Chapter 5
Basic Processor Design

*People who are really serious about software should make their own hardware.*

— A. Kay

**Abstract** In previous chapters we introduced the physical characteristics of digital circuits and the mechanisms used to design and implement them, followed by a functional description of how a computer processor executes programs. We can now combine this experience by examining how to design and implement a concrete processor. In this chapter we focus on the high-level design issues rather than the design of individual components. As such, we cover micro-architecture design (the design of the physical hardware) and instruction set design (the design of the interface to the hardware offered to the programmer). In other books, it is popular to introduce assembly language at this point as a means of programming the resulting system. We take a different approach, opting to describe instructions (and hence programs) using the pseudo-assembly language introduced previously.

### 5.1 A Concrete Stored Program Architecture

To investigate some of the issues involved in designing a concrete instruction set and implementing a concrete processor, we will step through the development of a realisation of MIPS32 [42–44]. Micro-Processor without Interlocked Pipeline Stages (MIPS) is a name that relates to an implementation technique called pipelining which we will discuss in Chapter 9. The first MIPS processor that used the associated techniques was developed in the early 1980s by a team led by John Hennessy at Stanford University. Their focus was improvement of performance using pipelining techniques but, recalling that an ISA is simply an interface not a design, the MIPS32 ISA does not dictate this as necessary. This is highlighted by the fact that the company formed to commercialise the Stanford design, also called MIPS, has produced a series of backwardly-compatible MIPS32 ISAs which are now used in implementations by companies such as NEC, Toshiba and Phillips; their products
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>addl</td>
<td>add long integer registers</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>addw</td>
<td>add short integer registers</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>addu</td>
<td>add without carry</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>and</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>bcc</td>
<td>branch if condition is not set</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>beq</td>
<td>branch if equal</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>ble</td>
<td>branch if less or equal</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>bne</td>
<td>branch if not equal</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>blt</td>
<td>branch if less</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>bge</td>
<td>branch if greater or equal</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>bltz</td>
<td>branch if less than zero</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>breq</td>
<td>branch if reference equal</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>bgtz</td>
<td>branch if greater than zero</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>j</td>
<td>jump</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>jal</td>
<td>jump and link</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>jalr</td>
<td>jump and link register</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>jr</td>
<td>jump register</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>lw</td>
<td>load word from memory</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>mfhi</td>
<td>move from high integer register</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>mflo</td>
<td>move from low integer register</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>mult</td>
<td>multiply integers</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>mfhi</td>
<td>move from high integer register</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>mflo</td>
<td>move from low integer register</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>mult</td>
<td>multiply integers</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>mfhi</td>
<td>move from high integer register</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>mflo</td>
<td>move from low integer register</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>mult</td>
<td>multiply integers</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>mfhi</td>
<td>move from high integer register</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>mflo</td>
<td>move from low integer register</td>
</tr>
<tr>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>mult</td>
<td>multiply integers</td>
</tr>
</tbody>
</table>

Table 5.1: A table description of a reduced MIPS ISA.
drive consumer electronics such as the PlayStation2 and workstations such as the SGI Indy.

We selected MIPS32 as an example for several reasons. The design is fairly simple and very typical of modern processors; this is highlighted by the level of real-world use of the ISA. The documentation for the ISA is easily available and there are a number of good compilers, simulators and so on that can be used to explore the design without physical hardware. Unlike more complete references [28], we focus on a reduced version of the MIPS32 ISA to make it more easily accessible; this is shown in Table 5.1. For example, we omit all floating point operations. Where appropriate we are careful to differentiate between the full MIPS32 ISA and this reduced version.

5.1.1 Major Data-path Components

We often call the details of a processor implementation the **micro-architecture** to contrast with the **instruction set architecture** which represents an interface with the programmer potentially shared between many different implementations. The MIPS32 ISA dictates, or at least strongly hints toward a 32-bit micro-architecture: the data-path width is 32 bits in the sense that the accumulators hold 32-bit values, the buses usually transfer 32-bit values and the ALU can usually deal with 32-bit
inputs. In some sense, 32-bit is the natural size of things within the processor. To highlight where this size has an impact we say the data-path is \( n \) bits wide: keep in mind that in this case \( n = 32 \). Figure 5.1 shows a block diagram of a data-path which we use for our implementation. The components in, and structure of, the diagram closely follows the description of the more general stored program architecture. Sectioned to relate the concrete data-path to the abstract stored program architecture, the key components are:

**Accumulators** The accumulators within a stored program architecture are defined in a somewhat abstract way. Our data-path includes a set of concrete accumulators, normally called registers in a modern context, which are for general-purpose use. There are thirty two such registers grouped into a structure called a register file; we write \( GPR[i] \) to refer to the \( i \)-th general-purpose register, each value in \( GPR[i] \) is \( n \) bits in size. The register file is like a small, high-speed, short-term memory whose content we can access very quickly making it ideal to supply values to the ALU for computation and storing results of said computation before they are written into more long-term storage such as the main memory. The registers are general-purpose in the sense that their content and usage are freely determined by the programmer. Usually the general purpose registers are used to contain integer values; there may be a second register file to store floating point values.

The MIPS32 ISA demands that \( GPR[0] \) is the constant value zero; retrieving the value of \( GPR[0] \) gives zero, any stores to \( GPR[0] \) are discarded. This might appear a strange choice: why would we waste a register to store a seemingly useless value and nothing else? The answer is that by having the constant zero always available we can construct some operations within the ISA from others using no additional logic. For example consider the \texttt{nop} or NOP instruction which performs no operation. We can implement such an instruction with more or less any instruction which targets \( GPR[0] \) and has no other side effects. For example, using the instruction \texttt{sllv} to shift \( GPR[0] \) left by a distance of zero and store the result in \( GPR[0] \) will have no effect on the state of the processor: although we need some logic to implement the \texttt{sllv} instruction, we get the \texttt{nop} instruction for free.

As well as the general-purpose registers, the data-path includes a number of special-purpose registers whose usage is pre-determined by the design of the processor; these might not even be accessible to the programmer. The special-purpose register set includes a Program Counter (PC) that holds the address of the next instruction the processor will execute; an Instruction Register (IR) which holds the instruction currently being executed; the Memory Address Register (MAR) which holds addresses that are passed to the memory when a value is stored or retrieved; the Memory Buffer Register (MBR) which holds the actual values being stored or retrieved; and the HI and LO registers that act as temporary storage for specific instructions such as \texttt{mult}.

The issues of designing and constructing registers and register files are examined in more detail in Chapter 8.
5.1 A Concrete Stored Program Architecture

**Arithmetic and Logic Unit (ALU)** The ALU is shown as a dashed wedge shape. We will discuss implementation of devices that perform arithmetic and logic in Chapter 7; for now it suffices to understand that the ALU simply performs computation for us. More specifically, where we might write a Verilog statement to perform an addition we would instead have the ALU produce a result by passing it operands and using an opcode to select the addition operation.

**Control Unit** The control unit is shown alongside the data-path in a dashed box; it has implicit links into all the components so that it can control how they behave. Inside the control unit are two implicit structures whose implementation we leave as fairly abstract. Firstly, the control unit has a constant value register which it uses to send constant values onto either the A or B buses; in reality this might not be a physical register. Secondly, the control unit has a status register which details the state of the processor. Various conditions that are flagged by bits in a typical status register include arithmetic exceptions, for example division by zero, and the processor mode (either user or kernel), an issue which is discussed later in Chapter 12. A program can query the status register by extracting and examining bits and hence make decisions on how to proceed based on special events which might have occurred during execution.

**Buses** There are two main groups of buses within the data-path which are used to communicate values between the different components: the ALU buses allow communication of operands and results to and from the ALU, the memory bus allows communication of addresses and values to and from the memory. There is one slight subtlety in the design of our buses in that where a value can be communicated from two different places we need a multiplexer to select which one; control signals for the multiplexer are managed by the control unit. Examples include the input to the MBR which can take values from either the C bus or the memory.

**Input and Output Devices** In this basic model we largely ignore input and output devices and assume they interact with the processor. Simply put, this involves associating regions of memory with a hardware device rather than a memory location. So for example if one were to write into address 123 this might actually mean “communicate a value to the video device” rather than “store a value in memory”.

**Memory** The main memory is shown alongside the data-path as a dashed box since it is not actually part of the processor; memory is commonly placed on a different physical microchip to the processor. From here on we will consider a memory in which each location holds 8-bit values. Each location will have an address so this is called a byte addressed memory: we are able to access each byte individually. However, in order to remove the need for multiple accesses to retrieve or store a 32-bit value for example, the basic unit of transfer between the processor and memory typically matches the data-path width $n$. This does not change the fact that the memory is byte addressed however. We still view the memory fairly abstractly here, filling in the details of implementation in Chapter 8. For now it suffices to understand that the memory can be represented, if not actually implemented, as a 2-dimensional Verilog register.
The crucial thing to realise from this description is that the data-path contains only hardware devices we already know how to build: we know how to implement registers in Verilog; we know how to implement simple arithmetic circuits in Verilog; we know how to design and implement state machines in Verilog. Although we will use later chapters to go into more detail as to how these devices are actually realised, there should be nothing intimidating about the block diagram in Figure 5.1: it is just a big circuit.

### 5.1.2 Describing Instruction Behaviour

In other books, it is popular to introduce assembly language early on as a means of programming the processor under discussion. We break with this trend and take a slightly different approach. That is, we describe instructions (and hence programs) using a pseudo-assembly language that relates directly to the effect of each instruction on the processor. The aim is to show that machine code is the natural language at this level: only when we start to discuss the interface between hardware and software in Chapter 10 and Chapter 11 is assembly language introduced.

The general-purpose registers are viewed abstractly as an array such that use of an index identifies a specific register, $GPR[7]$ is the general-purpose register number seven for example. Any special-purpose registers are referred to by their name, for example $PC$ is the program counter. Memory is also viewed abstractly as an array. Recalling Table 5.1, an instruction with the meaning $GPR[2] ← GPR[3] + GPR[4]$ shows the use of operands read from general-purpose register three and four in performing an ALU operation (in this case an addition), and writing a result back into general-purpose register two. Roughly speaking, things on the right-hand side of the assignment symbol $←$ should be interpreted as being read (or as source operands) and those on the left-hand side as being written (or as target operands) by the instruction. This also implies that memory access can be written in the same way; for example the operation $MEM[64] ← GPR[2]$ reads general-purpose register two and writes the value into memory address sixty four.

