3. The CPU

Most people are blissfully unaware of the central processing unit. They write, compile, and run their programs with no worry about how they actually execute; and for the most part their programs run with reasonable efficiency. Generally speaking, this is also true of scientific computing. Yet there are times when seemingly minor modifications to a program can speed it up dramatically, and the sources of the speedups are often to be found in the behavior of the central processing unit or its interaction with the memory hierarchy. In this section we will treat the CPU.

The design and analysis of central processing units is a topic that can fill volumes. In this highly selective presentation we will concentrate on the aspects that are immediately useful to the scientific programmer. We begin this section with an overview of how a CPU processes data. We will then discuss a specific machine, the MIPS64. This machine will be used in the next section to discuss instruction pipelining, the most important of a number of techniques used to design fast CPU's. We will then turn to floating-point computations, illustrating our discussion with two widely used vector operations. This lays the basis for a treatment of vector computers.

3.1. Overview

Synopsis: The CPU processes data by executing a sequence of instructions stored in memory. Ordinarily instructions are executed in the order they appear sequentially in memory. But jump and branch instructions can be used to override the natural sequence. Today's CPU's are usually load-and-store machines. They have many registers where operations are performed. Special load and store instructions are used to move data to and from memory. These instructions have addressing modes that aid in computing the address of a memory reference.

The execution of the CPU is synchronized by a clock. The time between two ticks of a clock is called a cycle, and the speed of the CPU is measured in cycles per second, aka hertz. The execution of an instruction usually takes several cycles, the number of which may vary from CPU to CPU. Thus raw clock speed is not a reliable predictor of the performance of a computer.

—

We have already observed that a CPU does its work by modifying data in registers or main memory according to a sequence of instructions that are themselves contained in main memory. The address of the current instruction is contained in a special register called the **program counter (PC).** Ordinarily the instructions are executed in the order that they appear in memory. Equivalently, after each instruction is executed the PC is incremented by the length of the instruction.

This arrangement is sufficient to evaluate a sequence of expressions, but it precludes branching — the basis of control statements of a high-level programming language. Consequently, the CPU has instructions that override the default incrementation of the PC.

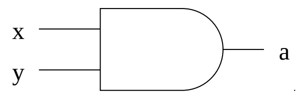
There are three types of such instructions. A **jump** changes the PC to some value specified by the programmer. It is the equivalent to a `goto` in a high-level programming language. A **branch** changes the PC to one of two values depending on whether a certain condition is satisfied. The condition varies with the architecture of the CPU. For example, a branch may depend on whether an arithmetic operation has just produced a negative result. Branches are used to implement `if` statements and loops. The final instruction is a **jump and save PC**, which saves the value of the PC before changing it, so that the program knows how to get back to where it was. It is used to implement subprogram calls. Note that the terminology for these instructions varies from machine to machine.

The CPU must also interact with main memory to fetch instructions and operands and to store results. There are two aspects to this interaction. The first concerns how operands are fetched and results are stored. One long defunct approach, called **memory-memory architecture,** has each individual instructions fetching its operands, performing its operation, and storing its result — all in one fell swoop. Another approach is to have a single register, usually called an **accumulator,** in which operations are performed. Both these approaches — especially the accumulator — had decided advantages in the days when hardware and registers were costly. Today the trend is to provide a plentiful supply of registers and to perform all operations in them. Special instructions load operands from memory and store back results. This decreases memory references and encourages the reuse of operands already in registers. Such a computer is called a **load-store computer.**

The second aspect is how memory addresses are computed. For example, if one is looping to sum the elements of an array, one must compute the address of each element to load it to a register. It is desirable that the load instruction lend an hand. Over the years, many different **addressing modes** have been devised. We will return addressing modes when we consider the MIPS computer.

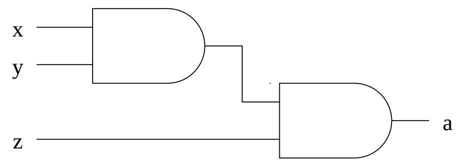
The execution of a CPU is synchronized by a **clock**, which beats time for the operations of the CPU. The interval between two beats is called a **cycle**. The speed of a CPU is generally measured in cycles per second or **hertz**. Thus a 2 GHz (giga hertz) system has a clock that runs through two billion cycles in a second. The **period** or **cycle time** of a clock is the time required for a single cycle. It is the reciprocal of its speed in hertz. Thus our 2 GHz clock has a period of 0.5 nanoseconds. To put this in perspective, a beam of light in a vacuum travels about half a foot in 0.5 ns. Since nothing travels faster than light, no 2 GHz CPU can be larger than half a foot (and in practice it must be considerably smaller).

How is an instruction executed? The actual execution is done by **logic circuits** that are built up from primitive units called **gates.** The following diagram illustrates an **AND gate.**



The gate has two inputs, x and y, and one output, a. The inputs and outputs are currents at one of two nonoverlapping voltage ranges: high and low. A low voltage represents a 0 bit and a high voltage represents a 1 bit. The output of the gate is the logical and of its inputs — that is, a is 1 if and only if both x and y are 1. There are a variety of gates: inverters, OR gates, XOR gates, and so on.

Gates can be combined to produce more complicated functions. As a simple example, the following combination of two AND gates



gives a three-input AND gate, whose output is 1 if and only if all its inputs are 1. More complicated combinations yield circuits that can perform shifts, additions, and many other operations. An important aspect of these circuits is that it takes time for the inputs to propagate through them — their outputs do not appear instantly.

Returning to the execution of an instruction, at the beginning of a cycle the instruction is in a special register called the instruction register (IR). At the tick of the clock, data flows from the IR and the registers through the logic circuits. Under control of the IR register, the logic circuits produced the desired results which pile up at the registers. At the end of the cycle, the results are latched into the registers and the CPU proceeds to the next instruction.

All this is an oversimplification. In practice an instruction usually takes several clock cycles to execute. For example, in one cycle the instruction may be fetched from memory and placed in the IR. In the next, the instruction may be decoded to send control signals to the logic circuits. Another cycle may be required to compute the results of the instruction, and another to deliver it to the appropriate registers. This multiplicity of cycles means that the CPU must have internal or working registers to store intermediate results. These registers are invisible to the programmer: they cannot be directly written or read. The instruction register is an example of such an internal register.

The fact that instructions require several cycles to execute highlights the limitations of the clock speed as a measure of performance. Performance also depends on the nature of the instructions the computer can execute, how much their executions can be overlapped or pipelined, and how quickly items can be transferred between the CPU and memory. Thus two computers with the same clock rate may perform quite differently on a given application. However, of two computers with the same basic architecture, the one with the faster clock will generally outperform the other.

