Amdahl's Law

The performance gain that can be obtained by improving some portion of a computer can be calculated using Amdahl's law. Amdahl's law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

Amdahl's law defines the *speedup* that can be gained by using a particular feature. What is speedup? Suppose that we can make an enhancement to a computer that will improve performance when it is used. Speedup is the ratio:

$$\text{Speedup} = \frac{\text{Performance for entire task using the enhancement when possible}}{\text{Performance for entire task without using the enhancement}}$$

Alternatively,

$$\text{Speedup} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}$$

Speedup tells us how much faster a task will run using the computer with the enhancement as opposed to the original computer.

Amdahl's law gives us a quick way to find the speedup from some enhancement, which depends on two factors:

1. *The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement*—For example, if 20 seconds of the execution time of a program that takes 60 seconds in total can use an enhancement, the fraction is 20/60. This value, which we will call $\text{Fraction}_{\text{enhanced}}$, is always less than or equal to 1.

2. *The improvement gained by the enhanced execution mode, that is, how much faster the task would run if the enhanced mode were used for the entire program*—This value is the time of the original mode over the time of the enhanced mode. If the enhanced mode takes, say, 2 seconds for a portion of the program, while it is 5 seconds in the original mode, the improvement is 5/2. We will call this value, which is always greater than 1, $\text{Speedup}_{\text{enhanced}}$.

The execution time using the original computer with the enhanced mode will be the time spent using the unenhanced portion of the computer plus the time spent using the enhancement:

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

The overall speedup is the ratio of the execution times:

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

The Processor Performance Equation

Essentially all computers are constructed using a clock running at a constant rate. These discrete time events are called *ticks*, *clock ticks*, *clock periods*, *clocks*, *cycles*, or *clock cycles*. Computer designers refer to the time of a clock period by its duration (e.g., 1 ns) or by its rate (e.g., 1 GHz). CPU time for a program can then be expressed two ways:

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

or

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

In addition to the number of clock cycles needed to execute a program, we can also count the number of instructions executed—the *instruction path length* or *instruction count* (IC). If we know the number of clock cycles and the instruction count, we can calculate the average number of *clock cycles per instruction* (CPI). Because it is easier to work with, and because we will deal with simple processors in this chapter, we use CPI. Designers sometimes also use *instructions per clock* (IPC), which is the inverse of CPI.

CPI is computed as

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

This processor figure of merit provides insight into different styles of instruction sets and implementations, and we will use it extensively in the next four chapters.

By transposing the instruction count in the above formula, clock cycles can be defined as IC × CPI. This allows us to use CPI in the execution time formula:

$$\text{CPU time} = \text{Instruction count} \times \text{Cycles per instruction} \times \text{Clock cycle time}$$

Expanding the first formula into the units of measurement shows how the pieces fit together:

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

As this formula demonstrates, processor performance is dependent upon three characteristics: clock cycle (or rate), clock cycles per instruction, and instruction count. Furthermore, CPU time is *equally* dependent on these three characteristics; for example, a 10% improvement in any one of them leads to a 10% improvement in CPU time.

Unfortunately, it is difficult to change one parameter in complete isolation from others because the basic technologies involved in changing each characteristic are interdependent:

- *Clock cycle time*—Hardware technology and organization
- *CPI*—Organization and instruction set architecture
- *Instruction count*—Instruction set architecture and compiler technology

Luckily, many potential performance improvement techniques primarily improve one component of processor performance with small or predictable impacts on the other two.

Sometimes it is useful in designing the processor to calculate the number of total processor clock cycles as

$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

where IC_i represents the number of times instruction i is executed in a program and CPI_i represents the average number of clocks per instruction for instruction i. This form can be used to express CPU time as

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time}$$

and overall CPI as

$$\text{CPI} = \frac{\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i}{\text{Instruction count}} = \sum_{i=1}^{n} \frac{\text{IC}_i}{\text{Instruction count}} \times \text{CPI}_i$$

The latter form of the CPI calculation uses each individual CPI_i and the fraction of occurrences of that instruction in a program (i.e., $IC_i \div$ Instruction count). CPI_i should be measured and not just calculated from a table in the back of a reference manual since it must include pipeline effects, cache misses, and any other memory system inefficiencies.

Consider our performance example on page 47, here modified to use measurements of the frequency of the instructions and of the instruction CPI values, which, in practice, are obtained by simulation or by hardware instrumentation.

- **Arithmetic average of execution time of all pgms?**
  - **But they vary by 4X in speed, so some would be more important than others in arithmetic average**
- **Could add a weights per program, but how pick weight?**
  - **Different companies want different weights for their products**
- **SPECRatio: Normalize execution times to reference computer, yielding a ratio proportional to performance =**

$$\frac{\text{time on reference computer}}{\text{time on computer being rated}}$$

- **If program SPECRatio on Computer A is 1.25 times bigger than Computer B, then**

$$1.25 = \frac{SPECRatio_A}{SPECRatio_B} = \frac{\dfrac{ExecutionTime_{reference}}{ExecutionTime_A}}{\dfrac{ExecutionTime_{reference}}{ExecutionTime_B}}$$

$$= \frac{ExecutionTime_B}{ExecutionTime_A} = \frac{Performance_A}{Performance_B}$$

  - **Note that when comparing 2 computers as a ratio, execution times on the reference computer drop out, so choice of reference computer is irrelevant**

- **Since ratios, proper mean is geometric mean (SPECRatio unitless, so arithmetic mean meaningless)**

$$GeometricMean = \sqrt[n]{\prod_{i=1}^{n} SPECRatio_i}$$

  - **2 points make geometric mean of ratios attractive to summarize performance:**

1. **Geometric mean of the ratios is the same as the ratio of the geometric means**
2. **Ratio of geometric means = Geometric mean of performance ratios ⇒ choice of reference computer is irrelevant!**

Weighted Arithmetic Mean. For many cases, computing an equal-weight arithmetic mean will give misleading results. Care must be taken when events occur at different fractions of the total events and each event requires a different amount of time. The weighted time per event is the weighted arithmetic mean, defined as

$$\text{weighted arithmetic mean} = \sum_{i=1}^{n} W_i T_i.$$

The weighted arithmetic mean is the central tendency of time per unit of work. W_i is the fraction that operation i is of the total operations, and T_i is the time consumed by each use. Note that $W_1 + W_2 + \cdots + W_n = 1$ and that W_i is not the fraction of time that the operation is in use.

EXAMPLE

A processor has two classes of instructions: class A instructions take two clocks to execute whereas class B instructions take three clocks to execute. Of all the instructions executed, 75% are class A instructions and 25% are class B instructions. What is the CPI of this processor?

Solution

The observations are in time and are weighted. Thus the CPI of the processor is determined by the weighted arithmetic mean:

$$\text{CPI} = W_A \text{ CPI}_A + W_B \text{ CPI}_B,$$

$$\text{CPI} = (0.75 \times 2) + (0.25 \times 3) = 1.5 + 0.75 = 2.25 \text{ clocks per instruction}$$

Comment

When solving a problem such as this one, add the event probabilities together and verify that the sum is one; if the sum is not equal to one, there is some error in the solution. A good practice is to use a table, as shown in Table 2.1, for the solution of these problems rather than attempt to bind variables to an equation.

Next: Many Examples: **Up:** Chapter 4: Instruction Set **Previous:** An Example: the M68000

Instruction Set of MIPS Processor

- **Register file (RF):** 32 registers ($0 through $31), each for a word of 32 bits (4 bytes);
  - $0 always holds zero
  - $sp (29) is the stack pointer (SP) which always points to the top item of a stack in the memory;
  - $ra (31) always holds the return address from a subroutine

  The table below shows the conventional usage of all 32 registers.

Register Number	Mnemonic Name	Conventional Use	Register Number	Mnemonic Name	Conventional Use
$0	zero	Permanently 0	$24, $25	$t8, $t9	Temporary
$1	$at	Assembler Temporary (reserved)	$26, $27	$k0, $k1	Kernel (reserved for OS)
$2, $3	$v0, $v1	Value returned by a subroutine	$28	$gp	Global Pointer
$4–$7	$a0–$a3	Arguments to a subroutine	$29	$sp	Stack Pointer
$8–$15	$t0–$t7	Temporary (not preserved across a function call)	$30	$fp	Frame Pointer
$16–$23	$s0–$s7	Saved registers (preserved across a function call)	$31	$ra	Return Address

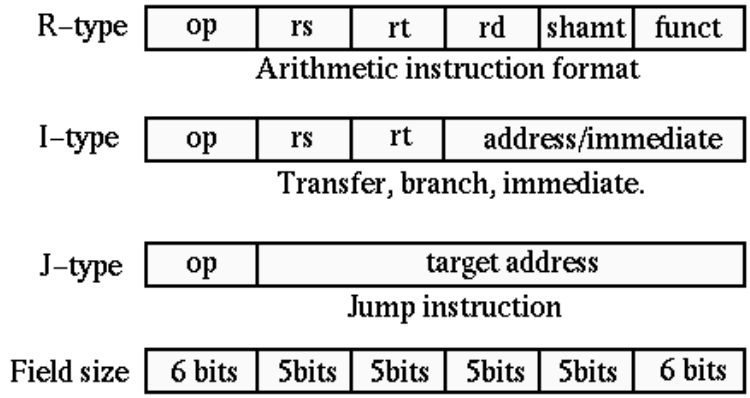
MIPS registers

- **Main memory (MM):** 2^{32} addressable bytes ($0, 1, 2, 3, \cdots, 2^{32}-1 = 4,294,967,295$, 4 Gb) or 2^{30} words ($0, 4, 8, 12, \cdots, 2^{32}-4 = 4,294,967,292$).

$2^{32}-1$	$2^{32}-2$	$2^{32}-3$	$2^{32}-4$
....	....	....	....
19	18	17	16
15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	0

MIPS virtual memory

- **Instruction set**: each instruction in the instruction set describes one particular CUP operation. Each instruction is represented in both **assembly language** by the *mnimonics* and **machine language** (binary) by a word of 32 bits subdivided into several fields.



There are different types of instructions:

1. Computational Instructions

   These instructions are for arithmetic or logic manipulations. In general they operate on two operands and store the result.

op $rd, $rs, $rt

   $rd ← $rs * $rt

   where

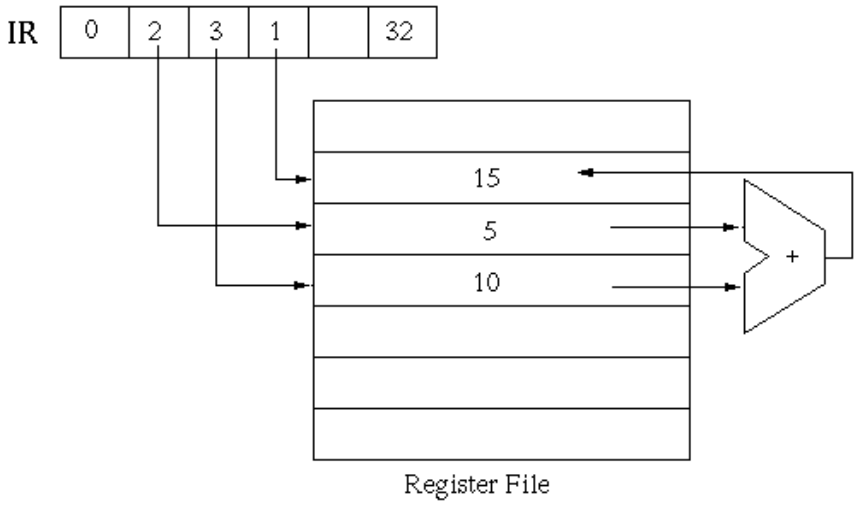
   - op: opcode specifying the arithmetic/logic operation to be performed;
   - $rd: destination register in which the result is to be stored;
   - $rs: source register containing the 1st operand;
   - $rt: source register containing the 2nd operand.

   Opcode can be: add, sub, mult, div, and, or, etc.

   $rd, $rs, $rt can be any of the 32 registers.

   The assembly instruction: add $1 $2 $3

   The machine instruction:

   Note:

   - The destination register is specified in the first field following the opcode field in the assembly language instruction, but the last 5-bit field in the binary machine language instruction.
   - In all R-type data manipulation instructions (arithmetic, logical, shift), the operations are specified by the function field (6 least significant bits) in the binary instruction, with the opcode field (6 most significant bits) all equal to zero.

Note:

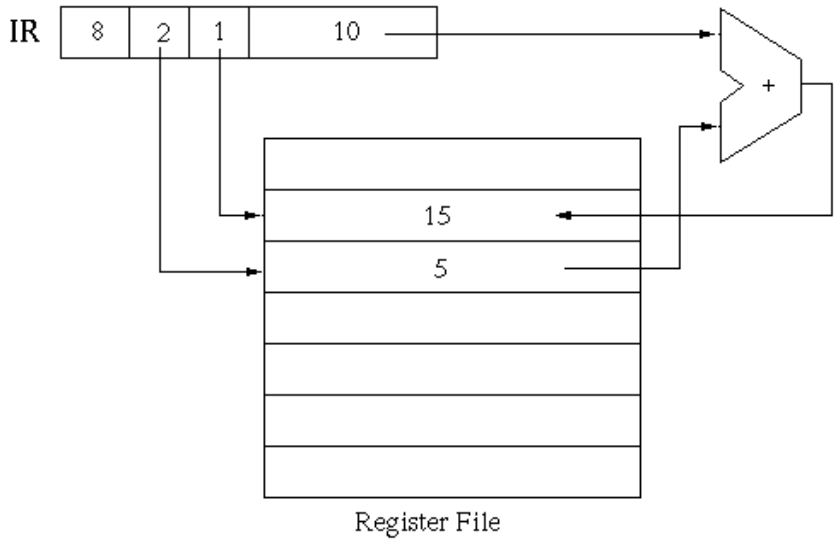
Immediate mode - replacing $rt by a constant:

op $rd, $rs, constant

where op can only be: add, and, or.

The assembly instruction: addi $1 $2 10

The machine instruction:



2. Shift Instructions

op $rd, $rs, shamt

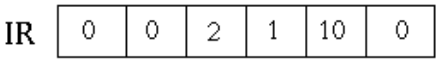
where op can be sll (shift left logical) srl (shift right logical) or sra (shift right arithmetic), and $shamt$ specifies the number of bits to shift. The shift amount can also be specified by a variable in a register, such as the example below ($t1 holds the shift amount):

srlv $rd, $rs, $t1

Examples:

The assembly instruction: sll $1 $2 10
($1 ← $2 << 10)

The machine instruction:



The assembly instruction: srl $1 $2 10
($1 ← $2 >> 10)

The machine instruction:

3. Data Transfer Instructions

These instructions transfer data back and forth between the MM and the CPU.

