Preface

These notes are to accompany the module CO 1016. They contain core material for the course. For more motivation and background please attend the lectures.

These notes are detailed. Thus, while they should be useful and have been checked carefully, there are bound to be some errors. Please do let me know about any typos or other mistakes which you find. If you have any other (constructive) comments, please tell me about them.

Books recommended for CO 1016 are listed in the Study Guide: ask if you need advice.

These notes have an index—please make use of it!! Definitions in the notes are in bold text, and appear in the index.

If you are to do well in this course,

YOU MUST ATTEND THE LECTURES.

They will give you additional examples, highlight key issues which may not appear quite as important from the notes as in fact the issues are, and give guidance towards what you need to know for the examinations.

You should note especially that with a course of this nature, notation, definitions and ideas can differ even among the experts. The lectures will clarify many of the more woolly areas.

Acknowledgements

Many of the books in the Study Guide have helped in the preparation of this module. The following book has been used closely, and should be purchased by students.

Hennessy and Patterson. **Computer Organization and Design.** Morgan Kaufmann, 1998 (second edition).

Contents

List of Tables

List of Figures

1
Background Mathematics

1.1 Denotation and Semantics

Look up the word (a verb) **denote** in a dictionary, and read the definition. Roughly, it means to “signify” or “indicate”. The idea of denotation pervades Computer Science and Mathematics. Let $\{0,1,2,3,\ldots\}$ be the set of natural numbers. We use symbols to denote things we are interested in. For example, we might say “let $\mathbb{N}$ denote the set of natural numbers”. Sometimes we will just say “let $\mathbb{N}$ be the set of natural numbers”—clearly $\mathbb{N}$ is just a symbol on paper and not the actual set, but the meaning of the sentence is clear. However, sometimes we might want to use a *different* symbol, say N, to denote the *same* set of natural numbers. Another example concerns the symbols used to denote the natural numbers themselves. For example, under appropriate circumstances, both $|||||$ and V denote the number five. The first symbol consists of five strokes, and the second is a Roman numeral.

The word **semantics** indicates “meaning”. We sometimes say that the semantics of V is five. We refer to V as **syntax**. Syntax refers to symbols, or notation, which will usually have a meaning, or semantics. *<u>Exercise:</u> Look up any words which are new to you in a dictionary, and read the definitions.* A more real world example concerns natural language. Both *cheeze* and *fromage* are very different syntactically. But in the correct context, knowing that they are words of English and French, they have the same semantics—tasty, yellow gunge. As this course unfolds, we shall see many examples of syntax and semantics. Finally, note that *denotation* is a relationship between syntax and semantics; $|||||$ denotes five, II denotes two, III does not denote ten, and so on.

A **language** consists of a syntax and semantics. The syntax consists of a set of symbols called an **alphabet** and some **rules** which state how we use the alphabet to form words. In a natural language the rules are the rules of spelling, given by a dictionary. In a programming language the syntax rules appear in the programming manual; they tell us how to write computer programs which are syntactically correct. The semantics of a program is usually defined to be what happens when the program runs on a computer.

1.2 Sets

We shall assume that the reader has met the idea of a *set* before, and has some experience of the mathematical manipulation of sets. Here we recall some of the basic ideas of the theory of sets. For the purpose of these notes, a **set** is *an unordered collection of objects in which any object can appear at most once*—and for the time being, these objects can be any objects you care to think of. We shall not give a definition of the terms "unordered" or "collection." These concepts will be left as being understood intuitively, at least for this introductory course. Recall that in simple cases, the notation used to indicate a set is a pair of curly braces which encloses the objects which make up the set; some examples of sets are

- $\{$ 'a', 'b' $\}$ which is the set consisting of the first two **letters** (also called **characters**) of the alphabet—note the 'font' used to indicate actual letters,
- $\{1,2,3,4,5,6\}$ which is the set of the first six positive natural numbers, and
- $\{$ 'pen', 'paper', 'ruler' $\}$, a set consisting of three **strings** (a string is a sequence of letters) where each string denoting a well known writing implement.

Note that any given set (collection of objects) has an important property: if we pick any object we like, it is either in the set or it is not. For example, 'a' is in the set $\{$ 'a', 'b' $\}$ but both 4 and 'fudge' are not. Note also that, by definition, $\{$ 'a', 'a', 'b' $\}$ is not a set as the 'a' appears more than once.

The objects in a set are referred to as **elements**, so for example 'a' is an element of our first example of a set. We write 'a' $\in \{$ 'a', 'b' $\}$ to indicate this.

We use un-quoted letters to indicate **variables**. A variable is, roughly, an unknown quantity. Do not confuse letters and variables—they are different. We may write A to denote any set. We write $e \in A$ as shorthand for "e is an element in A". Here, e is a variable denoting an element. If e is any object not in A, we denote this by $e \notin A$. For example if A happens to be the set $\{5,7,88\}$ and e is 2, then $e \notin A$. Note that we shall often, but not always, use capital letters to denote sets, and small letters to denote elements of sets.

If A is a set, we write $S \subseteq A$ to mean that S is a set, and that any element of S is an element of A. We call such an S a **subset** of A. Note that any subset is, by definition, itself a set!

If we write $E \stackrel{\text{def}}{=} \{0,2,4,6,\dots\}$ this means that E is shorthand for the set of even natural numbers. For example $12 \in E$. We can also write

$$E \stackrel{\text{def}}{=} \{\, n \mid n \in \mathbb{N};\ n \text{ is divisible by } 2 \,\}$$

We read this as "E is the set of objects n such that n is a natural number and n is divisible by 2". The symbol $\mid$ means "such that". Note that the set

$$\{\, x \mid x \in \mathbb{N};\ x \text{ is divisible by } 2 \,\}$$

is equal to E. Why? This is an important point which also arises in programming.

Example 1.2.1 We have $\{2,4,6\} \subseteq \{1,2,3,4,5,6\}$.

We shall use the following notational definitions

$\mathbb{B}$	binary digits	$\{0,1\}$
$\mathbb{N}$	natural numbers	$\{0,1,2,3,\ldots\}$
$\mathbb{Z}$	integers	$\{\ldots,-2,-1,0,1,2,\ldots\}$
$\mathbb{Q}$	rationals (fractions)	$\{n/d \mid n,d \in \mathbb{Z}; d \neq 0\}$

1.3 Constructions on Sets

As we noted earlier, the critical attribute of a set is whether or not an object is in the set. So, for example, $\{2,1\} = \{1,2\}$. There is no notion of order of the elements. We assume the reader has met the idea of a *pair* before, for example $(1,2)$. Such a pair might represent a map reference, "1 along and 2 up," which is different from "2 along and 1 up." Such map references are also called **Cartesian coordinates**. A **pair** is anything of the form (a,b) where a and b are any objects. We call a **coordinate 1** of (a,b) and b **coordinate 0** of (a,b). In order to capture the idea of "order of the elements", we regard two pairs as equal iff their respective coordinates are equal. More formally, the pair (a,b) is equal to (a',b') when $a = a'$ and $b = b'$. For example $(1,2) \neq (2,1)$ but $(1,1) = (1,1)$.

Given sets A and B, the **cartesian product** of A and B is the set

$$A \times B \stackrel{\text{def}}{=} \{\ (a,b)\ \mid\ a \in A \text{ and } b \in B\ \}.$$

For example,

$$\{1,2\} \times \{\text{‘X‘}, \text{‘Y‘}, \text{‘Z‘}\} \stackrel{\text{def}}{=} \{(1,\text{‘X‘}),(1,\text{‘Y‘}),(1,\text{‘Z‘}),(2,\text{‘X‘}),(2,\text{‘Y‘}),(2,\text{‘Z‘})\}.$$

We call $A \times B$

• *cartesian* with reference to the set of pairs—think of a pair as a cartesian coordinate; and

• *product* because the number of elements of $A \times B$ is the (usual) product of the number of elements in A and B.

In the case when $A = B$ we sometimes write A^2 instead of $A \times A$. We sometimes call A^2 the 2**-ary product** of A. We can generalize to a 3**-ary product**. For example, the elements of the set $\mathbb{B} \times \mathbb{B} \times \mathbb{B}$ (or $\mathbb{B}^3$) are called **triples**. For example $(0,0,0) \in \mathbb{B}^3$ and $(1,0,1) \in \mathbb{B}^3$. You might like to think of the elements of $\mathbb{B}^3$ as the corners of a three dimensional cube with sides of length 1.

We can generalise these ideas to the n**-ary product** A^n. A typical element of A^n looks like $(a_{n-1},\ldots,a_1,a_0)$ where each $a_i \in A$ for $0 \le i \le n-1$. Each a_i is a variable, and there are n of them. You can think of these variables as a (Java[1]) *array* of length n. Such an element $(a_{n-1},\ldots,a_1,a_0)$ of A^n is called an n**-tuple**. We call each of the a_i a **component** or **coordinate** of the n-tuple. Sometimes we call a particular a_i the **ith** component of a. Suppose that $a \stackrel{\text{def}}{=} (a_{n-1},\ldots,a_1,a_0)$ and $b \stackrel{\text{def}}{=} (b_{n-1},\ldots,b_1,b_0)$. Then, by definition, $a = b$

[1] In Java, a_i would be written `a[i]`

if and only if $a_i = b_i$ for each $0 \leq i \leq n-1$. Informally, two n-tuples are equal if their components are equal taken in order. We shall sometimes refer to an n-tuple in A^n as a **sequence** of elements from A, and write $a_{n-1} \ldots a_1 a_0$ instead of $(a_{n-1}, \ldots, a_1, a_0)$. This is just a convenient shorthand for the actual mathematical notation $(a_{n-1}, \ldots, a_1, a_0)$. We shall use a sequence notation for tuples when we study digital electronics. The element $10001101 \in \mathbb{B}^8$ is a sequence of binary digits and is an example of a binary number—see Chapter 3.

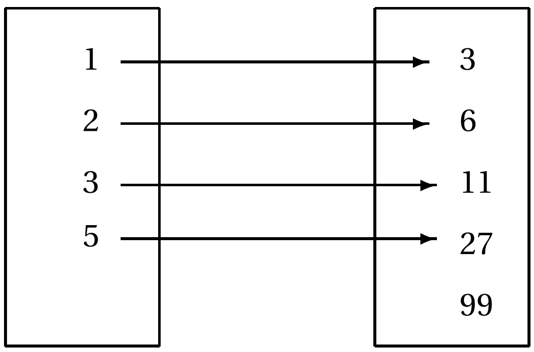
Exercise 1.3.1 Let $G \stackrel{\text{def}}{=} \{3,5,7\}$. Write down some examples of 4-tuples.

1.4 Functions

The notion of a function pervades mathematics and computer science. Before we define functions, we give an informal introduction. A function is a mathematical model of a machine (such as a computer), which takes an input (for example an integer) and produces a single output (such as the integer squared). We often give a name to a function, such as s. If s is our squaring example, then an input 3 to s gives output 9. The pair $(3,9)$ is an example of an *input/output* pair. Suppose that the inputs z are selected from the integers $\mathbb{Z}$. The outputs are also (non-negative) integers. We can describe s by saying that an input/output pair is given by (z, z^2) for every $z \in \mathbb{Z}$. In order to define a function we always give

- A set from which inputs can be drawn (eg $\mathbb{Z}$);
- a set containing all of the outputs (eg $\mathbb{Z}$);
- a set of input/output pairs which describes the function (eg $\{ (z, z^2) \mid z \in \mathbb{Z} \}$).

A function always satisfies the property that for *every possible input* there is *always an output* and *only one output*. Let's have a look at another example. $\{1,2,3,5\}$ is the set of inputs, $\{3,6,11,27,99\}$ the set of outputs, and G is the name of the function. We often write $G\colon \{1,2,3,5\} \to \{3,6,11,27,99\}$ to indicate this. For each input i the output will be i^2+2. This can be illustrated thus:



We can describe the function G mathematically as the set of (input/output) pairs

$$G \stackrel{\text{def}}{=} \{ (1,3), (2,6), (3,11), (5,27) \}$$

Notice that as the first components of the pairs come from the set of inputs, and the second components from the set of outputs, it follows that

$$G \subseteq \{ 1,2,3,5 \} \times \{ 3,6,11,27,99 \}$$

Exercise: Check this is correct. We also have other notation for writing down input/output pairs—some of this notation is used in programming! Instead of writing $(2,6) \in G$ we sometimes write $G(2) = 6$. Sometimes we might also write $G\,2 = 6$. These are all different ways of writing the same thing—the input 2 to G yields output 6.

Now we give the formal definition; hopefully this will be understandable given the previous discussion. For sets A and B we say f is a **function** when

- $f \subseteq A \times B$, and
- for all $a \in A$ there is a unique $b \in B$ for which $(a,b) \in f$.

The first statement is a precise way of saying that for any input/output pair (a,b) in f, the input a must be an element of A ("all inputs come from A"), and the corresponding output b must be an element of B ("all outputs come from B"). The second statement says that for each input a, we will *always get* the *same (ie unique)* output b.

We often write $f(a)$ for this unique $b \in B$. We sometimes refer to f as a set of **input/output** pairs, where a is the input to the function f and $f(a)$ is the (unique) output. Occasionally we write $f\,a$ instead of $f(a)$. We call A the **source** of the function f, and B the **target** of f. Note that we sometimes say that f is a function **between** A and B, or from A to B. If we are given sets A and B, and a function f between A and B, we write $f : A \to B$ to indicate this—regard f as acting on each $a \in A$ and returning a value $f(a) \in B$. You might like to think of f as a computer program which takes values in A as inputs, and returns values in B as outputs. We sometimes refer to the a in $f(a)$ as an **argument** of the function f.

Examples 1.4.1

(1) We described the function $s : \mathbb{Z} \to \mathbb{Z}$ given by $s(z) = z^2$. Notice that as the square of any number is non-negative, we can also write $s : \mathbb{Z} \to \mathbb{N}$ given by $s(z) = z^2$. *Exercise: Should we still call the function s? Should it have a new name?*

(2) There is a function $c : \mathbb{Z} \to \mathbb{Z}$ given by $c(z) = z^3$. c stands for the set of input/output pairs,

$$c = \{ (z, z^3) \mid z \in \mathbb{Z} \}.$$

Exercise: Is it true that $c : \mathbb{Z} \to \mathbb{N}$? If not, why not?

(3) Multiplication on the rationals is a function $M : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$, where $M((n/m, n'/m')) = n * n'/m * m'$ and the notation $x * y$ denotes, as usual, integer x multiplied by integer y.

(4) Consider the following set g of input output pairs

$$\{(3,8),(3,7),(9,10),(8,300)\}$$

This is not a function $g: \{3,8,9\} \to \{7,8,10,300\}$. Why? Because for the input 3 the output is not unique.

(5) Let $A \stackrel{\text{def}}{=} \{3,4\}$ and $B \stackrel{\text{def}}{=} \{4,7,6,\text{‘x‘},\text{‘y‘}\}$ and $f \stackrel{\text{def}}{=} \{(3,4),(4,\text{‘x‘})\}$. Then $f: A \to B$ is a function from A to B. *Check this!*

(6) Let $A \stackrel{\text{def}}{=} \{3,4,5\}$ and $B \stackrel{\text{def}}{=} \{4,7\}$ and $f \stackrel{\text{def}}{=} \{(3,4),(4,7),(5,4),(9,10)\}$. Then $f: A \to B$ is not a function from A to B because f is not a subset of $A \times B$. *Check this!*

(7) Let $A \stackrel{\text{def}}{=} \{3,4,5\}$ and $B \stackrel{\text{def}}{=} \{4,7\}$ and $f \stackrel{\text{def}}{=} \{(3,4),(4,7)\}$. Then $f: A \to B$ is not a function from A to B because there is no output specified for the input 5.

The definition of a (mathematical) function says, informally, that for each (single) input there is just one ouput. Our conceptual model was a machine which takes an input and produces an output: for example, a computer might take an integer as input and produce twice the integer as output. This model is quite limiting. What about a computer which takes three numbers as “inputs” and gives two “outputs” consisting of the sum and product of the three integers? We “feed in” n, m and l, and get back $n+m+l$ and $n*m*l$. We can model the computer c as a function with source $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ and target $\mathbb{Z} \times \mathbb{Z}$, that is

$$c: \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z} \times \mathbb{Z}$$

which can also be written as $c: \mathbb{Z}^3 \to \mathbb{Z}^2$. We call this function a $3 \mid 2$-ary function, indicating its input is a 3-tuple, and output a 2-tuple. The input $(1,2,1)$ is mapped to the output $(4,2)$.

More generally, we call a function of the form $f: A^n \to A^m$ an $n \mid m$-ary function. Given an input tuple $(a_{n-1}, a_{n-2}, \ldots, a_0)$ we call the a_i **input components** because they are the components of the input (tuple). In the remainder of this module, we shall study functions of the form $f: \mathbb{B}^n \to \mathbb{B}^m$ where $\mathbb{B} = \{0,1\}$ so please ensure you understand this important topic. *Digital circuits* can be modelled by such $n \mid m$-ary functions.

Exercise 1.4.2 A typical example of the effect of c is $c(3,4,5) = (12,60)$. The input tuple is $(3,4,5)$ and 3, 4 and 5 are the input components. *Exercise: What is the output tuple, and what are the output components?*

Example 1.4.3 We can define a function $f: \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z} \times \mathbb{Z}$ by specifying $f(n,m) \stackrel{\text{def}}{=} (n, n+m)$. Thus $f(10,100) = (10,110)$. *Exercise: If $f(2,5) = P$, what is P? Exercise: f is an $x \mid y$-ary function; what are x and y?*

We finish this section with a little more notation. Suppose that $f\colon A \times A \to A$ is any $2 \mid 1$-ary function. Given an input (a, a'), we have so far written $f(a, a')$ to denote the output. There are two other ways of denoting the output, and each is frequently used in computing

Descriptive Name	Notation
prefix	$f(a, a')$
infix	$a\ f\ a'$
postfix	$(a, a')f$

Note that addition is an example of a function of the form $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ whose outputs are usually written using infix notation. Input (a, a') gives output $a + a'$. Computer architects often call a and a' the **operands** of the function.

1.5 Binary Numbers

We give an extremely brief introduction to binary numbers which will be useful when reading Chapter 2, which comes before the main chapter on this topic. A **binary number** is a sequence of 0s or 1s, such as $1001^{\,bin}$, $111^{\,bin}$, $10101^{\,bin}$. Such a binary number denotes an integer. Starting from the right, each digit stands for a power of 2, namely $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, and so on. The integer denoted by $111^{\,bin}$ is $1+2+4 = 7$. $10101^{\,bin}$ denotes $16+0+4+0+1 = 21$.

2
Top-Level Computer Organization

2.1 A Brief History of Computing

Please see pages 13 to 24 of Tannenbaum, and the first chapter of Hennessy and Patterson. This material is non-examinable, but useful to know.

2.2 Overview of a Modern Computer

A modern computer consists of *hardware* and *software*. In this course, we concentrate on learning about the hardware, and certain aspects of the software. So

$$\boxed{\text{Computer} = \text{Hardware} + \text{Software}}$$

We begin with a few words which lead to a definition of software. A digital computer is a machine which performs a task by carrying out a sequence of instructions. The task is often to solve some kind of problem; a very simple problem might be to add up the first 100 natural numbers, and a difficult problem might be to provide a realistic flight simulator. The sequence of instructions is called a **program**. The instructions might be very simple, such as *subtract* 1 or *check if a number is strictly greater than* 0. Here is an example of a simple program which will be *explained in the lectures*.

```
        a := 0;
        c := 10;
L       a := a+c;
        c := c-1;
        if c >= 1 then L;
        exit;
```

A computer has a collection of very primitive instructions, known as a **machine language**. These instructions will be written in a specified *syntax*. Above, `a := 0` is a typical such instruction. A different syntax might be `a BECOMES 0`. Each instruction has a *semantics* which is a description of what happens when the instruction is run on the computer. A more formal word for "run" is **executed**. The letter `a` stands for a *memory location* which is a small part of a computer used to store data. When `a := 0` is executed, 0 will be stored in `a`. When `a := a + c` is executed, the values stored in `a` and `c` will be added, and the sum stored in `a` (thus the contents of `a` will alter). An **algorithm** is a set of instructions describing how to do something. A **computer program** is a particular representation of an algorithm within a computer. We say that the program **implements** the algorithm. In fact the program above is just one way we

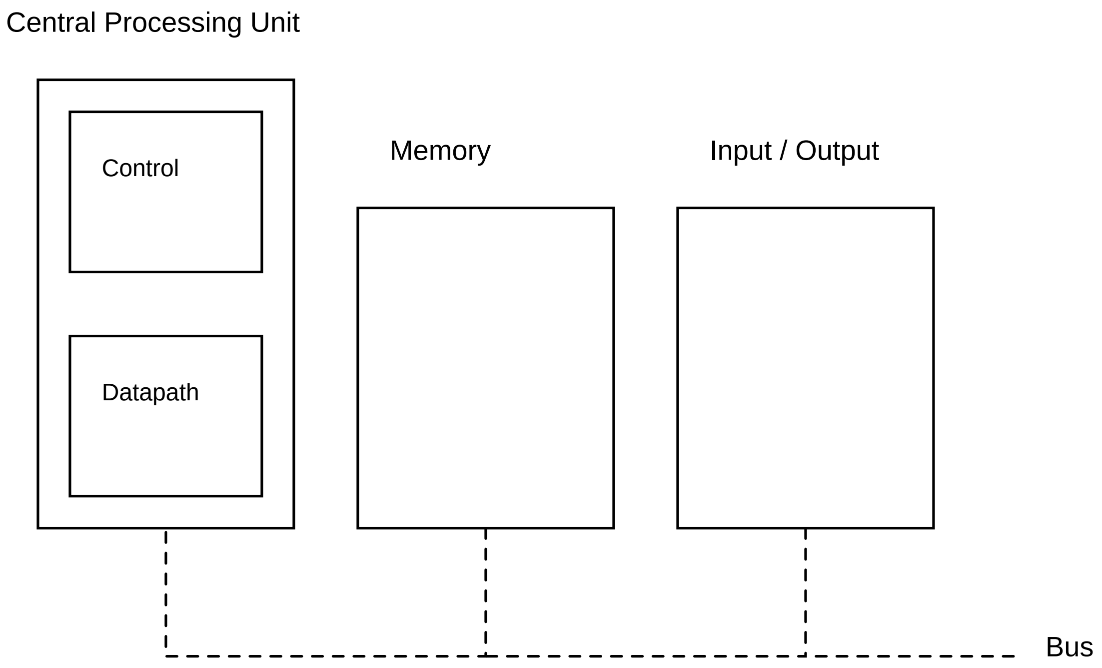


Figure 2.1: The Main Hardware Components

can represent an algorithm for adding the first ten numbers. The algorithm is: begin with 0, add 10, then 9, then 8, and finally 1. The final sum is stored in a. **Software** refers to the actual implementation of an algorithm on a computer. For example, if the program above is stored on a floppy disc we would refer to the stored program as software.

A computer is made up of physical components known as **hardware**. Examples are memories, cables, printer, integrated circuits, monitor and so on. We shall talk about some of these things in detail later on. For now, we describe hardware in very general terms. At its simplest, hardware can be seen as three top-level components, given in Figure 2.1. These are **Memory**, **Input/Output Devices**, and the **Central Processing Unit** or **CPU**. Note that the CPU is a combination of two sub-components, namely Control and the Datapath. We will give precise definitions of these components in later chapters. We often simply refer to the CPU as the **processor**.

$$\boxed{\text{Hardware} = \underbrace{\text{CPU}}_{\text{Control + Datapath}} + \text{Memory} + \text{I/O}}$$

At its simplest, how does a real computer work? We answer this by drawing an analogy between the main components (described above) and a more familiar real life situation. The analogy involves Dr. Doitall, who goes to MFI and purchases a do it yourself kit for a new bookcase. The analogies are

- Control $\leftrightarrow$ Dr. Doitall;
- Datapath $\leftrightarrow$ Dr. Doitall's toolbox;
- Memory $\leftrightarrow$ workshop storage space;
- Input $\leftrightarrow$ buying the MFI kit, Output $\leftrightarrow$ producing the new bookcase; and
- Program and Data $\leftrightarrow$ the MFI kit instruction manual and parts.