3.2. The MIPS64 computer

Synopsis: The MIPS64 is a load-store reduced instruction set computer that works with 64 bit registers. It has 32 general-purpose registers and 32 floating-point registers. In describing the MIPS instruction set, we use assembly language conventions.

MIPS has a displacement addressing mode in which a register is added to a base contained in the instruction to get the final memory address. It also has an immediate addressing mode, in which an operand is contained in the instruction.

MIPS has a full complement of load-store, arithmetic-logical, branch, and jump instructions, along with instructions for comparing the contents of registers.

Some of the features of MIPS are illustrated by an implementation of AXPY, which overwrites a vector y with $ax + y$.

—

3.2.1. Registers

The MIPS64 computer (here shortened to MIPS) is a embedded computer — that is a computer designed to be incorporated into devices, like cell phones and hand calculators, that require computer assistance. The 64 refers to the fact that it operates with 64 bit words. It is load-store **reduced instruction set computer (RISC).** The basic architecture was developed by John L. Hennessy, author with David A. Patterson of *Computer Architecture: A Qualitative Approach* (3rd edition), where it is used as running example. You should look to this excellent compendium for more details on some of the topics in this section.

The MIPS has 32 general purpose registers named R0, R1, ..., R31, each containing 64 bits. The register R0 always contains zero. The MIPS also has 32 double-precision floating-point registers F0, F1, ..., F31, which can also be used for single-precision numbers. In addition there are some special purpose registers that need not concern us here.

3.2.2. Assembly languages

In order to treat MIPS instructions we need a language to talk about them. In the very earliest days of computers, these instructions would have been coded as binary numbers. Needless to say it was quickly realized that writing out a program in binary was a time-consuming, error-prone task. The cure was to write the program in a coded form and let another program, called an **assembler,** translate it into the numerical form the machine could execute. Thus when people speak of machine language programs they usually mean **assembly language** programs. As it has turned out, assembly programs have been largely superseded by compiled programs written in high-level programming languages. Nonetheless, it is useful to be able to read assembly language programs — for example, to determine what a compiled program is actually up to. We will use assembly language constructs in describing the MIPS instructions.

3.2.3. Instructions

MIPS has a fixed-length instruction format of 32 bits. The first six bits form an operation code which tells what the instruction does. The fixed length makes it easy to fetch instructions. The uniform operation-code field helps in the decoding of instructions.

There are three distinct MIPS instruction formats, which are illustrated in Figure 3.1. We will examine the I-type instruction in detail because it illustrates a number of points about the operation of the MIPS machine. We will then briefly discuss the other two.

I-type instructions perform four general functions: reading and writing memory, performing immediate operations, implementing branches, and implementing register jumps. We will consider each of these functions in turn.

Instructions that move data between registers and memory are called **load-store instructions.** For example, the MIPS instruction

```
LD  R5, 24(R17)
```

causes a double word to be loaded from memory into register `R5`. The abbreviation `LD` stands for Load Double, and it corresponds to a particular configuration of bits in the operation-code field of the instruction. The register `R5` is the destination register, and the corresponding field of the instruction contains 00101_2. Similarly, `R17` is the source register, so that the corresponding field contains 01001_2. Finally, `24` is the immediate value, so that the immediate field contains 18_{16}.

The address of the word to be loaded is computed by adding the contents of the immediate field, 24, to the contents of the source register `R5`. This **addressing mode** is useful in loops where elements of an array must be loaded in succession. For example, if the array consists of contiguous double words in memory, one only has to increase `R5` by 8 to point the instruction to the next word in the array. Note that if `R17` is replaced by `R0` (which is always zero), the instructions loads the word at the absolute address of 24.

MIPS has a variety of load-store instructions to manipulate words of all sizes from bytes to double words.

I-type instructions also implement **immediate operations**, another addressing

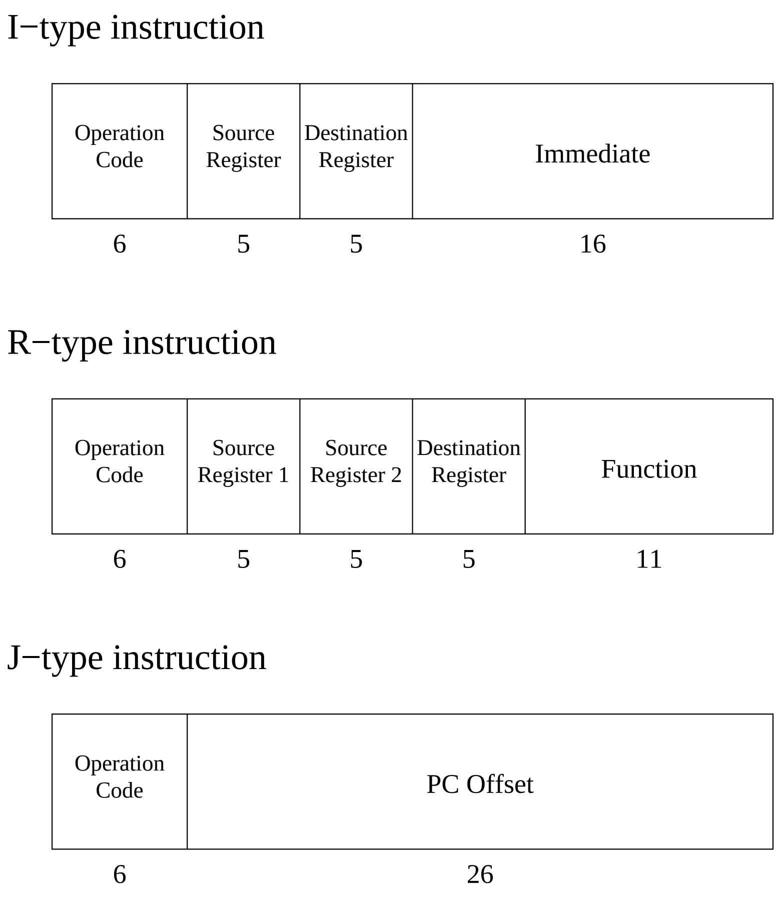


Figure 3.1: Instruction formats for the MIPS

mode in which one of the operands is contained in the instruction itself. For example the instruction

```
DADDI R17, R5, #7
```

places 7 plus the contents of `R7` in `R17`. The number 7 occupies the immediate field, which is always treated as a 2's complement signed integer. Again

```
DSRLI R3, R2, #4
```

places the result of performing a logical right shift on the contents of `R2` in `R3`. The source and destination registers can be the same, so that

```
DADDI R1, R1, #-1
```

decrements the contents of R1 by one.