- Load word from MM to register:

```
lw   $rd, offset($rs)
```

$rd ← Memory[offset + $rs]

where

- lw: the opcode for *load word*;
- $rd: destination register into which the word is to be loaded;
- $rs: source register (e.g., an index number of an array);
- offset: (e.g., the beginning address of the array in memory).
- effective MM address: offset + $rs

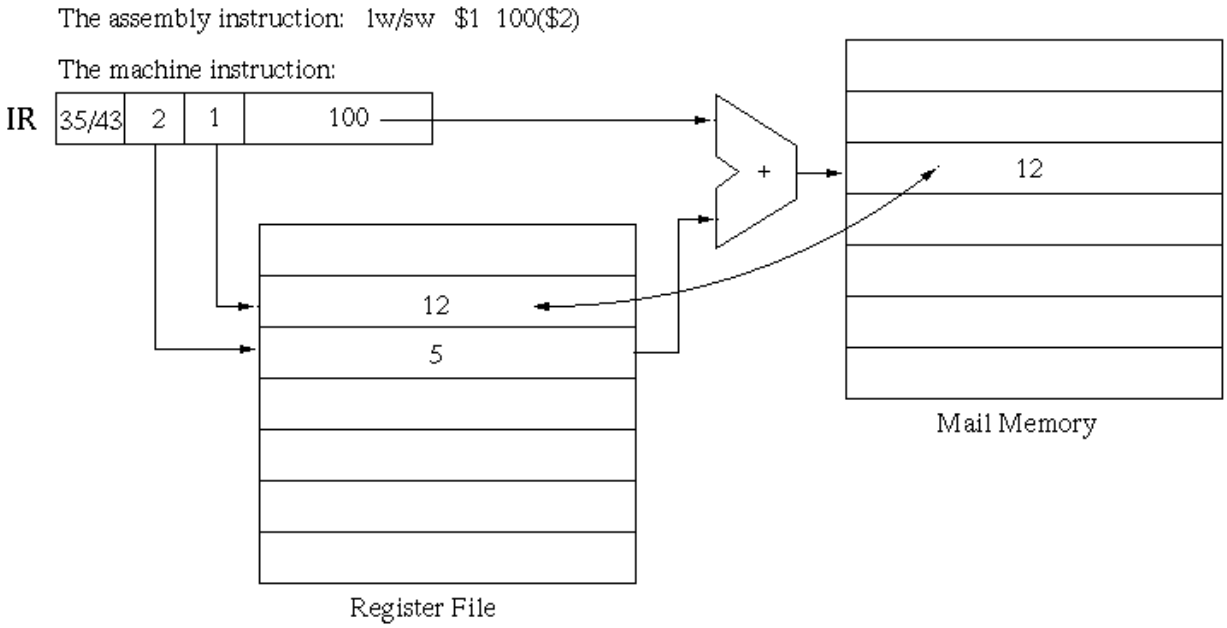
- Store word from register to memory:

```
sw   $rs, offset($rd)
```

$rs → Memory[offset + $rd]

where

- sw: the opcode for *save word*;
- $rs: source register whose content is to be stored;
- $rd: destination register, e.g., an index number of an array;
- offset: e.g., the beginning address of the array in memory.
- MM address: offset + $rd



4. Program Control

Usually the program is executed in the straight line fashion, i.e., the next instruction to be executed is the one that follows the previous one currently being executed. But some time it is needed to conditionally or unconditionally jump to some other part of the program (e.g., functions, loops, etc.) by the program control instructions.

- Branch to a labeled statement if two variables are equal:

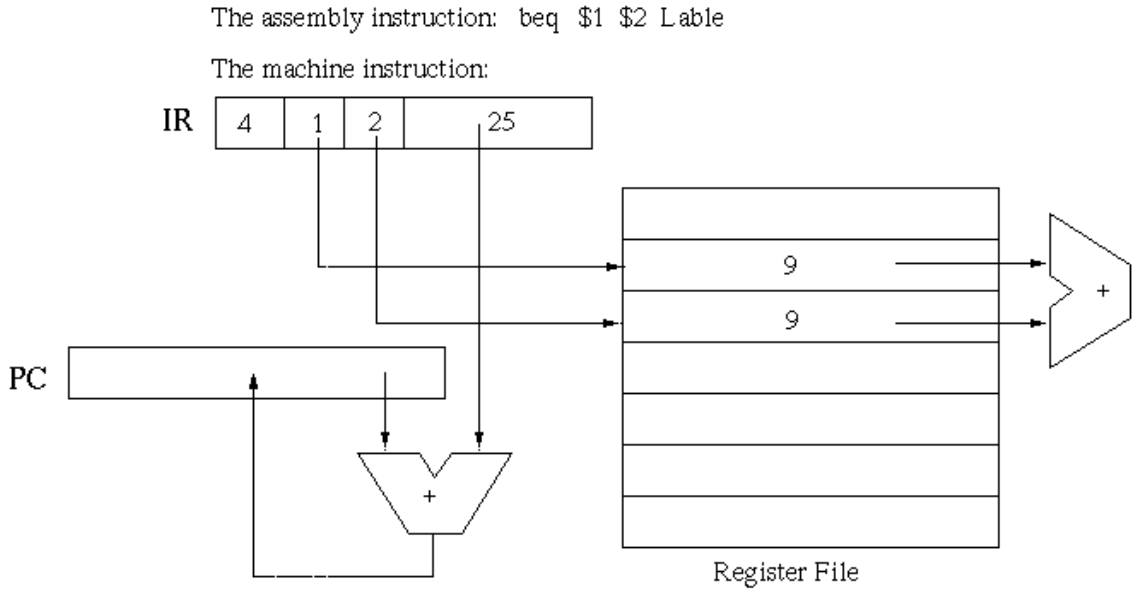
```
beq   $rd, $rs, L1
```

or not equal:

```
bne   $rd, $rs, L1
```

where

- beq (bne): branch if equal (not equal);
- $rd, $rs: two registers holding the variables;
- L1: label for the target statement.



- Set a register to 1 if first variable is smaller than the second, set the register to 0 otherwise.

```
slt   $rd, $rs, $rt
```

where

- slt: set if less than
- $rs, $rt: registers containing two variables to be compared;
- $rd: register to be set to 1 if $rs $<$ $rt, or 0 otherwise.

Immediate mode -- replacing $rt by a constant:

```
slti   $rd, $rs, constant
```

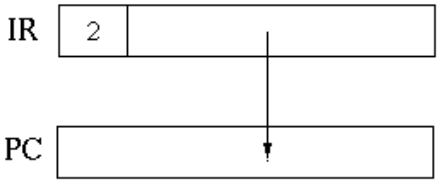
set $rd to 1 if $rs $<$ constant.

- Jump unconditionally to a certain labeled statement:

```
j   label
```

The assmebly instruction: j Label

The machine instruction:



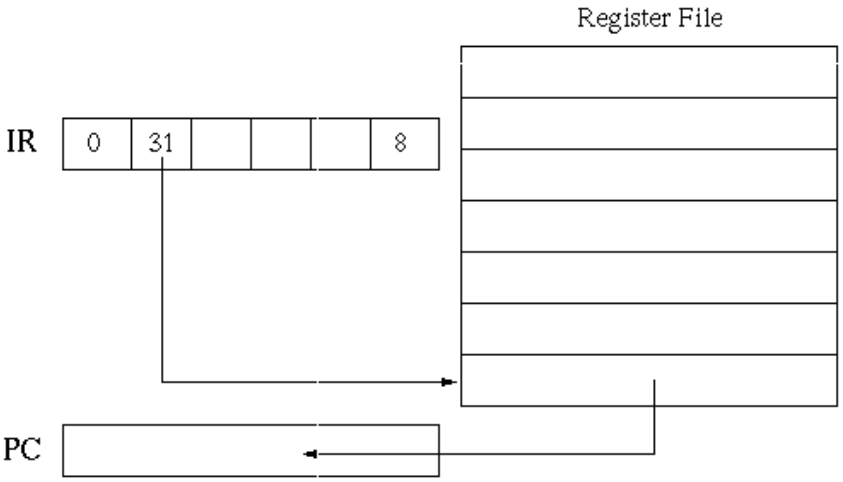
- Jump unconditionally to a certain statement whose address is given in a register:

```
jr   $rd
```

- Jump unconditionally to a subroutine (or function, procedure, etc.) whose address is given as a symbol SubroutineAddress:

The assmebly instruction: jr $31

The machine instruction:




```
jal   SubroutineAddress
```

Operations related to function calls:

- Branch to a subroutine by jal (jump and link)
  1. update PC ($PC \leftarrow PC + 4$) to point to the next instruction;
  2. save content of PC into register $ra ($31) as the return address;
  3. load starting address of subroutine (symbolized as SubroutineAddress) to PC.
- Return to calling routine:

  To return from subroutine to the calling routine, the last statement of the subroutine must be:

```
jr   $ra
```

- Recursion: For recursive subroutine calls, previous content of $ra ($31) needs to be stored in a *stack* in the memory.

MIPS R3000 Instruction Set

MIPS Machine Language

Next: Many Examples: **Up:** Chapter 4: Instruction Set **Previous:** An Example: the M68000

Ruye Wang 2003-10-30

MIPS Assembly/Instruction Formats

This page describes the implementation details of the MIPS instruction formats.

Contents

- 1 R Instructions
  - 1.1 R Format
  - 1.2 Function Codes
  - 1.3 Shift Values
- 2 I Instructions
  - 2.1 I Format
- 3 J Instructions
  - 3.1 J Format
- 4 FR Instructions
- 5 FI Instructions
- 6 Opcodes

R Instructions

R instructions are used when all the data values used by the instruction are located in registers.

All R-type instructions have the following format:

```
OP rd, rs, rt
```

Where "OP" is the mnemonic for the particular instruction. *rs*, and *rt* are the source registers, and *rd* is the destination register. As an example, the **add** mnemonic can be used as:

```
add $s1, $s2, $s3
```

Where the values in *$s2* and *$s3* are added together, and the result is stored in *$s1*. In the main narrative of this book, the operands will be denoted by these names.

R Format

Converting an R mnemonic into the equivalent binary machine code is performed in the following way:

opcode	rs	rt	rd	shift (shamt)	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

opcode
: The opcode is the machinecode representation of the instruction mnemonic. Several related instructions can have the same opcode. The opcode field is 6 bits long (bit 26 to bit 31).

rs, rt, rd
: The numeric representations of the source registers and the destination register. These numbers correspond to the $X representation of a register, such as $0 or $31. Each of these fields is 5 bits long. (25 to 21, 20 to 16, and 15 to 11, respectively). Interestingly, rather than rs and rt being named r1 and r2

(for source register 1 and 2), the registers were named "rs" and "rt" because t comes after s in the alphabet. This was most likely done to reduce numerical confusion.

Shift (shamt)
Used with the shift and rotate instructions, this is the amount by which the source operand *rs* is rotated/shifted. This field is 5 bits long (6 to 10).

Funct
For instructions that share an opcode, the **funct** parameter contains the necessary control codes to differentiate the different instructions. 6 bits long (0 to 5). Example: Opcode 0x00 accesses the ALU, and the funct selects which ALU function to use.

Function Codes

Because several functions can have the same opcode, R-Type instructions need a function (Func) code to identify what exactly is being done - for example, 0x00 refers to an ALU operation and 0x20 refers to ADDing specifically.

Shift Values

I Instructions

I instructions are used when the instruction must operate on an immediate value and a register value. Immediate values may be a maximum of 16 bits long. Larger numbers may not be manipulated by immediate instructions.

I instructions are called in the following way:

```
OP rt, rs, IMM
```

Where *rt* is the target register, *rs* is the source register, and *IMM* is the immediate value. The immediate value can be up to 16 bits long. For instance, the **addi** instruction can be called as:

```
addi $s1, $s2, 100
```

Where the value of $s2 plus 100 is stored in $s1.

I Format

I instructions are converted into machine code words in the following format:

opcode	rs	rt	IMM
6 bits	5 bits	5 bits	16 bits

Opcode
The 6-bit opcode of the instruction. In I instructions, all mneumonics have a one-to-one correspondence with the underlying opcodes. This is because there is no **funct** parameter to differentiate instructions with an identical opcode. 6 bits (26 to 31)

rs, rt
The source and target register operands, respectively. 5 bits each (21 to 25 and 16 to 20, respectively).[1] (http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/format.html)

IMM
The 16 bit immediate value. 16 bits (0 to 15). This value is usually used as the offset value in various instructions, and depending on the instruction, may be expressed in two's complement.

J Instructions

J instructions are used when a jump needs to be performed. The J instruction has the most space for an immediate value, because addresses are large numbers.

J instructions are called in the following way:

```
OP LABEL
```

Where *OP* is the mnemonic for the particular jump instruction, and *LABEL* is the target address to jump to.

J Format

J instructions have the following machine-code format:

Opcode	Pseudo-Address

Opcode
: The 6 bit opcode corresponding to the particular jump command. (26 to 31).

Address
: A 26-bit shortened address of the destination. (0 to 25). The two most LSBits are removed, and the 4 MSBits are removed, and assumed to be the same as the current instruction's address.

FR Instructions

FR instructions are similar to the R instructions described above, except they are reserved for use with floating-point numbers:

Opcode	fmt	ft	fs	fd	funct

FI Instructions

FI instructions are similar to the I instructions described above, except they are reserved for use with floating-point numbers:

Opcode	fmt	ft	Imm

Opcodes

The following table contains a listing of MIPS instructions and the corresponding opcodes. Opcode and funct numbers are all listed in hexadecimal.

Mnemonic	Meaning	Type	Opcode	Funct
`add`	Add	R	0x00	0x20
`addi`	Add Immediate	I	0x08	NA
`addiu`	Add Unsigned Immediate	I	0x09	NA
`addu`	Add Unsigned	R	0x00	0x21
`and`	Bitwise AND	R	0x00	0x24
`andi`	Bitwise AND Immediate	I	0x0C	NA
`beq`	Branch if Equal	I	0x04	NA
`bne`	Branch if Not Equal	I	0x05	NA
`div`	Divide	R	0x00	0x1A
`divu`	Unsigned Divide	R	0x00	0x1B
`j`	Jump to Address	J	0x02	NA
`jal`	Jump and Link	J	0x03	NA
`jr`	Jump to Address in Register	R	0x00	0x08
`lbu`	Load Byte Unsigned	I	0x24	NA
`lhu`	Load Halfword Unsigned	I	0x25	NA
`lui`	Load Upper Immediate	I	0x0F	NA
`lw`	Load Word	I	0x23	NA
`mfhi`	Move from HI Register	R	0x00	0x10
`mflo`	Move from LO Register	R	0x00	0x12
`mfc0`	Move from Coprocessor 0	R	0x10	NA
`mult`	Multiply	R	0x00	0x18
`multu`	Unsigned Multiply	R	0x00	0x19
`nor`	Bitwise NOR (NOT-OR)	R	0x00	0x27
`xor`	Bitwise XOR (Exclusive-OR)	R	0x00	0x26
`or`	Bitwise OR	R	0x00	0x25
`ori`	Bitwise OR Immediate	I	0x0D	NA
`sb`	Store Byte	I	0x28	NA
`sh`	Store Halfword	I	0x29	NA
`slt`	Set to 1 if Less Than	R	0x00	0x2A
`slti`	Set to 1 if Less Than Immediate	I	0x0A	NA
`sltiu`	Set to 1 if Less Than Unsigned Immediate	I	0x0B	NA
`sltu`	Set to 1 if Less Than Unsigned	R	0x00	0x2B
`sll`	Logical Shift Left	R	0x00	0x00
`srl`	Logical Shift Right (0-extended)	R	0x00	0x02
`sra`	Arithmetic Shift Right (sign-extended)	R	0x00	0x03
`sub`	Subtract	R	0x00	0x22
`subu`	Unsigned Subtract	R	0x00	0x23

Mnemonic	Meaning	Type	Opcode	Funct
sw	Store Word	I	0x2B	NA

MIPS R3000 Instruction Set Summary

MIPS Operands

Name	Example	Comments
32 registers	$0, $1, $2,..., $31	Fast location for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $0 always equal 0. Register $1 is reserved for the assembler to handle pseudo instructions and large constants
2^{30} memory words	Memory[0], Memory[4],..., Memory[4293967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls

