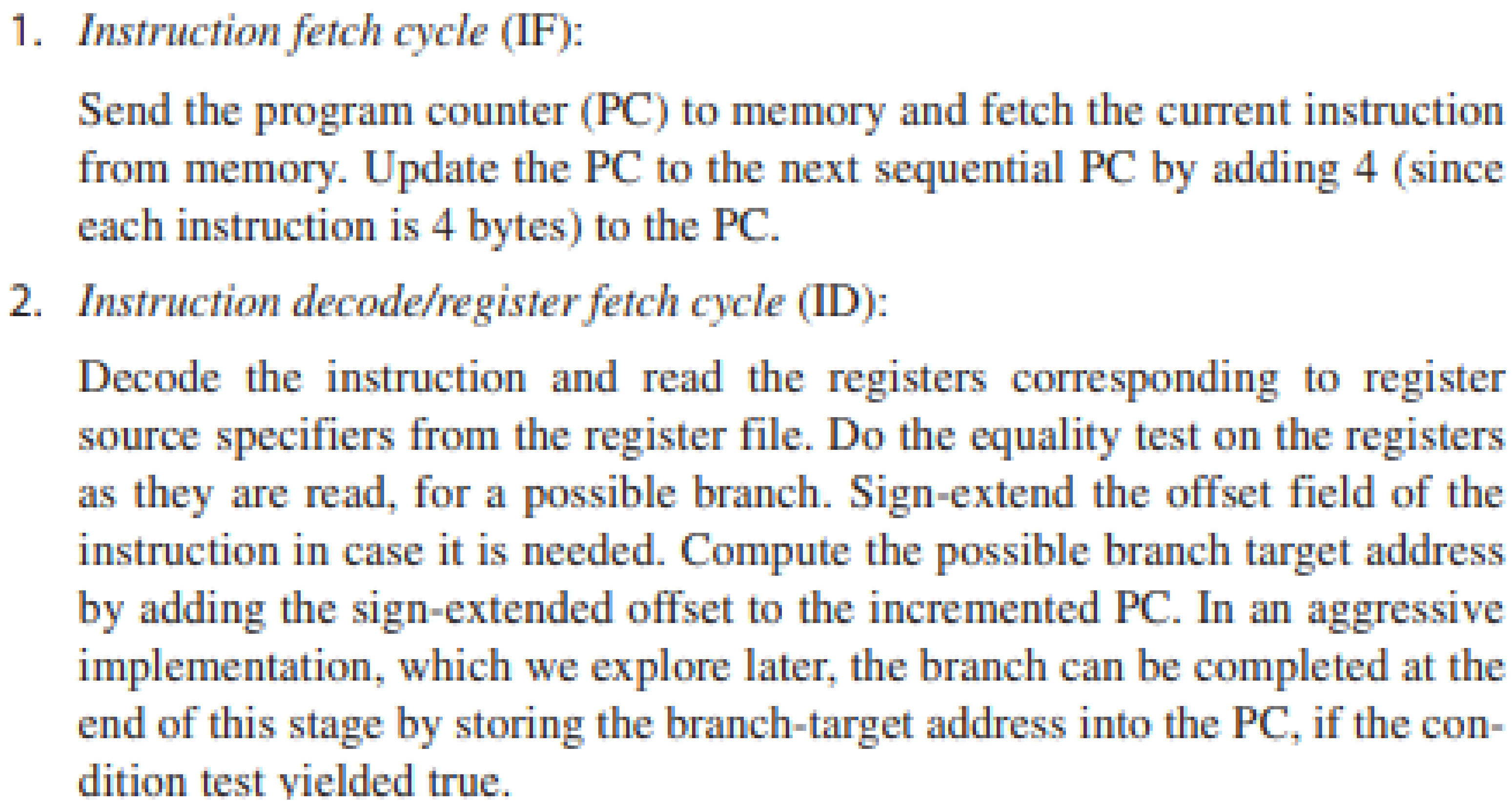
The Basics of a RISC Instruction Set

1. ALU instructions—These instructions take either two registers or a register and a sign-extended immediate (called ALU immediate instructions, they have a 16-bit offset in MIPS), operate on them, and store the result into a third register. Typical operations include add (DADD), subtract (DSUB), and logical operations (such as AND or OR), which do not differentiate between 32-bit and 64-bit versions. Immediate versions of these instructions use the same mnemonics with a suffix of I. In MIPS, there are both signed and unsigned forms of the arithmetic instructions; the unsigned forms, which do not generate overflow exceptions—and thus are the same in 32-bit and 64-bit mode—have a U at the end (e.g., DADDU, DSUBU, DADDIU).

2. Load and store instructions—These instructions take a register source, called the base register, and an immediate field (16-bit in MIPS), called the offset, as operands. The sum—called the effective address—of the contents of the base register and the sign-extended offset is used as a memory address. In the case of a load instruction, a second register operand acts as the destination for the data loaded from memory. In the case of a store, the second register operand is the source of the data that is stored into memory. The instructions load word (LD) and store word (SD) load or store the entire 64-bit register contents

3. Branches and jumps—Branches are conditional transfers of control. There are usually two ways of specifying the branch condition in RISC architectures: with a set of condition bits (sometimes called a condition code) or by a limited set of comparisons between a pair of registers or between a register and zero. MIPS uses the latter

A Simple Implementation of a RISC Instruction Set



1. *Instruction fetch cycle* (IF):

   Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by adding 4 (since each instruction is 4 bytes) to the PC.