By incorporating one of its operands into itself an immediate instruction saves having to load the operand from memory. However, the small size of the immediate field limits the size of the operand. Moreover, the operand is hard wired into the program and cannot be changed.[4]

The MIPS has branch instructions that test whether a register is zero or nonzero and branch accordingly. For example, the code

```
BEQZ R1, #-100                                        (3.1)
```

adds $-100{\cdot}4 + 4$ to the PC if the contents of R1 (which is the source register in the I-format) are zero and otherwise does nothing. This effects a conditional backward jump of 99 instructions in the program. The product $-100{\cdot}4$ comes from the fact that instructions are a fixed 4 bytes in length. A branch of this kind is said to be **PC relative.** The fact that the immediate field contains only 16 bits, restricts a branch induced jump to about 130,000 instructions in either direction. However, a branch can be combined with other jump instructions to produce larger conditional jumps.

Finally, the I-type format implements jump-register instructions. Specifically, the code

```
JR R1
```

causes an absolute jump to the address contained in R1.

The R-type instructions implement operations involving registers. For example, the code

```
DSUB R5, R3, R20
```

subtracts the contents of source register R20 from those of source register R3 and places the result destination register in R5. Again the code

```
MUL.D F2, F4, F6
```

multiplies the double-precision floating-point numbers contained in the registers F4 and F6 and places the result in F2.

An important class of R-type instructions is the set conditionals. For example,

```
SNE R1, R2, R3
```

[4]Actually, since instructions reside in memory, a program can, in principle, change the immediate field of one of its instructions. In the early days of digital computing this was regarded as a virtue of von Neumann machines. Nowadays, however, code is always left inviolate, so that different processes can share it. Such code is called **reentrant.** For more see §8.2.3.

sets `R1` to 1 if the contents of `R2` and `R3` are not equal and otherwise sets `R1` to zero. These tests, which include `LT, GT, LE, GE, EQ,` in addition to `NE`, can be combined with the branch instructions to implement branches for all six conditions. There are analogous set conditionals for floating-point numbers.

The J-type instructions implement jumps. For example,

```
J #100                                                        (3.2)
```

changes the PC to $\texttt{PC} + \texttt{4}{\cdot}\texttt{100} + \texttt{4}$. This handling of the PC is analogous to the branch instructions. However, since the PC offset field contains 26 bits the possible size of a jump is over 130 M instructions. A corresponding jump-and-link behaves in the same way, but is also saves $\texttt{PC} + \texttt{4}$ in `R31`. As we have already mentioned, jumps of this type can be used to implement calls to subprograms.

3.2.4. An example: AXPY

As a nontrivial example of MIPS code we will consider the process of overwriting a vector y with $ax + y$, where x is a vector and a is a scalar. This operation is common in scientific computing and is called an AXPY for A X Plus Y. (When it is necessary to distinguish single from double precision versions, AXPY becomes SAXPY and DAXPY respectively.)

Using Matlab notation, an AXPY can be written in scalar form as

```
for i = 1:n
   y(i) = y(i) + a*x(i);                                      (3.3)
end
```

where `n` is the dimension of the vectors x and y. Figure 3.2 gives a MIPS implementation of this loop. From the accompanying comments it should not be difficult to puzzle out how this code works. But here are some additional points.

- `Loop` is a symbolic name for the address of the instruction beginning the AXPY loop, and it is referenced in the branch at the end of the loop. The assembler is responsible for reconciling these references so that the branch goes to the right place.

- The number 8 in the `DADDI`'s is the length of a double-precision number in bytes.

- The variable `i` has disappeared from the implementation. Its place has been taken by the registers `R1` and `R2`, which contain the addresses of `x(i)` and `y(i)`. The fact that two registers, `R1` and `R2`, instead of the single integer `i` must be updated at each stage of the loop, shows that there is more to indexing a variable than meets the eye in a high-level programming language.

- Since, at the end of the loop, there is no `i` to compare with `n`, the address of `y(n+1)` is compared with the address `R2`.

This code implements the AXPY loop (3.2) in double precision. Initially it is assument that

> `R1` contains the address of `x(1)`
> `R2` contains the address of `y(1)`
> `R3` contains the address of `x(1)` plus `8*n`
> `F0` contains the constant `a`

```
Loop: L.D    F1, 0(R1)  ; load x(i)
      L.D    F2, 0(R2)  ; load y(i)
      MUL.D  F1, F0, F1 ; a*x(i)
      ADD.D  F2, F2, F1 ; y(i) + a*x(i)
      S.D    0(R2), F2  ; store new y(i)
      DADDI  R1, #8     ; update address of x(i)
      DADDI  R2, #8     ; update address of y(i)
      DSUB   R4, R1, R3 ; zero at end of loop
      BNZE   R4, Loop
```

Figure 3.2: MIPS implementation of AXPY

3.3. Instruction pipelines

Synopsis: The classical example of a pipeline is the automobile assembly line. Cars are moved from stage to stage of the line. At each stage a specific part of the assembly is performed. Although it takes time proportional to the number of stages for a car to be assembled, cars pop out of the assembly line at the rate they move from stage to stage.

Instructions in the MIPS take five cycles to execute, corresponding to five distinct stages. Consequently, after one instruction completes the first state, a second can follow it into the instruction pipe line. After five instructions have done so, MIPS will be executing instructions at the rate of one per cycle.

Unfortunately, certain hazards can cause the pipeline to stall. There are three kinds. A structural hazard occurs when two instructions must use the same resource at the same time, as when one instruction must be read from memory while the another also needs to read memory. A data hazard occurs when and instruction needs the results of a previous instruction that is still in the pipeline. Finally, branch hazards occur because then next instruction cannot be brought into the pipeline until the branch address has been computed. There are various fixes for these problems. But even the best planned pipelines stall from time to time.

Instruction pipelining is only one (very important) technique for designing fast computers. Combining it with other techniques results in superscalar computers that can at times complete more than one instruction per cycle.

—

3.3.1. Pipelining

We have mentioned (p. 37) that instructions in a CPU execute in stages, each stage requiring one cycle. This suggests that once an instructions has left the first stage, another instruction may begin executing and follow the first instruction one stage behind. This process is called **pipelining.** In this section, we will show how instructions can be pipelined on the MIPS computer. But first some words on pipelines in general.

The classic example of a **pipeline** is an assembly line in an automobile factory. The assembly of the car is broken down into independent stages, each taking roughly the same time, and each stage is assigned a position in the assembly line. As the car moves from position to position, workers perform the tasks particular to their stage. At the end the result is a fully assembled car. The beauty of the assembly line is that once a car has moved from stage one to stage two another care can be started in stage one. With cars following one another through the assembly line, they emerge at the rate of one car in the time required to complete one stage, even though any individual car requires n stages to complete, where n is the length of the assembly line.[5]

There is some terminology associated with pipelines. The number of stages in a pipeline is called the **length** of the pipeline. The time it takes for an item to be appear at the end of the pipeline, which is generally a multiple of the length, is called the **latency** or **startup time** of the pipeline. The rate at which the pipeline produces its results is called its **bandwidth.** In computer pipelines, latency is usually measured in cycles.