MIPS Assembler Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
	subtract	sub $1,$2,$3	$1 = $2 - $3	3 operands; exception possible
	add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
	add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
	subtract unsigned	subi $1,$2,$3	$1 = $2 - $3	3 operands; exception possible
	add immediate unsigned	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
	Move from coprocessor register	mfc0 $1,$epc	$1 = $epc	Used to get of Exception PC
Logical	and	and $1,$2,$3	$1 = $2 & $3	3 register operands; Logical AND
	or	or $1,$2,$3	$1 = $2 | $3	3 register operands; Logical OR
	and immediate	and $1,$2,100	$1 = $2 & 100	Logical AND register, constant
	or immediate	or $1,$2,100	$1 = $2 | 100	Logical OR register, constant
	shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
	shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
Data transfer	load word	lw $1, (100)$2	$1 = Memory[$2+100]	Data from memory to register
	store word	sw $1, (100)$2	Memory[$2+100] = $1	Data from memory to register
	load upper immediate	lui $1,100	$1 = 100 * 2^{16}	Load constant in upper 16bits

Conditional branch	branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+100	Equal test; PC relative branch
	branch on not equal	bne $1,$2,100	if ($1 != $2) go to PC+4+100	Not equal test; PC relative
	set on less than	slt $1,$2,$3	if ($2 < $3) $1 = 1; else $1 = 0	Compare less than; 2`s complement
	set less than immediate	slti $1,$2,100	if ($2 < 100) $1 = 1; else $1 = 0	Compare < constant; 2`s complement
	set less than unsigned	sltu $1,$2,$3	if ($2 < $3) $1 = 1; else $1 = 0	Compare less than; natural number
	set less than immediate unsigned	sltiu $1,$2,100	if ($2 < 100) $1 = 1; else $1 = 0	Compare constant; natural number
Unconditional jump	jump	j 10000	goto 10000	Jump to target address
	jump register	j $31	goto $31	For switch, procedure return
	jump and link	jal 10000	$31 = PC + 4;go to 10000	For procedure call

MIPS Floating-Point Operands

Name	Example	Comments
32 floating-point registers	$f0, $f1, $f2,..., $f31	MIPS floating point register are used in pairs for double precision numbers. Odd numbered registers cannot be used for arithemetic or branch, just for data transfer of the right "half" of double precision register pairs.
2^{30} memory words	Memory[0], Memory[4],..., Memory[4293967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls

MIPS Floating-Point Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	add.s $f2,$f4,$f6	$f2 = $f4 + $f6	Floating-Point add (single precision)
	FP subtract single	sub.s $f2,$f4,$f6	$f2 = $f4 - $f6	Floating-Point sub (single precision)
	FP multiply single	mul.s $f2,$f4,$f6	$f2 = $f4 * $f6	Floating-Point multiply (single precision)
	FP divide single	div.s $f2,$f4,$f6	$f2 = $f4 / $f6	Floating-Point divide (single precision)
	FP add double	add.d $f2,$f4,$f6	$f2 = $f4 + $f6	Floating-Point add (double precision)
	FP.dubtract double	.dub.d $f2,$f4,$f6	$f2 = $f4 - $f6	Floating-Point sub (double precision)
	FP multiply double	mul.d $f2,$f4,$f6	$f2 = $f4 * $f6	Floating-Point multiply (double precision)
	FP divide double	div.d $f2,$f4,$f6	$f2 = $f4 / $f6	Floating-Point divide (double precision)
Data	load word coprocessor 1	lwc1	$f1 =	32-bit data to FP register

transfer		$f1,100($2)	Memory[$2+100]	
	store word coprocessor 1	swc1 $f1,100($2)	Memory[$2+100] = $f1	32-bit data to memory
Arithmetic	branch on FP true	bc1t 100	if (cond == 1) go to PC+4+100	PC relative branch if FP condition
	branch on FP false	bc1f 100	if (cond == 0) go to PC+4+100	PC relative branch if not condition
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s $f2,$f4	if ($f2 < $f4) cond=1; else cond=0	Floating-point compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d $f2,$f4	if ($f2 < $f4) cond=1; else cond=0	Floating-point compare less than double precision

Summary of Addressing Modes in MIPS

Introduction

MIPS has only a small number of ways that is computes addresses in memory. The address can be an address of an instruction (for branch and jump instructions) or it can be an address of data (for load and store instructions).

We'll look at the four ways addresses are computed

- **Register Addressing** This is used in the **jr** (jump register) instruction.
- **PC-Relative Addressing** This is used in the **beq** and **bne** (branch equal, branch not equal) instructions.
- **Pseudo-direct Addressing** This is used in the **j** (jump) instruction.
- **Base Addressing** This is used in the **lw** and **sw** (load word, store word) instructions.

We'll also consider *indirect addressing*, a popular addressing mode in CISC ISAs.

Register Addressing

Register addressing is used in the **jr** instruction. Because a register stores 32 bits, and because an address in a MIPS CPU is also 32 bits, you can specify any address in memory.

The typical call is:

```
jr $rs
```

where **$rs** is replaced by any register.

The semantics of this is:

```
PC <- R[s]
```

This means the PC (program counter) is updated with the contents of register **s**. Recall that a jump or branch is updated by modifying the contents of the program counter.

Register addressing gives you the ability to generate any address in memory. An *address exception* occurs if the two low bits are not 00.

PC-Relative Addressing

PC-relative addressing occurs in branch instructions, **beq** and **bne** (and other variations of branch instructions).

These instructions are I-type instructions, with the following format.

Opcode	Register s	Register t	Immediate
B_{31-26}	B_{25-21}	B_{20-16}	B_{15-0}
ooo ooo	sssss	ttttt	iiii iiii iiii iiii

The immediate value is only 16 bits, which means we can't address a large range of memory. The immediate value is in 16 bits 2C, so the range of values is -2^{15} up to $2^{15} - 1$.

Branch instructions are used primarily to implement loops or if-else statements. When you jump to a statement in an if-else or loop, you typically jump to a nearby instruction from the one you jumped from. The instruction you jumped from must have been the one at PC, thus it makes sense to jump *relative* to the PC.

For example, if you had a branch instruction at address 1000, then it's likely you'll jump somewhere near the branch instruction.

A naive (and thus incorrect approach) is to do the following:

$$PC \leftarrow PC + \text{sign-ext}_{32}(IR_{15-0}) \text{ WRONG!!}$$

where **sign-ext$_{32}$(X)** means to sign extend the X to 32 bits.

Why is this naive? We know the following facts:

- MIPS instructions are 32 bits.
- 32 bit quantities in MIPS must be stored in memory at word-aligned addresses.
- Word aligned addresses end in 00 when written in binary.
- Thus, the PC, which stores addresses of instructions must hold a word-aligned address.

If we're going to branch to some address, and PC is already word aligned, then the immediate value has to be word-aligned as well.

However, it makes no sense to make the immediate word-aligned because we're wasting the low two bits by forcing it to be 00.

It makes more sense to allow the 16-bit immediate to be any of the 2^{16} possible immediate values. That way you make full use of the immediate value. It's similar to IEEE 754 floating point where we don't represent the hidden 1 in the fraction for normalized floating point numbers. Since it's always there, why waste a bit to represent it?

The real PC-relative addressing

Here's how the address is really computed.

$$PC \leftarrow PC + \text{sign-ext}_{32}(IR_{15-0}::00) \text{ CORRECT!!}$$

You take the 16 bit immediate value, add two zeroes to the end (which is the same as shifting it logical left 2 bits). This creates a value that's divisible by 4. Then, you sign-extend it to 32 bits, and add it to the PC.

Thus, the range of possible addresses is: **PC - 2^{17}** up to **PC + (2^{17} - 4)**.

2^{17} is 128 K. So you can jump back roughly -128,000 bytes backwards up to about 128,000 bytes forward. That's large, but still a small fraction of memory. Fortunately, for branch instructions, you don't need to jump that far, if you've written reasonably good code.

Pseudo-Direct Addressing

Direct addressing means specifying a complete 32 bit address in the instruction itself. However, since MIPS instructions are 32 bits, we can't do that. In theory, you only need 30 bits to specify the address of an instruction in memory. However, MIPS uses 6 bits for the opcode, so there's still not enough bits to do true direct addressing.

Instead, we can do pseudo-direct addressing. This occurs in **j** instructions.

Opcode	Target

B_{31-26}	B_{25-0}
ooo ooo	tt tttt tttt tttt tttt tttt tttt

26 bits are used for the target. This is how the address for pseudo-direct addressing is computed.

$$PC <- PC_{31-28}::IR_{25-0}::00$$

Take the top 4 bits of the PC, concatenate that with the 26 bits that make up the target, and concatenate that with 00. This produces a 32 bit address. This is the new address of the PC.

This allows you to jump to 1/16 of all possible MIPS addresses (since you don't control the top 4 bits of the PC).

Base Addressing

The other three addressing modes modify the PC. They create addresses for branch/jump instructions.

However, load/store instructions also generate addresses in memory.

Let's consider the following instruction.

```
lw $rt, offset($rs)
```

where **$rs** and **$rt** are any two registers.

The offset is stored in 16 bits 2C. Thus, **lw** and **sw** are I-type instructions.

The address computed is:

$$addr <- R[s] + sign\text{-}ext_{32}(offset)$$

$rs is the base register, which is where the name base addressing comes from.

The **offset** is the 16 bit immediate value from the instruction. Unlike branch instructions, we don't add a 00 to the end of the immediate value, even though we can only load and store are word-aligned addresses.

The reason is because there are other load/store instructions that load/store halfwords and bytes, and it makes sense to compute the addresses the same way, regardless of what you're loading.

So what happens an address is computed that's not word aligned? An address exception occurs.

Indirect Addressing

MIPS does not support indirect addressing. RISC ISAs generally do not support such an instruction. However, it's a popular addressing mode for CISC ISAs.

Let's see how this indirect addressing works. First, recall how **lw** works. It adds the contents of a register to a sign-extended offset. This results in an address. The word stored at that address is loaded into a register.

Let's make up a new instruction called **indlw** which means "indirect load word". This instruction doesn't really exist in MIPS, but pretend it does.

```
indlw $rt, offset($rs)
```

In this case, we'll do the same thing. Add the sign-extended offset to **$rs**. But instead of loading the word at that address into a register, we load the word (say, to some temporary location like $at), and use that word as

an address, then we load that word from memory.

Thus, we go to memory twice. First time, we load a word that represents an address, and second time, we use that address to load a word of data.

The semantics are:

$$R[\ t\ ] \leftarrow M_4[\ M_4[\ R[\ s\ ] + \text{sign-ext}_{32}(\ \text{offset}\ )\ ]\ ]$$

Indirect addressing might seem odd, but if you increment the address stored in memory, it's one way to process an array. That address could be a pointer.

The major reason indirect addressing isn't used in RISC ISAs is because it accesses memory twice. The goal is to make each instruction fast, and accessing memory is considered slow. As it is, even load and store word don't really fully execute in one clock cycle (it takes about two, and that assumes it's in cache).

Still, this kind of addressing mode and far more are seen in CISC ISAs. The goal in those ISAs is to provide a person who programs in assembly language a lot of choices of instructions. CISC ISAs are meant to minimize the amount of memory used for a program by providing a large number of instructions.

The goal of RISC ISAs is minimize instructions to improve performance by making hardware design simpler (fewer instructions means simpler hardware, which means greater opportunities for optimization).

Nevertheless, money and brainpower can compesate for this seemingly good idea. Intel makes CISC processors that are now currently faster than RISC processors. Part of this is because Intel chips do some conversion to RISC-like instructions behind the scenes. Thus, you code in x86 (or, more properly, IA32) and the hardware converts it to RISC like internal instructions which you don't see.

Summary

All assembly languages give you a variety of way to generate addresses in memory. This can be the address of an instruction or the address of data.

We considered the four major ways to compute addresses in MIPS.

- register addressing
- PC-relative addressing
- pseudo-direct addressing
- base addressing

These four ways are much smaller than CISC ISAs, which may provides ten or more addressing modes. We discussed one of the more popular CISC addressing mode: indirect addressing.

C

Pipelining: Basic and Intermediate Concepts

It is quite a three-pipe problem.

Sir Arthur Conan Doyle
The Adventures of Sherlock Holmes

C.1 Introduction

Many readers of this text will have covered the basics of pipelining in another text (such as our more basic text *Computer Organization and Design*) or in another course. Because Chapter 3 builds heavily on this material, readers should ensure that they are familiar with the concepts discussed in this appendix before proceeding. As you read Chapter 2, you may find it helpful to turn to this material for a quick review.

We begin the appendix with the basics of pipelining, including discussing the data path implications, introducing hazards, and examining the performance of pipelines. This section describes the basic five-stage RISC pipeline that is the basis for the rest of the appendix. Section C.2 describes the issue of hazards, why they cause performance problems, and how they can be dealt with. Section C.3 discusses how the simple five-stage pipeline is actually implemented, focusing on control and how hazards are dealt with.

Section C.4 discusses the interaction between pipelining and various aspects of instruction set design, including discussing the important topic of exceptions and their interaction with pipelining. Readers unfamiliar with the concepts of precise and imprecise interrupts and resumption after exceptions will find this material useful, since they are key to understanding the more advanced approaches in Chapter 3.

Section C.5 discusses how the five-stage pipeline can be extended to handle longer-running floating-point instructions. Section C.6 puts these concepts together in a case study of a deeply pipelined processor, the MIPS R4000/4400, including both the eight-stage integer pipeline and the floating-point pipeline.

Section C.7 introduces the concept of dynamic scheduling and the use of scoreboards to implement dynamic scheduling. It is introduced as a crosscutting issue, since it can be used to serve as an introduction to the core concepts in Chapter 3, which focused on dynamically scheduled approaches. Section C.7 is also a gentle introduction to the more complex Tomasulo's algorithm covered in Chapter 3. Although Tomasulo's algorithm can be covered and understood without introducing scoreboarding, the scoreboarding approach is simpler and easier to comprehend.

What Is Pipelining?

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. Today, pipelining is the key implementation technique used to make fast CPUs.

A pipeline is like an assembly line. In an automobile assembly line, there are many steps, each contributing something to the construction of the car. Each step operates in parallel with the other steps, although on a different car. In a computer pipeline, each step in the pipeline completes a part of an instruction. Like the

assembly line, different steps are completing different parts of different instructions in parallel. Each of these steps is called a *pipe stage* or a *pipe segment*. The stages are connected one to the next to form a pipe—instructions enter at one end, progress through the stages, and exit at the other end, just as cars would in an assembly line.

In an automobile assembly line, *throughput* is defined as the number of cars per hour and is determined by how often a completed car exits the assembly line. Likewise, the throughput of an instruction pipeline is determined by how often an instruction exits the pipeline. Because the pipe stages are hooked together, all the stages must be ready to proceed at the same time, just as we would require in an assembly line. The time required between moving an instruction one step down the pipeline is a *processor cycle*. Because all stages proceed at the same time, the length of a processor cycle is determined by the time required for the slowest pipe stage, just as in an auto assembly line the longest step would determine the time between advancing the line. In a computer, this processor cycle is usually 1 clock cycle (sometimes it is 2, rarely more).