To get a computer to perform a task, we *Input* data, and a program; this corresponds to the bookcase parts, and instruction manual (ie the MFI kit). The computer needs to store and organize the data and program in *Memory*; this corresponds to opening up the MFI kit, and laying the parts and instructions out in a workshop. The computer then needs a way of reading the program instructions, one by one, and performing (*Control*ing) the tasks indicated; this corresponds to Dr. Doitall reading the manual, and building the bookcase step-by-step. The execution of the individual program instructions requires the selection of various simple operations, such as addition and subtraction, together with certain operands, and performing the arithmetic (these tasks are performed by the *Datapath*); this corresponds to Dr. Doitall selecting various tools, and using them to build parts of the bookcase. Finally, the *Output* corresponds to the completion of the bookcase.

In the remainder of this chapter, we will give more details of some of the main components, although in this course we do not look at input/output in any detail, nor large scale memory components.

2.3 The Central Processing Unit

2.3.1 Central Processing Unit Outline

At a superficial level, a computer works by performing very simple arithmetic operations on numerical data. That's it! So how does a computer manage to display extremely complex graphical images, for example? Ultimately, everything that happens can be seen to reduce to simple arithmetic and numerical data. For example, graphical images are built up from thousands of small dots of light. The position and colour of the dots (which change many times a second to give the eyes an impression of a smoothly changing image) are all given inside the computer as simple numerical quantities. The changes are calculated using complex arithmetic—but ultimately these complex calculations reduce to many, many simple ones, such as adding two integers.

Mathematicians are able to show that in order to compute a wealth of complex functions on just the natural numbers, all you need are some very simple functions (such as just adding 1!) from which all the complex ones can be built (a typical building construction might be the composition of functions). Using just the natural numbers, we can then define mathematically the integers, the rationals and the reals. So, in principle at least, if we can make a computer calculate the simple functions described above, we can then go on to make it perform some very complex calculations. And if we can represent many real life situations using things such as graphical images, it follows that a computer can do quite a lot for us.

As we have seen, the CPU consists of control and a datapath. The CPU executes a computer program stored in main memory—it can be thought of as the "brain" of the computer. It will execute a program by using control to fetch the program instructions, one by one from the main memory, executing each one using the datapath before fetching another instruction. By **fetching** an instruction we mean that a *copy* of the instruction which is held in main memory is made within the datapath. When the instructions are fetched, copies are stored within the datapath in special memory components called **registers**. For the time being, think of a register as a container that may hold any kind of data. A register has a **name** such as R. The **data** may be a number, a letter, or even a program instruction. We call the data the **contents** of the register. The datapath usually contains a small, fixed number of registers; 32 or thereabouts is typical. The instruction currently being executed is stored in a special register called the **Instruction Register (IR)** . The next instruction to be executed is specified by the contents of a register called the **Program Counter (PC)**. The PC *does not* store the next instruction. The contents of the PC contains an "address" which tells us where to find the next instruction in main memory. The next instruction will then be fetched into the IR. Many of the other registers are used for different tasks at different times; they are called **general registers**.

Note that the components in Figure 2.1 are joined by a bus. A **bus** is a collection of wires arranged in parallel, along which all kinds of signals (data, instructions etc) can be passed. Busses not only join the main components, but also the smaller components of a computer. For example, the circuits making up a datapath will themselves be joined by busses. We will discuss busses in greater detail in Chapter 4, when a number will come along all at once.

2.3.2 Machine and Assembly Instructions

The instructions which will be executed by the datapath can be presented in two forms. The first is a symbolic form which programmers can understand. An example instruction is $\mathtt{r} := \mathtt{a} + \mathtt{b}$ or, in a different syntax, `add` R, A, B. Symbols denote addition, and the registers used when the instruction is executed. This kind of notation is an example of **assembly language**.

We shall see later on that a computer stores data by representing it by using two symbols, usually denoted by 0 and 1. Inside the machine, these symbols are recorded

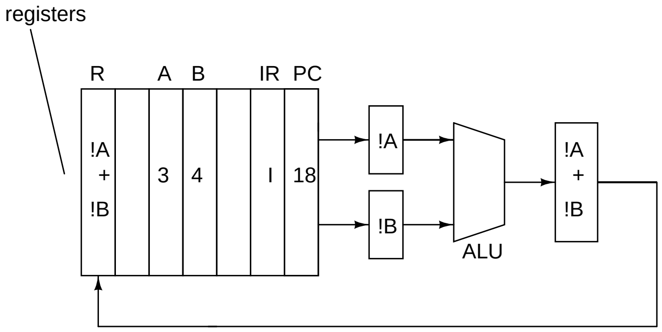


Figure 2.2: A Simple Datapath

as low and high voltages. Thus, in order for the computer to store the assembly language instruction, we have to represent it using 0s and 1s. The computer will record each "part" of an instruction as a sequence of 0s and 1s, for example `add` might be 00111, R might be 0001, and A and B might be 0011 and 0100. The instruction will then be represented by 00111000100110100, which is an example of a **machine language instruction**. We shall explain these ideas in much greater detail later on in the course.

2.3.3 The Datapath

The key role of the datapath is to execute an individual instruction. A very simple datapath is given in Figure 2.2. It has a program counter, and an instruction register. The IR contains an instruction I which is about to be executed; we call it the **current** instruction. Inside a real machine this will be stored as machine language. Suppose here that I is `add` R, A, B and it is the 17th instruction in a program. Here, A, B and R are general registers. The datapath has an **ALU** or **arithmetic logic unit**. This is a circuit which can perform functions such as addition. The rectangles to the left and right of the ALU are also registers. Note that we write $!A$ to denote the contents of A.

We drew an analogy between Dr. Doitall's toolbox, and the datapath. It is the job of the datapath to actually execute the tasks given by the instruction I's semantics. The *syntax* of our example instruction is `add` R, A, B. The *semantics* of the instruction is to pass the contents of A and B to the ALU, use the ALU to calculate the sum, then store the result in a register R. *Note that these steps occur in stages, and in the figure we see the state of the datapath at the end of all steps; we will give more details in the lectures.* In the figure, the contents of R will actually be $3+4=7$. The contents of the PC gives the main memory "address" of the next instruction—more details later on!

2.3.4 Control

You should read pages 56 to 60 of Stallings. The key purpose of control is to execute a complete computer program, making use of the datapath to execute each individual instruction. In general, control will perform a sequence of small steps, together called the **fetch-decode-execute (FDE)** cycle. These are

- fetch an instruction from main memory, and store it in the IR;
- update the PC to show where to find (in main memory) the next instruction;
- determine the instruction category (explained in lectures);
- execute the instruction; and
- repeat the FDE-cycle.

These steps give an algorithm for control. In fact the collection of FDE-steps is rather like a simple program, consisting of a sequence of "instructions". In fact we can write a program whose instructions are precisely the steps of the FDE-cycle. We can illustrate this with the simple JAVA method (program) given in Figure 2.3. We call such a program a **microprogram**. Thus its purpose is to execute other programs written by programmers!

2.4 An Overview of Computer Memory

2.4.1 Introduction

We are familiar with the idea of storing things. For example, an office will have a number of filing cabinets, in which documents are kept. The filing cabinets will have labelled draws, and may also be lockable. If we wish to find some document, we look at the labels on the draws to find its location, open the draw, and extract the document. The key concepts here are

- *storage facilities* (filing cabinets)
- *locations for storing objects* (draws)
- *addresses of locations* (draw labels)
- *contents of the locations* (documents)

Computer inboard memory works along similar principles, but instead of storing (paper) documents, stores "information". How does it do this? We will see in detail, later on, how all kinds of information can be represented as sequences of 0s and 1s. A computer can store 0s and 1s as low and high voltages. A **bit** is a small device in a computer which can "store" a 0 or 1 by producing either a low or high voltage. We call 0 and 1 **binary digits**. You can think of a bit as a "box" in which a single binary digit

```
public class microprogram {
         .
         .
         .
  public static void microprog (int memory[], int starting_address) {

  PC = starting_address;

    while (run_bit) {

      instr = memory[PC];
      PC = PC + 1;
      instr_category = get_instr_category(instr);
      data_loc = find_data(instr,instr_category);
      execute(instr_category,data);

    }

  }
         .
         .
         .
}
```

Exercise: *Fill in the missing parts of the JAVA code.*

Figure 2.3: A Simple Microprogram, written in Java

can be stored. Books often confuse bits and binary digits—in the lectures we shall see why it is useful to keep a clear distinction between the two. Such "0/1" information is usually stored in an electrical or also magnetic form, and we shall give further details later on. Computer inboard memory consists of collections of locations. Each location is a short sequence of bits in which data can be stored. So a location might consist of just a single bit (rather like a draw with just one document) or a finite number of bits regarded as a single entity (one draw, many documents). These groups of bits are known as **cells**. If there are k bits in a group, we refer to a k-bit cell. In practice, k is very unlikely to be 1, and is often 8 or 32. Each cell has an address so that we are able to find it (just as draw labels allow us to find the draw we are looking for). In a computer an **address** is given by a number. Often addresses are numbers which are obtained by physically counting up the cells in a particular computer memory.

Although there are many forms of memory, such as floppy discs, CDs etc, all memories have certain characteristics, and we look at these now.

2.4.2 Characteristics of Memory Systems

Memory Hierarchy Memory can be divided into two kinds. These are known as **primary memory** and **secondary memory**. Below, (i) and (ii) describe the former, and (iii) and (iv) the latter. Note that primary memory is sometimes called **inboard memory** because all the data is stored inside the computer's circuit boards. We shall describe the so-called "memory hierarchy" in detail quite shortly. For the time being it is enough to know that it is made up of the memories in the following list.

(i) **CPU memory**, where data is stored within the CPU inside actual computer circuits. Some of these circuits are registers, which we have already met. The other kind of CPU memory circuit is known as a cache, and we will say more about this later on.

(ii) **Main memory**, where data is stored inside other computer circuits, known as **memory chips**

(iii) **Outboard memory** consists of storage which is often located on the computer, such as **magnetic discs**, **compact discs**, **digital versatile discs** and so on.

(iv) **Off-line memory** consists of storage which is often located away from the computer, such as **magnetic tape**.

Capacity This is a measure of how big or small the memory is. A memory will have a certain capacity, which tells us how much data it can store. It will either be measured as a number of bits or a number of bytes—a byte is eight bits. We sometimes refer to a **bit count** per chip. We also talk about the **density** of the bits on a chip. Obviously the greater the density the greater the capacity of the chip.

Unit of Transfer This is a measure of how much information can be moved into or out of memory, each time data is read or written. For example, the unit of transfer might be a byte of data.

Method of Access There are different ways in which the units of data transferred out of memory can be found.

(i) **Sequential Access** Data is stored in a linear sequence, like books in a long corridor. To access the data you need, an accessing mechanism must bypass other data which is not required.

(ii) **Direct Access** Data is grouped into blocks, rather like books on separate library shelves. An accessing mechanism can access a block of data directly, but access the block sequentially.

(iii) **Random Access** Each transferable unit of data can be accessed at random, with no need to "bypass" redundant data.

Performance This refers to the time taken to find and retrieve data from a memory. We do not study this in detail.

Physical Type Memories are constructed in a variety of ways, and employ different methods for storing data. Some methods make use of **semi conductors**, **magnetic materials** and **optical materials**. We look at some of these in detail later on.

Physical Characteristics We call a memory **volatile** if data is lost when power is switched off; and **non-volatile** if data is kept when power is switched off. We call a memory **erasable** if data can be removed when required; and **non-erasable** if data is permanent.

2.4.3 The Memory Hierarchy

When designing a computer memory, three constraints are important. These are *capacity*, *performance* and *cost*. Better performance usually means higher costs per unit of data stored. So, for example, if we need high capacity memory, we may have to reduce our costs by lowering overall performance. Different physical memory types provide memories with these constraints in varying proportions. A typical modern PC must run programs quickly, but must also have some memory of high capacity. This is achieved by using different types of memory within the whole system. The list of memories given on page 15 is a **memory hierarchy**, with registers at the top, and tapes at the bottom. As you move down, the following occur:

- decreasing cost per unit of storage;
- increasing capacity;

- increasing access time;
- decreasing frequency of access of the memory by the CPU.

How do we use the hierarchy to ensure we can build a fast PC with large capacity memory at a good price? In more detail, we want a PC which will run a program quickly, but which can also store a large amount of data. The key here is that many programs have a fairly typical behaviour, namely that small groups of program instructions are often executed repeatedly. We refer to these groups of program instructions as “clusters”. The idea that program instructions often appear in such clusters is known as the **locality principle**. So at any point in time, we ensure that the instruction cluster, whose individual instructions are about to be repeatedly executed by the CPU, is held in memory found at the hierarchy top, that is fast access memory. All of the remaining instructions, which (for the time being) the CPU is much less likely to require, are stored lower down in slower access memory.

This fast memory is usually called a **cache**. It is usually physically located near to the CPU. When a cluster is finished with, it is **swapped** for a new cluster which is copied into the cache. In this way, most instruction access by the CPU is from fast cache memory, ensuring that program run time is small.

2.5 Primary Memory

2.5.1 Main Memory Details

In this section we describe main memory in some detail. Suppose that a computer memory has n cells. By convention, these will have addresses 0 to $n-1$, and adjacent cells are numbered consecutively. Also, all cells will contain the same number of bits. If a memory contains n bits in total, we call it an **n-bit memory**. Figure 2.4 gives two examples of a 96-bit memory. The first example is a memory layout of 12×8-bit-cells; the second example gives a memory layout of 3×32-bit-cells. These examples give memory addresses as decimal natural numbers. Note that a computer will need to refer to memory addresses, and thus the addresses themselves must be stored! In a computer, an address is actually stored as binary natural number a (see Section 1.5). The machine will store the address by using a sequence of bits to represent each of the binary digits in a.

The cell is significant as it is the smallest *individual, addressable* part of a memory. Given any cell, it is composed of a finite number of bits, by definition. We refer to the sequence of binary digits contained in these bits as the **contents** of the cell. Note that a k-bit cell can hold 2^k different binary digit sequences; *Exercise: why is this?*.

Most computers have a standard 8-bit cell. Each such cell is called a **byte location**. Groups of consecutive cells are called **word locations**. Typically, a word location will be 4 byte locations long, and hence consist of 32 bits, but many other possibilities

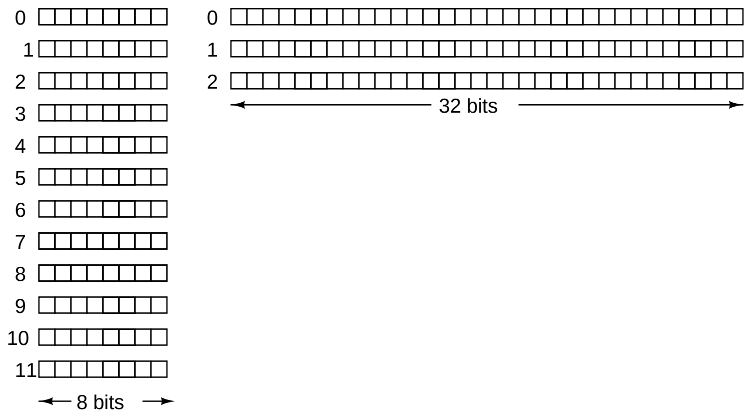


Figure 2.4: Memory Cell Organization

exist.

By definition, the **address of a byte location** is precisely the address of the corresponding 8-bit cell. The **address of a word location** is given by the smallest address of the cells making up the word location. A computer **location** is a general term for an "area of memory" which is used for storing data. Each location has an address, and some means of identifying the size of the location. Thus registers, cells, byte locations and word locations are all examples of the general concept of a computer location—each has an address, and a given size. A register address is usually called a **register number**. In particular, we refer to the **contents** of a byte location to mean the contents of the corresponding cell, and the **contents** of a word location to mean the sequence of binary digits given by the contents of the corresponding cells. We call the contents of a byte location a **byte**, and the contents of a word location a **word**. Note that by definition a byte is simply a sequence of eight binary digits.

Word locations are especially significant. In practice, an integer is typically stored in a fixed amount of memory, usually a word location. This imposes a limit on the size of integers which can be stored. Note also that individual program instructions will also, typically, be stored in word locations. Thus if a computer has n-bit word locations, we often refer to it as an **n-bit computer**.

We have said that word locations are groups of cells. How do we arrange cells into word locations? Recall that cells are fixed locations in memory, and the addresses increase by 1 as we move from one cell to the next. Let's fix on word locations composed of *four* cells as an illustration. To arrange the cells into consecutive word locations, we can group consecutive cells into blocks of four. But how do we do this? The first cell has address 0. The second cell could then be "placed" to the left *or* right of the

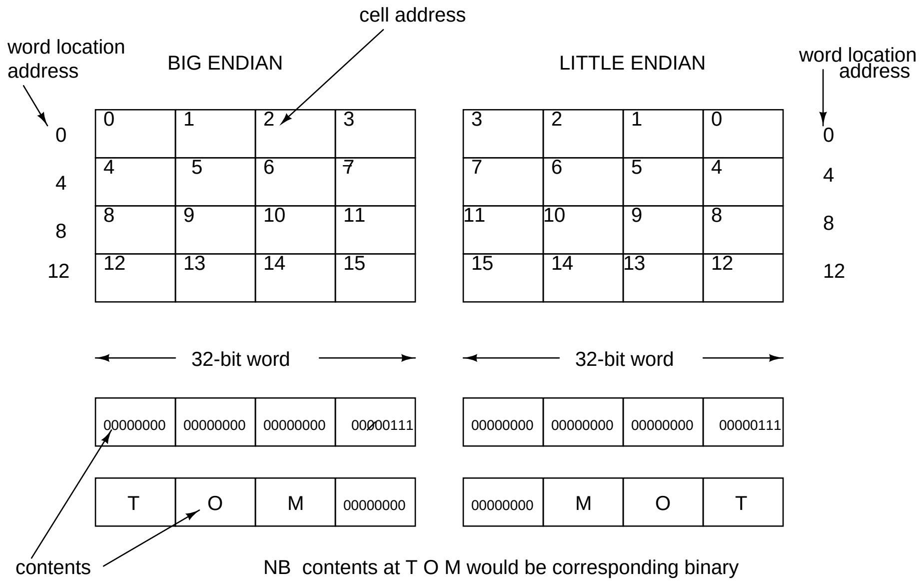


Figure 2.5: Endian Byte Location Ordering

first. And then the third cell could again be “placed” to the left *or* right. Suppose that we picture each word location as a horizontal group of four cells—remember, memory is not like this physically, but such a picture is a useful descriptive tool. In general, the cells in any word location can then be addressed from left to right, or right to left.

Figure 2.5 shows part of the memory of a 32-bit computer; in the figure, word locations are composed of four cells, and each cell has eight bits (so each cell is in fact a byte location). The left to right numbering is called **big endian**, and the right to left **little endian**.

Big endian is used by some IBM mainframes, and SPARC stations. Little endian is used by Intel. Within a single computer, it does not matter which system we choose, providing that the designer chooses just one system and builds the machine correctly. However, problems can arise when data is transferred from a big endian to a little endian computer. To see this, we first examine in detail how various kinds of data are stored in memory.

We mentioned that integers are often stored in word locations. If the word locations consist of four byte locations, then a sequence of 32 binary digits will be used to store any particular integer. These digits will always be recorded in a word from left to right. Thus, with reference to Figure 2.5, if we store 7 as $\vec{0}111^{bin}$ (where there are 29 zeros) then the 1s will appear in byte locations 3 or 7 or 11 or 15 in the big endian system, and 0 or 4 or 8 or 12 in little endian.

Characters are stored in cells, not word locations. For example, each of the characters A to Z will be denoted by an integer, known as a **character code**. Each code can be stored in a cell, by using the integer's binary representation. There are various systems in use for assigning character codes. A well known one is **ASCII**, standing for *American Standard Code for Information Interchange.* In fact each ASCII code requires 7 bits, yielding 128 characters. The letters A to Z are represented by the codes 65^{dec} to 90^{dec}. An English word such as TOM will be stored by placing T, O and then M in consecutive cells, using a binary representation of the ASCII codes.

Sometimes we want to transfer data from one computer to another. There is a potential problem with this. A naive way to move data in a big endian machine to a little endian machine, is to transfer the contents of an individual cell to the corresponding cell with the same address. Then all characters will be moved correctly. But an individual integer is stored in a *word location.* Thus although the contents of each *cell* making up the word location will be transferred correctly, the order of the cells making up the word location will be *reversed.* Thus the integer will be scrambled. Examples will be given in lectures—try drawing pictures of examples.

Exercise 2.5.1 Write down a selection of examples involving character and integer transfers, and make sure you understand the problems. Also make sure you understand the differences between bytes, words, cells, byte locations and word locations.

Example 2.5.2 Suppose that the 32-digit word

$$01000000.00000000.00000100.00000001$$

is stored in a big endian machine in byte locations 0,1,2 and 3. The word denotes the integer $2^{30} + 2^{10} + 1$. It is moved to a little endian machine, with cell a moved to cell a. If integers are stored in word locations, what integer is denoted by the contents of word location zero of the little endian system? Is the integer copied correctly? What is the (binary) content of cell 3?

Answer: The new word location contains

$$00000001.00000100.00000000.01000000$$

which represents $2^{24} + 2^{18} + 2^{6}$. The integer is not copied correctly. Cell 3 holds 00000001.

2.5.2 Physical Types of Primary Memory

The primary memory found in a computer comes in a variety of forms. Registers and cache memory are typically made from a form known as SRAM, which is *one* of those described below. Inboard main memory can be constructed from *any* of the forms below.

- **Random Access Memory** (**RAM**) is memory that can be both read and written at random, through the use of electrical signals. (Note that the name is unfortunate; all of the main memories described here have random access in the sense of the earlier definition.) RAM is volatile, and comes in S and D types. RAM is provided in the form of small, plastic devices which slot inside computers; each device is known as a **chip**.

- **Static** RAM (**SRAM**) is made from circuits known as *D latches* or *flip flops*. We will describe these in detail later on. SRAM is very fast and is often used in processor registers and in cache memory, but correspondingly costly.

- **Dynamic** RAM (**DRAM**) is an array of transistors and capacitors. A capacitor can store a small electrical charge, and the presence or otherwise of this charge is interpreted as a binary digit. Capacitors can lose their charge, and so circuits are required to refresh the stored data every few millisecs. Because each DRAM bit is physically simple, it is also small and cheap. Being small we can get a very high bit count per chip—thus we can fit a lot of memory in a small space. Thus DRAM is often used for main memory, but it is also slow.

- The oldest RAM is **Fast Page Mode DRAM**. The individual bits are arranged in a physical rectangular fashion on the chip, and are accessed by providing a row address and column address.

- This has been replaced by **Extended Data Output DRAM**. Here, a second memory access can begin before the first has finished. This is a simple form of *pipelining* which means that more accesses per second can occur than in an ordinary system.

- Both of these kinds of RAM are asynchronous. This means that the control of memory references is not timed against a single clock. **SyncDRAM** or **SDRAM** is a hybrid of static and dynamic RAM, and is controlled by single clock. It has a number of technical advantages, and its use is increasing.

- **ROM** or **Read Only Memory** stores data specified at manufacture. The bit pattern is often made by a photosensitive masking process. ROMs are used for *microprogramming* (see later) and to store libraries of common functions and system operations. The advantage of a ROM is that stored data and programs need not be retrieved from a slow secondary device. *<u>Exercise:</u> Think up some disadvantages.*

- The data stored on a **PROM** or **programmable read only memory** is set once by a user or supplier. The PROM chip contains little fuses. The user can use electricity to blow some of the fuses, and this leaves a pattern of working fuses which represents the data to be stored.