Where an instruction is only executed if some condition is true, we adopt a pseudo-code style and write the condition before the instruction. For example, in writing the instruction

$$\text{if } GPR[2] = 0, PC ← PC + 12$$
Figure 5.2 Execution of an addition instruction on the example data-path.

we mean that the \( PC \) is only incremented by twelve if the related condition is satisfied. That is, if the contents of general-purpose register two is equal to zero, then the current \( PC \) value is incremented, otherwise no action is performed.
5.1.3 The Fetch-Decode-Execute Cycle

To see how the data-path is typically used to execute instructions, consider the example of \( GPR[2] \leftarrow GPR[3] + GPR[4] \). Figure 5.2 details the familiar steps from Chapter 4 in a graphical way; they are also described below:

**Fetch** The PC is fed through the ALU unaltered and stored in the MAR since there is no direct connection between PC and MAR. Once stored, the control unit initiates a load from memory; this results in the instruction at the address in PC being loaded into the MBR. The instruction is again fed through the ALU and stored in the instruction register IR. Finally, the PC is incremented by four so it points at the next instruction to be executed: the PC is fed onto the A bus while the constant 4 is fed onto the B bus, the ALU is invoked to perform an addition with the result on the C bus stored back in the PC. Note that we increment the PC by four because each instruction is 32 bits in size and our PC counts in bytes: a 32-bit instruction is four 8-bit bytes.

**Decode** The control unit now examines the instruction in IR and determines the opcode and operands: in this case the opcode tells us we should do an addition and the operands tell us to source values from GPR[3] GPR[4] and produce a result in GPR[2].

**Execute** Execution proceeds by placing GPR[3] onto bus A and GPR[4] onto bus B, then provoking the ALU to add the two inputs together; the result is produced on bus C.

**Write** Finally, the write phase takes the value on bus C and stores it into GPR[2]. Execution of this instruction is now complete and the processor loops back to the fetch stage to retrieve and execute the next instruction.

The actions within the execute and write stages will change depending on the instruction type. For example, if we are required to perform a load from memory, then it is possible the ALU will not be needed whereas if we are performing an arithmetic operation it will. The actions within the fetch and decode stages are the same for every instruction. This should make sense since until the instruction decode has taken place, the control unit does not even know what the instruction means.

5.1.4 Controlling the Data-path

Construction of the control unit, which constitutes the control-path of the processor, is crucial: if the control unit cannot efficiently manage the resources in the data-path to implement the fetch-decode-execute cycle, the speed of the entire processor is compromised even if those resources are individually capable of being more efficient. There are two main approaches to designing the physical logic within the control unit; these are commonly termed hard-coded and micro-code designs.

In a hard-coded design one uses techniques outlined in Chapter 3 to directly implement a state machine capable of operating the data-path components in the right
way. This technique is easy to implement, although the state machine can become complicated, and is efficient in the sense that the amount of logic used can be small since the state machine is tailored closely to what is required. However, once the state machine is implemented in logic it is fixed forever: altering it means building a new processor.

A micro-coded design approaches the same problem in a different way. It essentially allows the data-path to be programmed using very simple instructions called micro-instructions; an instruction in the ISA is therefore implemented by a number of micro-instructions. One must contrast the level at which these two types of instruction work: an instruction in the ISA operates at a fairly high-level, the operands involved are typically values from the register set; an instruction in the micro-code operates at a low-level, typical operands are control signals in the data-path. A micro-coded design is more complex to implement but offers some significant advantages in terms of flexibility. For example, one can fix problems with the processor by upgrading the micro-code rather than re-engineering the actual hardware. This exact issue reared its head in 1994 when the Intel Pentium range of processors was found to have a bug in the division instruction; when using certain inputs the instruction would produce the wrong result. Today, Intel processors allow upgradeable micro-code which could potentially have solved this problem without the associated cost of product recall: in theory they might have simply issued software that would have updated the micro-code and fixed the flaw.

5.2 Buses

As previously stated, a bus is a communication link between devices; the most basic example is a wire which links the devices. We are usually careful to categorise a bus by the type of information it is used to communicate. The bus which communicates addresses from the processor to the memory is called an address bus; the bus which communicates data values to and from the processor and memory is called the data bus. The buses which communicate operands and results to and from the ALU is also a data bus. In each case, the bus width is the size of information that can be communicated along the bus: a single wire can communicate 1-bit values, a 32-bit wire vector can communicate a 32-bit value.

Although there are many ways to construct a communication channel between two (or more) devices, we need to at least make the distinction between two major classes.

5.2.1 Synchronous Buses

When deploying a synchronous bus there must be a shared, unskewed clock accessible to the two devices. The devices use this clock to schedule their operations;
each operation involving signals on the bus will always take a known, integer number of **bus cycles** to complete. This fact ensures that one device can operate with a guarantee that the other will remain in step with it.

Imagine a contrived example of two devices $P$ and $Q$ communicating values between themselves: $P$ sends a value $x$ to $Q$, $Q$ computes $y = x^2$ and sends $y$ back to $P$, finally $P$ checks that $y$ is the same as $x \cdot x$ and sends 1 or 0 back to $Q$ in the case that the test is true or false respectively. Using a synchronous bus, all these operations take say $b$ bus cycles so we can describe our communication as follows:

<table>
<thead>
<tr>
<th>Time</th>
<th>$P$</th>
<th>$Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b</td>
<td>$x \rightarrow$</td>
<td></td>
</tr>
<tr>
<td>1b</td>
<td></td>
<td>$y \leftarrow x^2$</td>
</tr>
<tr>
<td>2b</td>
<td>$y \equiv x \cdot x \rightarrow$</td>
<td></td>
</tr>
</tbody>
</table>

Note that $Q$ knows that the result $y$ will be communicated at time $2b$ because the two devices are synchronised by the shared clock, so it can read this value from the bus without further control. Also note that we are using a single shared bus to do communication in both directions. The time slices allocated to each device via the clock allow this to occur without conflict.

Examples of synchronous buses within our data-path include the $A$, $B$ and $C$ buses used to communicate with the ALU. These are synchronous because their operation is governed by the system clock used to drive the processor. For example, Figure 5.2 shows the $A$ bus being used to communicate both values read from the general-purpose register file and constant values from the control unit. These do not interfere with each other because the system clock determines the ordering of the fetch, decode and execute stages so they cannot overlap.

### 5.2.2 Asynchronous Buses

Devices connected by an asynchronous bus do not share a common clock and cannot therefore use it for synchronisation of their operations. Using the above example again but with an asynchronous bus, $Q$ would not be able to make the assumption that at time $2b$ the value $y$ would be communicated to it. That is, the value might be placed on the bus at time $2b + \varepsilon$ for some small $\varepsilon$ in which case $Q$ could read the wrong value. To combat this problem, extra control signals are added to the bus that allow each device to tell the other when it is ready to communicate. This introduces a **bus protocol** which must be adhered to.

As a simple example, when $P$ wants to communicate $x$ to $Q$ it first raises a control signal $c_1$ to show it is ready to send the data. $Q$ will have been waiting for this event to occur: as soon as it sees that $c_1 = 1$ it raises a control signal $c_2$ in the opposite direction to show it is read to receive the data. When $P$ notices the change in $c_2$, it knows the communication can proceed. Thus, the communication of $x$ is performed using a sequence such as:
5.3 Addressing Modes

Considering some collection of objects, addressing refers to identifying a particular member of the collection using a label called an address. As a concrete example one can imaging the 4-tuple of integers

\[ A = (6, 7, 8, 9). \]

Denoting the \( i \)-th entry of \( A \) as \( A_i \), in order to access an element of the tuple we need to provide an index, or address, i.e., an \( i \). For example, to access element number two we use \( A_2 \) where 2 is the address and 8 is the resulting value. An addressing mode encompasses the mechanism for taking an address of some sort and using it to retrieve an element from some collection. In the context of processor design, this means providing an address so that we can retrieve an operand ready to perform computation with. There are three major classes of operands: constant or immediate values, general and special-purpose register content, and memory content.

The MIPS32 ISA implies a load-store architecture. This means that instructions either access register content or memory content but not a mix of both: the \( lw \) and \( sw \) instructions are the only ones that access memory. This makes life easier since within a given instruction we limit the types of operand and hence addressing modes used. Not all ISAs are the same: traditionally Intel Pentium processor designs have included instructions that, for example, add the content of a memory location to the content of a register.
5.3.1 Immediate Addressing

An immediate operand is simply a constant value embedded into the instruction that is used in computation. For example, consider executing the addition operation $GPR[2] ← GPR[3] + 1$. The value 1 is implicit in the instruction: we do not need to fetch the value from a register or from memory, it is immediately available once the instruction is decoded. As such, calling this an addressing mode might seem somewhat overkill; we do not really provide an “address” as such since the value is available without retrieving it from somewhere.

5.3.2 Register Addressing

Addressing the general-purpose registers is a simple task: there are $r$ registers in total so to specify which one we mean, we simply supply an address in the range $0, 1, \ldots, r - 1$ which requires $\lceil \log_2(r) \rceil$ bits to represent it as an unsigned integer. This is commonly termed register direct addressing. There is a slight subtlety when it comes to accessing special-purpose registers however: we cannot simply provide an address or name, their access must be implicit in the instruction itself. As an example, the MIPS32 ISA has instructions named `mthi` and `mfhi` that move values to and from the special-purpose register `$HI$` that is used to store one half of a multiplication result. One cannot execute an instruction with the meaning $GPR[2] ← GPR[3] + HI$ since the register $HI$ is not generally addressable. One must first execute an `mfhi` instruction to perform something like $GPR[4] ← HI$ and then execute $GPR[2] ← GPR[3] + GPR[4]$.

This represents a trade-off in the design of the ISA. If one defines the special-purpose registers as disjoint from the general-purpose registers, then more special instructions are needed to access them. If one defines them in the same set as the general-purpose registers, then our register addresses need to come from a larger range (since they need to address more registers) or the number of general-purpose registers decreases. However, the second option means that the standard instruction set can access the special-purpose registers as easily as the general-purpose registers. The trade-off is therefore roughly one of orthogonality or regularity versus the constraints of usefulness.