A difficulty with pipelines is that they must be kept well fed to achieve their peak bandwidth. If the pipeline is not provided with input at one of its cycles, an empty bubble passes through the pipe, producing no output at the end and hence reducing the average bandwidth. A related difficulty is that if one of the stages cannot finish on time, all the stages before it must wait until it finishes. This situation is sometimes called a stall.

3.3.2. The MIPS instruction pipeline

In describing the MIPS instruction pipeline, we will exclude floating-point operations, which will be treated in the next subsection. There are five stages in the execution of a MIPS instruction.

1. Instruction fetch (IF): The instruction is fetched from memory and placed in the instruction register (IR).
2. Instruction decode (ID): The bits of the instruction are decoded into control signals. Operands are moved from registers or immediate fields to working

[5]Although Henry Ford is usually regarded as the inventor of the assembly line, Ransom E. Olds (of Oldsmobile fame) first introduced it in 1901, where it increased his production by sixfold.

registers. For branch instructions, the branch condition is tested and the branch address computed.

3. Execution (EX): The instruction is executed. Specifically, if the instruction is an arithmetic or logical operation, its results are computed. If it a load-store instruction, the address is computed. All this is done by an elaborate logic circuit called the arithmetic-logical unit (ALU).
4. Memory read/write (ME): If the instruction is a load-store, the memory is read or written.
5. Write back (WB): The results of the operation are written to the destination register.

In the MIPS pipeline these instructions follow one another through the pipeline. We can represent the procession of a sequence of instructions I1, I2, ... through the pipeline pictorially as follows.

cycle	IF	ID	EX	ME	WB
1	I1				
2	I2	I1			
3	I3	I2	I1		
4	I4	I3	I2	I1	
5	I5	I4	I3	I2	I1
6	I6	I5	I4	I3	I2
.	.	.	.	.	.

(3.4)

It takes five cycles to fill the pipe, after which instructions complete at the rate of one per cycle.

Although we cannot go into the details of how an instruction pipline is implemented, it is useful to think of the pipline as consisting of logic circuits connecting internal registers, as illustrated below.

IR L_{if} R_{ii} L_{id} R_{ie} L_{ex} R_{me} L_{wb} Visible registers

The L's represent logic circuits and are subscripted by their stages. The R's represent internal buffer registers between the stages. The subscripts are formed from the first letters of the stages they separate. At the right end, is an incoming instruction in the instruction register and at the left are the visible registers where the results are to be placed. At the beginning of the a cycle, the contents of the registers are allowed to flow through the logic circuits. After they have propagated through the circuit, they are latched into the registers at the right of the stage. This process keeps the execution of the instructions in order — no instruction can race ahead of another.

3.3.3. Impediments to instruction pipelining

In practice, certain impedements, called **hazards,** may keep pipelining from achieving its full potential. Hazards may be divided into three types: structural hazards, data hazards, and branch hazards.

Structural hazards occur when two instructions must use the same part of the computer at the same time. For example, suppose that I1 in (3.4) is a load-store instruction. Then at cycle 4, I1 must access memory. But at the same time I4 has to be fetched from memory. Unless the machine is organized so that instructions can be fetched simultaneously with memory reads and writes, the entry of I4 into the pipeline will have to be delayed or **stalled** for a cycle (or more, if, say, I2 is a load-store instruction). Since this kind of hazard will occur whenever a load-store instruction is executed, it is not surprising that computers have a separate channel for fetching instructions.

Data hazards occur when data for needed by one instruction has not been computed by a previous instruction. For example consider the sequence of instruction

```
DADD   R1, R2, R3
DSUB   R1, #10                                   (3.5)
```

Let's follow these two instructions through the pipeline (we abbreviate `AD` and `SB` for `DADD` and `DSUB`).

cycle	IF	ID	EX	ME	WB
1	AD				
2	SB	AD			
3	XX	SB	AD		
4	XX	SB	–	AD	
5	XX	SB	–	–	AD
6	XX	SB	–	–	–
7	XX	XX	SB	–	–
.	.	.	.	.	.

At cycle 3, `SU` needs to move the contents of `R1` to a working register as part of the decoding process. But `R1` does not contain the results of `AD`—and will not until `AD` completes its WB stage. Thus the pipeline stalls with `SB` in the ID stage until `AD` completes the WB, with a loss of three cycles.

A cure for this problem can be found in the fact that at the end of cycle 3, the result of `AD` is about to be latched into a working register as its EX stage is completed. With the help of some extra logic circuits the result can be delivered to the working register that would ordinarily be loaded by `SB` from `R1`. This process that is called **forwarding.** Now the pipeline proceeds with no stalls.

It must not be thought, however, that forwarding resolves all data hazards. Let's change the `DADD` in (3.5) to a load.

```
LD      R1, 0(R2)
DSUB    R1, #10
```

Then the pipeline behaves as follows.

cycle	IF	ID	EX	ME	WB
1	LD				
2	SB	LD			
3	XX	SB	LD		
4	XX	SB	–	LD	
5	XX	XX	SB	–	LD
6	XX	XX	XX	SB	–
.	.	.	.	.	.

The difficulty here is that the ultimate contents of `R1` are no longer available at the end of the EX stage. Instead, we have to wait until the end of the ME stage to forward the result of the load. This causes a stall of one cycle.

A possible cure for this problem is to find an instruction after `DSUB` and place it between `LW` and `DSUB`, a process known as **rescheduling.** This instruction must not use or alter `R1`. In more formal language we say there must be no **data dependencies** between the instruction and `LD` and `LSUB`. Since most programs are not written in assembly language, the determination of an appropriate instruction must be left to the compiler. This illustrates an important point about machines with instruction pipes. They need smart compilers.

Branch hazards occur because we do not know the status of the branch condition and the branch address until the end of the ID stage. As the following diagram shows this creates a stall. Here BR stands for the branch instruction and TG for the target instruction at the address actually taken by the branch.

cycle	IF	ID	EX	ME	WB
1	BR				
2	–	BR			
3	TG	–	BR		
4	XX	TG	–	BR	
5	XX	XX	TG	–	BR
6	XX	XX	XX	TG	–
.	.	.	.	.	.

The cure for a branch hazard is to schedule an instruction to fill the stall. There are two ways of doing this: static and dynamic.

In static scheduling the compiler choses an instruction to fetch. Ideally, this instruction would be one that was independent of the branch. But if such an instruction does not exist, the compiler can assume one of two things. First, it may assume that the branch will not be taken and schedule the instruction following the branch. Second, in may assume that the branch will be taken and schedule the target of the branch. If the assumption about the branch turns out to be wrong, the scheduled instruction must be cancelled, creating a stall.