- **EPROM** or **Erasable PROM** is similar to PROM, but can be programmed many times. Before each programming, the chip is exposed to UV light for 15 mins which resets all of the bits. The chip is then written to electrically, and the new data remains stable until another exposure to UV takes place.

- **EEPROM** is erased by electrical voltage, and can be reprogrammed in place. It is typically half as fast as EPROM. EEPROMs are 10 times slower and 100 times smaller than D/SRAMS. They are byte erasable, meaning that individual bytes can be erased and re-programmed, rather than the whole chip at once. However, they are expensive and not very dense.

- **Flash** memory is block erasable and writable, meaning that blocks of bytes can be reprogrammed instead of the whole chip, but not individual bytes. The erasure is very quick, and the chips are dense.

2.5.3 Cache Memory

We have already met the idea of a **cache**, which is extra fast memory found near, or on, the CPU. We will give a few more details concerning the operation of a cache. Recall the *locality principle*. If control requires an instruction from main memory, it is likely that the next few instructions are nearby. In fact control will check to see if the instruction is in the cache. If so, it will be fetched from the cache and brought into the CPU. If not, control will fetch the instruction from main memory *together with a cluster of other instructions found nearby (local)* into the cache, and place the (current) instruction into the CPU. We shall explain more precisely how this works.

Consider the cache in Figure 2.6. For simplicity, *we will assume that a program instruction can be stored in a single cell.*

Suppose main memory has 2^n cells. These are divided into **blocks** of K cells where 2^n is divisible by K. Thus there are $B \stackrel{\text{def}}{=} 2^n/K$ blocks, called block 0 , $1, \ldots (2^n/K) - 1$. The cache consists of a set of C **cache lines**. Each line is a memory location with a capacity equal to one block of memory. We often ensure that the number of blocks is divisible by the number of cache lines C.

Informally, the consecutive main memory blocks are copied into consecutive cache lines. When the cache is full (note that $C \leq B$) the process repeats itself. Data already in a cache line will be over-written with new data. Thus each cache line may hold the contents of a different block on different occasions.

In order to know which block is in a cache line, we use a *tag* which is simply the block number. The idea is that each memory block b is copied to a unique cache line with number l and tag b.

With some thought we can see that if an instruction I is held in the cell with address a, then

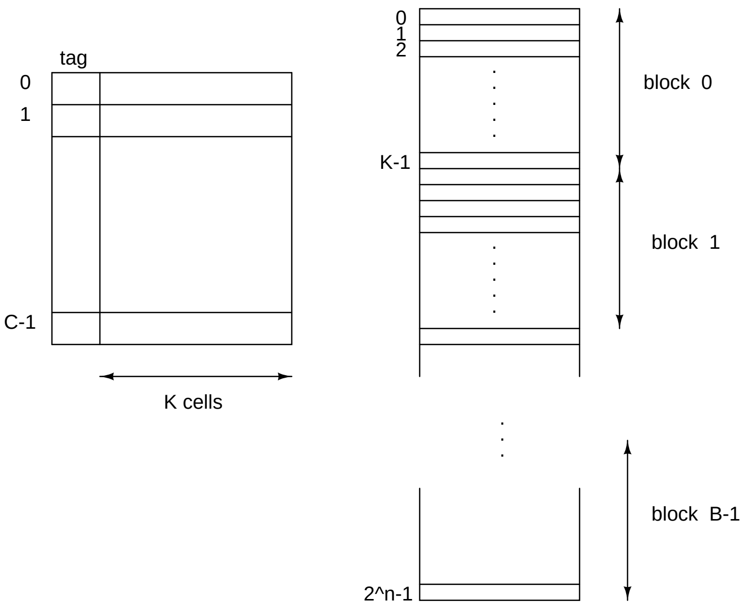


Figure 2.6: A Simple Cache

- I appears in block $a\ DIV\ K$;
- block b is copied into cache line $b\ MOD\ C$; and
- the first cell in block b has address $b * K$.

where *DIV* means *integer division* and *MOD* means *remainder*.

Example 2.5.3 Consider a cache with $n = 10$, $C = 4$, $K = 8$. The CPU requires cell 52 which is not in the cache. Into which cache line is the cell's block copied? What is the value of the cache line tag, and which cells are copied into the line?

Answer: Cell 52 appears in block $52\ DIV\ 8 = 6$, which is copied into line $6\ MOD\ 4 = 2$. The tag is 6. The cells copied in to the line have addresses $6 * 8 = 48$ to $48 + (K - 1) = 55$.

Exercise 2.5.4 Draw a picture of the last example.

2.5.4 Memory Packaging

In early times to early 1990s, most memory was supplied as single chip memories, from 1K bits to 1M bits. We now get 8 to 16 chips on a circuit board. These are either **single inline memory module (SIMM)** which have connectors on one board side, or **dual inline memory module (DIMM)** with connections both sides. A SIMM is depicted in Figure 2.7.

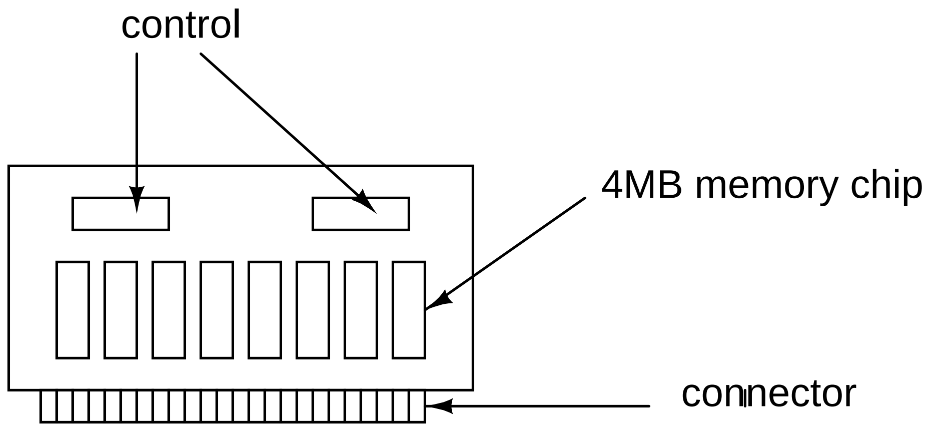


Figure 2.7: SIMM

2.6 Secondary Memory

Read Chapter 5 of Stallings; this is non-examinable. Additional information can be found on pages 69 to 88 of Tannenbaum.

2.7 Translation, Interpretation and Virtual Computers

Throughout this chapter we have been describing a computer by looking at the main hardware components. Our final task in this chapter is to describe a computer in terms of software. This will give us a very different but very useful perspective.

If you look back at the program in Section 2.2 you can see that it is composed of very simple instructions, such as $\mathtt{a} := \mathtt{0}$. These very simple machine (assembly) instructions can often be executed directly on a CPU—each machine instruction is executed by an electronic circuit. Designers often keep these machine instructions simple, as this can reduce the complexity and cost of the electronics. The downside is that the very simple instructions make it very tedious for people to write complicated programs. Such programs are likely to be long, because of the large number of simple instructions required to make up more complex ones. The length makes it difficult to understand what the program is supposed to do, that is, its semantics. There is potentially a gap between the ways people can describe problems easily, and a description of the problem which a computer can understand. This is better illustrated by an example. Consider our program again. Suppose that we want to multiply the first ten numbers. We could "do this" by changing $\mathtt{a} := \mathtt{a} + \mathtt{c}$ to $\mathtt{a} := \mathtt{a} * \mathtt{c}$. But if our computer does not have circuits for multiplication built in, we can't do this. One way of proceeding is to implement multiplication in terms of addition. Instead of writing $\mathtt{a} := \mathtt{a} * \mathtt{c}$ which our computer cannot recognize, we replace this "invalid instruction" with a sequence of valid instructions which perform the same task, by adding up `c` copies of `a`.

Exercise 2.7.1 Work out what this sequence might be.

Abstraction can be described as the process of building something new from smaller parts, and then ignoring the details of the parts. For example, a car designer needs to know the details of all the components. But a driver can deal with the car at a more abstract level: (s)he can learn to drive without detailed understanding of the clutch, engine etc. In computing, we have a similar situation. We might say that multiplication is more abstract than addition because we can define the former in terms of the latter. We can manipulate and understand multiplications without (necessarily) knowing the details of the associated additions. We might say that multiplication is a **higher level** instruction than addition. A programmer can be given an abstract programming language in which (s)he can deal with high level concepts. Providing the semantics of the high level instructions are understood, the details of the sequences of simple instructions they represent can be ignored. For example, consider

Language	Instructions
L	$I_1, I_2, I_3, I_4, +$
$L_{abstracted}$	$I_1, I_2, I_3, I_4, I_5, +, *$

Here I_5 means “run I_1 a hundred times” and $a * b$ will mean $a + a + \ldots a$ where a appears b times. A programmer begins with language L but soon finds that (s)he needs to run I_1 many, many times, and frequently needs to multiply numbers. Using $L_{abstracted}$ many programs will become much shorter.

We often refer to the less abstract language as a **low(er)-level** language, and the more abstract one as a **high(er)-level**. A real example, illustrating a very big abstraction, is JAVA (high-level) and Pentium-II machine code (low-level). The low level language has instructions which can often be executed directly by a CPU. The high level language has instructions, each of which can only be executed by running a corresponding sequence of low level instructions. So we need to understand in general terms how we can implement a high level language on a real machine.

Before we continue, look at Figure 2.8. This gives a very abstract view of a computer. Data together with a program are fed in, and an output is produced. The output is produced when the program is executed on the computer. Note that the input data can consist of almost any kind of information—including program instructions! We use this abstract picture to explain how we might implement a high level language. As the module progresses we shall explain in greater detail what data and programs are. For the time being, we shall assume the reader has an intuitive idea.

A high level language can be implemented in (at least) two ways, known as *translation* and *interpretation*. Suppose we have a computer C which can be programmed in a language L. We want to be able to program in a more abstract language which

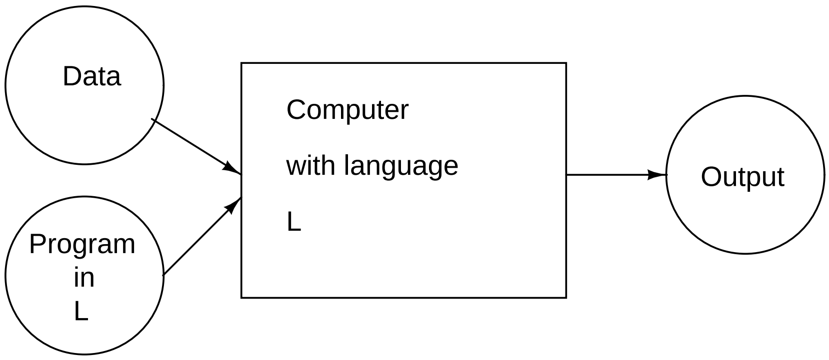


Figure 2.8: Abstract View of Computer Input/Output

we call L'. Roughly, any instruction in L' should correspond to a finite sequence of instructions from L. We can achieve this by

• Given a program $P_{L'}$ written in L', replace each of the $P_{L'}$ instructions by an equivalent sequence of L instructions to produce a new program P_L consisting entirely of L instructions. This is known as a **translation**. The translation is performed by an L program $Trans_L$ running on C, with $P_{L'}$ as data. This process is known as **compilation**. The resulting output, namely the program P_L, is then itself executed on C. We call P_L the **compiled program**. The time of the execution of P_L is known as **run time**. You will meet these ideas in MC103.

• Given a program $P_{L'}$ written in L', we run a program $Interp_L$ on C whose data is $P_{L'}$, which examines each of the instructions in $P_{L'}$ in turn and for each instruction immediately executes the corresponding sequence of L instructions. This process is known as **interpretation**.

Example 2.7.2 We have already seen a simple example of translation. We wanted a program to multiply the first ten numbers, but we only had a running language, L say, with addition and subtraction. We could see how to write such a program in a similar language, say L', which also had multiplication, but which was not implemented. We translated a program in L' into L by leaving all instructions alone (we might say we applied the identity translation) but translating the single multiplication instruction into a sequence of L instructions.

Exercise 2.7.3 Draw instances of the diagram in Figure 2.8 to illustrate translation and interpretation. In each diagram, think carefully about the what the input program and data are, and what the output is.

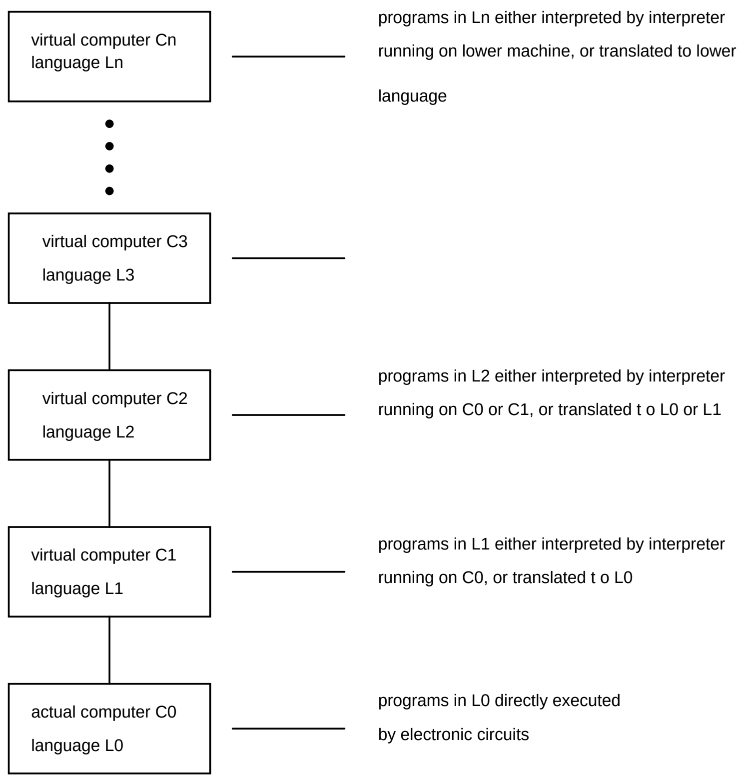


Figure 2.9: Layers of Virtual Computers

Suppose we have a computer C and a language L. C runs P_L programs. By writing either a translator or an interpreter, we end up with a new language L' in which we can write $P_{L'}$ programs and have them executed. This gives rise to a **virtual computer** (call it C') that will take data, and a $P_{L'}$ program as input, and produce output. *Exercise: Draw an abstract picture of these ideas*.

We might hope that the language L' could be "very" high level, and L' "very" low level, and hence L' could be easily implemented on an actual computer with just one translation. In practice, in turns out that L and L' have to be quite similar in order for L' to be translated into L in one step. We will see why this is so as the course progresses; for the moment, take it on trust that it is difficult to either translate or interpret a high level language directly into a low level one. We thus have to repeat the translation/interpretation process. Given any (virtual) computer C_n with language L_n, we can produce a virtual computer C_{n+1} with language L_{n+1}. We begin with $n = 0$ so that L_0 is a low level language, and produce a high level L_n which (hopefully) computer users are happy to use. This is illustrated in Figure 2.9. Note that C_0 is the physical computer itself, made from circuits, often called the **microarchitecture**. In a typical modern computer, we might find that n is about 5. Each level has a name:

(a) Instruction set architecture

(b) Operating System

(c) Assembly Language

(d) Problem Oriented language (eg Java, C, Pascal etc)

The **instruction set architecture (ISA)** level provides a language which is sometimes referred to as **machine language**. This is the language described in manuals such as "Machine Language Reference Manual for the Model-T Computer". Thus the manual describes instructions which are either directly executed by the microarchitecture circuits (direct hardware processor control), or are interpreted by a program (called the **microprogram**) running on the microarchitecture. In the former case, the ISA language corresponds to L_0. In the latter, the microprogram is L_0, and the ISA L_1.

The **operating system language (OSL)** level provides a language which allows new forms of control and organization. For example, this new language will allow two programs (written in another language) to run concurrently. The language contains some new instructions, but also contains many of the ISA level instructions as well. Thus the ISA language is a subset of the OSL. This operating system language is partially interpreted by a program running at the ISA level (that is, the interpreter is written in machine code): the interpreter deals with the new OSL instructions. However, an ISA instruction that is also an OSL instruction is executed as though it were a "normal" ISA level instruction. Note that some people call this (partial) interpreter the *operating system*, as distinct from the *operating system language*.

The first three levels are known as **system** levels. The levels above are called **application** levels. The former are for specialist programmers to design virtual computers which will support good higher level programming languages. The latter are designed for programmers to write code which solves real problems, or leads to programs which have "real applications" (such as *Tomb Raider* for example `:-)`).

The **assembly language** level is really just a symbolic notation for one of the lower levels (often the ISA level). It simply makes it easier for programs to be written in the lower level. Note that this was the essential basis for our different levels from the start—thus the operating system language has a rather special status. Programs written in assembly language are translated into one of levels 0/1,2 or 3 as appropriate (usually 0/1), and then executed by the corresponding actual/virtual machine.

Finally the **problem oriented level** provides languages which are used by applications programmers—Java is an example and there are literally hundreds of others, such as Haskell, C++, ML, Ada, Prolog, ...

3

Digital Arithmetic

3.1 Radix Number Systems

You will be familiar with the natural numbers $\mathbb{N} \stackrel{\text{def}}{=} \{0,1,2,\dots\}$ and the integers $\mathbb{Z} \stackrel{\text{def}}{=} \{\dots,-1,0,1,\dots\}$, where $\mathbb{N} \subset \mathbb{Z}$. In this course, we will be mainly concerned with the integers. However, for the first two sections of this chapter we will only consider integers which are either 0 or positive, that is elements of $\mathbb{N}$.

Each integer can be *represented* in different ways. For example, you will be familiar with *decimal* and *Roman* representation. If we write 9 and IX, although these are different symbolic representations, you understand the concept of an *integer quantity* of *nine*. Both 9 and IX *denote* the quantity *nine*. Again, 12 and XII both denote the quantity *twelve*; and *twelve* is *represented* by both 12 and XII. We could make this idea of representing integers a little more precise, by saying that a **representation** of $\mathbb{N}$ is given by a set of symbols S (such as {I,II,III,IV,... }) and a function $S \to \mathbb{N}$ which maps each symbol to the integer it denotes. If XII is the symbol input to the function, then the output integer is written $[\![XII]\!]$ where $[\![XII]\!] = \mathit{twelve}$.

We introduce *binary numbers* which are a symbolic representation for integers. The **binary digits** are 0 or 1. The symbols used to represent integers are *sequences of binary digits*—each sequence is called a **binary number**. Typical binary numbers are $10010^{\,bin}$ and $1001^{\,bin}$. Recall that $\mathbb{B} = \{0,1\}$ is the set of binary digits. Note that $10010^{\,bin}$ is by definition a sequence of five binary digits, so $10010^{\,bin} \in \mathbb{B}^5$. We often use d or d_i to stand for a binary digit. For example if $d_0 = 1$, $d_1 = 0$ and $d_2 = 1$, then $d_2 d_1 d_0^{\,bin}$ stands for $101^{\,bin}$. Note that there are *three* digits labelled using 0,1 and 2. We shall write a k-digit binary number as $d_{k-1}\dots d_0^{\,bin} \in \mathbb{B}^k = \underbrace{\mathbb{B}\times\mathbb{B}\times\dots\times\mathbb{B}}_{k}$.

Notice how careful we shall be with notation. The binary number $d_{k-1}\dots d_0^{\,bin}$ is a *sequence of digits*. It *denotes* an integer which will be written $[\![d_{k-1}\dots d_0^{\,bin}]\!]$. What is the integer? We proceed by example. The idea is that the digits stand for powers of 2. For example

$$
\begin{aligned}
[\![10^{\,bin}]\!] &= 1*2^1+0*2^0 = 2^1+0 = 2+0 = 2\\
[\![10100^{\,bin}]\!] &= 2^4+0+2^2+0+0 = 16+0+2+0+0 = 20
\end{aligned}
$$

A digit d_i stands for 2^i. In general we have

$$
[\![d_{k-1}d_{k-2}\dots d_0^{\,bin}]\!] = d_{k-1}*2^{k-1} + d_{k-2}*2^{k-2} + \dots + d_0*2^0 = \sum_{i=0}^{i=k-1} d_i*2^i
$$

Example 3.1.1 Calculate the integer denoted by $101010^{\,bin}$.

Answer: The integer is

$$[\![101010^{\,bin}]\!] \;=\; 1*2^5+0*2^4+1*2^3+0*2^2+1*2^1+0*2^0 = 32+8+2 = 42.$$

Now we shall look at how to generalize these ideas. The integer 2 is referred to as a **radix** or **base**. In general, any positive integer can be used as a radix. To represent an integer in a general radix r, we take a fixed set of r different symbols. Each symbol is called a **digit**. Each digit denotes an integer; we write $[\![d]\!]$ for the integer denoted by the digit d. In general, an integer is represented with radix r as a finite sequence $d_{k-1}\dots d_0{}^{\,r}$ of digits, where

$$[\![d_{k-1}\dots d_0{}^{\,r}]\!] = \sum_{i=0}^{i=k-1} [\![d_i]\!] * r^i$$

In binary we work with radix 2, and the set of two digits is traditionally $\{0,1\}$. We can write $[\![0]\!] \stackrel{\text{def}}{=} 0$ and $[\![1]\!] \stackrel{\text{def}}{=} 1$, and the formula above simplifies to the original one

$$[\![d_{k-1}\dots d_0{}^{\,bin}]\!] = \sum_{i=0}^{i=k-1} d_i * 2^i$$

In computing, we also make frequent use of *hexadecimal.* The radix in hexadecimal is 16. Thus we need 16 digits to represent integers in hexadecimal, and we choose the symbols $0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F$. The digits 0 to 9 denote the expected numbers (!) and the letters A to F denote 10 to 15. Thus we define $[\![0]\!] \stackrel{\text{def}}{=} 0$, $[\![1]\!] \stackrel{\text{def}}{=} 1, \dots$ $[\![E]\!] \stackrel{\text{def}}{=} 14$, and $[\![F]\!] \stackrel{\text{def}}{=} 15$. Thus the sequence of digits $B1A^{\,hex}$ denotes the number 2842:

$$[\![B1A^{\,hex}]\!] \stackrel{\text{def}}{=} \sum_{i=0}^{i=2}[\![d_i]\!]*16^i = [\![B]\!]*16^2+[\![1]\!]*16^1+[\![A]\!]*16^0 = 11*16^2+1*16^1+10*16^0$$

where, for example, $d_2 = B$ and $[\![B]\!] = 11$. *Check this!*

In decimal notation, the usual representation of integers, the radix is 10. The digits we use are 0, 1, … 9, which denote the obvious integers, namely $[\![0]\!] \stackrel{\text{def}}{=} 0$, $[\![1]\!] \stackrel{\text{def}}{=} 1$ etc. Thus, for example,

$$[\![25^{\,dec}]\!] = [\![2]\!]*10^1+[\![5]\!]*10^0 = 2*10^1+5*10^0 = 25$$

as expected!

Exercises 3.1.2

1. Calculate $[\![1AF3^{\,hex}]\!]$ showing your steps.

2. Work out how to convert a number into a representation using a radix r and sequences of r digits. Hint: First work out how to do this when r is 10 and the representation is the usual decimal one, as this is easy! Then move on to $r = 2$, and finally explain what to do for a general r.

3.2 Binary Numbers in Computers

We can represent an integer in binary notation within a computer, by having a finite collection of bits, each storing a binary digit. One thing to note here is that we can only represent integers n within a finite range. For example, suppose we have a sequence of eight bits. The smallest representable integer is clearly 0. The largest is represented by $11111111^{\,bin}$, that is

$$2^7+2^6+2^5+\ldots+2^1+2^0$$

This sum is equal to 2^8-1, so that $0 \le n \le 2^8-1$. (Check that the sum is indeed 2^8-1. We will explain why this is so in the lectures.) In general, if integers are represented using k bits in a computer, we shall say the computer uses **k-bit binary numbers**. We shall often write a typical k-digit binary number as $d_{k-1}\ldots d_0{}^{\,bin}$, and if the value of k is fixed, we may abbreviate this to $\vec{d}^{\,bin}$.