The pipeline designer's goal is to balance the length of each pipeline stage, just as the designer of the assembly line tries to balance the time for each step in the process. If the stages are perfectly balanced, then the time per instruction on the pipelined processor—assuming ideal conditions—is equal to

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

Under these conditions, the speedup from pipelining equals the number of pipe stages, just as an assembly line with n stages can ideally produce cars n times as fast. Usually, however, the stages will not be perfectly balanced; furthermore, pipelining does involve some overhead. Thus, the time per instruction on the pipelined processor will not have its minimum possible value, yet it can be close.

Pipelining yields a reduction in the average execution time per instruction. Depending on what you consider as the baseline, the reduction can be viewed as decreasing the number of clock cycles per instruction (CPI), as decreasing the clock cycle time, or as a combination. If the starting point is a processor that takes multiple clock cycles per instruction, then pipelining is usually viewed as reducing the CPI. This is the primary view we will take. If the starting point is a processor that takes 1 (long) clock cycle per instruction, then pipelining decreases the clock cycle time.

Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream. It has the substantial advantage that, unlike some speedup techniques (see Chapter 4), it is not visible to the programmer. In this appendix we will first cover the concept of pipelining using a classic five-stage pipeline; other chapters investigate the more sophisticated pipelining techniques in use in modern processors. Before we say more about pipelining and its use in a processor, we need a simple instruction set, which we introduce next.

The Basics of a RISC Instruction Set

Throughout this book we use a RISC (reduced instruction set computer) architecture or load-store architecture to illustrate the basic concepts, although nearly all the ideas we introduce in this book are applicable to other processors. In this section we introduce the core of a typical RISC architecture. In this appendix, and throughout the book, our default RISC architecture is MIPS. In many places, the concepts are significantly similar that they will apply to any RISC. RISC architectures are characterized by a few key properties, which dramatically simplify their implementation:

- All operations on data apply to data in registers and typically change the entire register (32 or 64 bits per register).
- The only operations that affect memory are load and store operations that move data from memory to a register or to memory from a register, respectively. Load and store operations that load or store less than a full register (e.g., a byte, 16 bits, or 32 bits) are often available.
- The instruction formats are few in number, with all instructions typically being one size.

These simple properties lead to dramatic simplifications in the implementation of pipelining, which is why these instruction sets were designed this way.

For consistency with the rest of the text, we use MIPS64, the 64-bit version of the MIPS instruction set. The extended 64-bit instructions are generally designated by having a `D` on the start or end of the mnemonic. For example `DADD` is the 64-bit version of an add instruction, while `LD` is the 64-bit version of a load instruction.

Like other RISC architectures, the MIPS instruction set provides 32 registers, although register 0 always has the value 0. Most RISC architectures, like MIPS, have three classes of instructions (see Appendix A for more detail):

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. For this appendix, we consider only comparisons for equality between two registers. In all RISC architectures, the branch destination is obtained by adding a sign-extended offset (16 bits in MIPS) to the current PC. Unconditional jumps are provided in many RISC architectures, but we will not cover jumps in this appendix.

A Simple Implementation of a RISC Instruction Set

To understand how a RISC instruction set can be implemented in a pipelined fashion, we need to understand how it is implemented *without* pipelining. This section shows a simple implementation where every instruction takes at most 5 clock cycles. We will extend this basic implementation to a pipelined version, resulting in a much lower CPI. Our unpipelined implementation is not the most economical or the highest-performance implementation without pipelining. Instead, it is designed to lead naturally to a pipelined implementation. Implementing the instruction set requires the introduction of several temporary registers that are not part of the architecture; these are introduced in this section to simplify pipelining. Our implementation will focus only on a pipeline for an integer subset of a RISC architecture that consists of load-store word, branch, and integer ALU operations.

Every instruction in this RISC subset can be implemented in at most 5 clock cycles. The 5 clock cycles are as follows.

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.

   Decoding is done in parallel with reading registers, which is possible because the register specifiers are at a fixed location in a RISC architecture.

This technique is known as *fixed-field decoding*. Note that we may read a register we don't use, which doesn't help but also doesn't hurt performance. (It does waste energy to read an unneeded register, and power-sensitive designs might avoid this.) Because the immediate portion of an instruction is also located in an identical place, the sign-extended immediate is also calculated during this cycle in case it is needed.

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).

In this implementation, branch instructions require 2 cycles, store instructions require 4 cycles, and all other instructions require 5 cycles. Assuming a branch frequency of 12% and a store frequency of 10%, a typical instruction distribution leads to an overall CPI of 4.54. This implementation, however, is not optimal either in achieving the best performance or in using the minimal amount of hardware given the performance level; we leave the improvement of this design as an exercise for you and instead focus on pipelining this version.

The Classic Five-Stage Pipeline for a RISC Processor

We can pipeline the execution described above with almost no changes by simply starting a new instruction on each clock cycle. (See why we chose this design?)

	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

Figure C.1 Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its five-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unpipelined implementation: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write-back.

Each of the clock cycles from the previous section becomes a *pipe stage*—a cycle in the pipeline. This results in the execution pattern shown in Figure C.1, which is the typical way a pipeline structure is drawn. Although each instruction takes 5 clock cycles to complete, during each clock cycle the hardware will initiate a new instruction and will be executing some part of the five different instructions.

You may find it hard to believe that pipelining is as simple as this; it's not. In this and the following sections, we will make our RISC pipeline "real" by dealing with problems that pipelining introduces.

To start with, we have to determine what happens on every clock cycle of the processor and make sure we don't try to perform two different operations with the same data path resource on the same clock cycle. For example, a single ALU cannot be asked to compute an effective address and perform a subtract operation at the same time. Thus, we must ensure that the overlap of instructions in the pipeline cannot cause such a conflict. Fortunately, the simplicity of a RISC instruction set makes resource evaluation relatively easy. Figure C.2 shows a simplified version of a RISC data path drawn in pipeline fashion. As you can see, the major functional units are used in different cycles, and hence overlapping the execution of multiple instructions introduces relatively few conflicts. There are three observations on which this fact rests.

First, we use separate instruction and data memories, which we would typically implement with separate instruction and data caches (discussed in Chapter 2). The use of separate caches eliminates a conflict for a single memory that would arise between instruction fetch and data memory access. Notice that if our pipelined processor has a clock cycle that is equal to that of the unpipelined version, the memory system must deliver five times the bandwidth. This increased demand is one cost of higher performance.

Second, the register file is used in the two stages: one for reading in ID and one for writing in WB. These uses are distinct, so we simply show the register file in two places. Hence, we need to perform two reads and one write every

clock cycle. To handle reads and a write to the same register (and for another reason, which will become obvious shortly), we perform the register write in the first half of the clock cycle and the read in the second half.

Third, Figure C.2 does not deal with the PC. To start a new instruction every clock, we must increment and store the PC every clock, and this must be done during the IF stage in preparation for the next instruction. Furthermore, we must also have an adder to compute the potential branch target during ID. One further problem is that a branch does not change the PC until the ID stage. This causes a problem, which we ignore for now, but will handle shortly.

Although it is critical to ensure that instructions in the pipeline do not attempt to use the hardware resources at the same time, we must also ensure that instructions in different stages of the pipeline do not interfere with one another. This separation is done by introducing *pipeline registers* between successive stages of the pipeline, so that at the end of a clock cycle all the results from a given stage are stored into a register that is used as the input to the next stage on the next clock cycle. Figure C.3 shows the pipeline drawn with these pipeline registers.

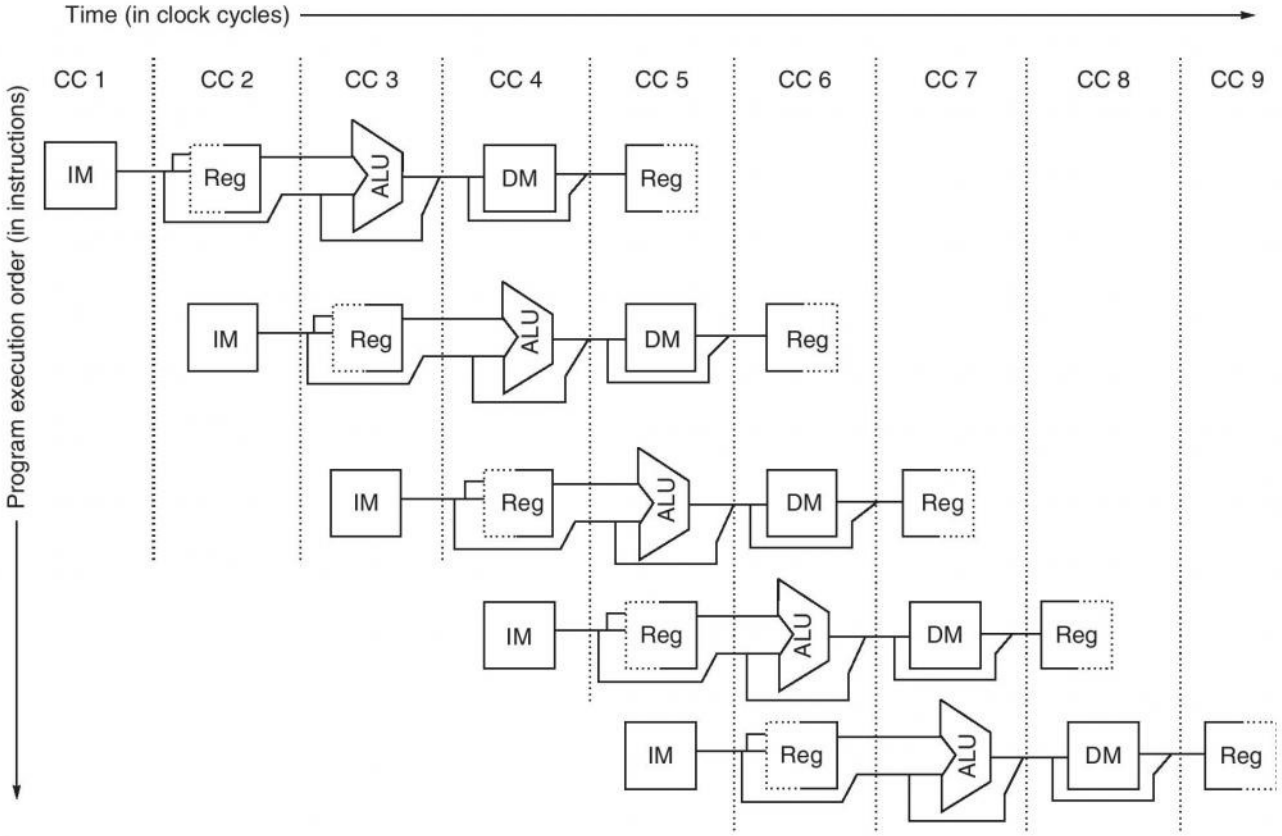


Figure C.2 The pipeline can be thought of as a series of data paths shifted in time. This shows the overlap among the parts of the data path, with clock cycle 5 (CC 5) showing the steady-state situation. Because the register file is used as a source in the ID stage and as a destination in the WB stage, it appears twice. We show that it is read in one part of the stage and written in another by using a solid line, on the right or left, respectively, and a dashed line on the other side. The abbreviation IM is used for instruction memory, DM for data memory, and CC for clock cycle.

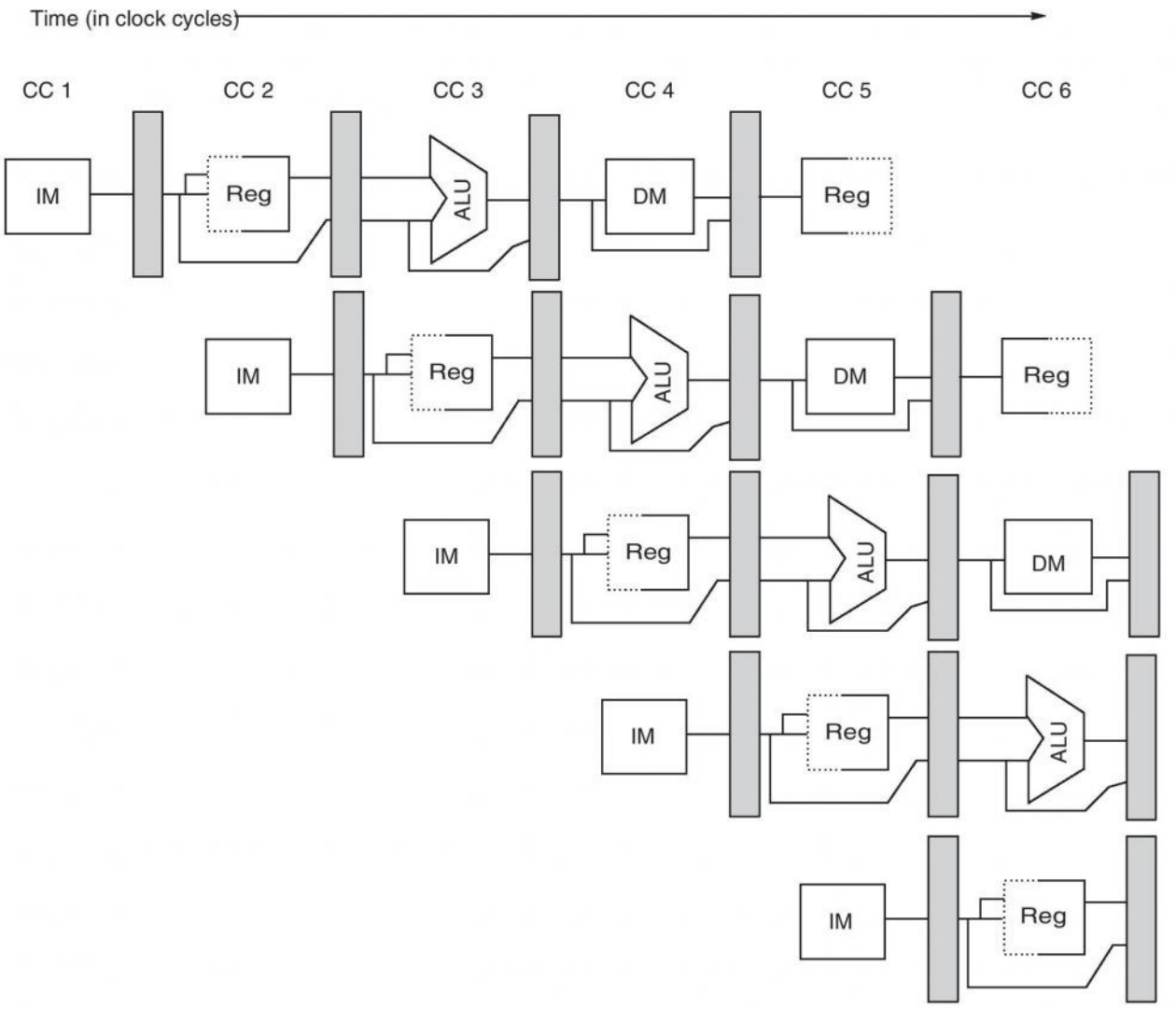


Figure C.3 A pipeline showing the pipeline registers between successive pipeline stages. Notice that the registers prevent interference between two different instructions in adjacent stages in the pipeline. The registers also play the critical role of carrying data for a given instruction from one stage to the other. The edge-triggered property of registers—that is, that the values change instantaneously on a clock edge—is critical. Otherwise, the data from one instruction could interfere with the execution of another!