5.3.3 Memory Addressing

Previously, we viewed memory as an array called $MEM$ which was accessed by providing an index or address $x$. The retrieval of a value at address $x$ was denoted $MEM[x]$ while $MEM[x] ← y$ represented a store of value $y$ into the memory location $x$. Here we are only interested in the address $x$ and so consider both forms of access at once. The key thing to realise is that there are different methods, or address-
Addressing modes, by which one can construct the address \( x \) which we call the **effective address** and is passed to memory.

It is easy to start introducing more and more addressing modes to make life easier for the programmer. However, doing so is a balancing act. More addressing modes typically means a more complex processor and a requirement for the programmer or compiler to understand and use them effectively. Less addressing modes means the program must be longer since it will need to implement the behaviour of missing modes using instructions that are provided. Different processors make different trade-offs and one might encounter a huge range of different addressing modes as a result:

**Direct Addressing**  Consider the most basic form of addressing, **direct addressing**. In this case, the effective address passed to memory is the same address as the one we want. That is, we want to access \( \text{MEM}[x] \) so just feed our address \( x \) as the effective address. Very roughly, we can describe the process as a diagram:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 20 )</td>
<td>28</td>
</tr>
</tbody>
</table>

This is a simple addressing mode to use, and requires no addition to the standard way we interface with memory. However, the trade-off for this simplicity is that all the work in constructing the effective address \( x \) needs to be done by the programmer.

**Indirect Addressing**  One way to extend direct addressing is to allow an indirect addressing mode instead. The memory treats an effective address as the address of the address of the element we want. You can think of this as the same way a pointer works in a C program: if the pointer is stored somewhere in memory, then to load the data it points to we need to first load the pointer and then load the data it points to. Again, we can describe this process diagrammatically as:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 20 )</td>
<td>28</td>
</tr>
<tr>
<td>16</td>
<td>28</td>
</tr>
<tr>
<td>20</td>
<td>28</td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Rightarrow 60 )</td>
<td>60</td>
</tr>
<tr>
<td>24</td>
<td>60</td>
</tr>
<tr>
<td>28</td>
<td>60</td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>

Although some processors might perform both memory accesses as the result of one instruction, and hence decrease the number of instructions the programmer needs to provide, the clear drawback to indirect addressing is the sequential nature of the two memory accesses: this can have a significant performance impact.
Indexed Addressing Perhaps a more attractive extension of direct addressing is **indexed addressing**. In this case, the effective address passed to memory is the sum of a base address and an offset. Again described diagrammatically, a memory access is performed as follows:

\[
\begin{align*}
\text{base} &= 16 \\
\text{offset} &= 4
\end{align*}
\]

\[x = 16 + 4 = 20 \rightarrow 28\]

To see why this might be a useful thing to offer, consider a typical array access in a C program; for example a reference to the tenth element of an array of bytes called \(A\) written as \(A[10]\). The element of \(A\) we want is ten locations on from wherever \(A\) is located and we know \(A\) is located at some fixed address. So to perform an access to \(A[10]\) we simply use an indexed addressing mode and set the base address to the address of \(A\), say \&\(A\), and the offset to ten. As a result we access \(MEM[\&A + 10]\) which is exactly what we wanted. The MIPS32 ISA uses this form of addressing mode in the \(lw\) and \(sw\) instructions: the base is allowed to be any register value while the offset is an immediate, their sum is passed to the memory as the effective address.

There is a trade-off however. We have introduced a new addressing mode that is doing some work for us by adding the base address and offset together. This is more complex than direct addressing for example, and probably requires invocation of the ALU to perform addition. On the other hand, it means programs can be written using less instructions: if we had no indexed addressing more then we would need to add the base address and offset ourselves using an extra addition instruction.

Scaled Addressing Consider what happens to the example access to byte array \(A\), discussed in the context of indexed addressing, if we change the array so it holds 32-bit integers rather than bytes. The element \(A[10]\) is no longer ten locations on from the address of \(A\) but forty locations on: each element of \(A\) requires four locations in memory. Accessing \(A[10]\) using the indexed mode is now a little tricky, essentially we need to scale the offset by a factor of four to get the right address. Clearly this could be done by adding another instruction to the program, but we could also introduce a scaled addressing mode by adding a scaling factor which is applied using an inexpensive shift:
\[
\begin{align*}
\text{base} &= 16 \\
\text{offset} &= 4 \\
\text{scale} &= 2
\end{align*}
\quad \begin{array}{c|c}
\text{Address} & \text{Value} \\
\hline
& \\
28 & \vdots \\
32 & 99 \\
36 & \vdots \\
& \\
\end{array}
\quad \rightarrow 99
\]

In the context of our example we can now still access \( A[10] \) using one instruction by setting the scaling factor to 2 so as to multiply the offset by 4 as above. As a result we access \( MEM[&A + 10 \cdot 4] = MEM[&A + 40] \) as required.

### 5.4 Instruction Encoding

We know how to specify an opcode which tells the control unit which type of instruction to execute: this is just a number. We also know how to specify the operands of an instruction in terms of where their value can be retrieved from using some addressing mode: these are just numbers as well. Now we need to consider how these items of information are encoded, or formatted, so the control unit can understand what we mean. One can view this as an extension of the simple method from Chapter 4.

#### 5.4.1 Instruction Selection

A major task in designing an ISA is selecting which instructions it will include. This will have a massive impact on many other aspects including any actual implementation. For example, one clearly needs to include hardware that matches the requirements of instructions within the ISA. There are few concrete rules for deciding which instructions should be included; for a special-purpose ISA it might suffice to exclude instructions which are extraneous for that domain but would be included in a general-purpose ISA.

However, one can quite easily start to look at the types of instructions in terms of the number of operands they use and the capabilities that result. Examining the MIPS32 ISA, it is clear that there are instructions with zero, one, two and three operands. For example, the \texttt{nop} instruction has zero operands, the \texttt{mfhi} instruction has one operand (a destination register), the \texttt{bltz} has two operands (a source register and an immediate value), the \texttt{add} instruction has three operands (two source registers and one destination register). Noting this fact seems like a triviality but in fact can have a major impact on the design of the rest of the processor and on how it is programmed.
We already have the instructions `mtlo` and `mthi` which have the meaning \( LO \leftarrow GPR[x] \) and \( HI \leftarrow GPR[x] \) for register operand \( x \), and `mflo` and `mfhi` which have the meaning \( GPR[x] \leftarrow LO \) and \( GPR[x] \leftarrow HI \) again for some register operand \( x \). To see the impact on having instructions with different numbers of operands, again consider having a requirement to execute the addition operation \( GPR[2] \leftarrow GPR[3] + GPR[4] \). A processor that implements a 0-operand ISA is often called a stack machine; everything has to be implicit in the instruction. So to add two values together, we might invent a fictitious instruction that has the meaning \( LO \leftarrow LO + HI \). Note that we do not specify the registers \( LO \) and \( HI \) in the instruction, you should think of them as being hard-coded in some sense: the addition operation cannot add values taken from any other registers. To use the instruction one would therefore need to execute the sequence

\[
\begin{align*}
LO & \leftarrow GPR[3] \\
HI & \leftarrow GPR[4] \\
LO & \leftarrow LO + HI \\
GPR[2] & \leftarrow LO
\end{align*}
\]

We execute four instructions in total. With one operand, things become a bit easier. Consider a processor that implements a 1-operand ISA which is often called an accumulator machine. To model such a processor, we introduce another fictitious instruction that has the meaning \( LO \leftarrow LO + GPR[y] \) where \( y \) is the one operand we specify in the instructions, \( LO \) is again hard-coded in some sense. Now we can perform the required operation with only three instructions:

\[
\begin{align*}
LO & \leftarrow GPR[3] \\
LO & \leftarrow LO + GPR[4] \\
GPR[2] & \leftarrow LO
\end{align*}
\]

As one might have expected, with two operands things are even easier still. A processor that implements a 2-operand ISA is often called a register machine. Again we invent a fictitious instruction to model the ISA; this time the instruction means \( LO \leftarrow GPR[x] + GPR[y] \) where \( x \) and \( y \) are our two specified operands. The required operation can now be executed in two instructions:

\[
\begin{align*}
GPR[2] & \leftarrow LO
\end{align*}
\]

Finally, with a 3-operand ISA (implementations of which are usually still classed as register machines) we can perform the required operation with just one instruction:

\[
\]

Notice the trade-off which occurs. The more operands in our instructions the less instructions are typically needed to execute an operation; however, we need to specify all these operands so our instructions may be bigger. The less operands we have,
the more information is provided implicitly in the instruction rather than explicitly as operands. Thus the instructions can be smaller.

Just as the number of operands has an impact on things, so does the type of those operands. In the above example we are only using register operands whereas in the actual ISA we have immediate operands and register operands. By collating the different instruction types, we can classify the instructions in the reduced MIPS32 ISA as follows:

- 3 instructions with 0 register operands + 0 immediate operands
- 5 instructions with 1 register operand + 0 immediate operands
- 1 instruction with 2 register operands + 0 immediate operands
- 9 instructions with 3 register operands + 0 immediate operands
- 1 instruction with 0 register operands + 1 immediate operand of 16 bits
- 2 instruction with 0 register operands + 1 immediate operand of 26 bits
- 4 instructions with 1 register operand + 1 immediate operand of 16 bits
- 5 instructions with 2 register operands + 1 immediate operand of 16 bits

Clearly we can reduce these seven instruction classes to even less; if we have a class of three register operand instructions, this can also accommodate those with two, one and zero register operands simply by ignoring those operands which are not needed.
5.4.2 Instruction Formats

An instruction format is basically a template which can be used to encode a class of instructions. Within the template the concrete opcode and operands are placed into fields to form an encoded instruction. The MIPS32 ISA uses three different instruction formats, called the R-type, I-type and J-type formats, and which are detailed in Figure 5.3 and Listing 5.1. These formats correspond to the actual encodings in Table 5.1 in the sense that it shows which fields of a format should be filled with which information to encode a given instruction.