We call d_{k-1} the **most significant** digit, and d_0 the **least significant**. We also refer to most and least significant bits. Note that in a computer, binary numbers stored in k-bits may have zeros in the most significant bits. If $k \stackrel{\text{def}}{=} 6$, then the number 3 is represented by $000011^{\,bin}$ in the machine. However, we would write just $11^{\,bin}$ if we were not concerned with the issues of using k bits to represent the number in a machine.

We have one final definition. We call 0 and 1 **complements** of each other, and write $\overline{0} \stackrel{\text{def}}{=} 1$ and $\overline{1} \stackrel{\text{def}}{=} 0$. If $\vec{d}^{\,bin} = d_{k-1}\ldots d_0{}^{\,bin}$ is a binary number, then its **digitwise complement** is defined to be $\overline{d_{k-1}}\ldots\overline{d_0}^{\,bin}$. We will sometimes denote the digitwise complement of $\vec{d}^{\,bin}$ by $\overline{\vec{d}^{\,bin}}$ or just simply $\overline{\vec{d}}$.

Example 3.2.1 Work out the binary number representation of 9 in a machine with five-bit binary numbers. Give also the digitwise complement.

Answer: We first decompose 9 as powers of two; a little thought gives $9 = 8+1 = 2^3+2^0$. We model five bits by the set $\mathbb{B}^5$. Thus 9 is represented by

$$01001^{\,bin} \in \mathbb{B}^5$$

The digitwise complement, $\overline{01001^{\,bin}}$, is $\overline{0}\,\overline{1}\,\overline{0}\,\overline{0}\,\overline{1}^{\,bin} = 10110^{\,bin}$.

3.3 Binary Addition

First some notation. In the expression $a+b$, where a and b denote numbers, we call a and b **operands** and $+$ the **operator**. Note also that addition can be regarded as a function $+\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ with typical input (a,b) and output $a+b$.

Recall that to add two decimal natural numbers, say $\vec{d} \stackrel{\text{def}}{=} d_k d_{k-1}\ldots d_0{}^{\,dec}$ and $\vec{d'} \stackrel{\text{def}}{=} d'_k d'_{k-1}\ldots d'_0{}^{\,dec}$ we add corresponding pairs of digits $d_i+d'_i$ (starting with the 0th digits). So, for example, if $d_2d_1d_0{}^{\,dec} = 423^{\,dec}$ and $d'_2d'_1d'_0{}^{\,dec} = 111^{\,dec}$, then the sum is

$$s_2s_1s_0{}^{\,dec} = (4+1)(2+1)(3+1)^{\,dec} = 534^{\,dec}$$

Recall that there may be carried 1s. For example if $d_2d_1d_0{}^{dec} = 987^{dec}$ and $d'_2d'_1d'_0{}^{dec} = 456^{dec}$, then the sum is

$$s_3s_2s_1s_0{}^{dec} = (0+0+1)(9+4+1\ MOD\ 10)(8+5+\boxed{1}\ MOD\ 10)(7+6\ MOD\ 10)^{dec} = 1443^{dec}$$

where eg $7+6\ MOD\ 10$ is the remainder 3 when dividing by 10 and the carried $\boxed{1}$ comes from $7+6\ DIV\ 10$.

Suppose that s_i represents the ith digit of the representation of the sum. Let $\vec{s}^{dec}$ be the sequence of digits representing the sum. We describe formally the algorithm to compute $\vec{s}^{dec}$ by showing how to calculate each s_i:

- $carry_0 \stackrel{\text{def}}{=} 0$.
- If $i \geq 0$ then $s_i \stackrel{\text{def}}{=} (d_i + d'_i + carry_i)\ MOD\ 10$ and $carry_{i+1} \stackrel{\text{def}}{=} (d_i + d'_i + carry_i)\ DIV\ 10$ where $d_i = d'_i = 0$ if $i > k$.

Because we compute the individual digits of $\vec{s}^{dec}$, we sometimes refer to an algorithm for **digitwise addition**.

Example 3.3.1 Take the decimal numbers $\vec{d}^{dec} \stackrel{\text{def}}{=} 473^{dec}$ and $\vec{d'}^{dec} \stackrel{\text{def}}{=} 49^{dec}$. As $3+9+0 = 12$, the least significant digit of the sum, s_0, is given by $12\ MOD\ 10 = 2$, and $carry_1 = 1$.

Exercises 3.3.2

(1) For the previous example, work out the values of all the d_i, d'_i and s_i.

(2) Take the decimal numbers 473^{dec} and 49^{dec} as above. What is d'_2? Compute the decimal sum $\vec{s}^{dec}$ using the algorithm above, making sure you understand the values of each of the variables.

To convert the algorithm above to work out the sum of two binary numbers, simply change the 10 to 2.

Exercises 3.3.3 Check that $1010^{bin} + 1111^{bin} = 10001^{bin}$ by showing that the algorithm gives rise to the following table

$\vec{d}^{bin}$		1	0	1	0
$\vec{d'}^{bin}$		1	1	1	1
$\vec{s}^{bin}$	1	1	0	0	1
$\vec{carry}^{bin}$		1	1	0	0

Do not forget that binary numbers are just a notation, or representation, for integers. Suppose that integer z is represented by binary number $\vec{d}^{bin}$, and z' by $\vec{d'}^{bin}$. The "digitwise" algorithm for addition takes the binary number representations and produces a binary number $\vec{s}^{dec}$. In order for the algorithm to be useful, we need to know that the integer denoted by $\vec{s}^{\,dec}$ is indeed $z+z'$. In this case, we would say that the sum $z+z'$ is **correctly** represented by $\vec{s}^{\,dec}$, and that the algorithm is **correct**. Thus a formal statement of correctness is that

$$[\![\vec{s}^{\,bin}]\!] = [\![\vec{d}^{\,bin}]\!] + [\![\vec{d'}^{\,bin}]\!]$$

In fact this equality is always true, for any binary numbers $\vec{d}^{\,bin}$ and $\vec{d'}^{\,bin}$, and we say that the algorithm for adding binary numbers is **correct**. In a computer, we have already mentioned that binary numbers will be stored using k bits. However, as we have seen, if the operands of sum are large enough, the binary sum may have $k+1$ digits. Thus, if $k=3$,

$\vec{d}^{\,bin}$		1	0	1
$\vec{d'}^{\,bin}$	+	1	1	0
$\vec{s}^{\,bin}$	(1)	0	1	1

the digit $s_3 = 1$ will be lost, as there are only three bits for the binary sum. In a computer, this kind of problem is called *overflow*. We shall deal with this issue in more detail in the next section, where we also deal with subtraction.

3.4 2s-Complement Numbers

We need to be able to represent all integers in a computer—we restricted ourselves to the non-negative integers in the last two sections. Naively, we can represent all integers much as we explained above for non-negative integers, but using an extra bit to indicate whether the integer is positive or negative. This leads to various problems, such as two possibilities for zero ($+0$ and -0), and that the computer will need a special algorithm to work out the sign of any arithmetic calculation.

We shall in fact use a notation called **2s-complement**. In a $k+1$-bit 2s-complement system, the kth (most significant) bit will be 0 when representing positive integers and zero, and 1 in the case of negative integers. We call this the **sign** bit. For example, if $k=3$, then $1001^{\,bin}$ denotes a negative integer, and $0011^{\,bin}$ a positive integer. What are the integers denoted by these 2s-complement numbers? Given a 2s-complement number $d_k \dots d_0{}^{bin}$, the actual integer it denotes is, by definition,

$$(\!| d_k \dots d_0{}^{bin} |\!) \stackrel{\text{def}}{=} -d_k 2^k + [\![d_{k-1} \dots d_0{}^{bin}]\!] = -d_k 2^k + \sum_{i=0}^{i=k-1} d_i * 2^i$$

Note that we use $(\!|\vec{d}^{\,bin}|\!)$ to mean the 2s-complement interpretation of a digit sequence, and $[\![\vec{d}^{\,bin}]\!]$ to mean the "standard" interpretation.

Example 3.4.1 Thus, for example, if $k=5$, then $(\!|000111^{\,bin}|\!) = -0*2^5+7 = 7$, but $(\!|100111^{\,bin}|\!) = -1*2^5+7 = -32+7 = -25$. Note also that $25 = (\!|011001^{\,bin}|\!)$. We will come back to this example shortly. *Exercise: Make sure you understand these examples before moving on.*

If m is any integer, its **negation** is defined to be $-1*m$. So, for example, the negation of 3 is -3, and the negation of -3 is 3. We shall see later in the course that it is useful for a computer to always see subtraction as the process of adding the negation of an integer. Thus we need to know how to compute the negations of 2s-complement numbers. In fact we have the following result:

Fix a $k \geq 1$. Let $\vec{b}^{\,bin}$ be any $k+1$ digit 2s-complement number other than $1\underbrace{0\ldots0}_{k}$. Then the negation of the integer denoted by $\vec{b}^{\,bin}$ (that is $(-1)*(\!|\vec{b}^{\,bin}|\!)$) is represented by the $k+1$ digit 2s-complement number $\overline{\vec{b}^{\,bin}}+1^{\,bin}$, where $\overline{\vec{b}^{\,bin}}$ is the digitwise complement.

Example 3.4.2 As an example of this result, recall that $25 = (\!|011001^{\,bin}|\!)$. So the negation should be represented by

$$\overline{011001^{\,bin}}+1^{\,bin} = 100110^{\,bin}+1^{\,bin} = 100111^{\,bin}$$

and this agrees with our calculations above, namely $-25 = (\!|100111^{\,bin}|\!)$.

Exercise 3.4.3 Try to prove this result, ie that for any $k+1$ digit 2s-complement number $\vec{b}^{\,bin}$, other than $1\underbrace{0\ldots0}_{k}$, we have

$$(-1)*(\!|\vec{b}^{\,bin}|\!) = (\!|\overline{\vec{b}^{\,bin}}+1^{\,bin}|\!)$$

Use the definitions in the notes to do this.

In a $k+1$-bit 2s-complement representation, note that some integers will be too large to be represented. Any integer z, to be representable, must lie in the range below

$$-2^k \leq z \leq 2^k - 1 \qquad \textit{(range for } k+1\textit{-bit 2s-complement)}$$

which follows from the definition of 2s-complement.

Example 3.4.4 Explain why $-2^k \leq (\!|d_k \ldots d_0{}^{\,bin}|\!)$.

Answer: We have $(\!|d_k \ldots d_0{}^{\,bin}|\!) \stackrel{\text{def}}{=} -d_k 2^k + [\![d_{k-1}\ldots d_0{}^{\,bin}]\!]$. In order for this integer to be large and negative, we must make $-d_k 2^k$ "large", and $[\![d_{k-1}\ldots d_0{}^{\,bin}]\!]$ which is always non-negative, "small". This happens if $d_k = 1$ and all other digits are 0. The value is then -2^k as required.

Exercise 3.4.5 Show that $(\!|d_k \ldots d_0{}^{\,bin}|\!) \leq 2^k - 1$.

3.5 Correctness and Overflow

Recall that digitwise addition is a correct algorithm for ordinary binary numbers, that is $[\![\vec{s}^{\,bin}]\!] = [\![\vec{b}^{\,bin}]\!] + [\![\vec{b'}^{\,bin}]\!]$. We have to be careful with this statement. Suppose that $\vec{b}^{\,bin} = b_{k-1}\dots b_0{}^{bin}$ and $\vec{b'}^{\,bin} = b'_{k-1}\dots b'_0{}^{bin}$. Then $\vec{s}^{\,bin}$ may have a 1 in the k-th position, from a carry. So, to be more precise, we are really saying

$$[\![s_k\dots s_0{}^{bin}]\!] = [\![b_{k-1}\dots b_0{}^{bin}]\!] + [\![b'_{k-1}\dots b'_0{}^{bin}]\!]$$

because s_k might be 1.

If we were to work in a computer with k-bit numbers, this (non-zero) carry digit s_k would be lost. The machine can only store the digits $s_{k-1}\dots s_0{}^{bin}$. If there is no carry, so that the sum $\vec{s}^{\,bin}$ has just k digits, then the integer denoted by $\vec{s}^{\,bin}$ will indeed be $[\![\vec{b}^{\,bin}]\!] + [\![\vec{b'}^{\,bin}]\!]$. However, if there is a carry, then the integer denoted by $\vec{s}^{\,bin}$ will not be $[\![\vec{b}^{\,bin}]\!] + [\![\vec{b'}^{\,bin}]\!]$.

Example 3.5.1 Consider a computer with 3-bit numbers. Given

$\vec{b}^{\,bin}$		0	1	1
$\vec{b'}^{\,bin}$	+	1	0	0
$\vec{s}^{\,bin}$		1	1	1

the integer denoted by the three bit computer sum $111^{\,bin}$ is $[\![111^{\,bin}]\!] = 7$. The integers denoted by the operands are $[\![011^{\,bin}]\!] = 3$ and $[\![100^{\,bin}]\!] = 4$. As $3+4=7$ the computer sum is correct.

Given

$\vec{b}^{\,bin}$		1	0	1
$\vec{b'}^{\,bin}$	+	1	1	0
$\vec{s}^{\,bin}$	(1)	0	1	1

the integer denoted by the three bit sum is $[\![011^{\,bin}]\!] = 3$. The integers denoted by the operands are $[\![101^{\,bin}]\!] = 5$ and $[\![110^{\,bin}]\!] = 6$. As $5+6 \neq 3$ the 3-bit computer sum is not correct.

In fact the following conditions are *equivalent* ways of expressing k-bit correctness

(1) $[\![s_{k-1}\dots s_0{}^{bin}]\!] = [\![b_{k-1}\dots b_0{}^{bin}]\!] + [\![b'_{k-1}\dots b'_0{}^{bin}]\!]$

(2) $0 \leq [\![s_{k-1}\dots s_0{}^{bin}]\!] \leq 2^k - 1$

This means that if condition (1) is true then so is (2); and if (2) is true then so is (1). Recall that k-bit binary numbers denote integers in the range 0 to 2^k-1. So informally this is saying that a computer will give a correct answer to a sum of binary numbers exactly when the correct answer lies within the range of integers that the computer can represent. Common sense!

- if the sign bits of the two operands are complementary, then the computer sum will be correct;
- if the sign bits of the two operands are identical, and also the same as the sign bit of the computer sum, then the computer sum will be correct.

Table 3.1: Correctness Conditions for 2s-complement

Suppose now that we work in a computer with $k+1$-bit 2s-complement numbers. To perform addition of $k+1$-digit 2s-complement numbers, we can perform digitwise addition as described in Section 3.3. There may be a carry digit into the $k+1$ position, which we can compute on paper, but in a computer this extra digit will "disappear", and only the other least significant digits $s_k \ldots s_0$ of the digitwise sum will be recorded. If we look at these $k+1$ digits $s_k \ldots s_0$, which we call the *computer sum*, in some cases this will be correct, and in other cases it will not be. It will be correct if the conditions in Table 3.1 hold. In all other cases, the computer sum will be incorrect. In these cases, we say that **overflow** has occurred. Be careful with these facts, which are slightly subtle. It is quite possible for a 1 to carry into the $k+1$ position, yet the sum given by the least significant digits $s_k \ldots s_0$ in the computer to be correct! Do not confuse our definition of computer overflow, with an extra carried digit! They are different!!

Example 3.5.2 For example, let $k=2$, so that a computer uses $2+1=3$ bits for 2s-complement numbers. Consider

	1	1	1
+	1	1	0
(1)	1	0	1

where there is a lost carry of 1. However, the computer's sum is correct. The 2s-complement operands denote $-4+2+1=-1$ and $-4+2+0=-2$ with integer sum -3. The 3-bit 2s-complement digitwize sum stored in the computer denotes $-4+0+1=-3$. So the computer is correct!

Notice also that the sign bits are all 1, so that the second condition in Table 3.1 holds.

Exercise 3.5.3 Write down examples of 2s-complement calculations, and check for yourself that the overflow conditions work.

We finish by summarising some equivalent conditions which capture the correctness of $k+1$-bit 2s-complement addition. It is a fact that if $\vec{b}^{\,bin}$ and $\vec{b'}^{\,bin}$ are two $k+1$-bit 2s-complement numbers, then the following conditions are equivalent

- $(\!| s_k \dots s_0{}^{bin} |\!) = (\!| b_k \dots b_0{}^{bin} |\!) + (\!| b'_k \dots b'_0{}^{bin} |\!)$
- $-2^k \leq (\!| s_k \dots s_0{}^{bin} |\!) \leq 2^k - 1$
- the conditions of Table 3.1 hold.

3.6 Logical Operations

Computer Scientists often need to know if statements are *true* or *false*. We sometimes also use 1 to denote truth, and 0 to denote falsity. If we write $\centerdot$ to denote *and*, and d and d' are binary digits, then we get a function $\centerdot\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ which maps input (d, d') to $d \centerdot d'$ where we write the output using infix notation. We define $d \centerdot d'$ to be 1 precisely when both d *and* d' are 1, and otherwise $d \centerdot d'$ is 0. Informally, $d \centerdot d'$ is "true" when both d *and* d' are "true". We call $\centerdot$ the **and** function. A similar idea applies to *or*. The function is denoted by $+_{or}\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ which we call the **or** function; $d +_{or} d'$ is 1 if and only if either d *or* d' is 1.

Example 3.6.1 Some examples are $0 \centerdot 0 = 0$, $1 \centerdot 1 = 1$, $1 +_{or} 1 = 1$ and $1 +_{or} 0 = 1$.

When we come to the chapters on the instruction set architecture, we shall see that computers often need to compute "and" and "or" on pairs of digits selected from two binary numbers. We define

$$d_k \dots d_0{}^{bin} \ AND \ d'_k \dots d'_0{}^{bin} \ \stackrel{\text{def}}{=} \ (d_k \centerdot d'_k) \dots (d_0 \centerdot d'_0)^{bin}$$

$$d_k \dots d_0{}^{bin} \ OR \ d'_k \dots d'_0{}^{bin} \ \stackrel{\text{def}}{=} \ (d_k +_{or} d'_k) \dots (d_0 +_{or} d'_0)^{bin}$$

and say that the functions act **digitwise**.

Example 3.6.2

(1) $0101 \ AND \ 0110 = (0 \centerdot 0)(1 \centerdot 1)(0 \centerdot 1)(1 \centerdot 0) = 0100$.

(2) $0101 \ OR \ 0110 = 0111$.

Exercise 3.6.3 Both *AND* and *OR* are functions whose inputs are k-digit binary numbers. What size are the outputs? Thus what are the source and target of *AND* and *OR*?

We complete this section with a couple of definitions that we will make use of in the chapter on Microarchitecture. Please note that we make heavy use of notational conventions, and these will be explained in the lectures in detail—however, you *should* be able to understand this section with a little thought. Sometimes, either a 2s-complement number or a binary number will be written using just k digits. In the machine, the number will be stored in a memory space containing k bits. We will then want to move the number into a space with $k'+k$ bits *so that the integer denoted by the contents of the* $k'+k$ *bits is the same as the integer denoted by the original binary number*. In the case of a normal binary number, we will fill the extra bits with k' zeros. This is called a **zero extension**. We can model it as a function

$$\mathbb{B}^k \longrightarrow \mathbb{B}^{k'+k} \qquad \vec{b} \mapsto \vec{0}\vec{b}$$

where $\vec{0}$ denotes k' zeros. We will sometimes write the output as $zx(\vec{b})$.

Example 3.6.4 Let $k = 2$ and $k' = 3$. If $10 \in \mathbb{B}^2$, then $zx(10) = 00010 \in \mathbb{B}^5$. Note that $[\![00010]\!] = [\![10]\!] = 2$ as required.

What about 2s-complement numbers? We first look at an example.

Example 3.6.5 Let 1001^{bin} be a 2s-complement number, denoting -7. It has 4 digits. Suppose we want to store it in seven bits, with these bits also storing 2s-complement numbers, with the most significant bit now regarded as the sign bit. So what should the * below be?

*	*	*	1	0	0	1

We could try taking 1 for the sign bit (why?):

1	0	0	1	0	0	1

But this denotes $-64 + 8 + 1 = -55$. Or we could look at the sign bit from 1001 and repeat it:

1	1	1	1	0	0	1

This denotes $-64 + 32 + 16 + 8 + 1 = -7$. Yippee! The denotation is preserved.

In fact "repeating" the sign digit always works. Suppose that $\delta\vec{b}$ is a 2s-complement number, where δ is the sign bit, and there are $k+1$ digits in total. Now we want to use $k'+k+1$ digits to represent the same integer. We can do this by looking at the $k+1$th sign digit of $\delta\vec{b}$, namely δ, and copying it into the new k' positions. This is called a **sign extension**. We can model it as a function

$$\mathbb{B}^{k+1} \longrightarrow \mathbb{B}^{k'+k+1} \qquad \delta\vec{b} \mapsto \vec{\delta}\delta\vec{b}$$

where $\vec{\delta}$ means a sequence of k' δs. We will sometimes write the output as $sx(\delta\vec{b})$. In the example above, $\vec{b} = 001$, $\delta = 1$ and $sx(1001) = 1111001$.

Exercise 3.6.6 Why does our definition of $sx(\delta\vec{b})$ work correctly, in the sense that the denotations of 2s-complement numbers are preserved, that is $[\![\delta\vec{b}]\!] = [\![sx(\delta\vec{b})]\!]$?

We can multiply by powers of 2 by shifting digit sequences to the left. To multiply $\vec{b} \in \mathbb{B}^k$ by $2^{k'}$, apply the k'**-left-shift** function

$$\mathbb{B}^k \longrightarrow \mathbb{B}^{k+k'} \qquad \vec{b} \mapsto \vec{b}\vec{0}$$

where $\vec{0}$ is a sequence of k' zeros. We will sometimes write the output as $\vec{b} \ll k'$.

Example 3.6.7 Let $k = 4$ and $k' = 5$. Write $\ll : \mathbb{B}^4 \to \mathbb{B}^9$ for the shift left function. What is $1011 \ll 5$? Have we achieved multiplication by 2^5?

Answer: $1011 \ll 5 = 101100000$. Note that $[\![1011]\!] = 11^{\,dec}$. We have

$$[\![101100000]\!] = 256+64+32 = 352. \qquad \frac{352}{2^5} = 11^{\,dec}.$$

So, yes, the integer denoted by 1011 has been multiplied by 2^5.

3.7 Further Examples

Examples 3.7.1

(i) Suppose that a computer can only represent numbers n using k bits. What range of numbers can it represent if

1. $k = 2$,
2. $k = 3$, and
3. $k = 5$.

Explain your answers. What range does n lie in for general $k \geq 1$? Give a careful explanation.

Answer hints (you should give more details):

1. $k = 2$: $\quad 0 \leq n \leq 2+1 = 3$
2. $k = 3$: $\quad 0 \leq n \leq 4+2+1 = 7$
3. $k = 5$: $\quad 0 \leq n \leq 16+8+4+2+1 = 31$

The smallest value of n is given when all bits are 0; so smallest n is 0. Largest value of n is when all bits are 1. For general k, largest value must be $2^{k-1}+2^{k-2}+\ldots+2+1 = 2^k-1$.

(ii) Consider $\vec{d} \stackrel{\text{def}}{=} 346^{\,dec}$ and $\vec{d'} \stackrel{\text{def}}{=} 1664^{\,dec}$. With reference to Section 3.3 of the notes, write down with explanation the values of d_i, d'_i and s_i. What values does i take?