Although many figures will omit such registers for simplicity, they are required to make the pipeline operate properly and must be present. Of course, similar registers would be needed even in a multicycle data path that had no pipelining (since only values in registers are preserved across clock boundaries). In the case of a pipelined processor, the pipeline registers also play the key role of carrying intermediate results from one stage to another where the source and destination may not be directly adjacent. For example, the register value to be stored

during a store instruction is read during ID, but not actually used until MEM; it is passed through two pipeline registers to reach the data memory during the MEM stage. Likewise, the result of an ALU instruction is computed during EX, but not actually stored until WB; it arrives there by passing through two pipeline registers. It is sometimes useful to name the pipeline registers, and we follow the convention of naming them by the pipeline stages they connect, so that the registers are called IF/ID, ID/EX, EX/MEM, and MEM/WB.

Basic Performance Issues in Pipelining

Pipelining increases the CPU instruction throughput—the number of instructions completed per unit of time—but it does not reduce the execution time of an individual instruction. In fact, it usually slightly increases the execution time of each instruction due to overhead in the control of the pipeline. The increase in instruction throughput means that a program runs faster and has lower total execution time, even though no single instruction runs faster!

The fact that the execution time of each instruction does not decrease puts limits on the practical depth of a pipeline, as we will see in the next section. In addition to limitations arising from pipeline latency, limits arise from imbalance among the pipe stages and from pipelining overhead. Imbalance among the pipe stages reduces performance since the clock can run no faster than the time needed for the slowest pipeline stage. Pipeline overhead arises from the combination of pipeline register delay and clock skew. The pipeline registers add setup time, which is the time that a register input must be stable before the clock signal that triggers a write occurs, plus propagation delay to the clock cycle. Clock skew, which is maximum delay between when the clock arrives at any two registers, also contributes to the lower limit on the clock cycle. Once the clock cycle is as small as the sum of the clock skew and latch overhead, no further pipelining is useful, since there is no time left in the cycle for useful work. The interested reader should see Kunkel and Smith [1986]. As we saw in Chapter 3, this overhead affected the performance gains achieved by the Pentium 4 versus the Pentium III.

Example 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?

Answer 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

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

$$= \frac{4.4 \text{ ns}}{1.2 \text{ ns}} = 3.7 \text{ times}$$

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.

This simple RISC pipeline would function just fine for integer instructions if every instruction were independent of every other instruction in the pipeline. In reality, instructions in the pipeline can depend on one another; this is the topic of the next section.

C.2 The Major Hurdle of Pipelining—Pipeline Hazards

There are situations, called *hazards*, that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards:

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped 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.

Hazards in pipelines can make it necessary to *stall* the pipeline. Avoiding a hazard often requires that some instructions in the pipeline be allowed to proceed while others are delayed. For the pipelines we discuss in this appendix, when an instruction is stalled, all instructions issued *later* than the stalled instruction—and hence not as far along in the pipeline—are also stalled. Instructions issued *earlier* than the stalled instruction—and hence farther along in the pipeline—must continue, since otherwise the hazard will never clear. As a result, no new instructions are fetched during the stall. We will see several examples of how pipeline stalls operate in this section—don't worry, they aren't as complex as they might sound!

Performance of Pipelines with Stalls

A stall causes the pipeline performance to degrade from the ideal performance. Let's look at a simple equation for finding the actual speedup from pipelining, starting with the formula from the previous section:

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}\\ &= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}\\ &= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}\end{aligned}$$

Pipelining can be thought of as decreasing the CPI or the clock cycle time. Since it is traditional to use the CPI to compare pipelines, let's start with that assumption. The ideal CPI on a pipelined processor is almost always 1. Hence, we can compute the pipelined CPI:

$$\begin{aligned}\text{CPI pipelined} &= \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}\\ &= 1 + \text{Pipeline stall clock cycles per instruction}\end{aligned}$$

If we ignore the cycle time overhead of pipelining and assume that the stages are perfectly balanced, then the cycle time of the two processors can be equal, leading to

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

One important simple case is where all instructions take the same number of cycles, which must also equal the number of pipeline stages (also called the *depth of the pipeline*). In this case, the unpipelined CPI is equal to the depth of the pipeline, leading to

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

If there are no pipeline stalls, this leads to the intuitive result that pipelining can improve performance by the depth of the pipeline.

Alternatively, if we think of pipelining as improving the clock cycle time, then we can assume that the CPI of the unpipelined processor, as well as that of the pipelined processor, is 1. This leads to

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}\\ &= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}\end{aligned}$$

In cases where the pipe stages are perfectly balanced and there is no overhead, the clock cycle on the pipelined processor is smaller than the clock cycle of the unpipelined processor by a factor equal to the pipelined depth:

$$\text{Clock cycle pipelined} = \frac{\text{Clock cycle unpipelined}}{\text{Pipeline depth}}$$

$$\text{Pipeline depth} = \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

This leads to the following:

$$\text{Speedup from pipelining} = \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \text{Pipeline depth}$$

Thus, if there are no stalls, the speedup is equal to the number of pipeline stages, matching our intuition for the ideal case.

Structural Hazards

When a processor is pipelined, 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.*

The most common instances of structural hazards arise when some functional unit is not fully pipelined. Then a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle. Another common way that structural hazards appear is when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute. For example, a processor may have only one register-file write port, but under certain circumstances, the pipeline might want to perform two writes in a clock cycle. This will generate a structural hazard.

When a sequence of instructions encounters this hazard, the pipeline will stall one of the instructions until the required unit is available. Such stalls will increase the CPI from its usual ideal value of 1.

Some pipelined processors have shared a single-memory pipeline for data and instructions. As a result, when an instruction contains a data memory reference, it will conflict with the instruction reference for a later instruction, as shown in Figure C.4. To resolve this hazard, we stall the pipeline for 1 clock cycle when the data memory access occurs. A stall is commonly called a *pipeline bubble* or just *bubble*, since it floats through the pipeline taking space but carrying no useful work. We will see another type of stall when we talk about data hazards.

Designers often indicate stall behavior using a simple diagram with only the pipe stage names, as in Figure C.5. The form of Figure C.5 shows the stall by indicating the cycle when no action occurs and simply shifting instruction 3 to

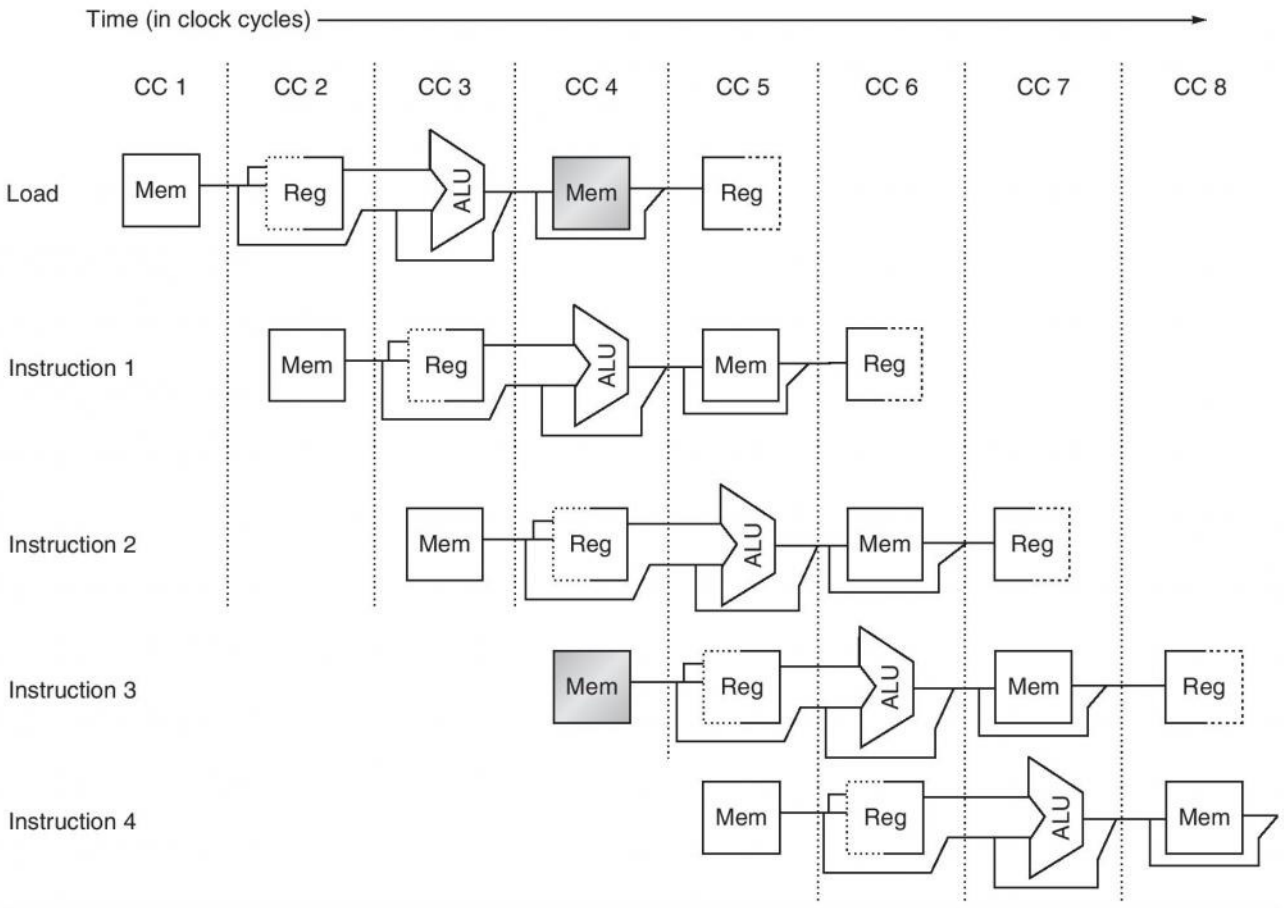


Figure C.4 A processor with only one memory port will generate a conflict whenever a memory reference occurs. In this example the load instruction uses the memory for a data access at the same time instruction 3 wants to fetch an instruction from memory.

the right (which delays its execution start and finish by 1 cycle). The effect of the pipeline bubble is actually to occupy the resources for that instruction slot as it travels through the pipeline.

Example Let's see how much the load structural hazard might cost. Suppose that data references constitute 40% of the mix, and that the ideal CPI of the pipelined processor, ignoring the structural hazard, is 1. Assume that the processor with the structural hazard has a clock rate that is 1.05 times higher than the clock rate of the processor without the hazard. Disregarding any other performance losses, is the pipeline with or without the structural hazard faster, and by how much?

Answer There are several ways we could solve this problem. Perhaps the simplest is to compute the average instruction time on the two processors:

$$\text{Average instruction time} = \text{CPI} \times \text{Clock cycle time}$$

	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 C.5 A pipeline stalled for a structural hazard—a load with one memory port. As shown here, the load instruction effectively steals an instruction-fetch cycle, causing the pipeline to stall—no instruction is initiated on clock cycle 4 (which normally would initiate instruction $i + 3$). Because the instruction being fetched is stalled, all other instructions in the pipeline before the stalled instruction can proceed normally. The stall cycle will continue to pass through the pipeline, so that no instruction completes on clock cycle 8. Sometimes these pipeline diagrams are drawn with the stall occupying an entire horizontal row and instruction 3 being moved to the next row; in either case, the effect is the same, since instruction $i + 3$ does not begin execution until cycle 5. We use the form above, since it takes less space in the figure. Note that this figure assumes that instructions $i + 1$ and $i + 2$ are not memory references.

Since it has no stalls, the average instruction time for the ideal processor is simply the Clock cycle $\text{time}_{\text{ideal}}$. The average instruction time for the processor with the structural hazard is

$$\begin{aligned}\text{Average instruction time} &= \text{CPI} \times \text{Clock cycle time} \\ &= (1 + 0.4 \times 1) \times \frac{\text{Clock cycle time}_{\text{ideal}}}{1.05} \\ &= 1.3 \times \text{Clock cycle time}_{\text{ideal}}\end{aligned}$$

Clearly, the processor without the structural hazard is faster; we can use the ratio of the average instruction times to conclude that the processor without the hazard is 1.3 times faster.

As an alternative to this structural hazard, the designer could provide a separate memory access for instructions, either by splitting the cache into separate instruction and data caches or by using a set of buffers, usually called *instruction buffers*, to hold instructions. Chapter 5 discusses both the split cache and instruction buffer ideas.

If all other factors are equal, a processor without structural hazards will always have a lower CPI. Why, then, would a designer allow structural hazards? The primary reason is to reduce cost of the unit, since pipelining all the functional units, or duplicating them, may be too costly. For example, processors that support both an instruction and a data cache access every cycle (to prevent the structural hazard of the above example) require twice as much total memory

bandwidth and often have higher bandwidth at the pins. Likewise, fully pipelining a floating-point (FP) multiplier consumes lots of gates. If the structural hazard is rare, it may not be worth the cost to avoid it.

Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	C.2
Delayed branches and simple branch scheduling	Control hazard stalls	C.2
Basic compiler pipeline scheduling	Data hazard stalls	C.2, 3.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	C.7
Loop unrolling	Control hazard stalls	3.2
Branch prediction	Control stalls	3.3
Dynamic scheduling with renaming	Stalls from data hazards, output dependences, and antidependences	3.4
Hardware speculation	Data hazard and control hazard stalls	3.6
Dynamic memory disambiguation	Data hazard stalls with memory	3.6
Issuing multiple instructions per cycle	Ideal CPI	3.7, 3.8
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls	H.2, H.3
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls	H.4, H.5

Figure 3.1 The major techniques examined in Appendix C, Chapter 3, and Appendix H are shown together with the component of the CPI equation that the technique affects.

The *ideal pipeline CPI* is a measure of the maximum performance attainable by the implementation. By reducing each of the terms of the right-hand side, we decrease the overall pipeline CPI or, alternatively, increase the IPC (instructions per clock). The equation above allows us to characterize various techniques by what component of the overall CPI a technique reduces. Figure 3.1 shows the techniques we examine in this chapter and in Appendix H, as well as the topics covered in the introductory material in Appendix C. In this chapter, we will see that the techniques we introduce to decrease the ideal pipeline CPI can increase the importance of dealing with hazards.

What Is Instruction-Level Parallelism?

All the techniques in this chapter exploit parallelism among instructions. The amount of parallelism available within a *basic block*—a straight-line code sequence with no branches in except to the entry and no branches out except at the exit—is quite small. For typical MIPS programs, the average dynamic branch frequency is often between 15% and 25%, meaning that between three and six instructions execute between a pair of branches. Since these instructions are likely to depend upon one another, the amount of overlap we can exploit within a basic block is likely to be less than the average basic block size. To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks.

The simplest and most common way to increase the ILP is to exploit parallelism among iterations of a loop. This type of parallelism is often called *loop-level parallelism*. Here is a simple example of a loop that adds two 1000-element arrays and is completely parallel:

```
for (i=0; i<=999; i=i+1)
        x[i] = x[i] + y[i];
```

Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little or no opportunity for overlap.

We will examine a number of techniques for converting such loop-level parallelism into instruction-level parallelism. Basically, such techniques work by unrolling the loop either statically by the compiler (as in the next section) or dynamically by the hardware (as in Sections 3.5 and 3.6).