5.4.3 Basic Encoding and Decoding

Consider encoding an add instruction to represent our running example which performs $GPR[2] \leftarrow GPR[3] + GPR[4]$. The add instruction uses an R-type format; to construct the instruction we first fill the opcode and funct fields with the 6-bit constants $000000_2$ and $100000_2$ respectively. These fields combine to tell the control unit that this is an add instruction. Then, we fill the register operand fields rs, rt and rd with the constants $00011_2$, $00100_2$ and $00010_2$ respectively and zero the remaining blank field. The end result is the 32-bit number
that represents our instruction. Filling in these fields is somewhat tricky so one would normally utilise some form of function to make things easier; Listing 5.2 details a simple C function which will return an encoded R-type instruction given a list of opcode and operand values. It first masks each value so that it is the right size and then shifts them into the right place; the parts are ORed together to form the end result.

Decoding an instruction is more or less the exact reverse process to encoding it: we simply extract information from the right place. However, the three MIPS32 instruction formats make the decoding process much easier than it might otherwise be. In particular, they provide a number of advantages typical of many instruction encoding strategies:

1. All instructions are of a fixed length; this means we never need to perform more than one memory access to retrieve them. The control unit can be simple since, as a result, it does not need to decide when and if more parts of an instruction are required.

2. Common fields between the formats are aligned in the same place, for example the opcode is always the most-significant six bits. This allows for more efficient decoding since the control unit never needs to conditionally shift around the fields

```c
uint32 encode_Rtype( int opcode, int rs, int rt, int rd, int funct )
{
    return ( ( opcode & 0x3F ) << 26 ) |
            ( ( rs & 0x1F ) << 21 )|
            ( ( rt & 0x1F ) << 16 )|  
            ( ( rd & 0x1F ) << 11 ) |
            ( ( funct & 0x3F ) << 0 ) ;
}
```

Listing 5.2 A simple C function to encode a MIPS32 R-type instruction.

```verilog
task decode_Rtype;
    input [31:0] instruction;
    output [ 5:0] opcode;
    output [ 4:0] rs;
    output [ 4:0] rt;
    output [ 4:0] rd;
    output [ 5:0] funct;
    begin
        opcode = instruction[31:26];
        rs = instruction[25:21];
        rt = instruction[20:16];
        rd = instruction[15:11];
        funct = instruction[ 5: 0];
    end
endtask
```

Listing 5.3 A simple Verilog task to decode a MIPS32 R-type instruction.
function ext;
  input [31:0] instruction;
begin
  extend_16bit = instruction[15] ? (16’hFFFF, instruction[15:0]) :
                        (16’h0000, instruction[15:0]);
end
endfunction

Listing 5.4 A simple Verilog task to decode a 16-bit sign-extended immediate value.

to get the right information: it just extracts the bits required from a known location.

As such, it is easy to write Verilog tasks that could be used within the control unit to
decode instructions. Listing 5.3 details an example which could decode our previously encoded R-type `add` instruction. Since Verilog offers a convenient mechanism
to describe the extraction of bits, the task amounts only to a few assignments.

However, things are somewhat complicated when we consider immediate val-
ues in the I-type and J-type formats: remember such immediates might represent
positive or negative numbers. Consider, for example, how to decode the 16-bit im-
mediate value in the `addi` instruction; naively one might simply extract the bottom
16 bits of the instruction. The problem comes when this value is used as a 32-bit
operand. The value $-1_{(10)}$ would be encoded as the 16-bit twos-complement con-
stant

$$FFFF_{(16)}.$$  

When naively padded with zeros to form a 32-bit value ready for performing com-
putation, we get

$$0000FFFF_{(16)}$$

which is no longer $-1_{(10)}$ but $65535_{(10)}$. The problem is that we need to pad with the
most-significant bit, or sign bit since this dictates the sign of the number, not zeros.
Application of this process is called **sign extension** and produces the constant

$$FFFFFFFF_{(16)}$$

which is still $-1_{(10)}$ as a 32-bit value. The sign extension process is also easy to
describe in Verilog; Listing 5.4 gives an example. The MIPS32 ISA description in
Table 5.1 denotes the application of the sign extension process to an immediate
operand $x$ as $ext(x)$. It is important to realise that not all immediate values need be
sign extended. For example, shift distances are *always* positive so may not require
sign extension but rather just naive extraction.
### 5.4 Instruction Encoding

<table>
<thead>
<tr>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
<th>00000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>111111</td>
<td>Rs</td>
<td>Rt</td>
<td>Rd</td>
<td>00000</td>
</tr>
<tr>
<td>opcode</td>
<td>Rs</td>
<td>Rt</td>
<td>Rd</td>
<td>shamt</td>
</tr>
<tr>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

**Figure 5.4** A fictitious example of variable length encoded instructions: a 1-word `add` instruction (with three register operands) is followed by a 2-word instruction (with six register operands), and finally by a 1-word `nop` instruction (with no operands).

### 5.4.4 More Complicated Encoding Issues

The MIPS32 encoding mechanism is typical of 32-bit processors in that the instruction size is large enough to comfortably accommodate all the required operands in a few “friendly” formats. Different designs require different approaches however; an instruction encoding mechanism for 8-bit instructions cannot use the same approach for example, since space is so limited. Although there are a huge range of options, we present a few techniques below that one might use. All of the presented techniques imply a trade-off in terms of their use: we might save space or pack more information into the same space but will inevitably complicate the control unit as a result. This is because the decoding step will become harder and less regular since special cases will need to be checked for and accommodated.

#### 5.4.4.1 Fixed and Variable Length Encodings

Roughly speaking, **code density** relates to the amount of memory taken up by the instructions for a program; higher code density, i.e., number of instructions or amount of computation carried out per unit space, means the program uses less space. We have already encountered a design decision that can impact on code density when we discussed different instruction types: the more operands an instruction has, the larger it is but typically the less instructions are actually needed to perform an operation. In an attempt to get the best of both worlds from this trade-off, some designs use what is called a **variable length encoding**. Previously, we looked at the MIPS32 encoding mechanism where every instruction was the same length. A variable length encoding breaks this constraint by allowing instructions that require many operands to be longer than those that require few operands. As a result, one can achieve high code density while still having many operands.

One way to achieve this is to reserve one value of the **opcode** field as some form of “escape code” which specifies the need to fetch more instruction from memory. Figure 5.4 attempts to illustrate this in the context of the MIPS32 instruction formats already presented. There are three instructions arranged in memory starting at some address. The first is a standard 1-word `add` instruction which can specify three
register addresses as operands. The second is some fictitious 2-word instruction; the first part uses the value $11111_{(2)}$ as the opcode to show that we need to fetch another word from memory to complete the instruction. The second part specifies the actual opcode with both parts containing three register address operands (a total of six operands). Finally, the third instruction is a standard 1-word \texttt{nop}.

In this contrived example one can clearly see the trade-off offered by having a variable length encoding. That is, we can now accommodate instructions with many operands, and still maintain good code density, but doing so means we might need two accesses to memory to load such instructions. Our control unit will naturally be more complicated as a result.

### 5.4.4.2 Compact Instruction Encodings

In the above discussion we used the concept of an escape code to denote that one opcode meant something special while all others could be interpreted as normal. We can re-use this concept to look at the issue of how one might design an instruction encoding mechanism for very compact instructions. That is, instructions that are perhaps only 8-bits long.

Consider a fictitious ISA with 8-bit instructions; it has four registers and a range of instructions that all fall into one of the following classes:

- 0 register operand + 0 immediate operands
- 1 register operands + 0 immediate operands
- 2 register operands + 0 immediate operands
- 0 register operands + 1 immediate operand of 7 bits
- 1 register operand + 1 immediate operand of 4 bits

Given register addresses can be represented as 2-bit values, a possible encoding might define five instruction formats:

1. $0\ldots0_{(2)}$
2. $10\ldots0_{(2)}$
3. $11ccrrrr_{(2)} \rightarrow 1100rrrr_{(2)} \ldots 1110rrrr_{(2)}$
4. $1111ccrr_{(2)} \rightarrow 111100rr_{(2)} \ldots 111110rr_{(2)}$
5. $111111cc_{(2)} \rightarrow 11111100_{(2)} \ldots 11111111_{(2)}$

where $c$ represents a field for opcodes, $x$ represents space for an immediate and $r$ represents space for a register address. From this we can see there are three possible two-operand instructions, three possible 1-operand instructions and four possible 0-operand instructions. Notice that in each case we are using an escape code to mark a different format. For example, $1100rrrr_{(2)}$, $1101rrrr_{(2)}$ and $1110rrrr_{(2)}$ are all two-operand instructions in the third class. However, we use what would have been the instruction $1111rrrr_{(2)}$ to signal that actually we mean something else: that is, one needs to read more opcode and hence translate into the fourth class of 1-operand instruction of the form $1111ccrr_{(2)}$. 

5.4 Instruction Encoding

One might view this as an application of variable length opcodes, rather than the variable length instructions as described previously; it is commonly called the **expanding opcode** approach. The key thing to realise is that even when we are allowing parts of the opcode to occur in various different places, every instruction is uniquely identifiable. This was trivial to see when looking at the MIPS32 ISA since the opcode was a fixed field. Here it is harder to see but this fact is vital if the encoding is to work correctly: if instructions are not uniquely identifiable, the control unit has no hope of telling which one we meant!

5.4.4.3 Compressing Register Addresses

Clearly space is at a premium as far as instruction encodings go, and in order to maximise the utilisation of this space, an ISA generally provides a general purpose register file whose size is a power-of-two. This means with \( n \)-bit register addresses we can address all \( 2^n \) registers. This has advantages and disadvantages. Although it ensures we get the best use of space since no bits are wasted, if we want less than \( 2^n \) registers, the next size down is \( 2^{n-1} \). If we start with thirty two registers, the next size down is sixteen: quite a jump. In ISAs that are designed for implementation in a context where resources are constrained, using less registers makes a lot of sense. But this must be balanced against performance so what happens if we want an ISA with twelve general-purpose registers for example, can we do better than use 4-bit addresses which could address sixteen registers and is hence wasteful?