Answer hints (you should give more details): $d_2 = 3$, $d_1 = 4$, $d_0 = 6$ and $d'_3 = 1$, $d'_2 = 6$, $d'_1 = 6$, $d'_0 = 4$. The digitwise sum $\vec{s}$ is 2010, that is $s_3 = 2$, $s_2 = 0$, $s_1 = 1$, $s_0 = 0$. i ranges over 0, 1, 2 and 3.

(iii) For the following (pairs of) integers, work out the 5-bit 2s-complement binary numbers which represent the two integers. In each case, state whether addition of the 2s-complement binary numbers gives computer overflow, with a brief explanation.

1. $(5,-12)$
2. $(-3,8)$
3. $(-15,-9)$
4. $(13,8)$
5. $(-15,11)$

Answer hints (you should give more details):

1. $(5,-12)$ given by 00101^{bin} and 10100^{bin}. Bitwise sum is 11001 which represents $-16+9=-7$. $5+(-12)=-7$, so no overflow.
2. $(-3,8)$ given by 11101^{bin} and 01000^{bin}. Bitwise sum is $(1)00101$ which represents $0+5=5$. $-3+8=5$, so no overflow. (Note that there is a lost carry).
3. $(-15,-9)$ given by 10001^{bin} and 10111^{bin}. Bitwise sum is $(1)01000$ which represents $0+8=8$. $-15+(-9)=-24$, so overflow occurs. (Note that there is a lost carry).
4. $(13,8)$ given by 01101^{bin} and 01000^{bin}. Bitwise sum is 10101 which represents $-16+5=-11$. $13+8=21$, so overflow occurs.
5. $(-15,11)$ given by 10001^{bin} and 01011^{bin}. Bitwise sum is 11100 which represents $-16+12=-4$. $-15+11=-4$, so no overflow occurs.

In each case, overflow occurs only when the 5-bits of the digitwise sum do not represent the correct answer, that is, for the pair $(\vec{b}^{\,bin},\vec{b'}^{\,bin})$ overflow occurs if and only if

$$(\!| s_4s_3s_2s_1s_0{}^{bin} |\!) \neq (\!| b_4b_3b_2b_1b_0{}^{bin} |\!) + (\!| b'_4b'_3b'_2b'_1b'_0{}^{bin} |\!)$$

Also, one can check that the "sign bit" conditions given on page 28 of the notes hold.

4

Digital Electronics

4.1 Motivation

As we have seen, a computer contains electrical circuits. These circuits contain wires which have a measurable voltage. It is possible to distinguish between a (fixed) high voltage and low voltage. We can use the **high** voltage to represent the integer 1, and the **low** voltage to represent 0. So far so good. What we want to do is use electrical circuits to perform computations of addition and subtraction on the integers, and to design circuits that will perform other kinds of computation. We also want memory circuits that will store data. In this chapter we shall explain how this can be done.

4.2 Boolean Algebras and the Switching Algebra

A *Boolean algebra* is a particular kind of mathematical structure. In this course, we shall make use of one particular example, called the **switching algebra**. This consists of the following five things: $(\mathbb{B}, \centerdot, +_{or}, \overline{}, 0, 1)$, where $\mathbb{B} = \{0, 1\}$. We have already seen the functions $\centerdot$, $+_{or}$ and $\overline{}$ in Chapter 3. The functions satisfy certain properties. Letting A, B and C denote elements of the set $\mathbb{B}$ (ie binary digits!), the basic properties are given below; notice that each property has two versions

Property	$\centerdot$	$+_{or}$
Identity	$A \centerdot 1 = A$	$A +_{or} 0 = A$
Idempotent	$A \centerdot A = A$	$A +_{or} A = A$
Complement	$A \centerdot \overline{A} = 0$	$A +_{or} \overline{A} = 1$
Commutative	$A \centerdot B = B \centerdot A$	$A +_{or} B = B +_{or} A$
Associative	$A \centerdot (B \centerdot C) = (A \centerdot B) \centerdot C$	$A +_{or} (B +_{or} C) = (A +_{or} B) +_{or} C$
Distributive	$A \centerdot (B +_{or} C) = (A \centerdot B) +_{or} (A \centerdot C)$	$A +_{or} (B \centerdot C) = (A +_{or} B) \centerdot (A +_{or} C)$

We sometimes call A, B, C **Boolean variables**. Note that because of associativity, we can sometimes omit brackets. For example, whatever A, B and C are,

$$A \centerdot (B \centerdot C) = (A \centerdot B) \centerdot C = A \centerdot B \centerdot C \qquad (\text{ eg } 1 \centerdot (0 \centerdot 1) = (1 \centerdot 0) \centerdot 1 = 0\,)$$

Also, we say that $\centerdot$ **binds** more tightly than $+_{or}$, which means that $A \centerdot B +_{or} C$ stands for $(A \centerdot B) +_{or} C$. Further, $\overline{}$ binds *most* tightly, so that $\overline{A} \centerdot B +_{or} C$ stands for $((\overline{A}) \centerdot B) +_{or} C$.

Exercise 4.2.1 Check that the functions $\centerdot$, $+_{or}$ and $\overline{}$, as defined in Chapter 3, *do* satisfy the properties in the table above.

We can give the functions $\centerdot$ and $+_{or}$ by explicitly computing all inputs and outputs in tables; see below. The columns to the left of the double lines give the possible

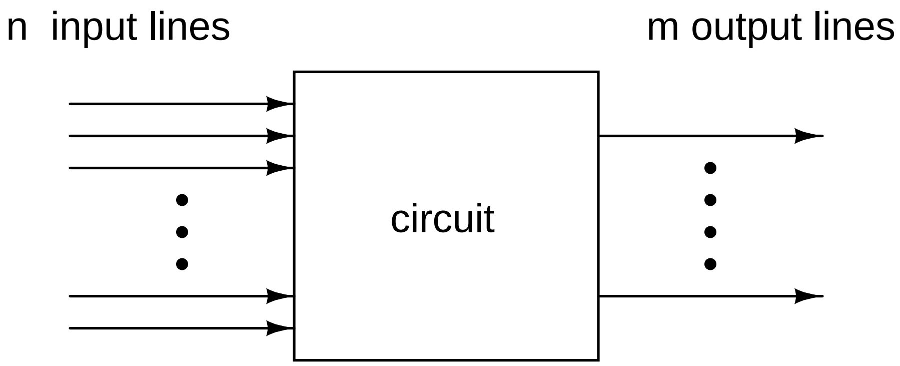


Figure 4.1: Circuit for an $n \mid m$-ary function

input components; each row specifies an input tuple (such as (A,B)). To the right of the double line we have the outputs (such as $A +_{or} B$). These tables are called **truth tables**.

A	$\overline{A}$
0	1
1	0

A	B	$A +_{or} B$
0	0	0
0	1	1
1	0	1
1	1	1

A	B	$A \mathbin{\bullet} B$
0	0	0
0	1	0
1	0	0
1	1	1

4.3 Implementation of Switching Algebra Functions

We shall soon see that we need to be able to implement $n \mid m$-ary functions over $\mathbb{B}$ as digital circuits. We shall represent such circuits using a picture such as the one in Figure 4.1. The box denotes the circuit whose details are omitted. It implements a function $f\colon \mathbb{B}^n \to \mathbb{B}^m$. The horizontal **input lines** denote wires which carry a voltage; each voltage indicates a binary digit, which will be one of the input components for f. We say that the line **carries** or **holds** a binary digit **value**. The values of the lines taken together specify an input tuple for f. The values on the m **output lines** specify the corresponding output tuple. We sometimes call a collection of n parallel wires an n-bit **bus**. For example, if $f\colon \mathbb{B}^3 \to \mathbb{B}^4$ and $f(1,0,1) = (1,1,0,0)$, there will be three input lines with values 1, 0, and 1, and four output lines with values 1, 1, 0, 0.

A **gate** is an electrical circuit which computes certain simple $m \mid 1$-ary functions over $\mathbb{B}$. Note that, by definition, complementation is a $1 \mid 1$-ary function, and the $\mathbin{\bullet}$ and $+_{or}$ functions above are $2 \mid 1$-ary functions. A simple **NOT** gate computes complementation, an **AND** gate computes $\mathbin{\bullet}$, and **OR** computes $+_{or}$. These circuits are denoted by the pictures in Figure 4.2. For example, if the values 0 and 1 are carried on the input lines to AND, the value $0 = 0 \mathbin{\bullet} 1$ will be carried on the output line.

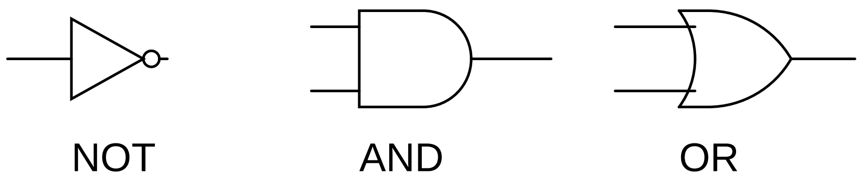


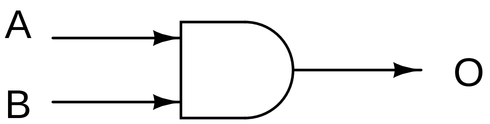
Figure 4.2: NOT, AND and OR Gates

There are also $m \mid 1$-ary versions of $\centerdot$ and $+_{or}$. Circuits which implement these are also called **gates**, and denoted by the same pictures. Informally, given an m-tuple over $\mathbb{B}$ as input to $\centerdot$, the output is 1 if all the input components are 1, and is otherwise 0. The $m \mid 1$-ary function $+_{or}$ is similar. If $m = 3$ then (eg) the function $\centerdot$ will map input $(1,0,1)$ to 0, that is $\centerdot(101) = 0$. We often write $A \centerdot B \centerdot C$ instead of $\centerdot(A,B,C)$—an infix notation. We can also draw up a truth table

A	B	C	$A \centerdot B \centerdot C$
0	0	0	0
0	0	1	0
0	1	0	0
⋮			
1	1	1	1

Exercise 4.3.1 Complete the table.

Note that we sometimes *label* input and output lines. An example is



We sometimes talk about the "input line" A to mean the physical wire at the "top" of the circuit. We shall sometimes talk about the "value" of A to mean the voltage (0 or 1) held on the input line at a given moment.

It is easy to see that an $n \mid m$-ary function f gives rise to m different $n \mid 1$-ary functions; informally, the output of each function is given by simply selecting one of the m output components of f. We call the functions f_0, f_1, …, f_{m-1}. By definition, $f_i(d_{n-1},\ldots,d_0)$ is the ith component of $f(d_{n-1},\ldots,d_0)$. *<u>Exercise:</u> Before moving on, make sure this is clear to you.* Thus we can present such $n \mid m$-ary functions f as truth tables which give the input components, and corresponding output components. For example,

A	B	C	$f_1(A,B,C)$	$f_0(A,B,C)$
0	0	0	0	1
0	1	0	0	0
0	0	1	0	1
0	1	1	0	1
1	0	0	0	0
1	1	0	0	0
1	0	1	0	0
1	1	1	0	1

Table 4.1: Truth table of a $3 \mid 2$-ary function f.

the table in Table 4.1 specifies a $3 \mid 2$-ary function $f\colon \mathbb{B}^3 \to \mathbb{B}^2$, where

$$f(A,B,C) = (f_1(A,B,C), f_0(A,B,C)) \qquad \text{eg } f(0,1,1) = (0,1)$$

Let us first see how to implement $n \mid 1$-ary functions. We begin by example. By computing the possible values of the expression on the right of $=$ below, you can check that the following equation holds

$$f_0(A,B,C) = \overline{A} \mathbin{.} \overline{B} \mathbin{.} \overline{C} +_{or} \overline{A} \mathbin{.} \overline{B} \mathbin{.} C +_{or} \overline{A} \mathbin{.} B \mathbin{.} C +_{or} A \mathbin{.} B \mathbin{.} C$$

Notice that there are four "$\mathbin{.}$" expressions in the right hand side—and that four of f_0's outputs are 1. Now look at each of the corresponding four inputs—the 1s and 0s match up with the normal and complemented variables in each of the "$\mathbin{.}$" expressions. We call the expression on the right a **sum-of-products**. We say it **represents** the function f_0.

In fact, a simple generalization of this example gives us a method for producing a sum-of-products expression which will represent any $n \mid 1$-ary function $f\colon \mathbb{B}^n \to \mathbb{B}$ on the switching algebra. The steps are

1. Write down the truth table for f using $A_{n-1},\ldots,A_0$ as the input variables;

2. count the number of lines (say l) in the table with $f(A_{n-1},\ldots,A_0) = 1$; write down

$$A_{n-1} \mathbin{.} A_{n-2} \mathbin{.} \ldots \mathbin{.} A_0 +_{or} \ldots +_{or} A_{n-1} \mathbin{.} A_{n-2} \mathbin{.} \ldots \mathbin{.} A_0 \qquad (*)$$

   in which there are l "$\mathbin{.}$" expressions; finally

3. for each of the l lines, look at the inputs holding a 0 and complement each of the A_i in the corresponding "$\mathbin{.}$" expressions in $(*)$.

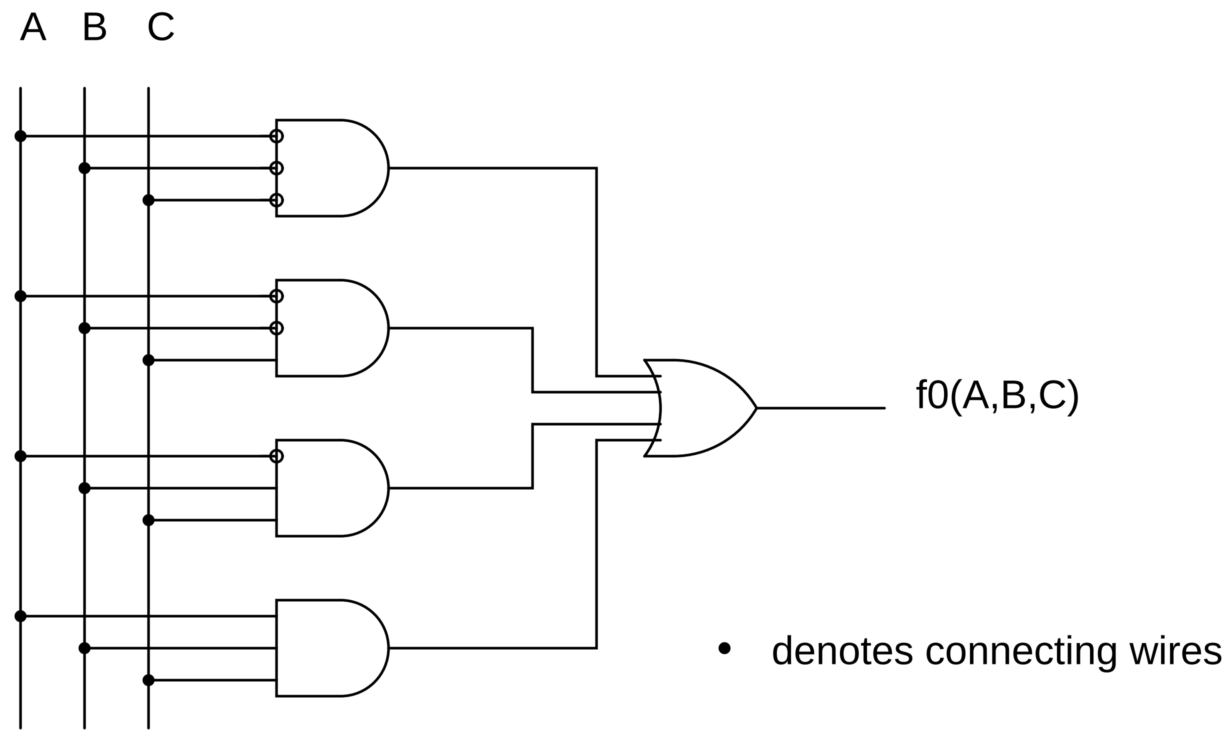


Figure 4.3: Circuit Implementing f_0

Example 4.3.2 We now show by example how one can "construct" a circuit which computes a sum-of-products expression. The circuit in Figure 4.3 will compute the function f_0, by implementing the sum-of-products expression for f_0. Look at the circuit, and you should be able to see that it is almost "obvious" given the sum-of-products expression. The small circles stand for NOT gates.

It is easy to describe a general method for producing a circuit to implement any $n \mid 1$-ary function f. First, write down a sum-of-products expression for f, using the variables A_{n-1} to A_0. Draw n vertical wires, labelling them A_{n-1} to A_0. Draw l different $n \mid 1$-ary AND gates to the right of these wires (l being the number of "**.**" expressions), and for each of these, draw horizontal lines connecting the n vertical lines to each AND gate, and inserting a NOT gate for any complemented variable in the sum-of-products expression. Finally, take the l outputs of the AND gates, and feed into an OR gate. The OR gate output will now be $f(A_{n-1}, A_{n-2}, \ldots, A_0)$.

Exercise 4.3.3 Apply this procedure to the sum-of-products expression in the last example.

A **Programmed Logic Array** or **PLA** is an integrated circuit which contains AND gates and OR gates which are arranged suitably for implementing circuits derived from sum-of-products expressions. The user simply applies current which blows out fuses in the PLA, leaving the required combinations of gates.

Such a naive method of circuit design will not normally be used in practice, because the circuit it produces will not be optimal. We illustrate this point with the function f_0. The sum-of-products expression can be simplified using the rules for Boolean algebra:

$$\begin{aligned}\overline{A}\,.\,\overline{B}\,.\,\overline{C} +_{or} \overline{A}\,.\,\overline{B}\,.\,C +_{or} \overline{A}\,.\,B\,.\,C +_{or} A\,.\,B\,.\,C &= \overline{A}\,.\,\overline{B}\,.\,(\overline{C} +_{or} C) +_{or} (\overline{A} +_{or} A)\,.\,B\,.\,C \\ &= \overline{A}\,.\,\overline{B}\,.\,1 +_{or} 1\,.\,B\,.\,C \\ &= \overline{A}\,.\,\overline{B} +_{or} B\,.\,C\end{aligned}$$

We can now implement a far simpler circuit from the simplified expression. In actual computer engineering, a variety of techniques will be used to perform this sort of simplification.

Exercise 4.3.4 Draw a circuit for the simplified expression.

4.4 Combinational Circuits

4.4.1 The Fundamental Idea

A circuit which implements an $n \mid m$-ary *function* is known as a **combinational** circuit. We have seen how a general $n \mid m$-ary function f gives rise to m different $n \mid 1$-ary functions $f_1, \ldots, f_m$. Moreover, we gave a method for implementing any $n \mid 1$-ary function. Thus we can certainly implement f by combining the circuits for the f_i "in parallel".

Exercise 4.4.1 Explain in detail what "in parallel" means. For example, suppose you have a function $f\colon \mathbb{B}^3 \to \mathbb{B}^2$ given by $f(x) = (f_1(x), f_0(x))$ and you have circuits C1 and C2 implementing f_1 and f_0. Use these in a drawing of a circuit for f.

We look at some actual examples of circuits which we shall use later on in this course.

4.4.2 Multiplexors

A **multiplexor** is an implemention of a $n+2^n \mid 1$-ary function M. The $n+2^n$ input lines divide up into n **control input lines** and 2^n **data input lines**.

Before explaining the general specification of M, let us look at one row of M's truth table in the example case when n is 3 (so $M\colon \mathbb{B}^{11} \to \mathbb{B}$)

C_2	C_1	C_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	$M(\vec{C},\vec{D})$
1	0	1	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0	d_5

Each d_i is a binary digit, that is either 0 or 1. The values on the control input lines are $1, 0, 1$ which is interpreted as representing $101^{\,bin}$. This denotes 5. The output value of the multiplexor is then defined to be the same as the value on the "corresponding" data line D_5, namely d_5.

In general, the output of the multiplexor M is precisely the value on one of the 2^n data input lines, namely the one "corresponding" to the integer denoted by the sequence of binary digits held on the n control lines. There are, of course, 2^n possible numbers.

Exercise 4.4.2 Explain *carefully* what "corresponding" means. (*Hint: We usually count the input lines down the page, starting from 0.*)

A circuit diagram of a $2+2^2$ multiplexor is given in Figure 4.4, along with an abstract diagram which we shall use to denote a general multiplexor. Note that we draw separate lines for the data lines, but we draw a single *bus* line for the n control lines. The diagonal stroke and the n symbolise that the "bus" line on the drawing denotes n input lines. We draw these as a bus to highlight the fact that the bus data is used to select just *one* of the data lines to supply the output value.

4.4.3 Decoders

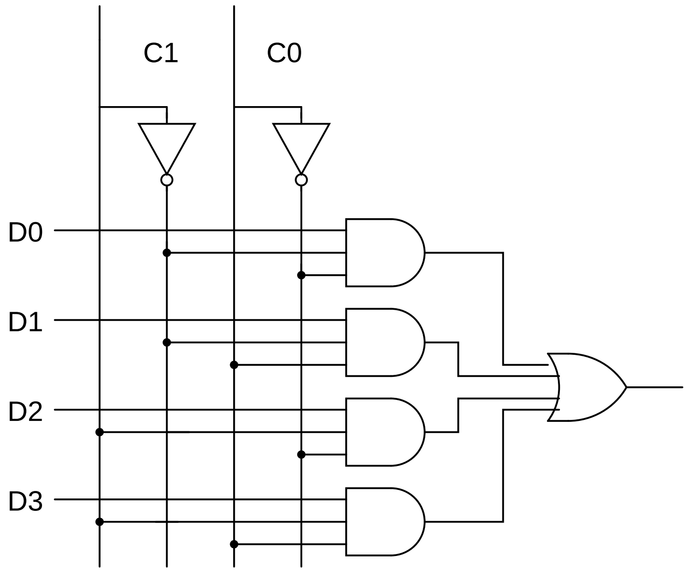
A **decoder** is a circuit with n input lines and 2^n output lines. For example, if $n=2$, the output lines might be A_0 and A_1, and output lines B_0, B_1, B_2, B_3. The input lines can therefore carry 2^n different n-tuples of binary digits. For each of the 2^n possible input tuples, exactly one of the 2^n output lines is set to 1 and all the others are set to 0. The line that is set to 1 is usually given by the integer denoted by the input. The input lines are A_0 and A_1, and we interpret them as a binary number A_1A_0. For example if $A_0 = 0$ and $A_1 = 1$ (denoting $[\![10]\!] = 2$) then $B_0 = 0, B_1 = 0, B_2 = 1, B_3 = 0$. In general, the line $B_{[\![A_1A_0]\!]}$ is set to 1, and all others to 0.

Example 4.4.3 A computer contains many individual circuits. Each circuit often has a switch which is controlled electronically. There is a special input wire. If its value is 0, the circuit is off; if 1 it is on. We can use a decoder to reduce wiring. Suppose we have $32 = 2^5$ circuits. Instead of having 32 on/off wires we can feed just 5 into a decoder, with just the 32 output wires from the decoder hooked up to the on/off wires. For each of the 32 binary numbers which can be input to the decoder, just one circuit will be on, and the others off.

Exercise 4.4.4 Draw a diagram of the example described here.

4.4.4 Clocks

In many digital circuits, the timing of various processes and tasks is crucial. A **pulse** is a voltage of 1 which lasts for a (usually short) time. A **clock** is a circuit which generates a series of pulses. Each pulse lasts for fixed time, and there is a fixed interval between



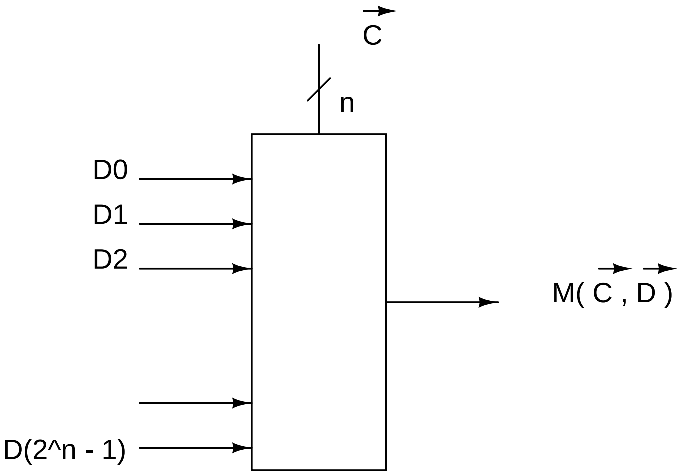


Figure 4.4: A Multiplexor

pulses during which the voltage produced is 0. The production of such a high and low voltage is called a **clock cycle**. The total time taken for this is called the **clock cycle time** or **period**. The **clock frequency** is the reciprocal of the period. The start time of a clock pulse is called a **rising edge**; the **falling edge** is when the pulse ends. In the lectures will will provide some diagrams, and explain how to use clock delays to provide finer timing than a standard clock.