2. *Instruction decode/register fetch cycle* (ID):

   Decode the instruction and read the registers corresponding to register source specifiers from the register file. Do the equality test on the registers as they are read, for a possible branch. Sign-extend the offset field of the instruction in case it is needed. Compute the possible branch target address by adding the sign-extended offset to the incremented PC. In an aggressive implementation, which we explore later, the branch can be completed at the end of this stage by storing the branch-target address into the PC, if the condition test yielded true.

3. *Execution/effective address cycle* (EX):

   The ALU operates on the operands prepared in the prior cycle, performing one of three functions depending on the instruction type.

   - Memory reference—The ALU adds the base register and the offset to form the effective address.
   - Register-Register ALU instruction—The ALU performs the operation specified by the ALU opcode on the values read from the register file.
   - Register-Immediate ALU instruction—The ALU performs the operation specified by the ALU opcode on the first value read from the register file and the sign-extended immediate.

   In a load-store architecture the effective address and execution cycles can be combined into a single clock cycle, since no instruction needs to simultaneously calculate a data address and perform an operation on the data.

4. *Memory access* (MEM):

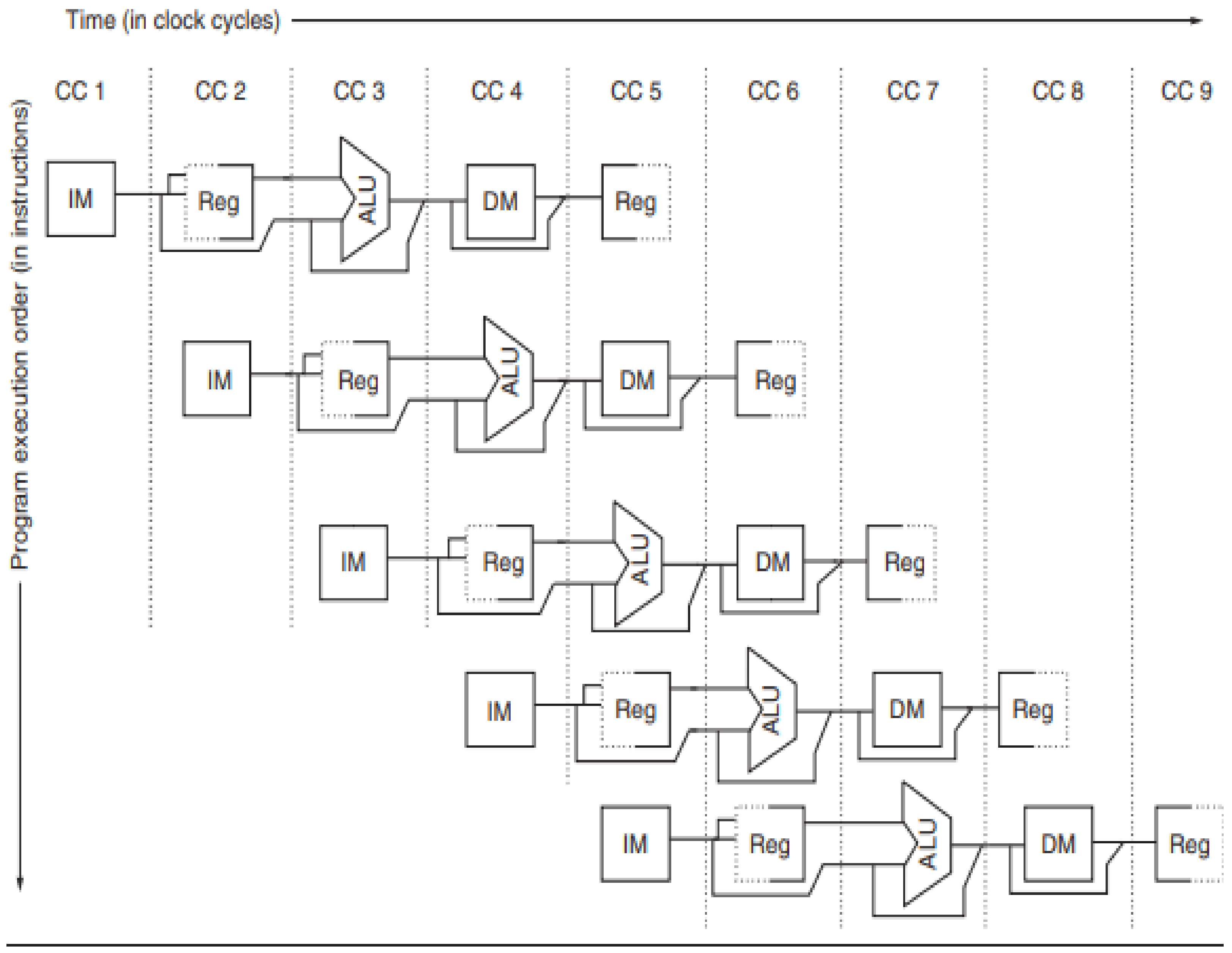
   If the instruction is a load, the memory does a read using the effective address computed in the previous cycle. If it is a store, then the memory writes the data from the second register read from the register file using the effective address.