If we assume there are formats for 3-operand instructions, using 4-bit register addresses would total twelve bits of operand in such an instruction; let us denote these three 4-bit register addresses as \( A_{3...0}, B_{3...0} \) and \( C_{3...0} \). The crucial thing to realise is that because
\[
A \in \{0_{10}, 1_{10}, \ldots, 11_{10}\}
\]
then the top two bits of \( A \), i.e., \( A_{3...2} \), cannot be \( 11_{(2)} \). That is, we always have that
\[
A_{3...2} \in \{00_{(2)}, 01_{(2)}, 10_{(2)}\}.
\]

So we can store the bottom two bits \( A_{1...0} \) as normal and then use the fact that there is an unused state in the top two bits to save space. Since we have \( A, B \) and \( C \) to encode we have that
\[
A_{3...2} \in \{00_{(2)}, 01_{(2)}, 10_{(2)}\}
\]
\[
B_{3...2} \in \{00_{(2)}, 01_{(2)}, 10_{(2)}\}
\]
\[
C_{3...2} \in \{00_{(2)}, 01_{(2)}, 10_{(2)}\}.
\]

There are thus only twenty seven possible combinations which can be encoded in only five bits, since \( 2^5 > 27 \), rather than the six which we would have used in the naive case. In total then, we need six bits for the bottom halves of the three register addresses and five bits to encode the top halves; this saves us one bit which can be used elsewhere, e.g., to double the number of opcodes we can specify.
5.5 Control-Flow

So far we have only considered programs, i.e., sequences of instructions, whose control-flow is a straight line. That is, the flow of execution starts at the top and proceeds through each instruction in sequence before ending at the bottom. In reality, this places a massive restriction on the types of programs we can write. For example we cannot conditionally execute sequences of instructions based on some input or intermediate result, or iterate over a sequence of instructions to form what we commonly call loops. Providing the ability to alter the control-flow of a program is therefore a vital part of both instruction set and micro-architecture design. This is particularly true since one can view instructions that alter control-flow as pure overhead: they are not really performing any useful computation themselves, just selecting which computation is performed. As such, ensuring that the control-flow mechanism does not have a significant impact on performance is also vital.

Roughly speaking, an instruction which might alter the flow of execution is called a branch instruction. Any branch will achieve a change in control-flow by updating the value of PC, the program counter, which holds the address of the next instruction to be executed. We can classify branches according to a number of criteria which dictate how and when they perform this update:

**Conditional Versus Unconditional** A branch is unconditional if it always updates the program counter; such a branch is sometimes called a jump. A branch is conditional if the update only occurs when some condition is true. Typically the condition will be a comparison between operands. When a branch is executed so that we know where next to fetch instructions from, i.e., the processor knows if it is taken or not, we say the branch has been resolved.
Examples of both forms are easy to see in the MIPS32 ISA. For example, the \( j \) instruction is an unconditional branch, it simply sets the \( PC \) to equal the immediate operand. Both \( beq \) and \( bltz \) are conditional branches since they compare two values to each other, two registers or a register and an immediate respectively, and only update the \( PC \) if the result of the comparison is true.

**Absolute Versus Relative**  
An absolute branch sets the \( PC \) to a value unrelated to the value it currently holds; the instruction simply replaces the value in \( PC \) with some new address. A \( PC \) relative (or offset) branch updates the \( PC \) by changing it by some amount (i.e., adding or subtracting some value); the end result is that the branch is relative to the \( PC \) rather than just any address.

Most branches are fairly local, it is uncommon for a branch to update the \( PC \) with an absolute address. As such, one can encode the target address more compactly as an offset from the \( PC \) than a full 32-bit address. This means that, for example, a single MIPS32 conditional offset branch \( beq \) can include a 16-bit immediate and cope with most offsets required.

There is one caveat to offset branches, which is related to when the branch is actually executed. In the fetch-decode-execute cycle we already said that the \( PC \) would be incremented by some amount during the fetch stage so it points to the next instruction to execute. By the time the processor enters the execution phase for a taken branch, we are essentially saying that this is not actually the instruction we want to execute at all, some other one is. But the \( PC \) has already been incremented so we need to take this into account when calculating the offset: typically the offset required will be one instruction, four bytes in the MIPS32 case, less than one might have thought. If this off-by-one problem is not taken into account, the processor will branch to the wrong place!

Figure 5.5 attempts to demonstrate the difference between absolute and offset branches; both Figure 5.5a and Figure 5.5b have two identical branches at addresses \( n \) and \( m \), the result of their execution is different however. In particular, both branches in Figure 5.5a result in control-flow being sent to address 100 while in Figure 5.5b the branch target depends on what address the branch instruction resides at.

**Immediate Versus Computed**  
The memory in a typical 32-bit processor has \( 2^{32} \) addressable locations. To branch to an arbitrary address therefore, one has to provide a 32-bit address. Encoding a 32-bit address inside a 32-bit instruction is impossible: there is no space for the opcode so some compromise must be made. The unconditional absolute branch \( j \) is a good example of this, it sets the \( PC \) to equal a 26-bit immediate which cannot address the whole memory but does cover a fairly large region.

To cope with the need to branch to arbitrary addresses, most ISAs include a computed branch instruction which takes the new \( PC \) value from a register operand. Since the register can contain a full 32-bit value, the branch can be to anywhere in memory. The trade-off is that one has to execute more instructions in order to compute this address in the register before the branch is executed; with immediate branches these extra instructions are not required.
Other than these different patterns of actual behaviour, the rest of the fetch-decode-execute cycle is unchanged. The completion of a branch in the execution stage means the right value is in the PC for the next fetch stage; the cycle simply carries on as normal except that we have altered where in the program we are now fetching instructions from.

One slight subtlety introduced by the MIPS32 ISA is the idea that we need not store the whole branch target as an immediate in the instruction encoding. Each instruction is a 32-bit value so if the PC starts at address zero, instructions always start on a multiple of four bytes; they are word aligned. If we are using a byte addressable memory where each byte has an address, the addresses of instructions are all 0 modulo 4, which is to say the least-significant two bits of the address are always zero. Since the PC is always word aligned, any offset from it will also be a multiple of four bytes and hence will also always have the least-significant two bits as zero. Given this fact, we do not need to store the bits in an encoded instruction. For example, the meaning of the \(j\) instruction is

\[
\text{PC} \leftarrow (\text{imm} : 00_{(2)})
\]

where \(\text{imm}\) is a 26-bit immediate stored in the instruction. This 26-bit immediate has the two implicit zero bits appended to the least-significant end via the concatenation operator before it is used to update the PC. This way, our 26-bit immediate does not waste space, and can specify a larger range of addresses as a result.

### 5.5.1 Predicated Execution

In the MIPS32 ISA, redirection of control-flow is done explicitly using dedicated branch instructions. Use of **predicated** or **guarded** execution offers an alternative to this model by associating a predicate function to each instruction and only executing the instruction if the predicate is satisfied, i.e., evaluates to true. Rather than set the PC to some new value, this approach allows one to “skip over” instructions based on the associated predicate. It is ideal for conditional execution of short sections of code due to the low overhead associated. Since MIPS32 does not support predicated execution, to show how the concept works we need several extensions: these are imaginary, but relate closely to the form of instruction set such as ARM.

Imagine we extend MIPS32 to include a carry flag \(CF\) which resides in the processor status register: this flag is set by arithmetic instructions such as addition when a carry-out is produced. If we also include a branch instruction based on the value of \(CF\), we can write a short fragment

\[
\begin{align*}
\text{...} \\
\text{if } CF = 0, PC & \leftarrow \text{skip} \\
\text{skip} : \text{...}
\end{align*}
\]
that performs an addition, then tests if the carry flag $CF$ was set in the status register: if the sum of $GPR[3]$ and $GPR[4]$ is too large to fit into 32 bits, a carry-out is produced and $CF$ will be set. If the carry flag is set, a second addition instruction adds the carry to $GPR[2]$; if the carry flag is not set, the branch instruction skips over this second addition. The fragment results in two or three instructions being executed depending on if the branch is taken or not.

Now imagine we extend MIPS32 further by allowing instructions to specify a predicate: every instruction can either be executed or skipped based on the value of $CF$. One might re-write the fragment from above as follows:

$$
\ldots \\
\text{if } CF = 1, GPR[5] \leftarrow GPR[5] + CF \\
\text{skip} : \ldots
$$

Now the second instruction has a predicate associated with it, only when the carry flag is set does this instruction get executed; if the predicate is not satisfied, then the processor just skips over it. This sequence is now more compact since it is only two instructions long: roughly speaking the processor executes only one or two instructions depending on if the predicate on the second instruction is satisfied. The trade-offs in deploying a system for predicated execution are quite involved. Typically programs using predicated execution exhibit better code density and, since they contain less branches, thus get more concrete computation done in the same time. On the other hand the predicates take up precious space in the instruction encoding and the control unit needs to be slightly more complicated to cope with skipping over instructions rather than passing them through the execute stage of the fetch-decode-execute cycle.

To better demonstrate the advantages of predicated execution, consider a more realistic example; the canonical such example is an implementation of the Greatest Common Divisor (GCD) algorithm due to Euclid. The idea is that given two integers $a$ and $b$, the algorithm finds the largest integer $c$ which exactly divides both $a$ and $b$; the case where $c = 1$ means $a$ and $b$ are co-prime. Figure 5.6a shows the Euclidean algorithm written in C. To implement this function we will again extend MIPS32 to include a rich set of flags in the status register which are set by a special compare instruction. Specifically, imagine the execution of

$$GPR[x] \overset{2}{=} GPR[y]$$

results in

- the $EQ$ flag being set when $GPR[x] = GPR[y]$,
- the $NE$ flag being set when $GPR[x] \neq GPR[y]$,
- the $LT$ flag being set when $GPR[x] < GPR[y]$,
- the $LE$ flag being set when $GPR[x] \leq GPR[y]$,
- the $GT$ flag being set when $GPR[x] > GPR[y]$,
- the $GE$ flag being set when $GPR[x] \geq GPR[y]$.
int GCD( int a, int b )
{
    while( a != b )
    {
        if( a > b )
            a -= b;
        else
            b -= a;
    }
    return a;
}

(a) The GCD function written in C.

    if EQ = 1, PC ← exit
    if LE = 1, PC ← skip

    PC ← loop

    PC ← loop

    exit: ...