4.4.5 Adders

We saw in Chapter 3 that in order to compute the sum of two binary numbers, we perform a bitwise addition with possible carries. Thus to design a circuit to perform addition of k-bit integers, we first need a circuit to add single binary digits. Using a notation similar to Chapter 3 (see page 32) this circuit will have three input lines and two output lines:

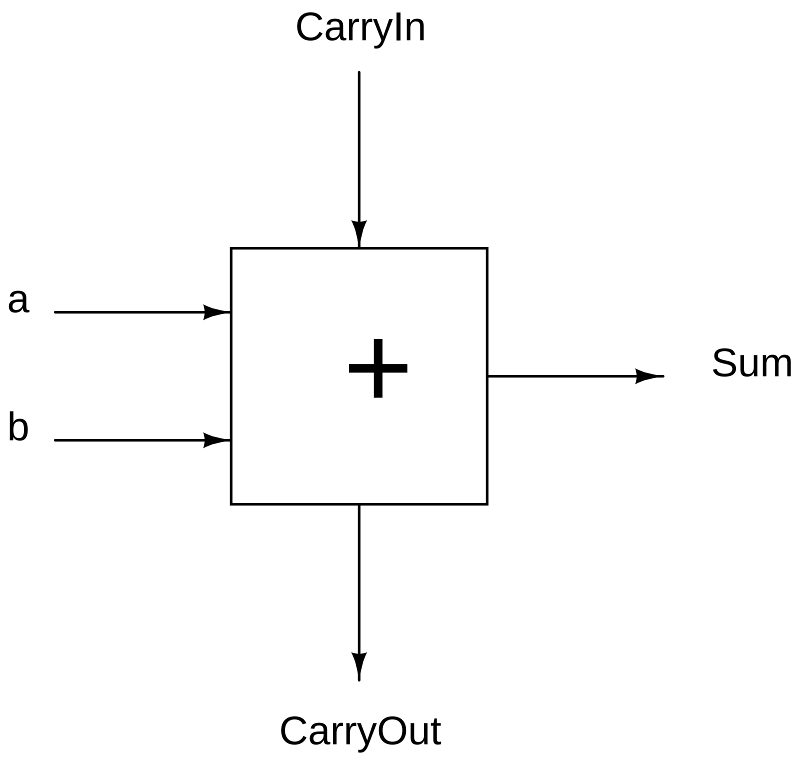
Line Name	**Holds Data**
`a`	d_i
`b`	d'_i
`CarryIn`	$carry_i$
`Sum`	s_i
`CarryOut`	$carry_{i+1}$

For example, if $d_i = 1$ and $d'_i = 0$ and $carry_i = 0$ then $1^{\,bin} + 0^{\,bin} + 0^{\,bin} = 01^{\,bin}$ and the carry out digit is $carry_{i+1} = 0$ and the sum is $s_i = 1$. If $d_i = 1$ and $d'_i = 1$ and $carry_i = 1$ then $1^{\,bin} + 1^{\,bin} + 1^{\,bin} = 11^{\,bin}$, the carry in is 1, the carry out is 1 and the sum 1. A circuit which implements this is a **full adder**. To implement a full adder, we can write down the truth table, and produce the relevant circuit as in Section 4.3. The truth table, and an abstract circuit diagram, appear in Table 4.2.

4.4.6 Arithmetic Logical Units

We need to construct an Arithmetic Logical Unit (**ALU**) in order to build a computer. This is a circuit which will perform operations (functions) such as addition, subtraction and so on. For any inputs, the desired operation can be selected. In a nutshell, the input lines to the ALU will be two k-bit busses (each holding a k-digit binary number). The main output value will be the selected operation applied to the two k-bit integers, carried on a k-bit bus. There will also be a special **control bus** which will be used to select the ALU operation (such as addition).

We shall design such an ALU where $k = 32$ and the operations are digitwize addition, subtraction, *AND* and *OR* (recall Chapter 3). There is also a new operation called "**set on less than**" which we shorten to `slt`. We explain what `slt` is. Suppose the inputs are $\vec{a}^{\,bin} = a_{31} \ldots a_0{}^{bin}$ and $\vec{b}^{\,bin} = b_{31} \ldots b_0{}^{bin}$, with an output $\vec{r}^{\,bin} = r_{31} \ldots r_0{}^{bin}$. The inputs are interpreted as 2s-complement numbers, with digit 31 being the sign digit.



a	*b*	*CarryIn*	*CarryOut*	*Sum*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 4.2: A Fulladder and Truth Table

The `slt` operation returns $\vec{r}^{\,bin}$ given by

$$\begin{array}{lll} r_{31} = 0, \ldots, r_1 = 0, r_0 = 1 & \text{if} & (\!|\vec{a}^{\,bin}|\!) < (\!|\vec{b}^{\,bin}|\!) \\ r_{31} = 0, \ldots, r_1 = 0, r_0 = 0 & & \text{otherwise} \end{array}$$

Informally, the 32-bit result denotes the integer 1 if the integer denoted by $\vec{a}^{\,bin}$ is strictly less than the integer denoted by $\vec{b}^{\,bin}$, and 0 if not.

There are are two other output lines, which we will make use of in Chapter 6.

The ALU will have a single bit output line `Zero` which will hold 1 when *any* result is $\vec{0}$, and will hold 0 otherwise. We call this a **test for zero**. We will use it in the chapter on microarchitecture.

The ALU will also have a single bit output line which will hold 1 when an operation causes overflow, and will hold 0 otherwise. Its use is obvious—it indicates when the ALU overflows!

ALU Step 1

The first step is to build an ALU which has single binary digit numbers and a carry in digit as input components. The output is a carry out digit, and the result of a choice of $\bullet$, $+_{or}$, $+$ or $-$ acting on the components. To perform the subtraction, we simply recall that we can compute $a-b$ as $a+\overline{b}+1$; and further that we can compute $\overline{b}$ using a single NOT gate. The ALU appears in Figure 4.5. Note the multiplexor which selects either b or $\overline{b}$. Note also that `Operation` is a 2-bit bus, used to select an output from one of the three circuits for $\bullet$, $+_{or}$, or $+$. (The `Operation` line *could* select up to four circuits—why?) In order to select the overall operation of the ALU, both `Binvert` and `Operation` are used together. These are sometimes called the ALU's **control lines**; together they form the **control bus**. *Exercise: Make sure this is clear to you—read the example.*

Example 4.4.5 With the usual conventions about multiplexors, what values appear on `Binvert` and `Operation` and `CarryIn` to select ALU subtraction of a and b?

Answer: We must compute $a+\overline{b}+1$. To complement b, `Binvert` must hold 1. `CarryIn` must hold 1. Then the full adder will compute $a+\overline{b}+1$ and this value will be sent to `Result` provided that `Operation` holds 10^{bin} (which denotes 2).

ALU Step 2

The second step is to modify the ALU of the first step. The new ALU should have an additional 1-bit input line, whose value can be passed directly as output. We call this line `Less`. Its exact purpose will be explained later on. It will be used on those occasions when we need to be able to set the result of the ALU to a specific value. The new ALU also has an additional output line, `Set`, which will always hold the output of the full adder. Finally, we add some further circuits which will test for overflow—

Input Line or Bus	Purpose
`a`	operation input
`b`	operation input
`CarryIn`	holds carry in digit
`Binvert`	used to select complementation of b
`Operation` (2-bit bus)	used to select (internal) operation on a and b
Output Line or Bus	**Purpose**
`Result`	holds result of operation
`CarryOut`	holds carry out digit

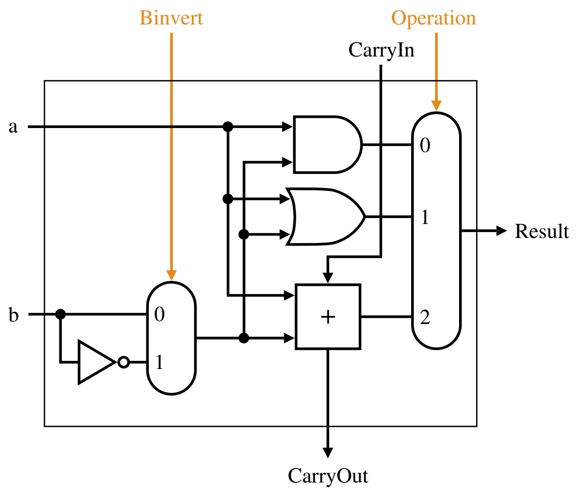


Figure 4.5: Step 1 ALU

recall that overflow was defined in Chapter 3. The ALU appears in Figure 4.6. *Exercise: Make sure that you understand the roles of the input lines to the overflow circuit.*

ALU Step 3

The final step is to link 31 of the step 1 ALUs and one of the step 2 ALUs to make a 32-bit ALU as specified at the start of the section. The ALU appears in Figure 4.7. The two 32-bit 2s-complement integers $a_{31} \ldots a_0$ and $b_{31} \ldots b_0$ are read in by feeding the values a_i and b_i as single digits into `ALUi`. The operation to be performed by the big ALU is selected using the `Bnegate` and `Operation` lines. These are sometimes called the ALU's **control lines**. The `Bnegate` line feeds the `Binvert` lines of each separate small ALUi. In ALU0, it is also wired to the `CarryIn` line. This is because we only want to complement the b_i when performing subtraction; and to perform subtraction, recall that we complement and add 1. So `Bnegate` is set to 1, and thus `CarryIn` of ALU0 is also set to 1 ensuring we add 1. The Zero output line will *always* tell us if the result, the 32-bit integer given by the lines `Result31` to `Result0`, is zero or not.

Example 4.4.6 Explain how to make the big ALU select the `slt` operation (by giving suitable values for its control lines) and how the `slt` operation is computed.

Answer: We want to check if $\llparenthesis \vec{a}^{\,bin} \rrparenthesis < \llparenthesis \vec{b}^{\,bin} \rrparenthesis$ or not. Using the method for computing subtractions, we equivalently check that $\llparenthesis \vec{a}^{\,bin} \rrparenthesis + \llparenthesis \overline{\vec{b}^{\,bin}} + 1^{\,bin} \rrparenthesis < 0$. In the cases where no overflow occurs, by correctness this is the same as checking if $\llparenthesis \vec{a}^{\,bin} + \overline{\vec{b}^{\,bin}} + 1^{\,bin} \rrparenthesis < 0$. Thus we set `Bnegate` to 1, and `Operation` to 11 which copies the value on each `Less` line into the result lines. So `Result1` to `Result31` are all set to 0. But the value of `Result0` comes from the `Less` line of ALU0, which is connected to the `Set` line of ALU31. By examining ALU31, and recalling that `Set` is connected to the full adder in ALU31, we see `Set` holds digit 31 of $\vec{a}^{\,bin} + \overline{\vec{b}^{\,bin}} + 1^{\,bin}$. This is the sign digit. So it will be 1 precisely when $\llparenthesis \vec{a}^{\,bin} + \overline{\vec{b}}^{\,bin} + 1^{\,bin} \rrparenthesis < 0$ and will be 0 if not. Magic!

Exercise 4.4.7 Write down a table which gives the values of `Bnegate` and `Operation` required to ensure the big ALU performs the various operations.

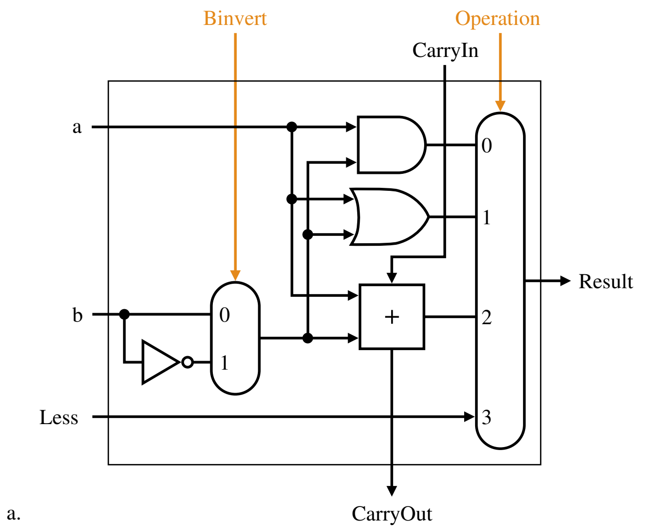
4.5 Sequential circuits

4.5.1 The Fundamental Idea

A combinational circuit implements a function between n-tuples with components from $\mathbb{B}$ and m-tuples with components from $\mathbb{B}$, that is a function $\mathbb{B}^n \to \mathbb{B}^m$. There are two key points encapsulated by the definition of *function*. These are

(i) for each input n-tuple, there *is* an output m-tuple; and

(ii) this m-tuple is *unique*.



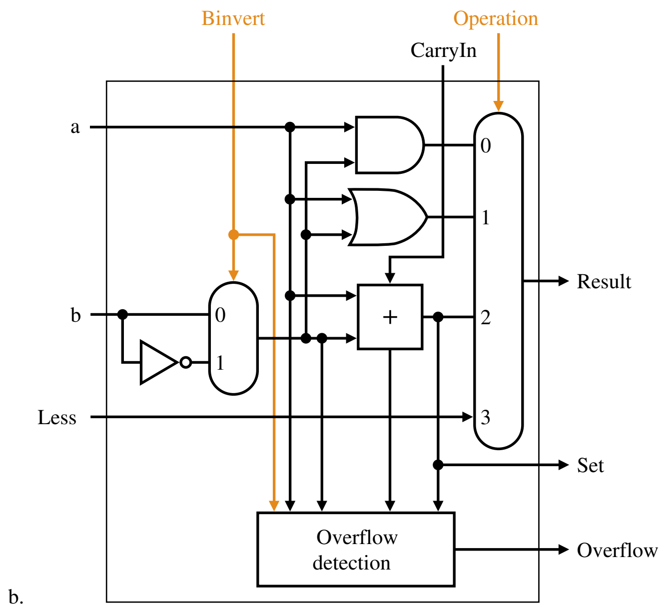


Figure 4.6: ALU Step 2

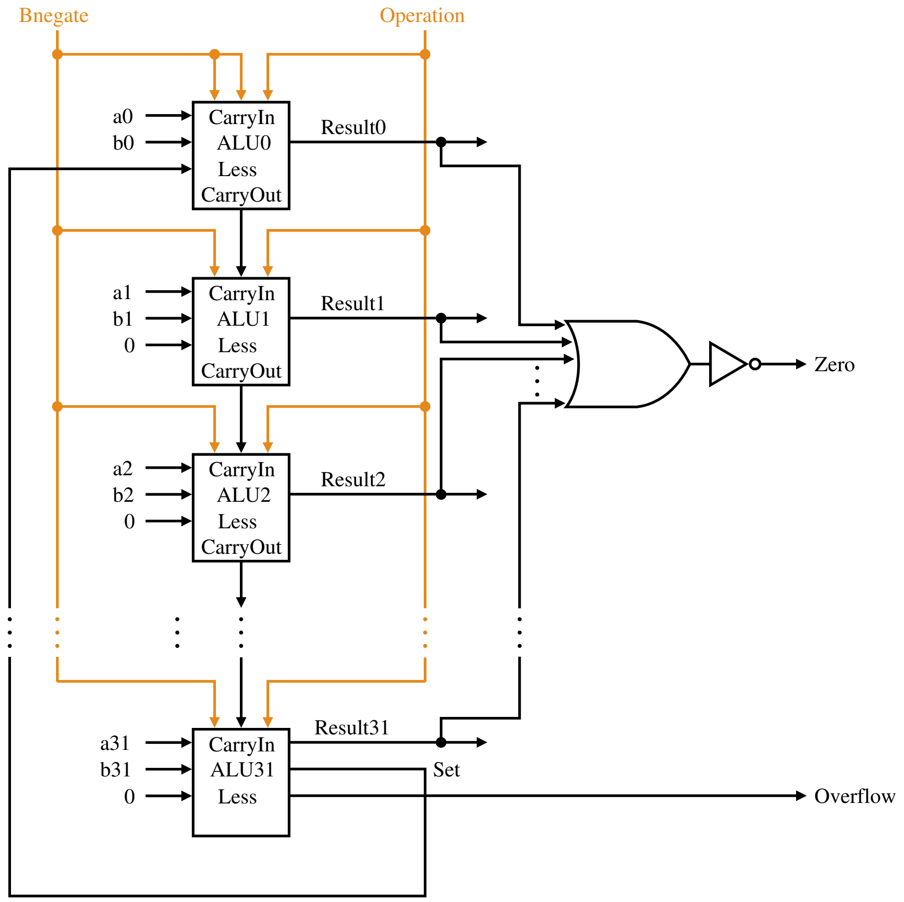


Figure 4.7: ALU Step 3

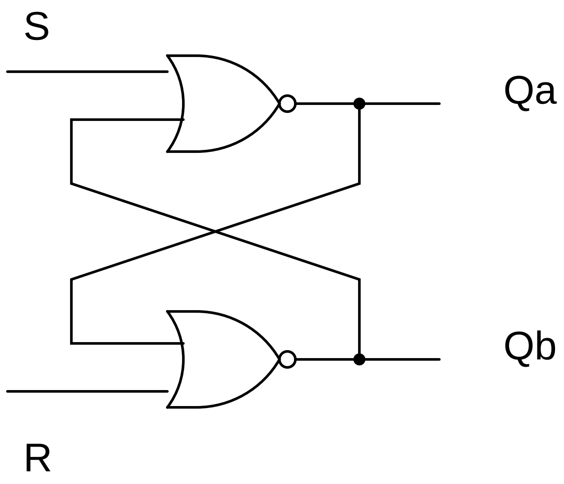


Figure 4.8: An SR Latch

A **sequential** circuit is a more general notion. Suppose that we specify an input tuple and output tuple. If the circuit "allows" this input and output, we say that the circuit is in a **stable** state. If not, we say that the circuit is in an **unstable** state. In practice, for each input tuple, there may be many output tuples giving a stable state. Thus sequential circuits are not functions (in fact they are simply relations). Examples appear in Section 4.5.2, where it will become clear what we mean by "allows."

4.5.2 Latches

SR Latches Figure 4.8 shows a circuit called an **SR latch**. It consists of two 2 | 1-ary **NOR** gates wired together, and has two input lines (S, R) and two output lines (Q_a, Q_b). A NOR gate is composed of an OR gate followed by a not gate. *Exercise: Write down the truth table for NOR before reading on.* An SR latch has only five stable states, and (thus) eleven unstable states; some of these are given in Table 4.3. We can check that the input tuple $(S, R) = (1, 0)$ and output tuple $(Q_a, Q_b) = (0, 0)$ give rise to an unstable state: the lower NOR gate has an input tuple $(Q_a, R) = (0, 0)$ and hence Q_b must be 1. But $Q_b = 0$ by assumption, a contradiction. Thus the I/O state $((1, 0), (0, 0))$ is *not allowed.*

Exercise 4.5.1 Check some of the other states to see if they are stable or not.

Look at the stable states, and in particular S, R and Q_b, and *ignore the final row of the stable state table.* Note that when S and R are both 0, Q_b can be anything. When S is 1, Q_b is 1, and when R is 1, Q_b is 0. We call S the **set** line and R the **reset** line, and think of Q_b as a signal which can be set (to 1) and reset (to 0).

S	R	Q_a	Q_b
0	0	1	0
0	0	0	1
1	0	0	1
0	1	1	0
1	1	0	0

Stable States

S	R	Q_a	Q_b
1	0	0	0
0	1	0	1
0	0	1	1

Unstable States

Table 4.3: Examples of SR Latch States

We shall soon use this property to design computer memory. The basic idea is that an SR latch will be used to make a *1-bit memory*. This will store a binary digit as the value of Q_b. We will change the stored value using the S and R lines.

Note that ignoring the final row of stable states, the values of Q_a and Q_b are complements, and are often written as $\overline{Q}$ and Q. Note that $\overline{Q}$ and Q are merely *symbols* for the output lines, and that it is quite possible for their *values* not to be complementary: this occurs in the final row above. However, in the circuits we use to build memory, we will force the values of Q_a and Q_b to be complementary, so the final stable state never arises in practice.

Clocked D Latches A good model of a simple 1-bit computer memory is a padlocked box in which one can store just 0s and 1s. This box has a glass window through which one can see its contents. Thus we can unlock the box and *store* either a 0 or a 1; locking the box ensures that the binary digit is *stored permanently*. One can then see the stored binary digit through the window, and *read* the value as required. *Note: The words **input/write/store** are often used in different situations, but mean the same thing. The same goes for **output/read**.*

We can use the SR-latch to construct a circuit which captures this idea; it is called a **Clocked D-Latch**, and is illustrated in Figure 4.9. Such a clocked D-Latch is a basic circuit for a **1-bit computer memory**. The input **data line** D holds the data (0 or 1) which we want to store (what we put in our box). The other input line, called the **clock line**, provides a signal which indicates when data can and cannot be stored (analogous to unlocking and locking the box). (It is sometimes called a **write enable** or **input enable** line.) The output line Q holds the value of the stored data (the contents of our box). *Note: Try not to be confused by the fact that the stored/written data corresponds to the value on the "output" line Q.* Measuring the voltage on the Q line amounts to reading the stored data (looking through the box window). Note

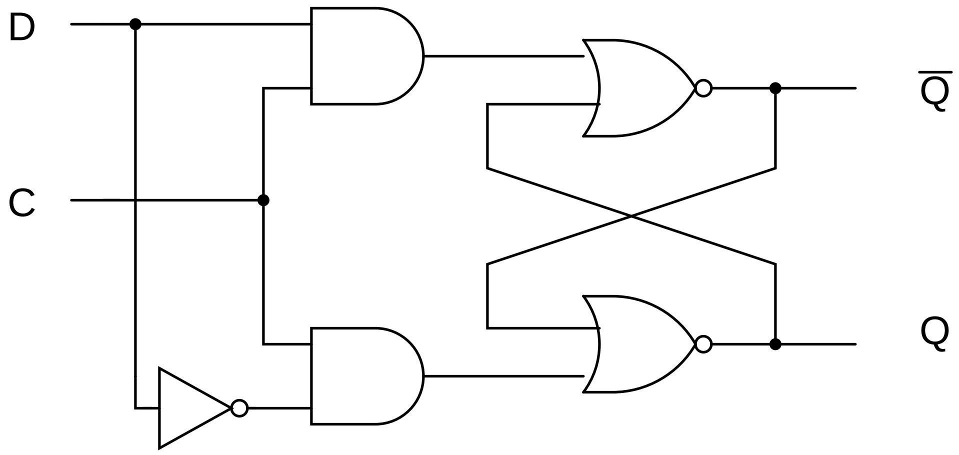


Figure 4.9: A Clocked D-Latch

the presence of the NOT gate—this ensures that the final stable state of Section 4.5.2 never arises.

Exercise 4.5.2 By considering the values that C and D can take, check that the final stable state can never arise.

When the clock line (write enable line) holds 1 (unlocked), you can check that we must have $Q = D$, ie Q takes on the value of the data line (data in box). When the clock is 0 (locked), the value of Q is whatever it was when the clock line last held 1, no matter the current value of D (we can continue to read the data through the box window, but cannot put anything new in the box). We will explain this carefully in the lectures.