5. *Write-back cycle* (WB):

   - Register-Register ALU instruction or load instruction:

   Write the result into the register file, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction).

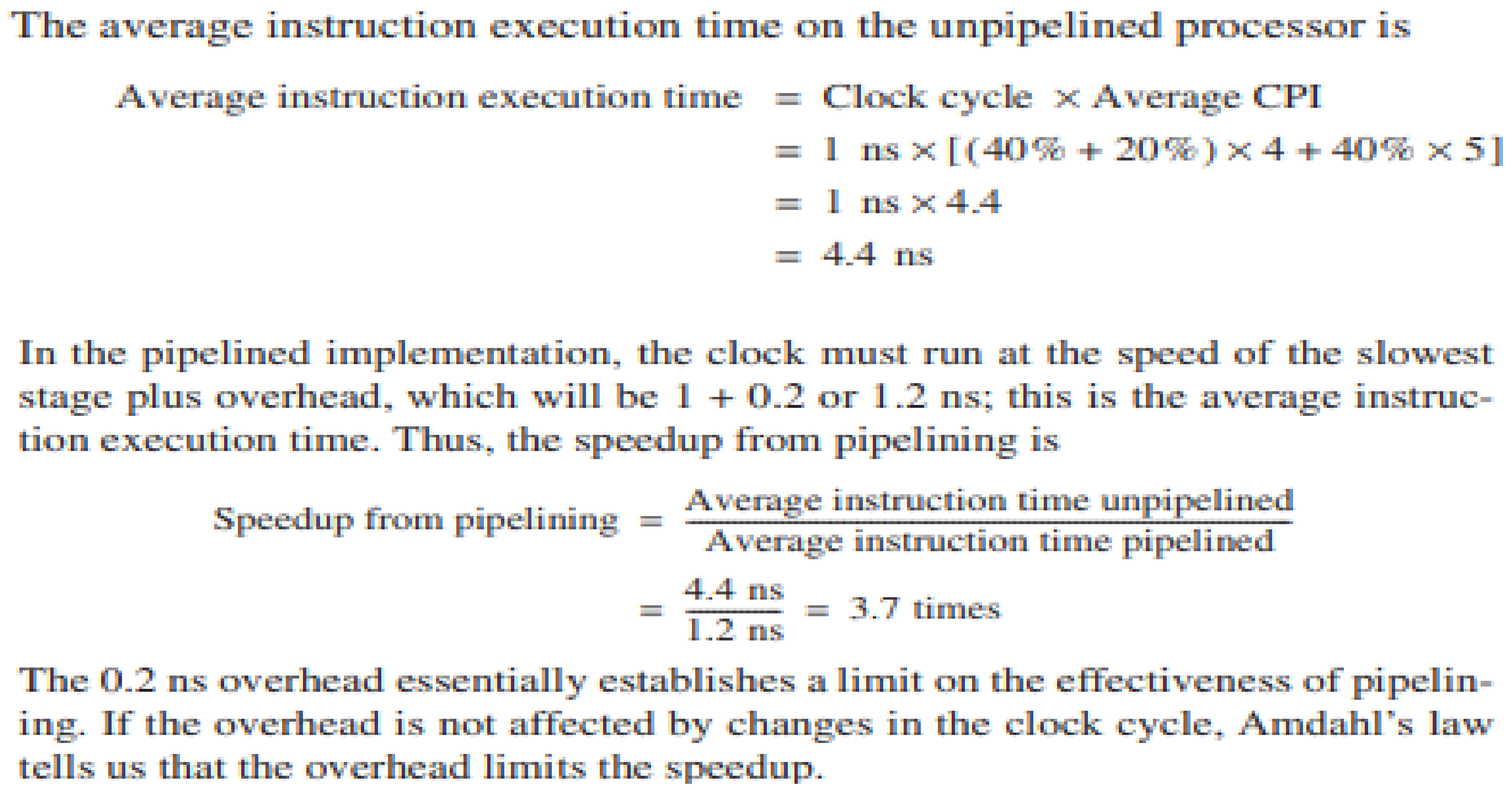
	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Program execution order (in instructions)
IM
Reg
ALU
DM
Reg

Basic Performance Issues in Pipelining

- Consider the unpipelined processor in the previous section. Assume that it has a 1 ns clock cycle and that it uses 4 cycles for ALU operations and branches and 5 cycles for memory operations. Assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively. Suppose that due to clock skew and setup, pipelining the processor adds 0.2 ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