(b) The GCD function implemented using branches.

    if NE = 1, PC ← loop

    exit: ...

(c) The GCD function implemented using predicated instructions.

Figure 5.6 A C function and ARM assembly language implementation of the Euclidean algorithm for computing GCD.

and that the flags are only set by the comparison instruction (i.e., not overwritten by execution of arithmetic). Again these are imaginary extensions, but again they closely follow real instruction sets such as ARM.

Figure 5.6b demonstrates a standard approach to implementing the function body under the assumption that a and b are stored in the registers GPR[2] and GPR[3]. The approach is controlled by comparing GPR[2] and GPR[3], i.e., a and b, and then using branch instructions to direct control-flow to the correct case: if GPR[2] = GPR[3], then the first branch exits the loop, if GPR[2] ≤ GPR[3] then the second
5.5 Control-Flow

branch directs control-flow to the instruction which subtracts a from b and then branches back to the start of the loop. Otherwise control-flow drops through to the instruction which subtracts b from a and then branches back to the start of the loop. In contrast, Figure 5.6c eliminates those branches by reimplementing the function body using three predicated instructions. Notice that the only branch is that which transfers control to the start of the loop: the branches required to select between the assignments a = a - b or b = b - a are managed by predicated execution. For example, the first subtract instruction is predicated on the GE or greater than or equal flag: if the comparison between a and b meant that a was greater than or equal to b, the GE flag would be set and hence the subtraction instruction (i.e the assignment a = a - b) would be executed, otherwise it is skipped.

5.5.2 Function Calls

Function calls are clearly another type of control-flow mechanism so we briefly look at the issue here with respect to support that is provided by the ISA. Considering a call to a sub-routine X, i.e., a function X with no arguments, execution looks like the following:

1. Branch to the sub-routine X, i.e., call the sub-routine.
2. Execute the sub-routine X.
3. Branch back to just after the call instruction.

The first two steps are simple enough and can be catered for using instructions already presented. The problem comes in the third step: a given sub-routine may be called from a number of different places, how do we know where to branch back to? The answer is to store the address to branch back to, the return address, somewhere when we call the sub-routine. There are three main options as regards where to store the return address:

• One could store the return address in a single fixed memory location or register. Although simple, this is a bad choice as it only allows the call of one sub-routine at a time; nested sub-routines are not really possible.
• One could store the return address in a per-sub-routine fixed memory location or register. This is a better option in that it allows nested sub-routine calls but still does not allow any form of recursion.
• Finally, one could use a stack; this is the most common solution since it allows total freedom with recursion and also facilitates argument passing when considering functions rather than simply sub-routines.

The MIPS32 ISA includes a common instruction called jal for jump-and-link that helps implement the first option above. The jal instruction performs two tasks: firstly it stores the return address in GPR[31], and then it performs a normal, unconditional absolute branch. The jr instruction offers a mechanism to branch to the contents of a register so this enables the sub-routine to branch back to the return
address by using $GPR[31]$. As such, these two instructions offer an efficient but limited form of sub-routine call. We will examine a more complete mechanism which solves the problems in Chapter 11 since more general function calls are essentially a mechanism implemented by the compiler rather than in the processor. Note that the $jal$ instruction is the first we have seen to blur the divide between general and special-purpose registers: $GPR[31]$ is a general-purpose register but it is reserved here for a special role.

## 5.6 Some Design Philosophy

### 5.6.1 Moore’s Law

Gordon Moore, co-founder of Intel, is credited with identifying an important trend with respect to development of transistor-based technology: roughly speaking, Moore’s Law [46] says that the number of transistors that one can fit in unit area roughly doubles every two years. This has become a form of a self-fulfilling prophecy however in the sense that that the “law” is now an accepted truth so indu-
try is forced to deliver improvements: industry is driven by the law rather than the other way around! Figure 5.7 demonstrates the manifestation of Moore’s Law on the development of Intel processors.

Of course Moore’s Law is not a “law” at all but has held for the last forty or so years all the same. The implications for design of a processor can be viewed in (at least) two ways:

- If one can fit more transistors in unit area, the transistors are getting smaller and hence working faster due to their physical characteristics. As a result, one can take a fixed processor design and over time it will get faster or use less power as a result of Moore’s Law.
- If one can fit more transistors in unit area, then one can design and implement more complex structures in the same fixed area. As a result, over time one can use the extra transistors to improve the processor design yet keep it roughly the same size.

The second option above is clearly more reasonable in the long term. That is, since there are physical limits that dictate how small and fast a transistor can be, better design is always the best long-term option. However, even in this case Moore’s Law can have an impact on thinking. For example, since the development of a new processor takes many years, Moore’s Law can invalidate a given design quite quickly: what is the best design one day might no longer be the best with respect to a new generation of improved technology two years later. Furthermore, not all components obey Moore’s Law. In particular, the rate at which memory access speed has improved does not match the rate at which memory density has improved; this leads to a widening disparity between processor and memory speed which can be a significant problem.

### 5.6.2 RISC versus CISC

Many of the trade-offs presented so far have been a matter of the length and complexity of programs, and hence performance in a rough sense, versus simplicity in design and implementation. This is a classic trade-off and one which has resulted in two schools of design methodology, the so-called **Reduced Instruction Set Computers (RISC)** and **Complex Instruction Set Computers (CISC)**.

**RISC**  
Put simply, RISC designs attempt to provide an ISA where instructions all do very simple operations. To write a program using such an ISA might take more instructions since they are so simple, but this same simplicity means a processor implementation can execute them very quickly; each instruction typically takes one clock cycle. RISC designs are characterised by the provision of many general-purpose registers, few addressing modes and a fixed length, regular instruction encoding. This makes it easy to program for both humans and compilers; performance of compiler-generated code can often match or better that of human-generated code.
CISC  In contrast, CISC designs offer instructions which perform much more involved operations. For example, a CISC ISA might include an instruction for computing square roots while a RISC design would demand you constructed an algorithm in software using only the basic instructions available. Although the CISC instruction set will be harder to implement and instructions might take longer to execute, a program will typically be shorter since there is more work being done by each instruction. Instructions might take many clock cycles to complete. A CISC design will typically have fewer general-purpose registers (perhaps using some of them for special-purposes), many addressing modes and might use a variable length instruction encoding. As a result, it will be harder for a compiler to generate efficient code.

Like many areas of computer science, research into ways of improving CISC designs such as the Intel 80486 and Motorola 68000 was initially undertaken by both industry and academia. Early experiments by IBM [17] into the removal of many under-utilised and over-designed features eventually led to the POWER and PowerPC RISC architectures. At the same time, research in Berkeley [50] and Stanford [27] heavily influenced the MIPS and SPARC RISC architectures. These processors command a significant market share; in particular they dominate the games console and embedded arenas.

Ultimately there is no right or wrong answer to the question of which is the better approach, it depends entirely on the design context and goals. However, since initial research and subsequent commercialisation in the late 1970s and early 1980s, RISC design has dominated the field of computer architecture. Even processors which were conventionally CISC-based, such as the Intel Pentium range, have gradually migrated to using an internal RISC core behind a translation layer for backward compatibility. Like most aspects of computer architecture, successful RISC designs rely heavily on selecting an effective trade-off between many competing factors. Such decisions are typically resolved by performing a workload characterisation of the sorts of program that the processor will execute. By considering an average program, the architecture is designed so that the average case is optimised while non-typical cases are marginalised or omitted.

5.7 Putting It All Together

As described, developing a Verilog processor model that implements the reduced MIPS32 ISA might seem a daunting task. Although we defer a detailed investigation of the ALU and main memory to Chapter 7 and Chapter 8, the goal of this section is to show that for the most part, such a processor model is simply a big state machine. As such, the resulting model could be more accurately termed a simulator or emulator: it does the right thing functionally but there is not enough detail to generate realistic hardware from the model. We make a further simplification by ignoring system calls, interrupts and the like until we encounter and explain them in Chapter 12.
module processor( input wire clk,
    input wire rst );

reg [31:0] PC;
reg [31:0] IR;
reg [31:0] HI;
reg [31:0] LO;
reg [7:0] MEM[0:1023];
reg [31:0] GPR[0: 31];
reg [2:0] state;
reg [5:0] dec_opcode;
reg [4:0] dec_rs;
reg [4:0] dec_rt;
reg [4:0] dec_rd;
reg [5:0] dec_funct;
reg [31:0] dec_imm16;
reg [31:0] dec_imm26;
reg [31:0] dec_imm16s;
reg [31:0] dec_imm26s;
reg [31:0] dec_ext16;
reg [31:0] dec_ext26;
reg [31:0] dec_ext16s;
reg [31:0] dec_ext26s;
reg [31:0] alu_dst_lo;
reg [31:0] alu_dst_hi;
...
endmodule

Listing 5.5 The interface to, and definitions for and within, a Verilog module that models a
MIPS32-based processor.
always @( posedge rst )
begin
  PC = 0;
  IR = 0;
  HI = 0;
  LO = 0;
  state = 'STATEFETCH;
  $readmemh( "processor.bin", MEM, 0 );
end

always @( posedge clk )
begin
  if( !rst )
  begin
    case( state )
      'STATEFETCH : begin
        fetch;
        state = 'STATE_DECODE;
      end
      'STATEDECODE : begin
        decode;
        state = 'STATE_EXECUTE;
      end
      'STATEEXECUTE : begin
        execute;
        state = 'STATEMEMORY;
      end
      'STATEMEMORY : begin
        memory;
        state = 'STATEWRITE;
      end
      'STATEWRITE : begin
        write;
        state = 'STATEFETCH;
      end
    endcase
  end
end

Listing 5.6 Verilog processes that determine the processor behaviour.