Notice that the clocked D-latch model as a "locked window box" is a little strange. We might normally think of actually removing the contents of a locked box, rather than just observing the contents through a window. In an actual circuit, the voltage on the Q line is always present, whether high or low; the continuing presence is analogous to peering through the window.

In fact, in an actual circuit, we shall sometimes want to be able to "turn off" the Q voltage completely—memory circuits are often connected together, and this will prevent data corruption. To do this, we use a **tri state buffer**, depicted in Figure 4.10. This is an electronic "switch". There is a data input line, a data output line, and a control line. When control is high, the buffer acts as a single wire, so that the data input value is precisely the output value. When control is low, the buffer breaks the connection between the data input and output, rather like a physical switch breaks a connection. We can add such a tri state buffer to the Q line of a clocked D-latch,

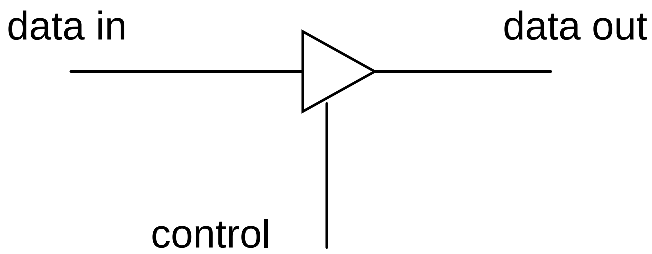
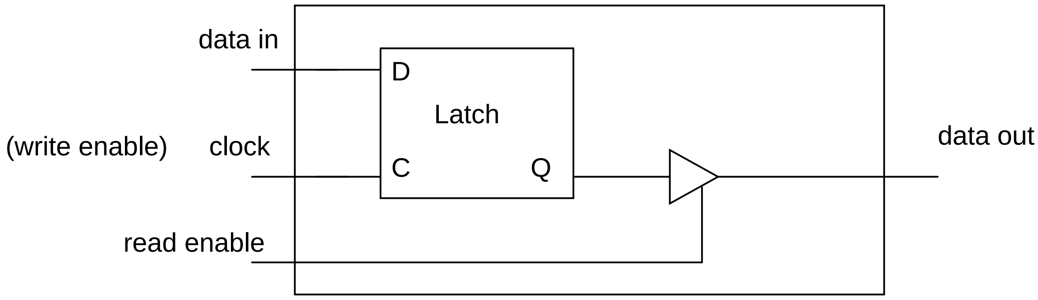


Figure 4.10: A Tri State Buffer

which will allow the stored data to be turned "on and off". In such a case, we call the control line a **read enable** line, or **output enable** line. In our model, you can think of the read enable line being used to cover up, and uncover the box window.

Example 4.5.3 Explain how to produce a 1-bit memory which has a read enable line; only when the line is high should the stored data be readable.

Answer:



Registers A k-bit **register** is simply a collection of k 1-bit memories, that is k clocked D-latches, which have their clock lines *wired together* to form a single write enable line. The idea is simply that we have k output lines (Q lines) each holding a "stored" digit, k input data lines (D lines), and just one write enable line which simultaneously "locks and unlocks (enables and disables)" the input lines. If the write enable line is high, each of the k clock lines is high, so each clocked D-latch can be written to.

In Chapter 6 we describe a CPU in detail. One of the main components is a **register file** which consists of

- a set of k-bit registers (each register has a **number**);
- a set of **read-register** busses and a set of **read-data** busses (the sets are the same size);

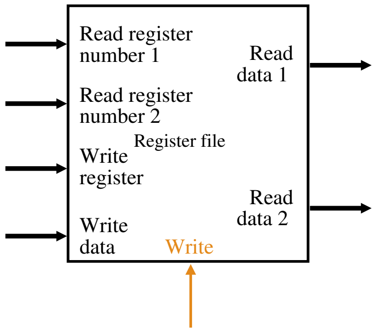


Figure 4.11: A Register File

- a set of **write-register** busses and a set of **write-data** busses; and
- a **write enable** line.

Here is the basic idea behind a register file. As a computer program runs, the CPU will sometimes require data from main memory. In order for the CPU to use these data, they will be *written* into registers in the register file. And the CPU will sometimes wish to send data from the register file back to main memory. In order for the CPU to send these data, they will be *read* from registers in the register file.

Writing to the Register File: In order to copy a word from main memory into the register file, we need a bus to carry each word—a*write-data bus*. We also need to specify into which register the word is to be stored. We do that by specifying the number of the register, which is carried on a *write-register bus*. We also need a *write enable line* to ensure that the registers can be written to, that is, writing is enabled. The *write enable line* for the register file is connected to the *write enable line* of each register.

Reading from the Register File: In order to read a word from the register file into main memory, we need a bus to carry each word—a*read-data bus*. We also need to specify from which register the word is to be read. We do that by specifying the number of the register, which is carried on a *read-register bus*. The *write enable line* is normally low when a read takes place. Sometimes there may be a read enable line.

In practice the sets of busses can be very small. A simple register file is given in Figure 4.11, in which there are just two read-register busses with corresponding data busses, and only one write-register bus, and one write-data bus. The write enable line is denoted by just `Write`. There is no read enable line.

Exercises 4.5.4 Consider a very simple register file. It has eight 1-bit registers (so each register is simply a clocked D-latch), a read-register bus of 3 lines, and a read-data line. It *also* has a read enable line: if this is high, output can be read from the read-data bus. Draw a circuit diagram which implements this register file, using eight 1-bit clocked D-latches, a multiplexor with $3+2^3$ inputs, and a tri state buffer.

4.6 Computer Memory Circuits

4.6.1 Building Static Random Access Memory

Note the meanings of the following prefixes.

Kilo (K)	2^{10}
Mega (M)	2^{20}
Giga (G)	2^{30}
Tera (T)	2^{40}

A finite number of 1-bit memories each of which is written to simultaneously, provides a basic building block for computer memory. For example, the MIPS computer has 8-bit cells (bytes) as the smallest addressable memory unit.

A computer's *fast* main memory is built from **SRAM**s, that is **Static Random Access Memory**s. An abstract diagram of an SRAM appears in Figure 4.12. An SRAM is an integrated circuit. It consists of a number of cells; in the example, the cells are 8 bits wide, that is bytes. Each SRAM provides a fixed number of cells; in the example there are $32K = 2^{15}$ cells. The chip has an address bus which is used to indicate which cell will be engaged in a read or write. The example has a bus with 15 lines, which can be decoded into 2^{15} addresses. Each cell can be written or read, requiring a data-input (data-write) and data-output (data-read) bus; the example busses are each 8 bits wide. There is a **chip select** line. If this is high, there is a possibility for the chip to be read or written to. If low, no reading or writing can take place at all, and the chip is essentially an isolated unit. Chip select "switches the chip on and off". There is a **write enable** line, which is connected to the cells' clock lines, and an **output enable** line. When Chip select is high, these two control lines determine whether writing or reading can actually take place. (Output enable might also be called read enable.)

The number of bits in a cell is sometimes called the chip's **width**, and the number of cells its **height**. We talk about an $h \times w$-SRAM; the example is a $32K \times 8$-SRAM. Figure 4.13 gives some details of the construction of a 4×2-SRAM. Note that the Chip Select and Output Enable lines have been omitted. Note that the `Enable` lines on each bit control *output* from the individual bits—see Example 4.5.3.

Exercise 4.6.1 Draw in the Chip Select and Output Enable lines, together with any circuits you need to implement these lines.

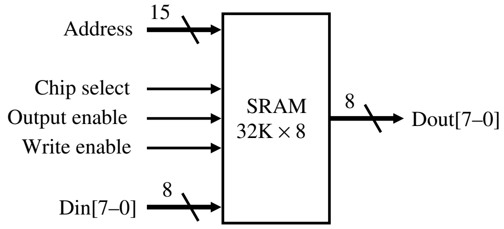


Figure 4.12: A 32K x 8 SRAM

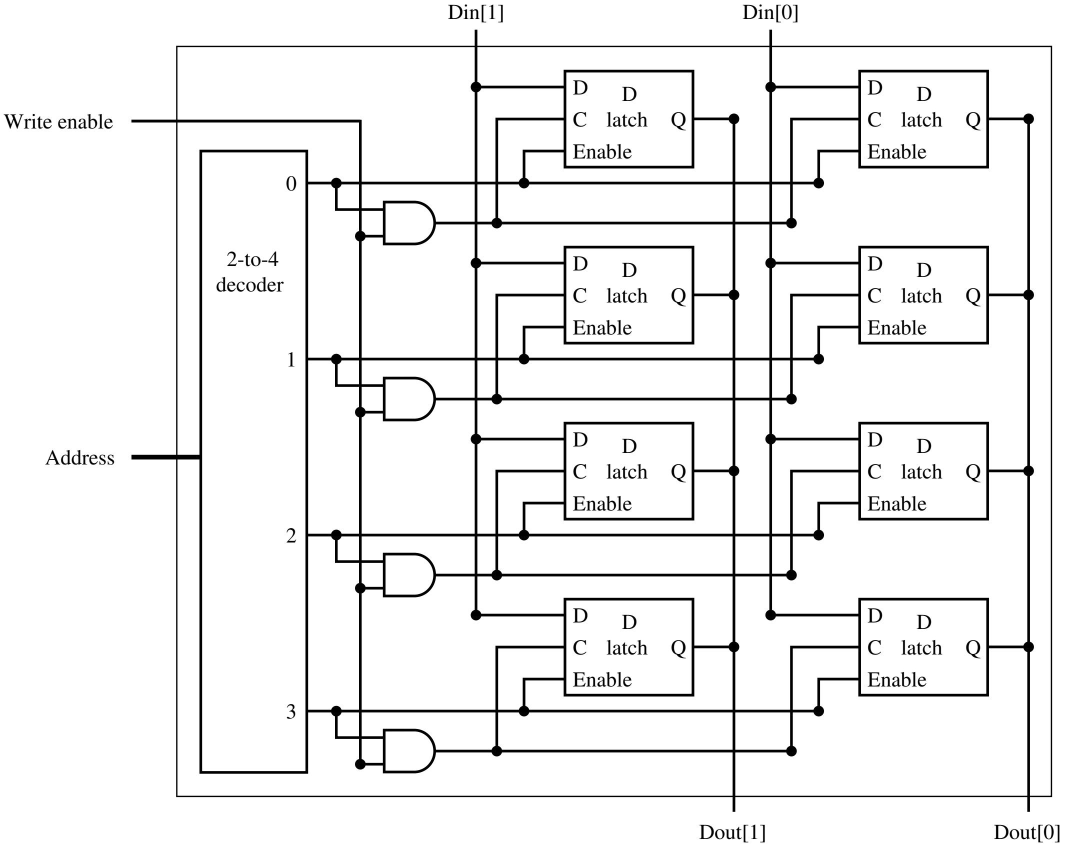


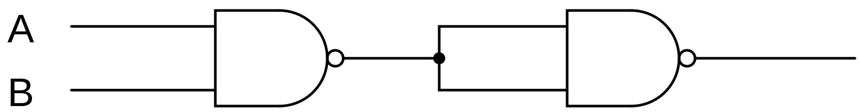
Figure 4.13: A 4K x 2 SRAM

4.7 Further Examples

Examples 4.7.1

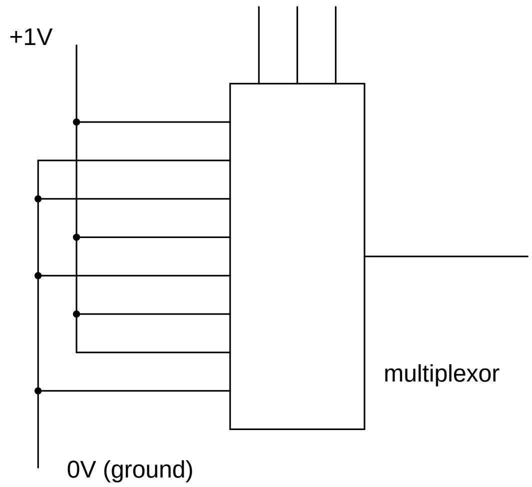
(i) A (binary) **NAND** gate implements a function $\mathbb{B}^2 \to \mathbb{B}$. The output value is the negation of the 'and' of the input pair. Show how to implement an AND gate using two NAND gates.

Answer hints (you should give more details):



(ii) The function $f\colon \mathbb{B}^3 \to \mathbb{B}$ has value 1 if an even number of input components are 1, and is 0 otherwise. Show how to wire up a $3+2^3 \mid 1$ multiplexor to produce a circuit which implements f. You may assume that you have a power supply.

Answer hints (you should give more details):

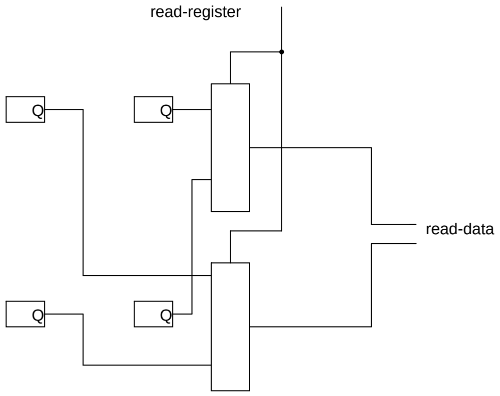


(iii) In this question, a bit is a clocked D-latch which does **not** have an output (read) enable line. In each answer, you should show one `Read Register` bus and one `Read Data` bus. *You are not asked to show any `Write` busses or their associated circuit details.*

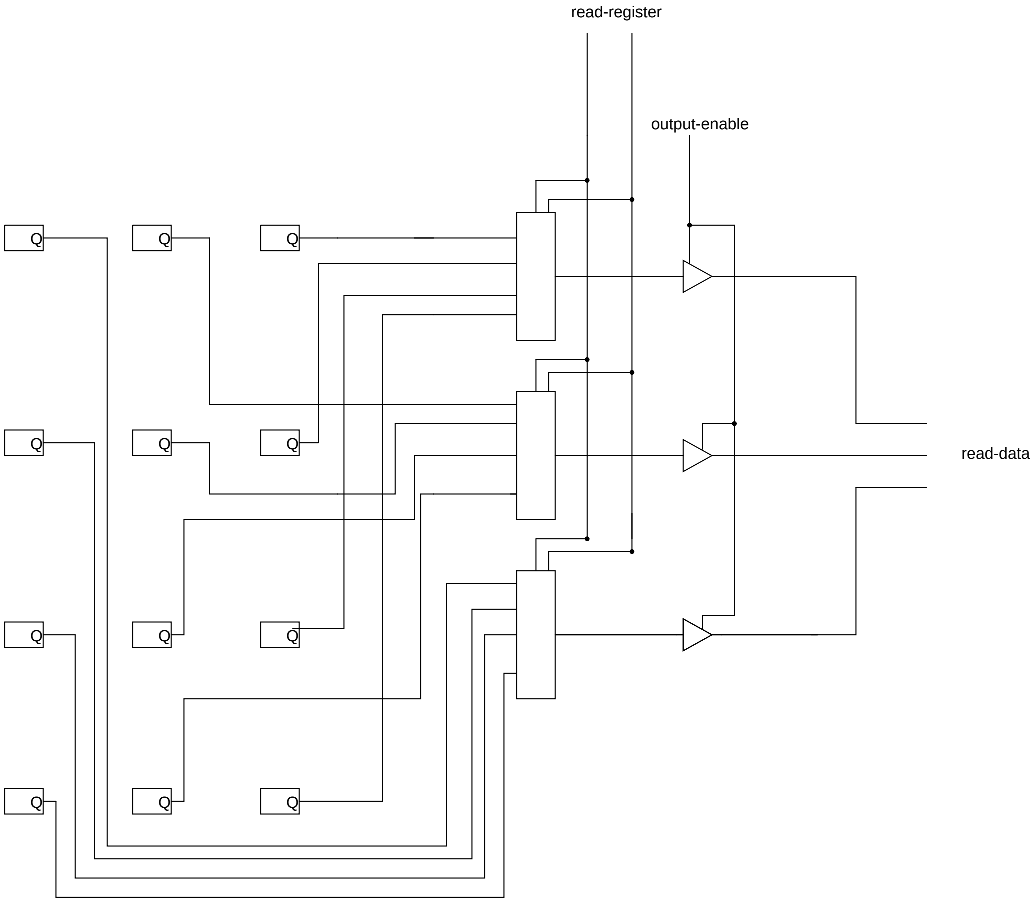
1. Draw a circuit diagram of a 2×2-bit register file, built from four bits and two $1+2 \mid 1$-ary multiplexors.
2. Draw a circuit diagram of a 4×3-bit register file with an `Output Enable` line, built from twelve bits, three $2+4 \mid 1$-ary multiplexors and three tri-state buffers.

Answer hints (you should give more details):

1.



2.



5

The Instruction Set Architecture Level

5.1 Introducing the MIPS Instruction Set Architecture

In Chapter 2 we saw that a modern computer contains a processor. We also saw that we can write simple programs whose instructions can be executed by the processor. In this chapter we explain in detail what many of these instructions look like, and how they are represented inside a computer. The instructions are regarded as a programming language; we call this language the **instruction set architecture (ISA)**. In this chapter we shall introduce the **MIPS** ISA. This is the language which is used to provide instructions for the *MIPS R2000* processor. Before looking at an example MIPS instruction, we explain a few details about the MIPS R2000. This processor has a register file which contains 32 registers. These registers are used to store

- data which has been copied from main memory;
- the results of intermediate computations performed by the CPU;
- the values of arguments supplied to functions, procedures and methods;
- the final values of computations; and
- signals used in the control of the datapath;

and we will explain the details during the course of this chapter.

The MIPS R2000 ISA instructions can be supplied in two forms, namely in that of *machine language* and *assembly language*, which are described in this chapter. We illustrate the idea using registers. First, recall that in Section 2.3 we defined a register as a sequence of k bits, with k being 32 for the MIPS R2000. In fact the MIPS R2000 provides a 32-bit computer, that is, all word locations are 32 bits long, both registers, and word locations in the main memory.

So far, we have referred to registers using a symbolic notation. For example, on page 12, we used A and B to denote registers. Using a symbol to denote a register is an example of *assembly language*. In the MIPS R2000, we mentioned that there are only 32 registers. Each of these is denoted by a special assembly language symbol. A typical example is `$s4`. Please see Table 5.3 for the complete set. If we wish to refer to an arbitrary register, we will call it R.

We can either refer to a specific register using its assembly symbol (eg `$s4`) or its **register number**. In the MIPS R2000 register file these numbers are 0 to 31 and their binary representations are the machine language numbers for the 32 registers. *Note:*

The MIPS R2000 has 32 registers, and each register has 32 bits. Do not let this confuse you!

Let us expand these ideas. Recall the simplified datapath of Section 2.3.3. When the computer executes an instruction involving addition, the operands are copied from the registers into the ALU, added, and the result stored in a register. MIPS provides an assembly language instruction to perform this task; an example is

```
add $t0, $t1, $t2.
```

The register `$t0` will hold the result of the addition. The operands are located in registers `$t1` and `$t2`. This assembly language instruction has a corresponding form in machine language, which looks like the middle row of

add	`$t1`	`$t2`	`$t0`	–	*add*
000000	01001	01010	01000	00000	1000000
6 bits	*5 bits*	*5 bits*	*5 bits*	*5 bits*	*6 bits*

in binary notation. There are six sections representing the instruction. Each section contains a fixed number of binary digits. The total number of digits is 32, so the machine language instruction will fit into a word location. Each section is called a *field*, and codes up part of the assembly instruction. For example, the first and last fields together tell the machine that the word represents an instruction to add (the reason that two fields are used will be explained later on). The second field is the actual machine language number of the (assembly) register `$t1`. This number is 9^{dec}. Notice that we said the MIPS R2000 has 32 registers. The register numbers can be represented in the machine using 5-bits (why?), just as in our example. We will explain all of these ideas in greater detail later on.

Now we look at an example MIPS program

```
begin:  addi $t0, $zero, 0
        addi $t1, $zero, 1
        addi $t2, $zero, 8
repeat: add $t1, $t1, $t0
        addi $t0, $zero, 1
        bne $t0, $t2, repeat
        sw $t1, 0, $t3
finish:
```

We shall look at this program in detail in the lectures. This will give you a flavour of how MIPS instructions work, before we go on to the precise details. *Exercise:* *Work out for yourself what the above program does; hint* `bne` *stands for "branch if not equal" and* `sw` *for "store word".*

In order to explain the MIPS assembly language, we need to give an abstract definition of a program. A MIPS *program* consists of a finite sequence consisting of either instructions or **labelled instructions**, each labelled instruction of the form `L : I`. Here `I` is a MIPS instruction, and `L` is a symbolic label. The program **executes** by having each instruction in the sequence executed on the datapath. By default, the instructions are executed in order, first to last. However, sometimes an instruction contains a label, and the next instruction to be executed is the one indicated by the label.

The instructions we look at fall into groups known as **categories**. These are **arithmetic**, **logical**, **data transfer**, **conditional branch** and **jump**. Later, we will see which kind of instructions appear in which category.

5.2 MIPS Assembly Language

5.2.1 Registers, Locations, Assignment and Semantics

We shall use the symbol R to represent any one of the 32 MIPS registers. We will be careful to distinguish a symbol denoting a location, and the contents of the location. Recall that each register is an example of a location. Given a register, such as R, we denote its contents by $!R$. This will be a 32-digit word.

As well as the 32 registers, we shall also want to refer to main memory word locations and byte locations. We have already defined a *byte location address* (a byte location being the smallest addressable unit in a MIPS computer, an 8-bit cell) and a *word location address*. It is convenient to have a notation to refer to the byte location at main memory address a. The notation we use is $B[a]$. Similarly, $W[a]$ denotes the word location with word address a. It is also convenient to have a special notation to denote the contents of byte locations and word locations: we write $!B[a]$ for the *contents* of $B[a]$, and similarly for word locations. We can picture this as in Figure 5.1. Note that MIPS R2000 provides a 32-bit computer. Program instructions will be held in consecutive word locations in main memory. Because the cells are 8 bits long, *the addresses of these word locations will always be multiples of* 4. Such addresses are said to be **aligned**. Finally, the MIPS R2000 processor will always operate in big endian format.

In this chapter we shall define the *semantics* of MIPS instructions. An instruction will be written down using syntax such as assembly language. Its **semantics** is a description of *what* actually happens in a computer when the instruction is executed. The semantics does not tell us *how* the instruction is executed—this is a topic of Chapter 6.

In order to describe the semantics of instructions we shall use **assignments**. An assignment takes the form $R := \omega$ where R denotes a register, and ω is a word. If the register has k bits, the word ω must be a sequence of k binary digits. The assignment means that the contents of R are changed or **updated** to ω. Thus we can describe the semantics of the instruction `add` R_1, R_2, R_3 as $R_1 := !R_2 + !R_3$. This describes *what* happens inside the machine when the instruction is executed. The $+$ means digitwize

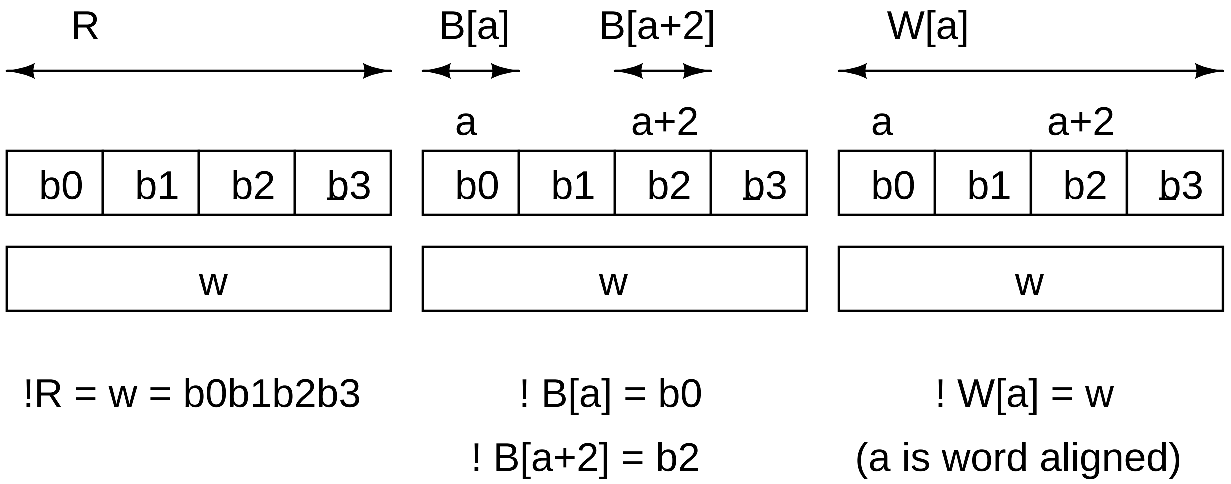


Figure 5.1: Locations and Contents

sum of two 32 digit binary numbers, namely $!R_2$ and $!R_3$.