Solution



The average instruction execution time on the unpipelined processor is

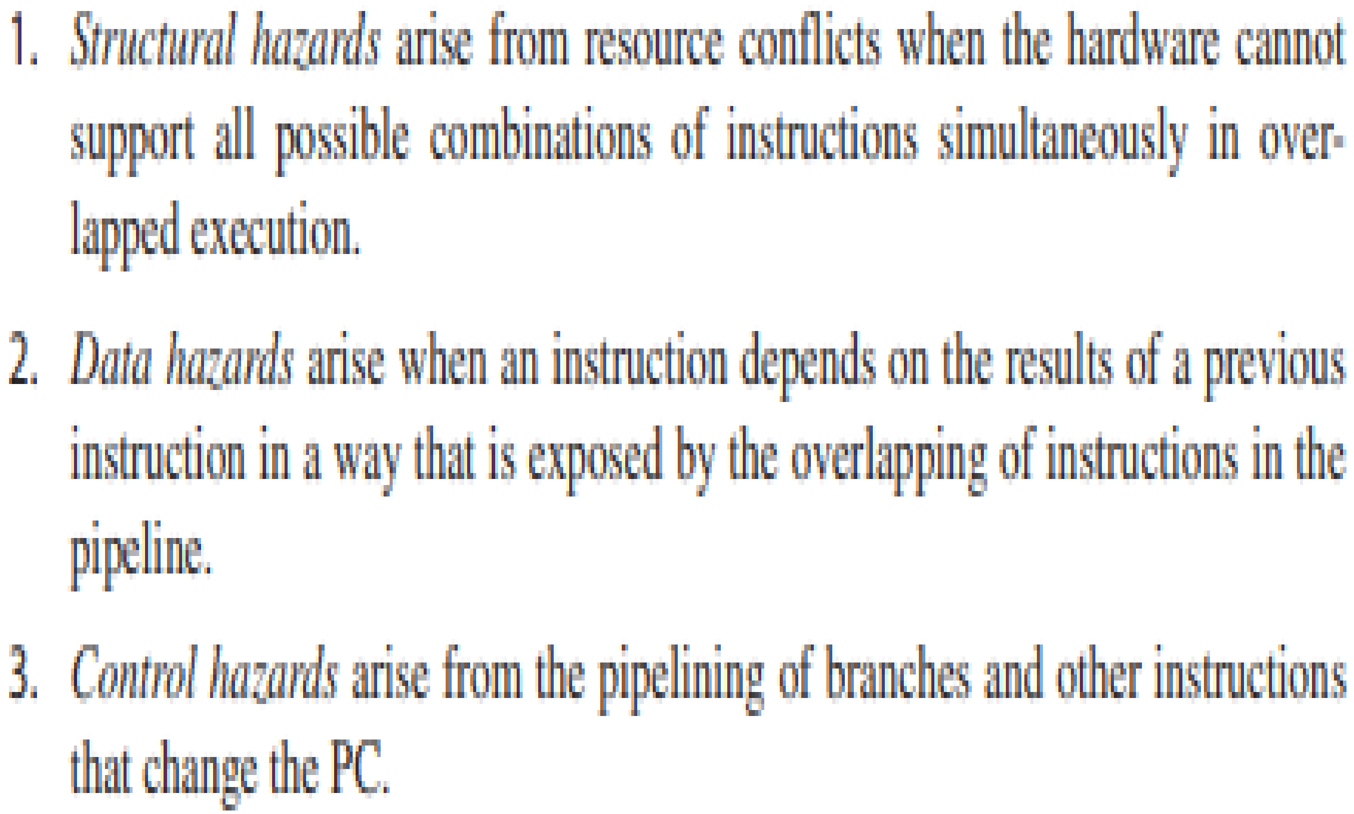
$$\begin{aligned}
\text{Average instruction execution time} &= \text{Clock cycle} \times \text{Average CPI} \\
&= 1\ \text{ns} \times [(40\% + 20\%) \times 4 + 40\% \times 5] \\
&= 1\ \text{ns} \times 4.4 \\
&= 4.4\ \text{ns}
\end{aligned}$$

In the pipelined implementation, the clock must run at the speed of the slowest stage plus overhead, which will be 1 + 0.2 or 1.2 ns; this is the average instruction execution time. Thus, the speedup from pipelining is

$$\begin{aligned}
\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} \\
&= \frac{4.4\ \text{ns}}{1.2\ \text{ns}} = 3.7\ \text{times}
\end{aligned}$$

The 0.2 ns overhead essentially establishes a limit on the effectiveness of pipelining. If the overhead is not affected by changes in the clock cycle, Amdahl's law tells us that the overhead limits the speedup.

Pipeline Hazards



1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in over-lapped execution.
2. *Data hazards* arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
3. *Control hazards* arise from the pipelining of branches and other instructions that change the PC.