Listing 5.5 describes the simple interface to the processor; it just includes external clock and reset signals. Normally this interface would include a lot of other connections with external devices, the main memory being the most obvious. However, we have not covered that material yet so we model the memory as an internal 2-dimensional Verilog register to the processor for now. The processor is driven by the processes in Listing 5.6. The first process resets the processor whenever a positive edge on the reset signal $rst$ is detected. The reset procedure involves zeroing the registers $PC$, $IR$, $HI$ and $LO$, initialising the state machine, represented by $state$, and then initialising the memory using the $readmemh$ system task: you can think of this as “magic” which would be replaced by an operating system running on the processor tasked with loading and starting execution of programs. The second task acts to update the state machine each time a positive edge on the clock signal $clk$ is detected; the faster the clock works, the faster the processor works. The state machine simply iterates around the fetch-decode-execute cycle calling the right task at
task alu( input [5:0] alu_oper,
        input [31:0] alu_src0,
        input [31:0] alu_src1 );
begin
  case( alu_oper )
    'FUNCT_SYSCALL : ;
    'FUNCT_BREAK : ;
    'FUNCT_ADD : alu_dst_lo = (alu_src0 + alu_src1);
    'FUNCT_SUB : alu_dst_lo = (alu_src0 - alu_src1);
    'FUNCT_MUL : {alu_dst_hi, alu_dst_lo} = (alu_src0 * alu_src1);
    'FUNCT_MULT : {alu_dst_hi, alu_dst_lo} = (alu_src0 * alu_src1);
    'FUNCT_AND : alu_dst_lo = (alu_src0 & alu_src1);
    'FUNCT_OR : alu_dst_lo = (alu_src0 | alu_src1);
    'FUNCT_XOR : alu_dst_lo = (alu_src0 ˆ alu_src1);
    'FUNCT_NOR : alu_dst_lo = ˜(alu_src0 | alu_src1);
    'FUNCT_SLLV : alu_dst_lo = (alu_src0 << alu_src1);
    'FUNCT_SRLV : alu_dst_lo = (alu_src0 << alu_src1);
    'FUNCT_MFHI : ;
    'FUNCT_MFLO : ;
    'FUNCT_MTHI : ;
    'FUNCT_MTLO : ;
    'FUNCT_JR : ;
  endcase
end
endtask

task lw( input [31:0] mem_addr,
          output [31:0] mem_data );
begin
  mem_data[ 7: 0] = MEM[{mem_addr[31:2],2'b00}];
  mem_data[15: 8] = MEM[{mem_addr[31:2],2'b01}];
  mem_data[23:16] = MEM[{mem_addr[31:2],2'b10}];
  mem_data[31:24] = MEM[{mem_addr[31:2],2'b11}];
end
endtask

task sw( input [31:0] mem_addr,
          input [31:0] mem_data );
begin
  MEM[{mem_addr[31:2],2'b00}] = mem_data[ 7: 0];
  MEM[{mem_addr[31:2],2'b01}] = mem_data[15: 8];
  MEM[{mem_addr[31:2],2'b10}] = mem_data[23:16];
  MEM[{mem_addr[31:2],2'b11}] = mem_data[31:24];
end
endtask

Listing 5.7 Verilog tasks to model the ALU and main memory.

Before we implement these tasks, we introduce three auxiliary tasks in Listing 5.7 that are used to model the ALU and access to memory. The alu task simulates the ALU performing some operation, determined by alu_oper, on the source values alu_src0 and alu_src1 to produce the results alu_dst_lo and alu_dst_hi. The ALU behaviour is implemented using built-in Verilog op-
task fetch();
begin
lw( PC, IR );
PC = PC + 4;
end
task decode();
begin
dec_opcode = IR[31:26];
dec_rs = IR[25:21];
dec_rt = IR[20:16];
dec_rd = IR[15:11];
dec_funct = IR[ 5: 0];
dec_imm16 = { 16'h0000, IR[ 15:0] };
dec_imm26 = { 6'h0000 , IR[ 25:0] };
dec_imm16s = { 14'h0000, IR[ 15:0], 2'h0 };
dec_imm26s = { 4'h0000 , IR[ 25:0], 2'h0 };
dec_ext16 = (( IR[15] == 1 ) ? 16'hFFFF : 16'h0000 ), IR[15:0];
dec_ext26 = (( IR[25] == 1 ) ? 6'h3F : 6'h00 ), IR[25:0];
dec_ext16s = (( IR[15] == 1 ) ? 14'hFFFF : 14'h0000 ), IR[15:0], 2'h0 ;
dec_ext26s = (( IR[25] == 1 ) ? 4'h3F : 4'h00 ), IR[25:0], 2'h0 ;
GPR[0] = 0;
end
task execute();
begin
case( dec_opcode )
  'OPCODE_SPECIAL1 : alu( dec_funct, GPR[dec_rs], GPR[dec_rt]);
  'OPCODE_SPECIAL2 : alu( dec_funct, GPR[dec_rs], GPR[dec Rt]);
  'OPCODE_ADDI : alu('FUNCT_ADD, GPR[dec_rs], dec_ext16 );
  'OPCODE_LW : alu('FUNCT_ADD, GPR[dec_rs], dec_ext16 );
  'OPCODE_SW : alu('FUNCT_ADD, GPR[dec_rs], dec_ext16 );
  'OPCODE_BEQ : alu('FUNCT_ADD, PC, dec_ext16s );
  'OPCODE_BNE : alu('FUNCT_ADD, PC, dec_ext16s );
  'OPCODE_BLEZ : alu('FUNCT_ADD, PC, dec_ext16s );
  'OPCODE_BGTZ : alu('FUNCT_ADD, PC, dec_ext16s );
  'OPCODE_REGIMM : alu('FUNCT_ADD, PC, dec_ext16s );
  'OPCODE_JAL : alu('FUNCT_ADD, PC, 8 );
endcase
end
task fetch();
begin
lw( PC, IR );
PC = PC + 4;
end
task decode();
begin
dec_opcode = IR[31:26];
dec_rs = IR[25:21];
dec_rt = IR[20:16];
dec_rd = IR[15:11];
dec_funct = IR[ 5: 0];
dec_imm16 = { 16'h0000, IR[ 15:0] };
dec_imm26 = { 6'h0000 , IR[ 25:0] };
dec_imm16s = { 14'h0000, IR[ 15:0], 2'h0 };
dec_imm26s = { 4'h0000 , IR[ 25:0], 2'h0 };
dec_ext16 = (( IR[15] == 1 ) ? 16'hFFFF : 16'h0000 ), IR[15:0];
dec_ext26 = (( IR[25] == 1 ) ? 6'h3F : 6'h00 ), IR[25:0];
dec_ext16s = (( IR[15] == 1 ) ? 14'hFFFF : 14'h0000 ), IR[15:0], 2'h0 ;
dec_ext26s = (( IR[25] == 1 ) ? 4'h3F : 4'h00 ), IR[25:0], 2'h0 ;
GPR[0] = 0;
end
task execute();
begin
case( dec_opcode )
  'OPCODE_SPECIAL1 : alu( dec_funct, GPR[dec_rs], GPR[dec_rt]);
  'OPCODE_SPECIAL2 : alu( dec_funct, GPR[dec_rs], GPR[dec rt]);
  'OPCODE_ADDI : alu('FUNCT_ADD, GPR[dec_rs], dec_ext16 );
  'OPCODE_LW : alu('FUNCT_ADD, GPR[dec_rs], dec_ext16 );
  'OPCODE_SW : alu('FUNCT_ADD, GPR[dec_rs], dec_ext16 );
  'OPCODE_BEQ : alu('FUNCT_ADD, PC, dec_ext16s );
  'OPCODE_BNE : alu('FUNCT_ADD, PC, dec_ext16s );
  'OPCODE_BLEZ : alu('FUNCT_ADD, PC, dec_ext16s );
  'OPCODE_BGTZ : alu('FUNCT_ADD, PC, dec_ext16s );
  'OPCODE_REGIMM : alu('FUNCT_ADD, PC, dec_ext16s );
  'OPCODE_JAL : alu('FUNCT_ADD, PC, 8 );
endcase
end

Listing 5.8 Verilog tasks to perform fetch, decode and execute stages.
5.7 Putting It All Together

```verilog

# Listing 5.9 Verilog tasks to perform memory access and write stages.

task memory();
begin
  case( dec_opcode )
    'OPCODE_LW : lw(alu_dst_lo, GPR[dec_rt]);
    'OPCODE_SW : sw(alu_dst_lo, GPR[dec_rt]);
  endcase
end
endtask

task write();
begin
  case( dec_opcode )
    'OPCODE_SPECIAL1 : case( dec_funct )
      'FUNCT_SYSCALL : ;
      'FUNCT_BREAK : ;
      'FUNCT_MULT : begin
        HI = alu_dst_hi;
        LO = alu_dst_lo;
      end
      'FUNCT_MFHI : GPR[dec_rd] = HI;
      'FUNCT_MFLO : GPR[dec_rd] = LO;
      'FUNCT_MTHI : HI = GPR[dec_rs];
      'FUNCT_MTL0 : LO = GPR[dec_rs];
      'FUNCT_JR : PC = GPR[dec_rs];
      default : GPR[dec_rd] = alu_dst_lo;
    endcase
    'OPCODE_SPECIAL2 : GPR[dec_rd] = alu_dst_lo;
    'OPCODE_ADDI : GPR[dec_rt] = alu_dst_lo;
    'OPCODE_BEQ : if( GPR[dec_rs] == GPR[dec_rt] )
      PC = alu_dst_lo;
    'OPCODE_BNE : if( GPR[dec_rs] != GPR[dec_rt] )
      PC = alu_dst_lo;
    'OPCODE_BLEZ : if( GPR[dec_rs] <= 0 )
      PC = alu_dst_lo;
    'OPCODE_BGTZ : if( GPR[dec_rs] > 0 )
      PC = alu_dst_lo;
    'OPCODE_REGIMM : case( dec_rt )
      'RT_BLTZ : if( GPR[dec_rs] < 0 )
        PC = alu_dst_lo;
      'RT_BGEZ : if( GPR[dec_rs] >= 0 )
        PC = alu_dst_lo;
      endcase
    'OPCODE_J : PC = dec_imm26s;
    'OPCODE_JAL : begin
      GPR[31] = alu_dst_lo;
      PC = dec_imm26s;
    end
  endcase
end
endtask
```
erators; in Chapter 7 we examine how this task can be replaced by a more efficient, more realistic module that does the same thing. The \texttt{lw} and \texttt{sw} tasks simulate memory access by loading and storing a 32-bit value to and from memory respectively. Again, in Chapter 8 we examine how this can be achieved more realistically by replacing the 2-dimensional \texttt{MEM} register with an accurate model with an appropriate interface.