In dynamic scheduling the CPU chooses the instruction to fill the stall. The CPU may use a fixed strategy. For example, assuming that backward branches will be taken is a good strategy, since most loops end with a backward branch to the beginning of the loop where an instruction is waiting to be executed. Or the CPU can try to predict which way the branch will go on the basis of past history. Some aspects of branch prediction are treated in connection with finite-state automata (see §7.2.3).

—

Instruction pipelining is the basic technique for speeding up CPU's. But it is not the only one. Parallel pipelines allow for the execution of more than one instruction at a time. Dynamic scheduling and branch prediction can reduce branch stalls beyond what a compiler can achieve. When these and other techniques are incorporated into a CPU, the result is a **superscalar computer** that can, in some applications, achieve execution rates faster than one instruction per cycle.

3.4. Floating-point pipelines

Synopsis: Floating-point operations can be pipelined, but it takes some care to realize any benefits. Two commonly used tricks are loop unrolling and loop splitting.

Loop unrolling is illustrated by AXPY. In its natural form it cannot be pipelined because at the ith stage of the loop the addition of y_i in $ax_i + y_i$ must wait for the multiplication by a to complete. Moreover, the y_i cannot be overwritten by $ax_i + y_i$. On the other hand the computation of $ax_i + y_i$, $ax_{i+1} + y_{i+1}$, and $ax_{i+2} + y_{i+2}$ are independent. Hence if we increment our loop by three instead of one and properly reschedule the loads, multiplies, adds, and stores, we can improve the pipelining.

Loop splitting is illustrated by illustrated by the problem of computing the sum $x_1 + x_2 + \cdots + x_n$. If this is done is the usual way by accumulating the sum in a register, there will be no benefits of pipelining. On the other hand if if we compute $x_1 + x_{n/2+1}, \ldots, x_{n/2} + x_n$, the sums are independent and their computation can be pipelined by loop unrolling. Applying this method recursively computes the sum. However, the numerical properties of this algorithm are different from straightforward accumulation.

—

Floating-point arithmetic is a complicated affair, and its operations tend to take longer than other machine operations. Consequently, if floating-point operations are plentiful in an application, floating-point arithmetic may dominate the calculations.

According to Amdahl's law, to speed up the computations we should start by speeding up floating point-arithmetic. One way of doing this is to pipeline floating-point operations.

The details of the stages of floating-point pipelines need not concern us here. Instead, in our discussion we will assume that floating-point additions and multiplications can be independently pipelined with a latency of three cycles, and that all other instructions require one cycle to execute. Although these assumptions are an oversimplification, they are are near enough the mark not to mislead us about the qualitative behavior of floating-point pipelines.

3.4.1. Loop unrolling

Let us return to the AXPY code in Figure 3.2, which we reproduce here with the number of cycles required by each instruction.

```
1. Loop: L.D    F1, 0(R1)  ; 1
2.       L.D    F2, 0(R2)  ; 1
3.       MUL.D  F1, F0, F1 ; 3
4.       ADD.D  F2, F2, F1 ; 3
5.       S.D    0(R2), F2  ; 1
6.       DADDI  R1, #8     ; 1
7.       DADDI  R2, #8     ; 1
8.       DSUB   R4, R1, R3 ; 1
9.       BNZE   R4, Loop   ; 1
```

The total number of cycles is 13. Our floating-point pipeline has not helped us at all! The reason is first that the `ADD.D` must wait for 3 cycles for the `MUL.D` to complete and to place its result in `F2`. Moreover, the `S.D` must wait for the `ADD.D` to complete — another example of data dependency.

A cure for this dependency problem is to note that there is no data dependency between the floating point operations of different iterations of the loop. Thus if we can collect iterations together, we can fill the pipeline. This commonly used process is called **loop unrolling**. The number of iterations we combine is the **unrolling factor.** Here is an example of a 3-fold unrolling of AXPY

```
for i = 1:3:n
   y(i) = y(i) + a*x(i);
   y(i+1) = y(i+1) + a*x(i+1);                    (3.6)
   y(i+2) = y(i+2) + a*x(i+2);
end
```

For simplicity we have assumed that `n` is is divisible by 3.

The corresponding MIPS code is contained in Figure 3.3. Note that the code is not

```
Loop: L.D     F1,  0(R1)
      L.D     F2,  8(R1)
      L.D     F3,  16(R1)
      L.D     F4,  0(R2)
      L.D     F5, 8(R2)
      L.D     F6, 16(R2)
      MUL.D   F1, F0, F1
      MUL.D   F2, F0, F2
      MUL.D   F3, F0, F3
      ADD.D   F4, F1, F4
      ADD.D   F5, F2, F5
      ADD.D   F6, F3, F6
      S.D     0(R2), F4
      S.D     8(R2), F5
      S.D     16(R2), F6
      DADDI   R1, #24
      DADDI   R2, #24
      DSUB    R4, R1, R3
      BNZE    R4, Loop
```

Figure 3.3: AXPY loop unrolled

the literal equivalent of (3.6), since the assembly language instruction are not in their natural order. This instruction rescheduling is necessary to get the benefits of loop unroling.

To count the number of cycles this loop takes, note that the `L.D`'s take 6 cycles. We must then issue 3 `MUL.D`'s requiring 3 cycles. Because the latency of the `MUL.D` pipeline is three, after the `MUL.D`'s have been issued, the first product is available to fed into the `ADD.D` pipeline. Three cycles later we can begin executing the 3 stores followed by the 4 loop control instructions. Thus we have a total of

$$6\,\texttt{L.D} + 3\,\texttt{MUL.D} + 3\,\texttt{ADD.D} + 3\,\texttt{S.D} + 4\,\texttt{loop} = 19\,\text{cycles}.$$

Since each iteration of this loop is equivalent to three iterations of the original loop, we divide by 3 to get a effective loop cycle count of $6\frac{1}{3}$ cycles. Thus unrolling the loop has reduced the original count of 13 by a factor of about 2.

It is instructive to ask what happens when we increase the unrolling factor to k, where $k > 3$. Reasoning as above, we find that our cycle count is

$$2k\,\texttt{L.D} + k\,\texttt{MUL.D} + k\,\texttt{ADD.D} + k\,\texttt{S.D} + 4\,\texttt{loop} = 5k + 4$$

cycles. Dividing by k we get an effective count of

$$5 + \frac{4}{k}$$

cycles.

In this count we recognize the Amdahl hyperbola. The best count we can get is 5, and when $k = 3$ we are quite near it. Increasing k to five gives a count of 5.8, which is not much improvement over 6.3.

Amdahl's law is not the only reason for limiting the degree of unrolling. Unrolling consumes registers. In the AXPY example it requires $2k+1$ registers to unroll the loop by a factor of k. On the MIPS, which has 32 floating-point registers, k is effectively limited to 15.