Structural Hazards

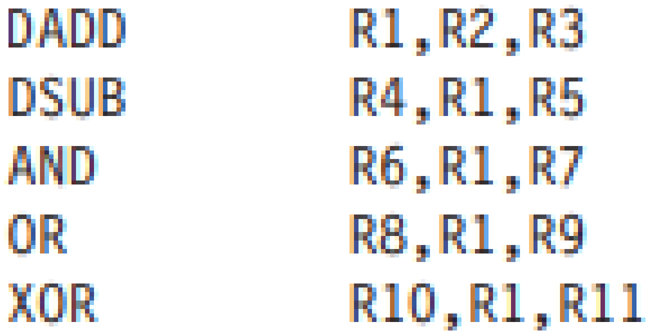
- the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline. If some combination of instructions cannot be accommodated because of resource conflicts, the processor is said to have a structural hazard.

	Clock cycle number									
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Load instruction	IF	ID	EX	MEM	WB					
Instruction *i* + 1		IF	ID	EX	MEM	WB				
Instruction *i* + 2			IF	ID	EX	MEM	WB			
Instruction *i* + 3				Stall	IF	ID	EX	MEM	WB	
Instruction *i* + 4						IF	ID	EX	MEM	WB
Instruction *i* + 5							IF	ID	EX	MEM
Instruction *i* + 6								IF	ID	EX

Figure A pipeline stalled for a structural hazard—a load with one memory port.

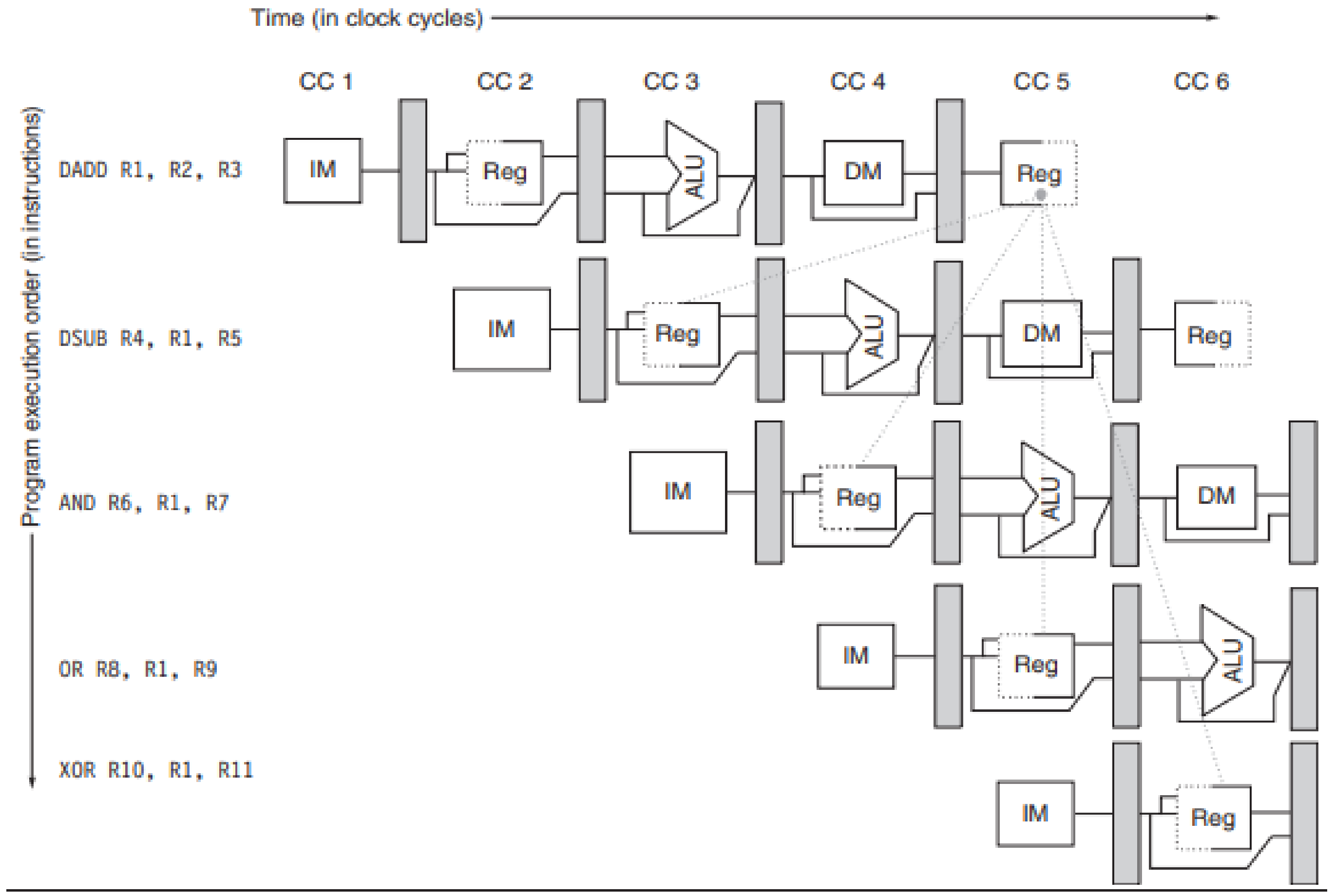
Data Hazards

- Consider the pipelined execution of these instructions:

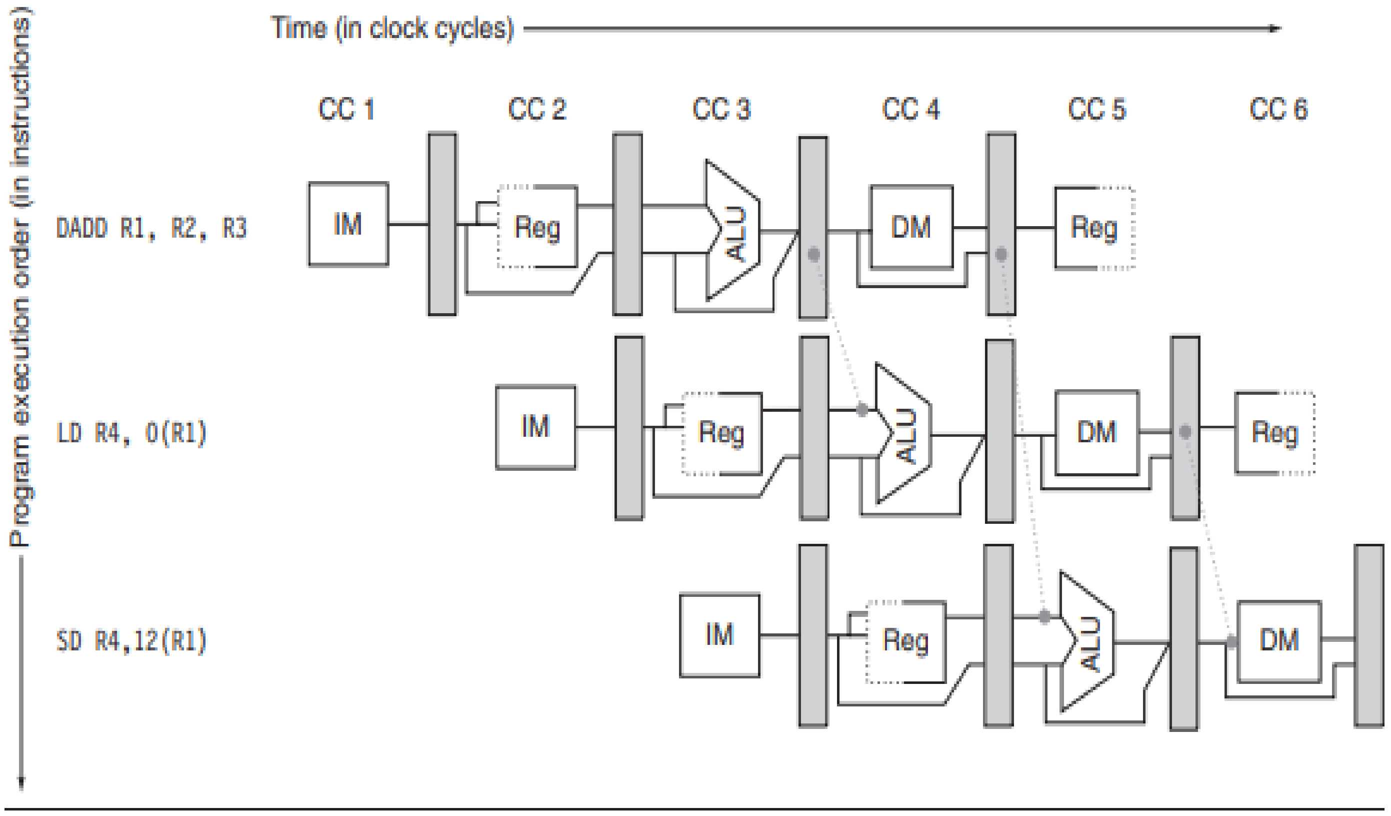


```
DADD    R1,R2,R3
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
XOR     R10,R1,R11
```

Minimizing Data Hazard Stalls by Forwarding



```
DADD            R1,R2,R3
LD              R4,0(R1)
SD              R4,12(R1)
```