Using the \texttt{alu}, \texttt{lw} and \texttt{sw} tasks as a means of abstraction, we can start to implement the processor behaviour. Listing 5.8 details the tasks used by the state machine for the fetch, decode and execute stages. Each performs exactly the behaviour we outlined in our previous description of execution within a stored program architecture. For example, the \texttt{fetch} task first loads the next instruction, using the address in \texttt{PC}, into the instruction register \texttt{IR}. It then uses the ALU to increment \texttt{PC}. The \texttt{decode} task decodes the instruction in \texttt{IR}, performing sign extension to produce values read to operate on. This task also cheats a little bit: to ensure that \texttt{GPR[0]} always holds the value zero, we reset it here so that when we come to executing an instruction it is always right. The \texttt{execute} task performs the first stage of execution by invoking the ALU on source operands to produce useful results. In the case of an R-type arithmetic instruction, the ALU operator is stored in the \texttt{funct} field so we can easily work out what to do; in the case of a branch, the ALU is used to compute the branch target by adding the offset to \texttt{PC}. Notice how the general-purpose registers are indexed by the decoded register address. Finally, the memory access and write tasks are detailed in Listing 5.9. The \texttt{memory} task implements memory access, both load and store, simply using a call to the appropriate \texttt{lw} or \texttt{sw} task. In both cases, the address to access will have been computed during the execute stage and generated by the ALU into \texttt{alu_dst_lo}. The write task \texttt{write} deals with storing the results of execution back and also completing any branches whose target address was computed during the execution stage.

In order to test the model, we first need to present it with a program to execute. To initialise the memory with a program, as triggered by a positive edge on the reset signal, we use the \texttt{$readmemh} system task. This has the effect of reading the file specified as the first argument into the 2-dimensional register specified as the second argument at an offset specified by the third argument. In this case, the file \texttt{processor.bin} is filled with 256 lines which look like

\begin{verbatim}
01 00 03 20
02 00 04 20
20 10 64 00
00 00 00 00
\ldots
\end{verbatim}

Given the bytes are presented in little-endian order, the first three 32-bit words represent the machine code instructions

\begin{verbatim}
0010000000000011000000000000000001(2)
001000000000010000000000000000010(2)
000000000110010000010000000100000(2)
\end{verbatim}

which represent the very basic program
5.7 Putting It All Together

Figure 5.8 Behaviour of the processor model.
\begin{align*}
GPR[3] &\leftarrow GPR[0] + 1 \\
GPR[4] &\leftarrow GPR[0] + 2 \\
\end{align*}

via two \texttt{addi} and one \texttt{add} instruction. Starting the processor executing on this program yields the behaviour in Figure 5.8. Iteration through the steps of execution is obvious with the \texttt{state} value being updated on clock edges once the processor is reset. One can see that during the fetch stage, i.e., when \texttt{state} equals 0, the \texttt{IR} value is read from memory and the \texttt{PC} is updated. On the first fetch this results in the instruction \texttt{00100000000000110000000000000001 (2)} = \texttt{20030001 (16)} being loaded from address \texttt{0 (16)} and then \texttt{PC} being set to \texttt{4 (16)}. After parts of the instruction are extracted during the decode stage, the ALU is invoked to perform the operation \( GPR[0] + 1 \) which produces the result \( 1 (16) \) in \texttt{alu\_dst\_lo}. The value is written-back and reused later as execution continues.

### 5.8 Further Reading

- S.P. Dandamudi.  
  *Guide to RISC Processors.*  

- D. Harris and S. Harris.  
  *Digital Design and Computer Architecture: From Gates to Processors.*  

- J.L. Hennessy and D.A. Patterson.  
  *Computer Architecture: A Quantitative Approach.*  

- D.A. Patterson and J.L. Hennessy.  
  *Computer Organization and Design: The Hardware/Software Interface.*  

- J. Stokes.  

- D. Sweetman.  
  *See MIPS Run.*  

- A.S. Tanenbaum.  
  *Structured Computer Organisation.*  
5.9 Example Questions

21. Imagine you are on the design team for a new 16-bit processor. The ISA for the processor includes specifications for:

- 16 general-purpose registers.
- 8 ALU instructions that take two input register operands and write a result into a third output register operand.
- 4 branch instructions that take an input register operand and an 8-bit immediate offset.
- 2 memory access instructions that take two input register operands and one output register operand. 2 memory access instructions that take one input register operand, one output register operand and a 4-bit immediate offset.

a. Devise an instruction encoding for the processor, taking care to explain any advantages and disadvantages of your design.
b. Draw a block diagram of the basic components within this processor, explain the function of each one and how they are connected together.
c. Using the example of an instruction that adds two registers together and writes the result in a third register, describe the steps involved during the following phases of execution:
   i. Fetch.
   ii. Decode.
   iii. Execute.
d. The control unit that operates the data-path can be implemented using hardwired logic or a micro-code based system. Explain the idea of both of these methods, including the advantages and disadvantages of each.

22. Explain what is meant by the following addressing modes, giving examples of their use in a C program:

a. Immediate.
b. Direct.
c. Indirect.
d. Indexed.

23. a. A change in the design of a particular processor means it will now use the idea of predicated execution. Explain what this means and translate the following C fragment into an assembly language style program for the processor:

```c
int t = 0;

for( int i = 0; i < n; i++ ) {
    if( ( x >> i ) & 1 )
        t = t + 1;
}
```
Use predicated execution or branch instructions where appropriate and take care to justify your choice in each case.

b. It is suggested that the code density of programs executed by the processor is too low. Explain what is meant by code density and outline two approaches to improving this situation.

24. You have been asked to define the instruction set of a new processor and the encoding of this instruction set. The processor contains a program counter (PC), four general-purpose registers (A, B, C and D), uses 8-bit instructions and has a 256-byte memory that is addressed indirectly via the registers. The instructions set includes the following instructions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Operands</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA</td>
<td>x</td>
<td>Loads constant x, with $0 \leq x \leq 127$, into register A.</td>
</tr>
<tr>
<td>MOV</td>
<td>R, S</td>
<td>Moves register S into register R.</td>
</tr>
<tr>
<td>NOT</td>
<td>R</td>
<td>Performs a bitwise complement of register R.</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td>Performs no operation.</td>
</tr>
<tr>
<td>NEG</td>
<td>R</td>
<td>Performs twos-complement negation of register R.</td>
</tr>
<tr>
<td>BEZ</td>
<td>R, x</td>
<td>Branches by incrementing the program counter by an offset x, with $-8 \leq x \leq 7$, if register R is equal to 0.</td>
</tr>
</tbody>
</table>

So one might write the program:

```
LDA 10
MOV B, A
NEG B
BEZ B, 4
```

to load the constant 10 into register A, then move this value into register B and finally negate it before jumping forward by four instructions if register B turns out to be zero.

a. Design an encoding for the instructions given above.
b. What is the maximum number of 2-operand instructions that you can add to the instruction set?
c. What is the maximum number of 1-operand instructions that you can add to the instruction set?
d. What is the maximum number of 0-operand instructions that you can add to the instruction set?
e. Write down a more complete instruction set, adding those which you view as currently missing and giving reasons why you want to add them.
f. Finally, write down the instruction encoding for this complete instruction set.

25. There are 32 pieces on a chess board represented by an $8 \times 8$ grid; a black and a white version of 1 king, 1 queen, 2 bishops, 2 knights, 2 rooks and 8 pawns.

You are involved in the design of a robot that automates the game of chess only in terms of moving the physical pieces around. The robot needs to be fed a list of
moves so that it knows how to move a piece on one board position to another position; it does not care about the legality of moves. However, it is intelligent enough to know whose turn it is (i.e., black or white), how to find a piece on the board given its unique name (e.g., pawn #1) and can cope with capturing pieces if a move places a piece on an already occupied square. It cannot play any version of chess with the advanced rules of promotion, castling, resignation or en passant: it simply moves and captures pieces. It can deal with the fact that pawns can move two spaces forward from their starting position but otherwise has no memory.

Design a compact encoding of instructions that can be fed to the robot to provoke the movement of any piece on the board. State any assumptions you make in your design and the advantages and disadvantages of the result.

26. Imagine you are asked to design a RISC style instruction set architecture for a new 32-bit processor which is modelled roughly on the MIPS32 design. The processor has 32 general-purpose registers and the instruction set is to include:

- Arithmetic, logic and comparison instructions that use a mix of general-purpose register and immediate operands.
- Memory access instructions that use an single addressing mode whereby a base address is specified by a general-purpose register operand and an offset is specified by an immediate operand.
- Branch instructions that include both conditional and unconditional variants and use mix of general-purpose register and immediate operands to specify the absolute and offset target address.

a. One designer wants to use a fixed length instruction encoding while another suggests that a variable length encoding would be better.

i. Explain the advantages and disadvantages of each approach.

ii. Select one approach and using your selection, design basic instruction encoding formats for the instruction types listed above.

b. The “add immediate” instruction adds the contents of a general-purpose register to a 16-bit immediate and stores the result back into a general-purpose register. For example, the instruction $GPR[1] \leftarrow GPR[2] + 3$ takes the contents of general-purpose register 2, adds the immediate value 3 to it and stores the result into general-purpose register 1.

Using just the instruction encoding formats from above, show how


are encoded into 32-bit “add immediate” instructions; state any assumptions you make.

c. A major customer for the processor has identified the following C function as crucial to the performance of their applications:
int f(int x) {
    int w = 0;
    for(int i = 0; i < 32; i++) {
        if((x >> i) & 1)
            w = w + 1;
    }
    return w;
}

Explain what this function does and how you might alter the processor design (including any changes to the ALU) to satisfy the requirements of this customer.

d. There is some debate about which endian convention the processor should use. Assuming the memory is byte addressed, consider the execution of an instruction that stores the 32-bit integer 305419896 at address 0. List the content of these addresses after the store instruction completes if the processor uses

i. Little-endian byte ordering.
ii. Big-endian byte ordering.

e. The team responsible for writing a compiler for the new processor is concerned about the number of general-purpose registers. Explain the positive and negative implications for the instruction set if the number of general-purpose registers is

i. Increased from 32 to 64.
ii. Decreased from 32 to 16.

In each case, describe how this might alter performance of programs that execute on the processor.