An important alternative method for exploiting loop-level parallelism is the use of SIMD in both vector processors and Graphics Processing Units (GPUs), both of which are covered in Chapter 4. A SIMD instruction exploits data-level parallelism by operating on a small to moderate number of data items in parallel (typically two to eight). A vector instruction exploits data-level parallelism by operating on many data items in parallel using both parallel execution units and a deep pipeline. For example, the above code sequence, which in simple form requires seven instructions per iteration (two loads, an add, a store, two address updates, and a branch) for a total of 7000 instructions, might execute in one-quarter as many instructions in some SIMD architecture where four data items are processed per instruction. On some vector processors, this sequence might take only four instructions: two instructions to load the vectors `x` and `y` from memory, one instruction to add the two vectors, and an instruction to store back the result vector. Of course, these instructions would be pipelined and have relatively long latencies, but these latencies may be overlapped.

Data Dependences and Hazards

Determining how one instruction depends on another is critical to determining how much parallelism exists in a program and how that parallelism can be exploited. In particular, to exploit instruction-level parallelism we must determine which instructions can be executed in parallel. If two instructions are *parallel*, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls, assuming the pipeline has sufficient resources (and hence no structural hazards exist). If two instructions are dependent, they are not parallel and must be executed in order, although they may often be partially overlapped. The key in both cases is to determine whether an instruction is dependent on another instruction.

Data Dependences

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:

- Instruction *i* produces a result that may be used by instruction *j*.
- Instruction *j* is data dependent on instruction *k*, and instruction *k* is data dependent on instruction *i*.

Data Hazards

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. Because of the dependence, we must preserve what is called *program order*—that is, the order that the instructions would execute in if executed sequentially one at a time as determined by the original source program. The goal of both our software and hardware techniques is to exploit parallelism by preserving program order *only where it affects the outcome of the program.* Detecting and avoiding hazards ensures that necessary program order is preserved.

Data hazards, which are informally described in Appendix C, may be classified as one of three types, depending on the order of read and write accesses in the instructions. By convention, the hazards are named by the ordering in the program that must be preserved by the pipeline. Consider two instructions i and j, with i preceding j in program order. The possible data hazards are

- 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—because all reads are early (in ID in the pipeline in Appendix C) and all writes are late (in WB in the pipeline in Appendix C). A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline *and* other instructions that read a source late in the pipeline, or when instructions are reordered, as we will see in this chapter.

Note that the RAR *(read after read)* case is not a hazard.

Data Hazards

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This overlap introduces data and control hazards. Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor. 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
```

All the instructions after the DADD use the result of the DADD instruction. As shown in Figure C.6, the DADD instruction writes the value of R1 in the WB pipe stage, but the DSUB instruction reads the value during its ID stage. This problem is called a *data hazard*. Unless precautions are taken to prevent it, the DSUB instruction will read the wrong value and try to use it. In fact, the value used by the DSUB instruction is not even deterministic: Though we might think it logical to assume that DSUB would always use the value of R1 that was assigned by an instruction prior to DADD, this is not always the case. If an interrupt should occur between the DADD and DSUB instructions, the WB stage of the DADD will complete, and the value of R1 at that point will be the result of the DADD. This unpredictable behavior is obviously unacceptable.

The AND instruction is also affected by this hazard. As we can see from Figure C.6, the write of R1 does not complete until the end of clock cycle 5. Thus, the AND instruction that reads the registers during clock cycle 4 will receive the wrong results.

The XOR instruction operates properly because its register read occurs in clock cycle 6, after the register write. The OR instruction also operates without incurring a hazard because we perform the register file reads in the second half of the cycle and the writes in the first half.

The next subsection discusses a technique to eliminate the stalls for the hazard involving the DSUB and AND instructions.

Minimizing Data Hazard Stalls by Forwarding

The problem posed in Figure C.6 can be solved with a simple hardware technique called *forwarding* (also called *bypassing* and sometimes *short-circuiting*). The key insight in forwarding is that the result is not really needed by the DSUB until

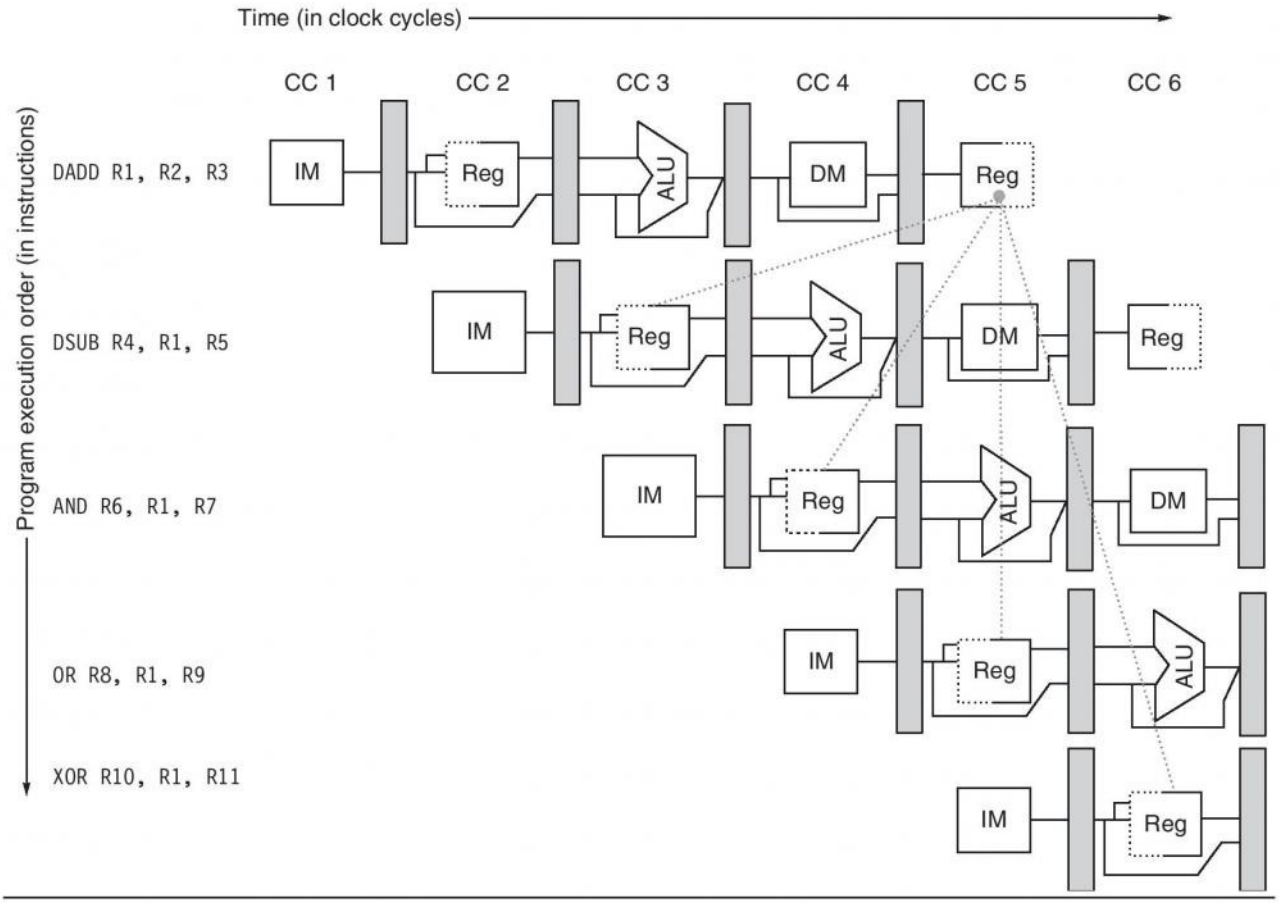


Figure C.6 The use of the result of the DADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

after the DADD actually produces it. If the result can be moved from the pipeline register where the DADD stores it to where the DSUB needs it, then the need for a stall can be avoided. Using this observation, forwarding works as follows:

1. The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs.
2. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.

Notice that with forwarding, if the DSUB is stalled, the DADD will be completed and the bypass will not be activated. This relationship is also true for the case of an interrupt between the two instructions.

As the example in Figure C.6 shows, we need to forward results not only from the immediately previous instruction but also possibly from an instruction

that started 2 cycles earlier. Figure C.7 shows our example with the bypass paths in place and highlighting the timing of the register read and writes. This code sequence can be executed without stalls.

Forwarding can be generalized to include passing a result directly to the functional unit that requires it: A result is forwarded from the pipeline register corresponding to the output of one unit to the input of another, rather than just from the result of a unit to the input of the same unit. Take, for example, the following sequence:

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

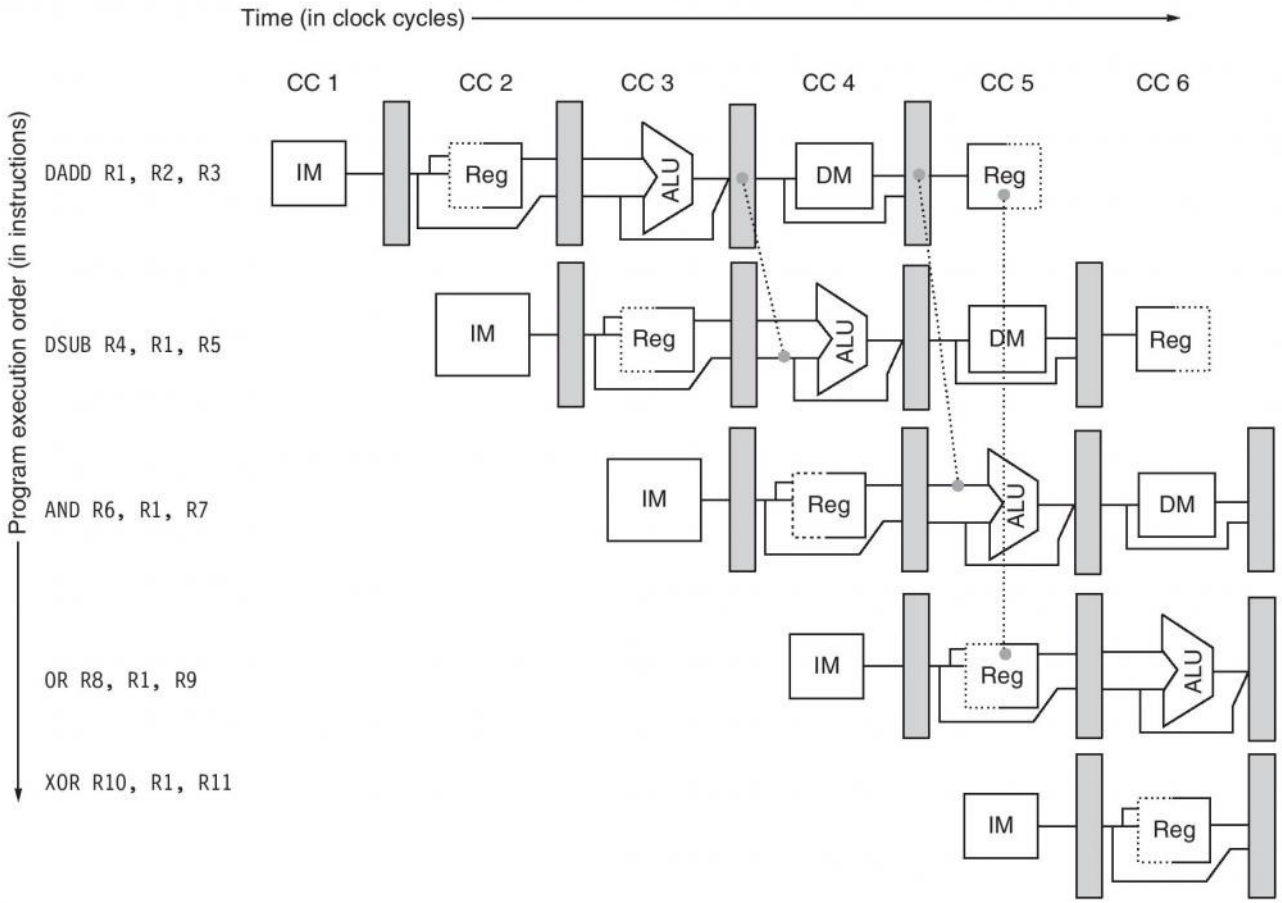


Figure C.7 A set of instructions that depends on the DADD result uses forwarding paths to avoid the data hazard. The inputs for the DSUB and AND instructions forward from the pipeline registers to the first ALU input. The OR receives its result by forwarding through the register file, which is easily accomplished by reading the registers in the second half of the cycle and writing in the first half, as the dashed lines on the registers indicate. Notice that the forwarded result can go to either ALU input; in fact, both ALU inputs could use forwarded inputs from either the same pipeline register or from different pipeline registers. This would occur, for example, if the AND instruction was AND R6,R1,R4.

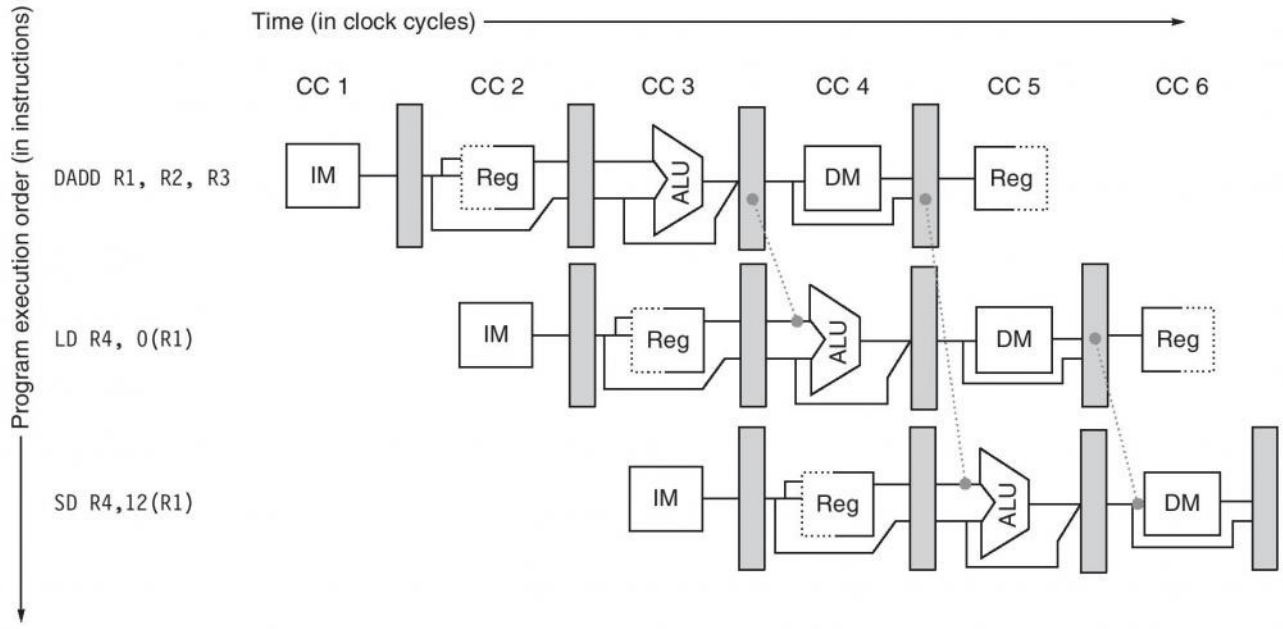


Figure C.8 Forwarding of operand required by stores during MEM. The result of the load is forwarded from the memory output to the memory input to be stored. In addition, the ALU output is forwarded to the ALU input for the address calculation of both the load and the store (this is no different than forwarding to another ALU operation). If the store depended on an immediately preceding ALU operation (not shown above), the result would need to be forwarded to prevent a stall.

To prevent a stall in this sequence, we would need to forward the values of the ALU output and memory unit output from the pipeline registers to the ALU and data memory inputs. Figure C.8 shows all the forwarding paths for this example.

Data Hazards Requiring Stalls

Unfortunately, not all potential data hazards can be handled by bypassing. Consider the following sequence of instructions:

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

The pipelined data path with the bypass paths for this example is shown in Figure C.9. This case is different from the situation with back-to-back ALU operations. The LD instruction does not have the data until the end of clock cycle 4 (its MEM cycle), while the DSUB instruction needs to have the data by the beginning of that clock cycle. Thus, the data hazard from using the result of a load instruction cannot be completely eliminated with simple hardware. As Figure C.9 shows, such a forwarding path would have to operate backward

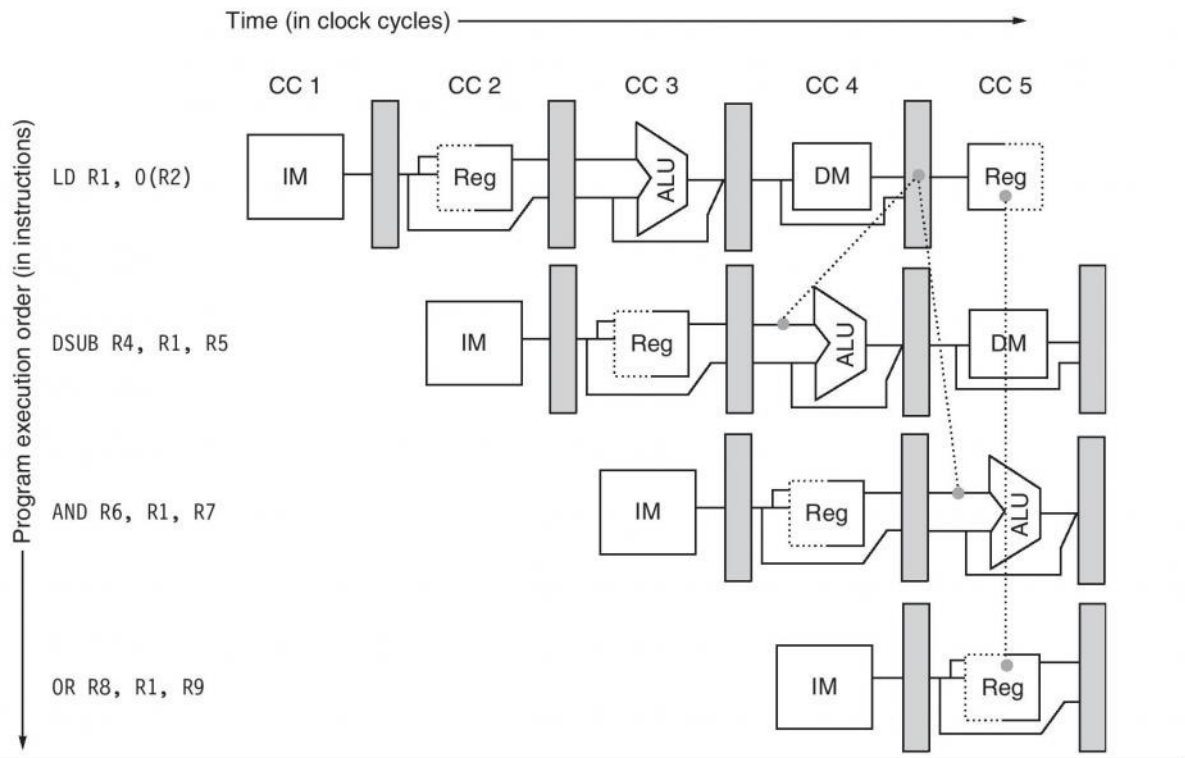


Figure C.9 The load instruction can bypass its results to the AND and OR instructions, but not to the DSUB, since that would mean forwarding the result in "negative time."

in time—a capability not yet available to computer designers! We *can* forward the result immediately to the ALU from the pipeline registers for use in the AND operation, which begins 2 clock cycles after the load. Likewise, the OR instruction has no problem, since it receives the value through the register file. For the DSUB instruction, the forwarded result arrives too late—at the end of a clock cycle, when it is needed at the beginning.

The load instruction has a delay or latency that cannot be eliminated by forwarding alone. Instead, we need to add hardware, called a *pipeline interlock*, to preserve the correct execution pattern. In general, a pipeline interlock detects a hazard and stalls the pipeline until the hazard is cleared. In this case, the interlock stalls the pipeline, beginning with the instruction that wants to use the data until the source instruction produces it. This pipeline interlock introduces a stall or bubble, just as it did for the structural hazard. The CPI for the stalled instruction increases by the length of the stall (1 clock cycle in this case).

Figure C.10 shows the pipeline before and after the stall using the names of the pipeline stages. Because the stall causes the instructions starting with the DSUB to move 1 cycle later in time, the forwarding to the AND instruction now goes through the register file, and no forwarding at all is needed for the OR instruction. The insertion of the bubble causes the number of cycles to complete this sequence to grow by one. No instruction is started during clock cycle 4 (and none finishes during cycle 6).

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

Figure C.10 In the top half, we can see why a stall is needed: The MEM cycle of the load produces a value that is needed in the EX cycle of the DSUB, which occurs at the same time. This problem is solved by inserting a stall, as shown in the bottom half.

Branch Hazards

Control hazards can cause a greater performance loss for our MIPS pipeline than do data hazards. When a branch is executed, it may or may not change the PC to something other than its current value plus 4. Recall that if a branch changes the PC to its target address, it is a *taken* branch; if it falls through, it is *not taken*, or *untaken*. If instruction *i* is a taken branch, then the PC is normally not changed until the end of ID, after the completion of the address calculation and comparison.

Figure C.11 shows that the simplest method of dealing with branches is to redo the fetch of the instruction following a branch, once we detect the branch during ID (when instructions are decoded). The first IF cycle is essentially a stall, because it never performs useful work. You may have noticed that if the branch is untaken, then the repetition of the IF stage is unnecessary since the correct instruction was indeed fetched. We will develop several schemes to take advantage of this fact shortly.

One stall cycle for every branch will yield a performance loss of 10% to 30% depending on the branch frequency, so we will examine some techniques to deal with this loss.

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

Figure C.11 A branch causes a one-cycle stall in the five-stage pipeline. The instruction after the branch is fetched, but the instruction is ignored, and the fetch is restarted once the branch target is known. It is probably obvious that if the branch is not taken, the second IF for branch successor is redundant. This will be addressed shortly.

Reducing Pipeline Branch Penalties

There are many methods for dealing with the pipeline stalls caused by branch delay; we discuss four simple compile time schemes in this subsection. In these four schemes the actions for a branch are static—they are fixed for each branch during the entire execution. The software can try to minimize the branch penalty using knowledge of the hardware scheme and of branch behavior. Chapter 3 looks at more powerful hardware and software techniques for both static and dynamic branch prediction.

The simplest scheme to handle branches is to *freeze* or *flush* the pipeline, holding or deleting any instructions after the branch until the branch destination is known. The attractiveness of this solution lies primarily in its simplicity both for hardware and software. It is the solution used earlier in the pipeline shown in Figure C.11. In this case, the branch penalty is fixed and cannot be reduced by software.

A higher-performance, and only slightly more complex, scheme is to treat every branch as not taken, simply allowing the hardware to continue as if the branch were not executed. Here, care must be taken not to change the processor state until the branch outcome is definitely known. The complexity of this scheme arises from having to know when the state might be changed by an instruction and how to "back out" such a change.

In the simple five-stage pipeline, this *predicted-not-taken* or *predicted-untaken* scheme is implemented by continuing to fetch instructions as if the branch were a normal instruction. The pipeline looks as if nothing out of the ordinary is happening. If the branch is taken, however, we need to turn the fetched instruction into a no-op and restart the fetch at the target address. Figure C.12 shows both situations.

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 C.12 The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom). When the branch is untaken, determined during ID, we fetch the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall 1 clock cycle.

An alternative scheme is to treat every branch as taken. As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and begin fetching and executing at the target. Because in our five-stage pipeline we don't know the target address any earlier than we know the branch outcome, there is no advantage in this approach for this pipeline. In some processors—especially those with implicitly set condition codes or more powerful (and hence slower) branch conditions—the branch target is known before the branch outcome, and a predicted-taken scheme might make sense. In either a predicted-taken or predicted-not-taken scheme, the compiler can improve performance by organizing the code so that the most frequent path matches the hardware's choice. Our fourth scheme provides more opportunities for the compiler to improve performance.

A fourth scheme in use in some processors is called *delayed branch*. This technique was heavily used in early RISC processors and works reasonably well in the five-stage pipeline. In a delayed branch, the execution cycle with a branch delay of one is