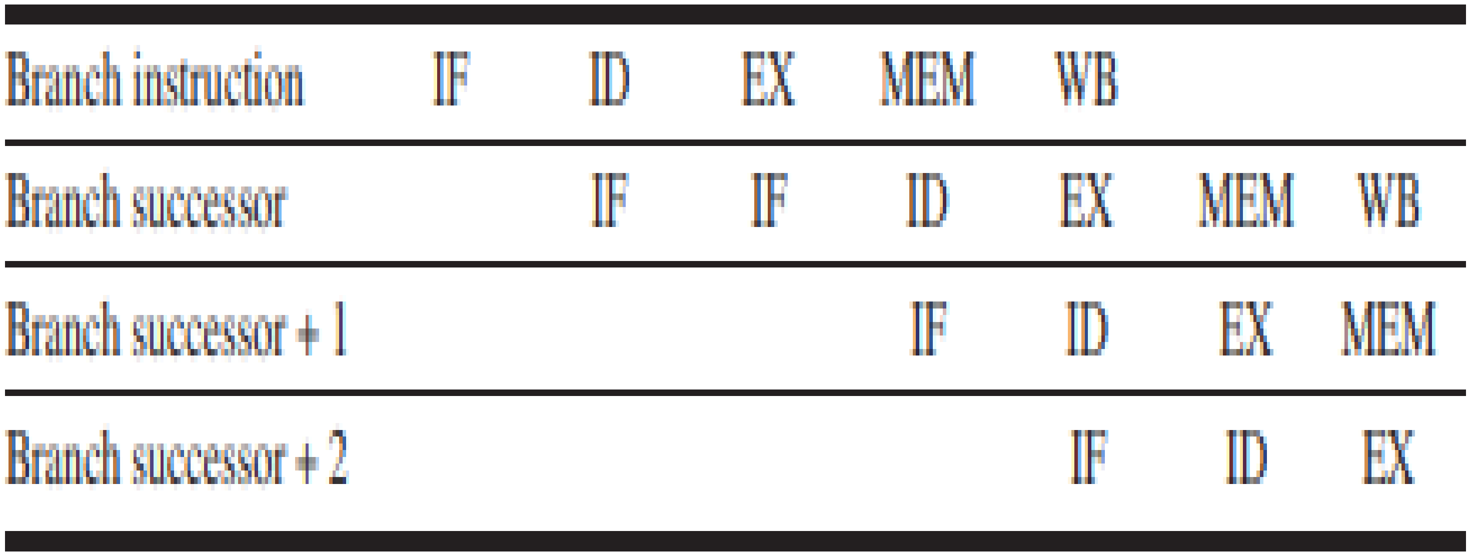
Data Hazards Requiring Stalls

```
LD          R1,0(R2)
DSUB        R4,R1,R5
AND         R6,R1,R7
OR          R8,R1,R9
```

LD R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB R4,R1,R5		IF	ID	EX	MEM	WB			
AND R6,R1,R7			IF	ID	EX	MEM	WB		
OR R8,R1,R9				IF	ID	EX	MEM	WB	

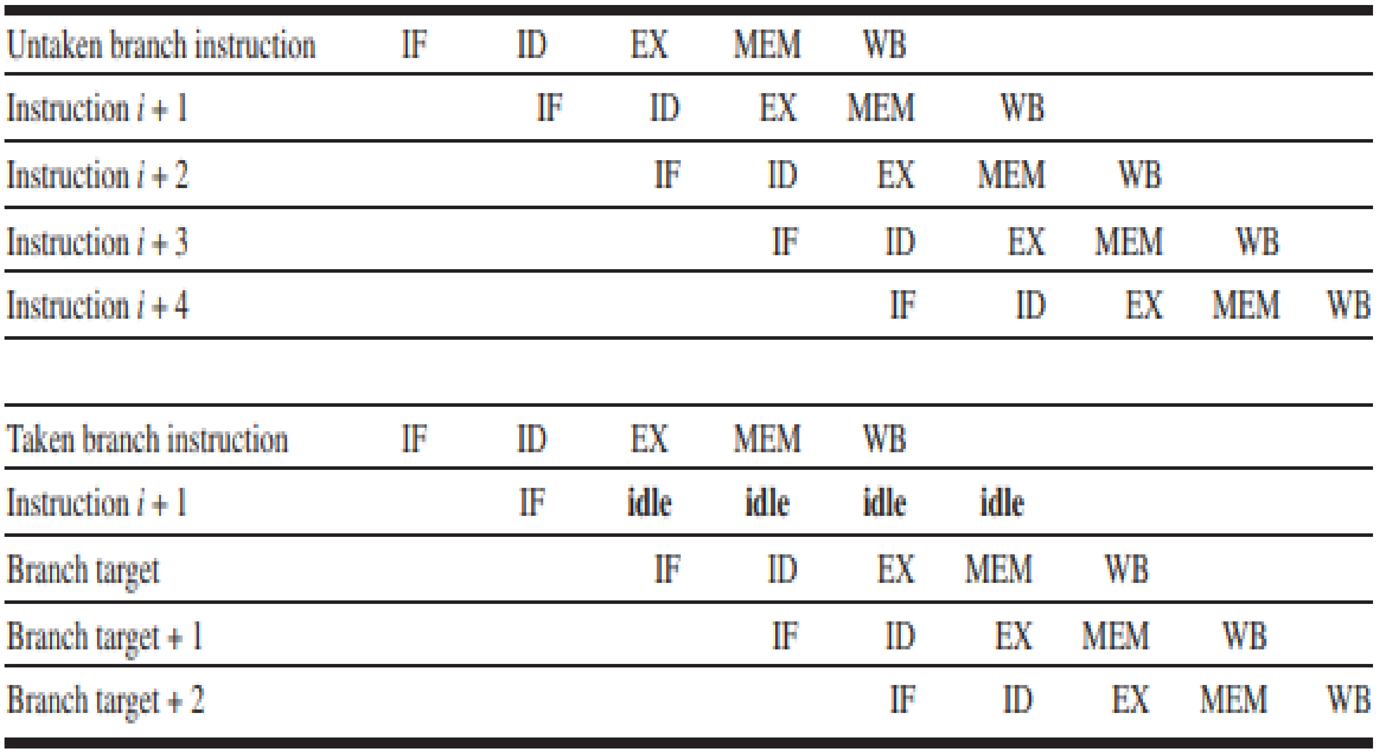
LD R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB R4,R1,R5		IF	ID	stall	EX	MEM	WB		
AND R6,R1,R7			IF	stall	ID	EX	MEM	WB	
OR R8,R1,R9				stall	IF	ID	EX	MEM	WB