It is also instructive to ask what happens when $k \leq 3$. In this case the sum is

$$2k\,\texttt{L.D} + 3\,\texttt{MUL.D} + 3\,\texttt{ADD.D} + k\,\texttt{S.D} + 4\,\texttt{loop} = 3k + 10.$$

The counts for `MULTD` and `ADDD` change from k to 3 because we must wait for the results to get through the floating-point pipeline. The average count is

$$3 + \frac{10}{k}.$$

Once again we have an Amdahl hyperbola. But because k is small and the multipier of k^{-1} is large we get a lot of bang for our bucks—passing from 13 to 8 to 6.3 for $k = 1, 2, 3$. The common sense of all this is that for $k \leq 3$ we gain both by using the pipeline more efficiently $(6/k)$ and by reducing the average loop overhead $(4/k)$. For $k > 3$ we only reduce the average loop overhead, which is already small for $k = 3$.

To simplify our exposition, we assumed that n was divisible by $k = 3$. When it is not, we must add a loop to take care of the part left out. Thus in general we modify (3.6) to read

```
m = rem(n, 3);
for i=1:m
    y(i) = y(i) + a*x(i);
end
for i = m+1:3:n                                    (3.7)
   y(i) = y(i) + a*x(i);
   y(i+1) = y(i+1) + a*x(i+1);
   y(i+2) = y(i+2) + a*x(i+2);
end
```

Actually with aggressive optimizing compilers it is not necessary to write unrolled code like (3.7). They will automatically unroll loops and reschedule the resulting instructions in the optimization process.

3.4.2. Loop splitting

We now turn to another vector operation — the dot product of two vectors defined by $x^{\mathrm{T}}y = \sum_{i=1}^{n} x_i y_i$. Sums like this are difficult to pipeline. The see why, let us consider the simpler problem of computing

$$\text{sum} = \sum_{i=1}^{n} x_i.$$

The natural code for this computation is

```
sum = 0;
for i = 1:n
   sum = sum + x(i)                                        (3.8)
end
```

Now consider the first two iterations of the loop.

```
sum(2) = sum(1) + x(1);
sum(3) = sum(2) + x(2);
```

We have indexed `sum` to distinguish its values at different iterations of the loop. We can only compute `sum(2)` after `sum(1)` has been computed. But that computation requires three cycles — the length of the floating-point pipeline. Similarly, we can only compute `sum(3)` after `sum(2)` has been computed, which causes another three cycle delay — and so on. The result is that the pipeline is never filled.

To solve this problem, we must radically rearrange our algorithm by a process called **loop splitting.** To keep things simple we will consider the problem of summing the elements of a vector x. We will also assume that the dimension n of x is 2^m. The idea is to divide the vector x into two vectors of dimension 2^{m-1}:

$$x = \begin{pmatrix} x' \\ x'' \end{pmatrix}.$$

We then calculate $y = x' + x''$. We now repeat the same procedure on y and continue until we are left with a scalar, which is the desired sum.

To see that this works, consider the case $m = 3$. We first compute

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} + \begin{pmatrix} x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix} = \begin{pmatrix} x_1 + x_5 \\ x_2 + x_6 \\ x_3 + x_7 \\ x_4 + x_8 \end{pmatrix}.$$

Next we compute

$$\begin{pmatrix} x_1 + x_5 \\ x_2 + x_6 \end{pmatrix} + \begin{pmatrix} x_3 + x_7 \\ x_4 + x_8 \end{pmatrix} = \begin{pmatrix} x_1 + x_5 + x_3 + x_7 \\ x_2 + x_6 + x_4 + x_8 \end{pmatrix}.$$

Finally we compute

$$\big(x_1 + x_5 + x_3 + x_7\big) + \big(x_2 + x_6 + x_4 + x_8\big) = x_1 + x_5 + x_3 + x_7 + x_2 + x_6 + x_4 + x_8,$$

which is the desired sum.

Here is the code implementing this process.

```
p = n;
while p~=1
   p = p/2;
   for i=1:p
      x(i) = x(i) + x(i+p);                    (3.9)
   end
end
sum = x(1);
```

The sums in the body of the inner loop are independent of one another, and therefore the inner loop can be unrolled and the additions pipelined. When n is not a power of two, there is a simple variant of (3.9) that does the job — see Exercise 5.

This algorithm has an operation count of approximately `n` additions, just as does the usual algorithm. However, its numerical properties are quite different — rounding error has less effect on the revised algorithm. Since compilers are not supposed to change the numerics of an algorithm without permission, loop splitting is not commonly used as a compiler optimization technique.

Finally, we note that the algorithm (3.9) access the elements of `x` sequentially. This, as we will see in §????, insures good cache performance.

3.5. Vector processors

Synopsis: A cure for the difficulty in pipelining floating-point operations is to work with collections of floating-point numbers as if they were vectors. The result is a vector computer, which has a number of vector registers consisting of, for example, sixty four scalar registers each. These vectors registers, can be loaded, stored, and combined by arithmetic and logical operations. Because, the operations on the components of the registers are independent, they can be pipelined, performed in parallel, or some combination of the two. To get more speed, some vector operations can be performed in parallel. Moreover, by a process called chaining, the results of one vector operation can be fed to another vector operation. The result is very efficient floating point computations.

When the actual vectors have more than 64 components, the computation can be broken up into subvectors with 64 components or less. Conditional execution allows operations to be performed on a subset of the components.

In some cases the components of a vector are not contiguous in memory. This can happen in two ways. First, they may be equally spaced in memory (the distance in memory between them is called their stride). In this case, and instruction to retrieve a vector requires only a base address and a stride. Second, they may be stored irregularly in memory. In this case the instruction requires a pointer to a vector of addresses of the desired components. Loading such a vector is called a gather operations; storing, a scatter.

—

3.5.1. Vector processors

We have seen in the last subsection that it is not easy to use a floating-point pipeline to implement operations on vectors. The problem is that is that MIPS and its relatives are scalar processors, and hence vector operations must be implemented as sequences of scalar operations. When these operations are performed in their natural order, they do not pipeline well.

A cure for this problem is to create a processor with instructions that manipulate vectors. This is feasible because the number of commonly used vector operations is small. It solves the pipelining problem because the hardware can be optimized to perform these operations efficiently. Computers that operate on vectors are called **vector processors** or **vector computers.**

Vector processors appeared in the early 1970's. The first ones were memory-memory architectures. Vectors in main memory were streamed into the processor and the results streamed back into memory. It was soon discovered that a load-store architecture using vector registers gave better results. Today's vector computers are all register oriented, and that is the kind we will treat in this subsection. The technical details of how vector operations are implemented on a vector processor are beyond the scope of this book, and we will restrict ourselves to describing the general features of vector processors.