```
branch instruction
sequential successor1
branch target if taken
```

The sequential successor is in the *branch delay slot*. This instruction is executed whether or not the branch is taken. The pipeline behavior of the five-stage pipeline with a branch delay is shown in Figure C.13. Although it is possible to have a branch delay longer than one, in practice almost all processors with delayed branch have a single instruction delay; other techniques are used if the pipeline has a longer potential branch penalty.

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 C.13 The behavior of a delayed branch is the same whether or not the branch is taken. The instructions in the delay slot (there is only one delay slot for MIPS) are executed. If the branch is untaken, execution continues with the instruction after the branch delay instruction; if the branch is taken, execution continues at the branch target. When the instruction in the branch delay slot is also a branch, the meaning is unclear: If the branch is not taken, what should happen to the branch in the branch delay slot? Because of this confusion, architectures with delay branches often disallow putting a branch in the delay slot.

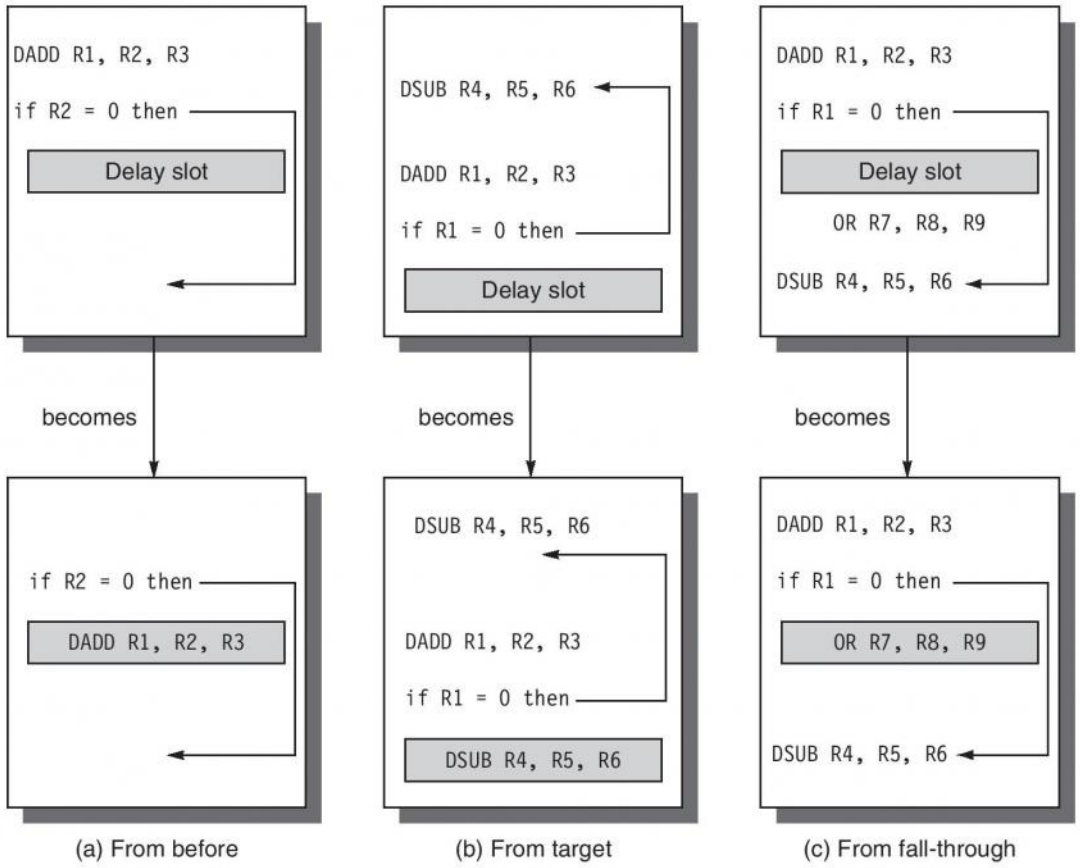


Figure C.14 Scheduling the branch delay slot. The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of R1 in the branch condition prevents the DADD instruction (whose destination is R1) from being moved after the branch. In (b), the branch delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall-through as in (c). To make this optimization legal for (b) or (c), it must be OK to execute the moved instruction when the branch goes in the unexpected direction. By OK we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, in (c) if R7 were an unused temporary register when the branch goes in the unexpected direction.

The job of the compiler is to make the successor instructions valid and useful. A number of optimizations are used. Figure C.14 shows the three ways in which the branch delay can be scheduled.

The limitations on delayed-branch scheduling arise from: (1) the restrictions on the instructions that are scheduled into the delay slots, and (2) our ability to predict at compile time whether a branch is likely to be taken or not. To improve the ability of the compiler to fill branch delay slots, most processors with conditional branches have introduced a *canceling* or *nullifying* branch. In a canceling branch, the instruction includes the direction that the branch was predicted. When the branch behaves as predicted, the instruction in the branch delay slot is simply executed as it would

normally be with a delayed branch. When the branch is incorrectly predicted, the instruction in the branch delay slot is simply turned into a no-op.

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.

Example For a deeper pipeline, such as that in a MIPS R4000, it takes at least three pipeline stages before the branch-target address is known and an additional cycle before the branch condition is evaluated, assuming no stalls on the registers in the conditional comparison. A three-stage delay leads to the branch penalties for the three simplest prediction schemes listed in Figure C.15.

Find the effective addition to the CPI arising from branches for this pipeline, assuming the following frequencies:

Unconditional branch	4%
Conditional branch, untaken	6%
Conditional branch, taken	10%

Branch scheme	Penalty unconditional	Penalty untaken	Penalty taken
Flush pipeline	2	3	3
Predicted taken	2	3	2
Predicted untaken	2	0	3

Figure C.15 Branch penalties for the three simplest prediction schemes for a deeper pipeline.

	Additions to the CPI from branch costs			
Branch scheme	Unconditional branches	Untaken conditional branches	Taken conditional branches	All branches
Frequency of event	4%	6%	10%	20%
Stall pipeline	0.08	0.18	0.30	0.56
Predicted taken	0.08	0.18	0.20	0.46
Predicted untaken	0.08	0.00	0.30	0.38

Figure C.16 CPI penalties for three branch-prediction schemes and a deeper pipeline.

Answer We find the CPIs by multiplying the relative frequency of unconditional, conditional untaken, and conditional taken branches by the respective penalties. The results are shown in Figure C.16.

The differences among the schemes are substantially increased with this longer delay. If the base CPI were 1 and branches were the only source of stalls, the ideal pipeline would be 1.56 times faster than a pipeline that used the stall-pipeline scheme. The predicted-untaken scheme would be 1.13 times better than the stall-pipeline scheme under the same assumptions.

Reducing the Cost of Branches through Prediction

As pipelines get deeper and the potential penalty of branches increases, using delayed branches and similar schemes becomes insufficient. Instead, we need to turn to more aggressive means for predicting branches. Such schemes fall into two classes: low-cost static schemes that rely on information available at compile time and strategies that predict branches dynamically based on program behavior. We discuss both approaches here.

Static Branch Prediction

A key way to improve compile-time branch prediction is to use profile information collected from earlier runs. The key observation that makes this worthwhile is that the behavior of branches is often bimodally distributed; that is, an individual branch is often highly biased toward taken or untaken. Figure C.17 shows the success of branch prediction using this strategy. The same input data were used for runs and for collecting the profile; other studies have shown that changing the input so that the profile is for a different run leads to only a small change in the accuracy of profile-based prediction.

The effectiveness of any branch prediction scheme depends both on the accuracy of the scheme and the frequency of conditional branches, which vary in SPEC from 3% to 24%. The fact that the misprediction rate for the integer programs is higher and such programs typically have a higher branch frequency is a major limitation for static branch prediction. In the next section, we consider dynamic branch predictors, which most recent processors have employed.

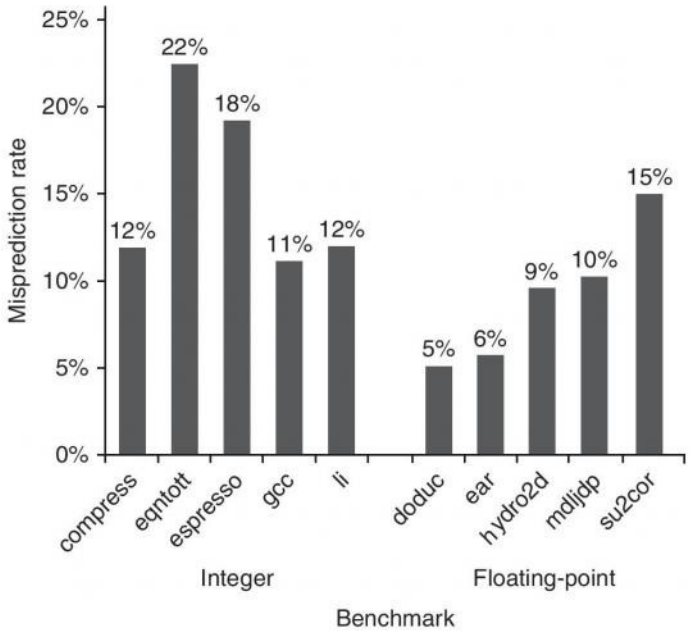


Figure C.17 Misprediction rate on SPEC92 for a profile-based predictor varies widely but is generally better for the floating-point programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%. The actual performance depends on both the prediction accuracy and the branch frequency, which vary from 3% to 24%.

Dynamic Branch Prediction and Branch-Prediction Buffers

The simplest dynamic branch-prediction scheme is a *branch-prediction buffer* or *branch history table*. A branch-prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not. This scheme is the simplest sort of buffer; it has no tags and is useful only to reduce the branch delay when it is longer than the time to compute the possible target PCs.

With such a buffer, we don't know, in fact, if the prediction is correct—it may have been put there by another branch that has the same low-order address bits. But this doesn't matter. The prediction is a hint that is assumed to be correct, and fetching begins in the predicted direction. If the hint turns out to be wrong, the prediction bit is inverted and stored back.

This buffer is effectively a cache where every access is a hit, and, as we will see, the performance of the buffer depends on both how often the prediction is for the branch of interest and how accurate the prediction is when it matches. Before we analyze the performance, it is useful to make a small, but important, improvement in the accuracy of the branch-prediction scheme.

This simple 1-bit prediction scheme has a performance shortcoming: Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken, since the misprediction causes the prediction bit to be flipped.

To remedy this weakness, 2-bit prediction schemes are often used. In a 2-bit scheme, a prediction must miss twice before it is changed. Figure C.18 shows the finite-state processor for a 2-bit prediction scheme.

A branch-prediction buffer can be implemented as a small, special "cache" accessed with the instruction address during the IF pipe stage, or as a pair of bits attached to each block in the instruction cache and fetched with the instruction. If the instruction is decoded as a branch and if the branch is predicted as taken, fetching begins from the target as soon as the PC is known. Otherwise, sequential fetching and executing continue. As Figure C.18 shows, if the prediction turns out to be wrong, the prediction bits are changed.

What kind of accuracy can be expected from a branch-prediction buffer using 2 bits per entry on real applications? Figure C.19 shows that for the SPEC89

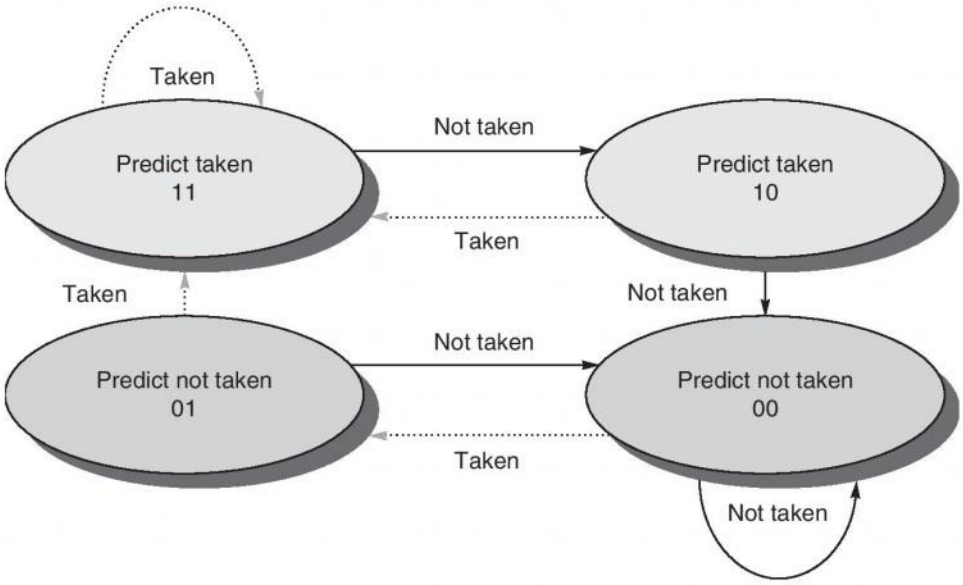


Figure C.18 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually a specialization of a more general scheme that has an n-bit saturating counter for each entry in the prediction buffer. With an n-bit counter, the counter can take on values between 0 and $2^n - 1$: When the counter is greater than or equal to one-half of its maximum value ($2^n - 1$), the branch is predicted as taken; otherwise, it is predicted as untaken. Studies of n-bit predictors have shown that the 2-bit predictors do almost as well, thus most systems rely on 2-bit branch predictors rather than the more general n-bit predictors.

Pipelining exploits the potential **parallelism** among instructions. This parallelism is called **instruction-level parallelism (ILP)**. There are two primary methods for increasing the potential amount of instruction-level parallelism. The first is increasing the depth of the pipeline to overlap more instructions. Using our laundry analogy and assuming that the washer cycle was longer than the others were, we could divide our washer into three machines that perform the wash, rinse, and spin steps of a traditional washer. We would then move from a four-stage to a six-stage pipeline. To get the full speed-up, we need to rebalance the remaining steps so they are the same length, in processors or in laundry. The amount of parallelism being exploited is higher, since there are more operations being overlapped. Performance is potentially greater since the clock cycle can be shorter.

Another approach is to replicate the internal components of the computer so that it can launch multiple instructions in every pipeline stage. The general name for this technique is **multiple issue**. A multiple-issue laundry would replace our household washer and dryer with, say, three washers and three dryers. You would also have to recruit more assistants to fold and put away three times as much laundry in the same amount of time. The downside is the extra work to keep all the machines busy and transferring the loads to the next pipeline stage.

Static Multiple Issue

Static multiple-issue processors all use the compiler to assist with packaging instructions and handling hazards. In a static issue processor, you can think of the set of instructions issued in a given clock cycle, which is called an **issue packet**, as one large instruction with multiple operations. This view is more than an analogy. Since a static multiple-issue processor usually restricts what mix of instructions can be initiated in a given clock cycle, it is useful to think of the issue packet as a single

instruction allowing several operations in certain predefined fields. This view led to the original name for this approach: **Very Long Instruction Word (VLIW)**.

Most static issue processors also rely on the compiler to take on some responsibility for handling data and control hazards. The compiler's responsibilities may include static branch prediction and code scheduling to reduce or prevent all hazards. Let's look at a simple static issue version of a MIPS processor, before we describe the use of these techniques in more aggressive processors.

An Example: Static Multiple Issue with the MIPS ISA

To give a flavor of static multiple issue, we consider a simple two-issue MIPS processor, where one of the instructions can be an integer ALU operation or branch and the other can be a load or store. Such a design is like that used in some embedded MIPS processors. Issuing two instructions per cycle will require fetching and decoding 64 bits of instructions. In many static multiple-issue processors, and essentially all VLIW processors, the layout of simultaneously issuing instructions is restricted to simplify the decoding and instruction issue. Hence, we will require that the instructions be paired and aligned on a 64-bit boundary, with the ALU or branch portion appearing first. Furthermore, if one instruction of the pair cannot be used, we require that it be replaced with a `nop`. Thus, the instructions always issue in pairs, possibly with a `nop` in one slot. Figure 4.68 shows how the instructions look as they go into the pipeline in pairs.

Static multiple-issue processors vary in how they deal with potential data and control hazards. In some designs, the compiler takes full responsibility for removing *all* hazards, scheduling the code and inserting no-ops so that the code executes without any need for hazard detection or hardware-generated stalls. In others, the hardware detects data hazards and generates stalls between two issue packets, while requiring that the compiler avoid all dependences within an instruction pair. Even so, a hazard generally forces the entire issue packet containing the dependent

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

FIGURE 4.68 Static two-issue pipeline in operation. The ALU and data transfer instructions are issued at the same time. Here we have assumed the same five-stage structure as used for the single-issue pipeline. Although this is not strictly necessary, it does have some advantages. In particular, keeping the register writes at the end of the pipeline simplifies the handling of exceptions and the maintenance of a precise exception model, which become more difficult in multiple-issue processors.

instruction to stall. Whether the software must handle all hazards or only try to reduce the fraction of hazards between separate issue packets, the appearance of having a large single instruction with multiple operations is reinforced. We will assume the second approach for this example.

To issue an ALU and a data transfer operation in parallel, the first need for additional hardware—beyond the usual hazard detection and stall logic—is extra ports in the register file (see Figure 4.69). In one clock cycle we may need to read two registers for the ALU operation and two more for a store, and also one write port for an ALU operation and one write port for a load. Since the ALU is tied up for the ALU operation, we also need a separate adder to calculate the effective address for data transfers. Without these extra resources, our two-issue pipeline would be hindered by structural hazards.

Clearly, this two-issue processor can improve performance by up to a factor of two. Doing so, however, requires that twice as many instructions be overlapped in execution, and this additional overlap increases the relative performance loss from data and control hazards. For example, in our simple five-stage pipeline,

loads have a **use latency** of one clock cycle, which prevents one instruction from using the result without stalling. In the two-issue, five-stage pipeline the result of a load instruction cannot be used on the next *clock cycle*. This means that the next *two* instructions cannot use the load result without stalling. Furthermore, ALU instructions that had no use latency in the simple five-stage pipeline now have a one-instruction use latency, since the results cannot be used in the paired load or store. To effectively exploit the parallelism available in a multiple-issue processor, more ambitious compiler or hardware scheduling techniques are needed, and static multiple issue requires that the compiler take on this role.

Simple Multiple-Issue Code Scheduling

How would this loop be scheduled on a static two-issue pipeline for MIPS?