Branch Hazards



Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

Reducing Pipeline Branch Penalties



Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

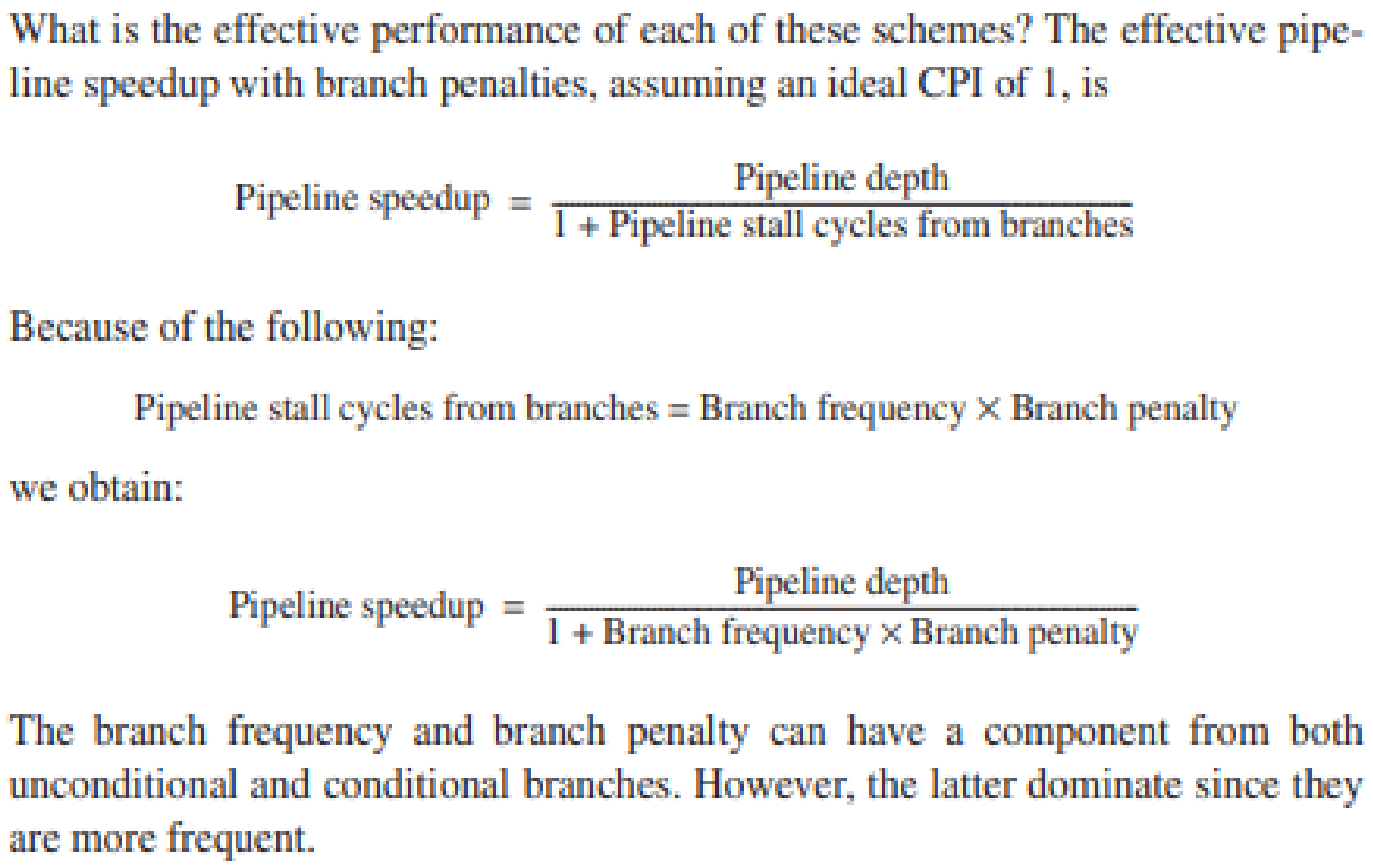
Figure The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom).

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (*i* + 1)		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (*i* + 1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Figure The behavior of a delayed branch is the same whether or not the branch is taken.

Performance of Branch Schemes



What is the effective performance of each of these schemes? The effective pipeline speedup with branch penalties, assuming an ideal CPI of 1, is

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}$$

Because of the following:

$$\text{Pipeline stall cycles from branches} = \text{Branch frequency} \times \text{Branch penalty}$$

we obtain:

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

The branch frequency and branch penalty can have a component from both unconditional and conditional branches. However, the latter dominate since they are more frequent.