A vector register consists of a number of scalar registers containing the components of the vectors, which may be single precision, double-precision, or integer. Depending on the machine the number of components in a vector register ranges from 64 to as many 4096. The number of vector registers ranges from 8 to 256.

The vector operations are performed by **vector function units.** The operations typically consist of vector addition, componentwise vector multiplication and division, and logical operations. Surprisingly, most vector processors do not have a dot product, and this lack can be a bottleneck in some applications (e.g., the method of conjugate gradients).

The componentwise additions that compose, say, a vector addition can be performed in parallel. But that would require as many scalar arithmetic units as there are components in a vector register, which is too costly. Instead, the arithmetic units exploit some combination of pipelining and parallelism. The fact that pipelining is involved means that vector operations have a latency or start up time. The fact that parallelism is involved means that once the pipeline is full, results can be generated at more than one component per clock cycle. But this gain in speed will be of little account if it is swallowed by the latency of the pipeline, which should be kept as small as possible.

In addition to arithmetic units, a vector processor have load-store units to communicate with memory. The success of a vector processor depends critically on how well they do the job. As we shall see later, many problems generate vectors that do not reside in a contiguous block of memory. Memory references that leap around in memory do not work well with cache memory, and consequently most vector processor communicate with a system of **memory banks**, which will be treated in §4.1.

A good vector processor also has to be a good scalar processor. In many applications scalar operations contribute a significant part of the computational overhead. By Amdahl's law speeding up the vector part without speeding up the scalar part will have diminishing impact on the overall speed of the vector computer.

3.5.2. AXPY on a vector processor

To see what AXPY looks like on a vector processor, we will use and extension of the MIPS computer devised by Hennessy an Paterson to illustrate vector computation. The VMIPS, as it is called, is a MIPS computer extended by eight vector registers `V0`,...,`V7` each holding 64 double words. The vector registers can hold integers and floating-point numbers. The registers are connected to functional units, which implement addition, multiplication, division, and integer and logical operation. They can run in parallel. A vector load-store unit communicates with main memory.

In implementing the AXPY we will assume that a is in `F0` and the starting addresses of `x` and `y` are in `R2` and `R3`. Here is the AXPY code

```
1.  LV        V1, 0(R2)    ; load x
2.  MULVS.D   V2, V1, F0   ; a*x
3.  LV        V3, 0(R3)    ; get y
4.  ADDV.D    V4, V2, V3   ; a*x + y
5.  SV        0(R3), V4    ; store y
```

The notation should be clear. `LV` and `SV` load and store vectors. `MULVS.D` multiplies a vector by a scalar, and `ADDV.D` adds two vectors.

It is not easy to predict how this loop will perform. To make a start let's assume that all the operations produce results at the rate of one component per cycle after a common startup time of λ cycles. There are five vector instructions. If they are executed strictly in sequence the total time is $5(\lambda + 64) = 5\lambda + 320$.

However, vector computers are aggressive about exploiting overlaps in computation. For example, since there are no data dependencies between the `MULVS.D` and the second `LV`, once the first `LV` has completed, we can run the `MULVS.D` and the second `LV` in

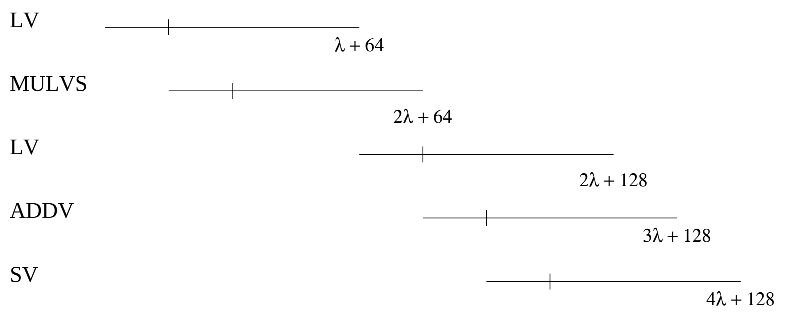


Figure 3.4: Chaining AXPY

parallel — provided the proper connections and control are available. In this case the time is reduced to $4(\lambda + 64) = 4\lambda + 256$.

Further reduction in time can be effected by directing the output of one instruction directly into another. This practice is called **chaining.** For example, the output from the first `LV` can be fed directly into the `MULVS.D`. The combined instructions finish at time $2\lambda + 64$.

Actually, the AXPY operation can be extensively chained, as is shown in Figure 3.4. The line following each instruction represents the period during which the instruction is executed. The tick in the line separates the startup period from the period of actual calculation. The formula at the end shows when the instruction completes. There are two chains in this calculation: the first `LV` $\rightarrow$ `MULVS.D` and the second `LV` $\rightarrow$ `ADDV.D` $\rightarrow$ `SV`. Note that the second chain cannot start until the first `LV` has finished, since VMIPS has only one load-store unit. The last instruction completes at $4\lambda + 128$. This is quite an improvement over our original $5\lambda + 320$.

This example shows the importance of keeping latency down, since 4λ can quickly approach 128. If, for example, $\lambda = 16$ then $4\lambda = 64$, which is one third the total computation time of $4\lambda + 128 = 192$.

3.5.3. Four problems

We now consider briefly four problems associated with vector processing and possible solutions.

1. What do you do when the vector size is less than 64?
2. What do you do when the vector size is greater than 64?
3. What do you do when an operation is to be performed on a subset of the components (aka conditional execution)?

4. What do you do when the components of a vector are not contiguous in memory?

3.5.4. Under and oversized vectors

The problem of undersized vectors is easily disposed of. The processor is provided with a programmable register that specifies the vector length, which, of course, must not be greater than the capacity of a vector register. If the value of this register is N, then only the first N components are processed by the arithmetic and load-store units.

Short vectors are processed faster than full vectors, but the latency of the arithmetic unit remains the same. As the vector becomes smaller a point will be reached in which it is faster to perform the operation in scalar mode. The vector size for which this happens is often written N_{v}.

When the vector size exceeds the register size, operations must be broken up into units that will fit into the vector registers. This must be done in software either by hand or by a vectorizing compiler. The following Matlab code illustrates the process for AXPY assuming 64 component registers.

```
m = rem(n, 64);
y(1:m) = y(1:m) + a*x(1:m)
for i = m+1:64:n
   j = i+64-1
   y(i:j) = y(i:j) + a*x(i:j)
end
```

Depending on the machine, it may be desirable to unroll the loop to take advantage of parallelism in the vector operations and load-stores.

3.5.5. Conditional execution

To illustrate the problem of conditional execution, consider the problem of computing the absolute value of the components of a vector x. This can be done as follows.

```
for i=1:64
   if x(i) < 0
      x(i) = -x(i);                                        (3.10)
   end
end
```

This calculation can be vectorized by introducing a vector mask register (VM) of 64 bits. For any vector operation if the ith bit of VM is zero, no result is stored for the ith component of the result; otherwise, the operation is performed and the result stored.