```
Loop: lw    $t0, 0($s1)    # $t0=array element
      addu  $t0,$t0,$s2# add scalar in $s2
      sw    $t0, 0($s1)# store result
      addi  $s1,$s1,-4# decrement pointer
      bne   $s1,$zero,Loop# branch $s1!=0
```

Reorder the instructions to avoid as many pipeline stalls as possible. Assume branches are predicted, so that control hazards are handled by the hardware.

The first three instructions have data dependences, and so do the last two. Figure 4.70 shows the best schedule for these instructions. Notice that just one pair of instructions has both issue slots used. It takes four clocks per loop iteration; at four clocks to execute five instructions, we get the disappointing CPI of 0.8 versus the best case of 0.5., or an IPC of 1.25 versus 2.0. Notice that in computing CPI or IPC, we do not count any nops executed as useful instructions. Doing so would improve CPI, but not performance!

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		lw $t0, 0($s1)	1
	addi $s1,$s1,-4		2
	addu $t0,$t0,$s2		3
	bne $s1,$zero,Loop	sw $t0, 4($s1)	4

FIGURE 4.70 The scheduled code as it would look on a two-issue MIPS pipeline. The empty slots are no-ops.

Loop Unrolling for Multiple-Issue Pipelines

See how well loop unrolling and scheduling work in the example above. For simplicity assume that the loop index is a multiple of four.

To schedule the loop without any delays, it turns out that we need to make four copies of the loop body. After unrolling and eliminating the unnecessary loop overhead instructions, the loop will contain four copies each of `lw`, `add`, and `sw`, plus one `addi` and one `bne`. Figure 4.71 shows the unrolled and scheduled code.

During the unrolling process, the compiler introduced additional registers (`$t1`, `$t2`, `$t3`). The goal of this process, called **register renaming**, is to eliminate dependences that are not true data dependences, but could either lead to potential hazards or prevent the compiler from flexibly scheduling the code. Consider how the unrolled code would look using only `$t0`. There would be repeated instances of `lw $t0,0($$s1)`, addu `$t0, $t0, $s2` followed by `sw t0,4($s1)`, but these sequences, despite using `$t0`, are actually completely independent—no data values flow between one set of these instructions and the next set. This case is what is called an **antidependence** or **name dependence**, which is an ordering forced purely by the reuse of a name, rather than a real data dependence that is also called a true dependence.

> Renaming the registers during the unrolling process allows the compiler to move these independent instructions subsequently so as to better schedule

	ALU or branch instruction		Data transfer instruction		Clock cycle
Loop:	addi	$s1,$s1,-16	lw	$t0, 0($s1)	1
			lw	$t1,12($s1)	2
	addu	$t0,$t0,$s2	lw	$t2, 8($s1)	3
	addu	$t1,$t1,$s2	lw	$t3, 4($s1)	4
	addu	$t2,$t2,$s2	sw	$t0, 16($s1)	5
	addu	$t3,$t3,$s2	sw	$t1,12($s1)	6
			sw	$t2, 8($s1)	7
	bne	$s1,$zero,Loop	sw	$t3, 4($s1)	8

FIGURE 4.71 The unrolled and scheduled code of Figure 4.70 as it would look on a static two-issue MIPS pipeline. The empty slots are no-ops. Since the first instruction in the loop decrements $s1 by 16, the addresses loaded are the original value of $s1, then that address minus 4, minus 8, and minus 12.

the code. The renaming process eliminates the name dependences, while preserving the true dependences.

Notice now that 12 of the 14 instructions in the loop execute as pairs. It takes 8 clocks for 4 loop iterations, or 2 clocks per iteration, which yields a CPI of 8/14 = 0.57. Loop unrolling and scheduling with dual issue gave us an improvement factor of almost 2, partly from reducing the loop control instructions and partly from dual issue execution. The cost of this performance improvement is using four temporary registers rather than one, as well as a significant increase in code size.