VMIPS has a VM and instructions to manipulate it. Let's see how they they can be used to implement (3.10). We will assume that scalar register `F0` contains zero and `F1` contains -1.

```
1.  LV       V1, x         ; load x
2.  SVLTS    V1, F0        ; VM(i) = 1 <==> x(i) < 0
3.  MULVS.D  V1, V1, F1    ; x(i) = -x(i) <==> VM(i) == 1
4.  CVM                    ; VM(1:64) = 1
5.  SV       x, V1         ; Store x
```

After loading `x`, the code uses the instruction SVLTS (Set Vector mask Less Than Scalar) to create a bit mask in VM that corresponds to the negative components of `x`. This is one of several vector compare instructions that set the VM. In the subsequent multiplication, the results are stored back only for those components which were originally negative. The `CVM` restores the VM to its default status of all ones. This is necessary so that the subsequent `SV` will store all the components of `V1`.

In a masked operation, the operation for an component with its VM bit zero may or may not be performed, depending on the machine. But in any event no result is stored. The execution of a masked operation for a component may be harmful if it causes and exception, such as overflow or divide by zero.

3.5.6. Vectors with a stride

The final problem concerns vectors that are not stored contiguously in memory. Because vector operations are performed in registers the real problem is how to move the vectors to and from memory. The solution depends on the regularity of the storage pattern.

In many cases, the components of a vector appear at equal intervals in memory. To see how this comes about consider the 5 by 4 C array defined by

```
double A[5][4]
```

The C language standard specifies that rectangular arrays be stored rowwise. Figure 3.5 shows how the array is stored beginning with address `a`. The reason for the increments of eight is that a double-precision number requires eight bytes to store.

If we pass through the array row by row, successive words are next to each other in memory, and we say that the references have a **stride** of one word. On the other hand, if we pass down a column successive words are four words from each other and we say that the references have a stride of four words. Some confusion can arise from not specifying the unit of the stride. For example, we could equally well say that the stride along a row is eight bytes, while down a column it is 32 bytes.

VMIPS has an instruction

```
LVWS V1, (R1,R2)
```

```
   a       a + 8    a + 16   a + 24
A[0][0]  A[0][1]  A[0][2]  A[0][3]

 a + 32   a + 40   a + 48   a + 56
A[1][0]  A[1][1]  A[1][2]  A[1][3]

 a + 64   a + 72   a + 80   a + 88
A[2][0]  A[2][1]  A[2][2]  A[2][3]

 a + 96  a + 104  a + 112  a + 120
A[3][0]  A[3][1]  A[3][2]  A[3][3]

a + 128  a + 136  a + 144  a + 152
A[4][0]  A[4][1]  A[4][2]  A[4][3]
```

Figure 3.5: Row-major storage of an array

that loads the vector register `V1` beginning with the address in `R1` and stride (in bytes) in `R2`. A corresponding instruction

```
SVWS (R1,R2), V1
```

stores a register with stride.

3.5.7. Scatter-gather operations

In some applications the components of a vector are stored irregularly. Their addresses can be specified by a vector of offsets from a base address. For example, suppose the maximum components of the columns of `A` in (3.5) occur at `A(2,1)`, `A(5,2)`, `A(3,3)`, `A(1,4)` and we want to form a vector of these components (as, for example, a preliminary to scaling the columns of `A` so that their largest component is one). Then the base address would be `a` and the offset vector would be

$$(32\ \ 136\ \ 80\ \ 24).$$

VMIPS has an instruction

```
LVI  V1, (R1+V2)
```

that loads `V1` with a vector whose base address is in `R1` and whose offset vector is in `V2`. In the parlance of vector computing, this operation is called a **scatter.** The corresponding store

```
SVI  (R1+V2), V1
```

is called a **gather.**

—

This completes our treatment of vector computers. They were once the crème de la crème of supercomputers, but today they are in partial eclipse. There are several reasons. They are expensive to build, owing to the large amount of hardware required for vector registers and units and the need for superfast memory to feed them. They are not always easy to use, since vectorizing compilers sometimes produce less than optimal results. Their performance is being challenged by much cheaper superscalar computers, at least in some applications. Finally, applications having coarse parallelism can be speeded up by spreading them out over networks of superscalar computers. On the other hand, cache-oriented, scalar computers cannot compete with vector computers in applications that require nonunit strides and scatter/gather operations. For this reason, if no other, we can expect vector computers to be around for some time to come.

Exercises

1. Among the things that affect the performance of a computer are the clock rate (CR), the number of cycles it requires to execute a typical instruction (CPI) that does not access memory, the average number of cycles required to satisfy a memory reference (CPM), and the fraction of instructions that make memory memory references (FM).

    1. Assuming that the number average number of cycles to complete a memory reference instruction is CPI + CPM and that CR is measured in hertz, develop a formula for the average number of instructions executed in a second.
    2. What does this suggest that you do as a programmer to speed things up.
    3. Give a critical assessment of this model.

2. The implementation of AXPY in Figure 3.2 uses two registers `R1` and `R2` to address `x(i)` and `y(i)`, both of which must be updated with each iteration of the loop. Show that if the distance between the addresses `x(1)` and `y(1)` is less than 2^{15}, we can get rid of `R2`.

3. The purpose of this problem is to compare load-store architecture with accumulator architecture in evaluating the cubic polynomial

$$p = ax^3 + bx^2 + cx + d = ((ax + b)x + c)x + d,$$

using the nested form on the right. The accumulator machine has four instructions (here a is an address in memory)

```
LDA  a  ;  loads the accumulator from a
STO  a  ;  stores the accumulator to a
MPY  a  ;  multiplies the acculator by the contents of a
ADD  a  ;  adds the contents of a to the accumulator.
```

Write accumulator and MIPS code to compute p. How many instructions are executed by each program? Assume that a memory reference executes in 2 cycles while an arithmetic operations executes in 1 cycle (so that for the accumulator machine `LDA` and `STO` run in 2 cycles while `MPY` and `ADD` require 3). Count the cycles for both programs. Comment on the results.

4. Show that moving a single instruction in the AXPY program in Figure 3.2 we can reduce the cycle count from 13 to 12.

5. The algorithm (3.9) for computing the sum of the components of a vector does not work when the dimension `n` of the vector is not a power of two. The problem can be cured by noting that a vector of even dimension can be divided into two equal parts, while a vector of odd dimension can be divided into two equal parts with one component left over, which can be added into an element of the equal vectors. Give Matlab code for

```
function sum = splitsum(x)
```

implementing this idea. [Hint: You will find the functions `bitshift` and `bitand` useful.]