We shall make use of a convention. We may write things like $R := 3$ where 3 is an integer. The convention is that this will be shorthand for $R := \vec{0}11^{bin}$ (where there are 30 zeros). We will also omit the ^{bin} when no confusion can arise. Thus $R := 3$ means that the binary representation of 3 will be stored in R.

5.2.2 Addressing Modes

Consider the two instructions

`addi` R_1, R_2, 5   `add` R_1, R_2, R_3

We refer to `add` and `addi` as instruction **names**. The registers R_i and the integer 5 are called instruction **arguments**. In each case, the instruction name tells us, in principle, what function or task or operation occurs at execution time. Here, it is some form of addition. This is part of the instruction's semantics. The arguments of the instruction tell us *where* we find the data to be processed or acted on by the instruction, and where to store the result. In the case of `add` above, the registers R_2 and R_3 contain data to be processed by addition. We shall call them **source** arguments. In both instructions, the register R_1 specifies where the resulting sum is to be stored. We shall call it a **destination** argument. Finally, we refer to the contents of a source argument as an **operand**.

For example, in `addi`, the number 5 is both an argument and operand. The `i` stands for *immediate*. We find the operand data "immediately", it being the number 5 itself, which will be used in the sum. In `add`, the register R_3 is a source argument. Here, the operand data is the contents of R_3. In both instructions, R_1 is the destination argument, and R_2 a source argument.

In general, there are a number of different ways in which source arguments allow us to *locate* operands. The operand might be given directly/immediately, reside in a register, or be in main memory. There are other possibilities, which we describe here. In general, each MIPS source argument has a specific **addressing mode** which tells us how to locate the corresponding operand.

Immediate Addressing The operand is specified *directly* as part of the instruction. For example, in `addi $t0, $t1, 24`, the number 24 is an operand which is part of the instruction. We sometimes refer to operands given by immediate addressing as **constants** . The "i" in `addi` means *immediate*. The semantics of the instruction is given by `$t0` $:=$`!$t1` $+ 24$, which shows "immediate" use of the 24 in the sum. Thus the 32 digit binary representation of 24 will be added digitwize to `!$t1` and the result stored in `$t0`.

Register Addressing The operand is given as the contents of a register, and the register is named in the instruction. If the register is called R, the operand is $!R$. As we have seen, `add` uses register addressing for its source arguments. The semantics of `add $t0, $t1, $t2` is given by `$t0` $:=!$`$t1`$+!$`$t2`. This notation shows very clearly the use of register addressing to obtain the two operands. Register addressing is sometimes also called register *direct* addressing.

Register Indirect Addressing Suppose that an instruction involves a source argument R. Register indirect addressing says that the operand is held in main memory at an address, a say, and the address a is itself given by the contents of R. In this case, we refer to R as a **pointer**. A main memory operand may come either from a byte location or word location. Thus the operand specified by R is either $!W[!R]$ or $!B[!R]$.

Indexed Addressing It is often the case that we will need to load registers with a sequence of words from memory, and they will all be found "quite near" to one particular address, called the *base* address. An analogous idea is getting books from a library on one subject; once you have found one, the others are likely to be nearby. Indexed addressing allows the computer to use this idea. The source argument will usually look something like $k(R)$ where R is a register, and k is an immediate constant. The contents of R are the **base**. k is called an **offset**, and indicates how far our data is located from the base address. Thus the operand is then held in memory at the address $!R+k$ which is the digitwize sum of $!R$ and the 32 digit binary representation of k. We will see that the operand may well be either a byte or a word, given by the contents of $B[!R+k]$ or $W[!R+k]$.

Example 5.2.1 Suppose that $!R=\vec{0}1101$ and that $k=2$. What is the binary main memory address given by the source argument $k(R)$?

Answer: The address is given by the digitwise sum $!R+k$ as defined above. The binary for k is $\vec{0}10$. So the address is $\vec{0}1111$.

Direct Addressing The source argument for **direct addressing** is always a number, say a. Unlike immediate "addressing", with direct addressing this number refers to a location whose address is a. Thus the operand is held in main memory at address a, and is defined to be $!W[a]$. This addressing mode is not used in MIPS.

Exercise 5.2.2 Draw pictures of memory to illustrate each of the addressing modes. Some of these will be given in lectures, but try the exercise *before* coming!

We shall now look at the syntax and semantics of various categories of instructions. Before doing this, we have to explain a rather subtle point. So far, k has denoted an integer, which may in fact be negative. Thus when we write, say, $!R+k$, we actually *mean add the 32 digits of* $!R$ *to a 32 digit 2s-complement representation of* k. In fact in all of the instructions we meet in this module, $k \in \mathbb{Z}$ must be stored in a computer *within 16 bits*. Thus we shall restrict k to the range

$$-2^{15} \leq k \leq 2^{15} - 1$$

which can be represented as a 16-bit 2s-complement number. Moreover, there will be situations when the 16-bit representation of k is copied into a 32-bit register. When this happens, the most significant 16 register bits will be filled with 0s or 1s by either zero extending or sign extending the 16-bit 2s-complement representation of k. Please re-read Section 3.6.

5.2.3 Arithmetic Category Instructions

These instructions all function in a similar manner to the `add` instruction. The table below shows some of those functions we shall be interested in

Syntax	Semantics
`add` R_1, R_2, R_3	$R_1 := !R_2 + !R_3$
`sub` R_1, R_2, R_3	$R_1 := !R_2 - !R_3$
`addi` R_1, R_2, k	$R_1 := !R_2 + sx(k)$

Note that R_2, R_3 and k are all source arguments. The registers R_i employ register addressing, and k immediate addressing. This is indicated in the semantics column. The integer k will be stored in the computer using 16 bits. In order for the 32-bit ALU to add k to $!R_2$, a circuit will be used to sign-extend the 16-bit 2s-complement number which represents k to 32 digits. We write $sx(k)$ for these 32 digits (even though, strictly speaking, the sign extension is a function $B^{16} \to \mathbb{B}^{32}$).

Example 5.2.3 Suppose that R_2 contains 00000000.01000000.00000000.00000111 and that $k = -32,760$. What does R_1 hold when `addi` R_1, R_2, k is executed?

Answer: We look for a power of 2 near 32760. In fact $k = -2^{15} + 8$. Thus the 16-digit 2s-complement representation of k is 10000000.00001000. It follows that the 32-digit representation of k is the sign extension of this value, namely

$$sx(10000000.00001000) = 11111111.11111111.10000000.00001000$$

Hence R_1 will hold 000000000.00111111.10000000.00001111. Note: You can check this by calculating the corresponding integer sum.

5.2.4 Logical Category Instructions

These instructions all function in a similar manner to the `add` instruction, but employ logical operations. The logical functions are the *digitwise AND* and *OR* (see page 37). The table below shows those functions we shall be interested in

Syntax	Semantics
`and` R_1, R_2, R_3	$R_1 := !R_2\ AND !R_3$
`or` R_1, R_2, R_3	$R_1 := !R_2\ OR !R_3$
`andi` R_1, R_2, k	$R_1 := !R_2\ AND\ zx(k)$
`ori` R_1, R_2, k	$R_1 := !R_2\ OR\ zx(k)$

The notation $zx(k)$ means "zero-extend the 16-bit 2s-complement number which represents k."

Example 5.2.4 Let $k = -32760$. Compute the contents of R_1 when `ori` R_1, R_2, k is executed. R_2 holds $b = 11111111.11111111.01111111.11110111$.

Answer: k is represented in 16-bits by $b' = 10000000.00001000$. Hence

$$zx(b') = 00000000.00000000.10000000.00001000$$

So R_1 will hold $b\ OR\ zx(b')$ shown below

$$\begin{array}{c} 11111111.11111111.01111111.11110111 \\ 00000000.00000000.10000000.00001000 \\ \hline 11111111.11111111.11111111.11111111 \end{array}$$

5.2.5 Data Transfer Instructions

We have seen that data (and instructions) must be copied from main memory and stored in registers to be processed by an ALU, and vice versa. When data is copied from main memory into a register, we say it is **loaded**. When data is copied from a register into main memory, we say it is **stored**. Try not to let these words confuse you—the word "stored" applies to many situations, not just copying data from registers to main memory. These forms of copying are referred to as **data transfers**. We examine various examples.

Loading and Storing Words A example of a *load word* instruction looks like

`lw $s1, 0($s2)`

Informally, the semantics of the instruction entails finding a word which is stored in main memory, and then copying it in the register `$s1`. The required word is found using the source argument `0($s2)` which employs indexed addressing. The base address is $a \stackrel{\text{def}}{=} !\$s2$ and the offset is 0. Thus the word we seek is $!W[a]$.

The general form of the load word instruction is `lw` R_1, $k(R_2)$. The semantics is similar to the above example, but we select the word location W with address the sum of k and $!R_2$, and store the contents of W in R_1. Note that R_2 is the base address, and that the constant k is the offset. Do not forget that k is stored in 16 bits, and that the machine will perform a sign extension.

Storing (copying from a register to main memory) is accomplished using instructions of the form `sw` R_1, $k(R_2)$. Informally, we calculate a memory (word) address as the sum of k and the contents of the base register R_2, and then copy the (word) contents of R_1 into main memory at the computed address. In summary:

Syntax	Semantics
`lw` R_1, $k(R_2)$	$R_1 := !W[sx(k)+!R_2]$
`sw` R_1, $k(R_2)$	$W[sx(k)+!R_2] := !R_1$

Loading and Storing Bytes Recall that we work with 8-bit cells, that is, a byte addressable machine. A example of a **load byte unsigned** instruction looks like

`lbu $s1, 0($s2).`

When this is executed, we examine the contents of `$s2`, which contains a memory address, say a. We then use a as a byte location address, and we store the byte located at a in the register `$s1`. The register contains a word, which consists of four bytes, so where do we place the byte $!B[a]$? The `u` stands for *unsigned*. We place $!B[a]$ in the least significant `$s1` bits, and set the other $32-8=24$ bits to 0. Thus the semantics of the instruction is `$s1` $:= \vec{0}\vec{0}\vec{0}!B[a]$, where $\vec{0}$ denotes a byte of 0s. $B[a]$ has been zero extended.

There is also a **load byte** instruction. An example is `lb $s1, 0($s2)`. In this case, we examine $!B[!\$s2]$ which we regard as an 8-bit 2s-complement binary number. We then copy it to the register `$s1` in the least significant bits (as before) but *sign extend* the register. Thus, for example, if $!B[!\$s2]$ represents a negative integer, then `$s1` $:= \vec{1}\vec{1}\vec{1}!B[!\$s2]$. In the table below, we use the functions for zero and sign extension given on page 38 to express these semantics.

Finally we have a **store byte** instruction. This works like `sw` except that the least significant byte in R_1 is stored into main memory. The contents of $!R_1$ consists of the four bytes β_3 to β_0.

Syntax	Semantics
`lb` R_1, $k(R_2)$	$R_1 := sx(!B[sx(k)+!R_2])$
`lbu` R_1, $k(R_2)$	$R_1 := zx(!B[sx(k)+!R_2])$
`sb` R_1, $k(R_2)$	$B[sx(k)+!R_2] := \beta_0; \qquad \beta_3\beta_2\beta_1\beta_0 \stackrel{\text{def}}{=} !R_1$

Loading Constants We can load a constant into a register using the instruction

`addi` R, `$zero`, k

because its semantics is $R := \vec{0}+k$. The contents of `$zero` are 32 zeroes.

5.2.6 Conditional Instructions

A conditional branching instruction is one that will perform a small computation, such as checking if two numbers are equal, and then depending on the result will make program execution change to one of two possible instructions. We say that execution has two **branches**. Where it branches to depends, that is, is *conditional* on, the result of the test. Such an instruction looks like `bne` R_1, R_2, L where R_1 and R_2 are any of the MIPS registers, and L is a symbol known as an **instruction label**. (Of course, there is a precise definition of the syntax for such labels, but this does not concern us here). We have seen an example of `bne` before

```
repeat :  add $t1, $t1, $t0
          addi $t0, $zero, 1
          bne $t0, $t2, repeat
          next − instruction
```

Here, we use a label to execute repeatedly a piece of code. To execute `bne` we check to see if !`$t0` ≠!`$t2`. If true, we execute (we say "branch to") `add $t1, $t1, $t0` again. If false, we execute `next − instruction`. There is also an instruction to branch if two numbers are equal, detailed below.

As well as branching to a new instruction, we may just want to record if a condition is true or not. The instruction `slt` R_1, R_2, R_3 checks to see if the contents of R_2 is strictly less than those of R_3. If so, R_1 is set to 1 (that is, it will hold $\vec{0}1$), and if not to 0. Please refer back to page 49.

Syntax	Semantics
`beq` R_1, R_2, L	if $!R_1 = !R_2$ then goto L
`bne` R_1, R_2, L	if $!R_1 \neq !R_2$ then goto L
`slt` R_1, R_2, R_3	if $!R_2 < !R_3$ then $R_1 := 1$ else $R_1 := 0$

5.3 MIPS Machine Language

5.3.1 Introduction

We said that a MIPS assembly program was a sequence of (possibly labelled) instructions. When writing an assembly program, we do not put labels on all the lines. However, we could be obtuse, and put labels on each line, some being redundant. Then a MIPS program will always look like this $\mathtt{L_1} : \mathtt{I_1}, \mathtt{L_2} : \mathtt{I_2}, \ldots \mathtt{L_n} : \mathtt{I_n}$. The corresponding machine language program *will* always have this kind of form. The labels will be compiled to actual machine addresses, and each instruction will be represented as a 32-digit word. Recall that the PC is a register whose contents indicate the next instruction to be executed; the contents will be machine addresses which correspond to the labels $\mathtt{L_i}$.

Format	Fields					
R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	address/immediate		
J	op	target address				
# Bits	6	5	5	5	5	6

Table 5.1: Instruction Formats

5.3.2 Instruction Fields

Given any MIPS assembly instruction, what is the corresponding 32-bit machine language instruction? To answer this, we first explain instruction *fields.* Each machine instruction belongs to one of three **formats**. These are known as **R**, **I** and **J** formats. In this module we discuss only R and I formats. Each format specifies how the 32 bits are divided up into sections known as **fields**. These are indicated in Table 5.1.

Before continuing, let us look at an example. Consider the instruction

$$\texttt{I} \stackrel{\text{def}}{=} \texttt{add \$t0, \$s1, \$s2}$$

This is an arithmetic category instruction, which by definition is R-format. The first field is known as the opcode (abbreviated to op); for this instruction it is 0^{dec}, and indicates an R-format instruction. The final field is the function field (abbreviated to funct); for this instruction it is 32^{dec} and indicates addition. The second and third fields indicate the source arguments, that is, they specify the *numbers* of the registers which hold the two source operands. For this instruction they are 17^{dec} and 18^{dec}. The fourth field, 8^{dec}, specifies the destination register. Finally, the fifth field is 0^{dec}. This field is called "shamt" standing for shift amount. It is not required in any instruction in this course, and will *always* hold 0. Putting this together, the above instruction is represented in the machine as the following binary sequence

$$\texttt{I} = 000000.10001.10010.01000.00000.100000$$

whose fields were given above in decimal notation. Thus the second field of 5 digits in positions 25 to 21 is 10001^{bin}. We will sometimes write $\texttt{I}[25-21]$ for these digits. See Chapter 6.

Now back to the general definitions. We explain the roles of the various fields within the three formats

R-Format In R-format instructions, the *name* of the instruction is indicated by using *both* the op field *and* the funct field.

• **op**: This field is the **opcode**. For R-format instructions, it indicates a set of possible operations which the instruction could perform. For example, if the opcode is 0^{dec}, then in this course the operation is one of the arithmetic or logical operations, or `slt`. There are other possibilities, but this course does not cover them. The exact operation performed is indicated by the funct field.

• **rs**: This field specifies the number of the first source argument register.

• **rt**: This field specifies the number of the second argument register.

• **rd**: This field specifies the number of the destination register.

• **shamt**: This field is not used in the instructions covered by this course; it will be set to zero.

• **funct**: The opcode indicates a set of operations, and the **function** field specifies the actual operation. For example, with opcode 0^{dec} and function field 32^{dec}, the operation is addition, and the instruction name is `add`.

The R-format instructions are `add`, `sub`, `and`, `or` and `slt`. Note that each instruction has the form *name* R_1, R_2, R_3. We specify the use of each field using a table:

`add, sub, and, or, slt`	R_2	R_3	R_1	$\vec{0}$	`add, sub, and, or, slt`
op	rs	rt	rd	shamt	funct
6 bits	*5 bits*	*5 bits*	*5 bits*	*5 bits*	*6 bits*

Example 5.3.1 Which field and bits are used to store the number of register R_1 in the R-format instruction `sub` R_1, R_2, R_3?

Answer: From the table, the register number is given by the rd field; this is stored in bits 15 to 11.

I-Format

• **op**: This determines the instruction name, eg load word, store word, etc.

• **rs**: This field specifies either a source or destination register.

• **rt**: This field specifies either a source or destination register.

• **address**: This field sometimes specifies an address offset, or sometimes a branch address. In both cases, it uses immediate addressing. It can also be used to specify an immediate constant in arithmetic and logical instructions; in such cases, the word "address" is rather unfortunate!

The I-format instructions appear in the following tables. Those with the form *name* R_1, $k(R_2)$ are

`addi, andi, ori, lw, sw, lb, lbu, sb`	R_2	R_1	k
op	rs	rt	address
6 bits	*5 bits*	*5 bits*	*16 bits*

The branch instructions have the form *name* R_1, R_2, L with fields

`beq`, `bne`	R_1	R_2	L
op	rs	rt	address
6 bits	*5 bits*	*5 bits*	*16 bits*

5.3.3 Translating Assembly to Machine Language

An **assembler** is a program that will translate assembly language into machine language. The assembler specifies a function which maps assembly language instructions into machine instructions. We can do this using the translation table given in Table 5.2. The details will be given in the lectures. The idea is quite simple. To translate an assembly instruction into machine language, use Table 5.2 to provide the opcode and function fields. Then use the field layouts given in this Section 5.3.2, together with the register number Table 5.3, to provide the values of the remaining fields.

Example 5.3.2 What is the machine code for `lw $s1, 13($t1)`?

Answer: The opcode for `lw` is 35. The number for `$s1` is 17 and `$t1` is 9. The 16 digit representation of $k = 13$ is 0000000000001101. Hence the answer is

10011	01001	10001	0000000000001101

Note the "reversal" of the register numbers in the machine code.

5.3.4 Branch Addressing

In the examples we have seen, symbolic labels were used in the branch instructions `beq` R_1, R_2, L. The label `L` is used to point to the next instruction to be executed. The assembler will translate such branch instructions into machine code; in particular, we have seen that a symbolic label L will be translated to a numerical (address) field which is only 16 bits long:

`beq`, `bne`	R_1	R_2	L
op	rs	rt	address
6 bits	*5 bits*	*5 bits*	*16 bits*

We might expect the address field to give the actual main memory address of the next instruction to be executed. Even if the 16 digits were used as a binary number, the largest address would be $2^{16}-1$ which is simply not big enough. And in fact because instructions are stored in word locations, whose addresses are 0,4,8,12, and so on, the largest meaningful address would be $2^{16}-4$.

10	16	op(31:26)	funct(5:0)	funct(5:0)
0	00		sll	add.f
1	01			sub.f
2	02	j	srl	mul.f
3	03	jal	sra	div.f
4	04	beq		
5	05	bne		abs.f
6	06	blez	srlv	mov.f
7	07	bgtz	srav	neg.f
8	08	addi	jr	
9	09	addiu	jalr	
10	0a	slti		
11	0b	sltiu		
12	0c	andi	syscall	
13	0d	ori	break	
14	0e	xori		
15	0f	lui		
16	10	z = 0	mfhi	
17	11	z = 1	mthi	
18	12	z = 2	mflo	
19	13	z = 3	mtlo	
20	14			
21	15			
22	16			
23	17			
24	18		mult	
25	19		multu	
26	1a		div	
27	1b		divu	
28	1c			
29	1d			
30	1e			
31	1f			
32	20		add	cvt.s.f
33	21	lb	addu	cvt.d.f
34	22	lh	sub	
35	23	lwl	subu	
36	24	lw	and	cvt.w.f
37	25	lbu	or	
38	26	lhu	xor	
39	27	lwr	nor	
40	28			
41	29	sb		
42	2a	sh	slt	
43	2b	swl	sltu	
44	2c	sw		
45	2d			
46	2e			
47	2f	swr		
48	30			c.f.f
49	31	lwc0		c.un.f
50	32	lwc1		c.eq.f
51	33	lwc2		c.ueq.f
52	34	lwc3		c.olt.f
53	35			c.ult.f
54	36			c.ole.f
55	37			c.ule.f
56	38			c.st.f
57	39	swc0		c.ngle.f
58	3a	swc1		c.seq.f
59	3b	swc2		c.ngl.f
60	3c	swc3		c.lt.f
61	3d			c.nge.f
62	3e			c.le.f
63	3f			c.ngt.f

	rs (25:21)	funct (4:0)	rt (20:16)
0	mfcz		bltz
1		tlbr	bgez
2	cfcz	tlbwi	
3			
4	mtcz		
5			
6	ctcz	tlbwr	
7		tlbp	
8			
9			
10			
11			
12			
13			
14			
15		rte	
16	copz		bltzal
17	copz		bgezal
18			
19			
20			
21			
22			
23			
24			
25			
26			
27			
28			
29			
30			
31			

	(16:16)
0	bczt
1	bczt

if z = 0

if z = l, f = d

if z = l, f = s

Table 5.2: MIPS Opcode Map

Register		Usage
$zero	0	contents always zero
$at	1	reserved for assembler
$v0	2	expression evaluation and function results
$v1	3	expression evaluation and function results
$a0	4	first argument component (preserved across call)
$a1	5	second argument component (preserved across call)
$a2	6	third argument component (preserved across call)
$a3	7	fourth argument component (preserved across call)
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area (preserved across call)
$sp	29	stack pointer (preserved across call)
$fp	30	frame pointer (preserved across call)
$ra	31	return address (after function call; preserved across call)

Table 5.3: Register Numbers and Their Usage

In practice it turns out that the next instruction to be executed is quite near to the current (branch) instruction (more detail will be given in the lectures). In fact the address field is used to measure *how far the next instruction is from the current instruction.* More precisely, it is used to indicate *how many word locations* the next instruction is from the current instruction. If the current instruction is stored in address a, and the next instruction at a', we have a picture

$$
\begin{array}{ll}
a & \boxed{\texttt{beq ......}} \\
 & \quad\vdots \\
 & \quad\vdots \\
a' & \boxed{\texttt{next ......}}
\end{array}
$$

where the boxes are word locations. Suppose that the next instruction is six word locations after the branch. What is its address a'? Because our computer's cells are byte locations, and each word location consists of four byte locations, we must have $a' = a + (4 * 6) = a + 24$. There two further complications.

- The next instruction might occur at an address *lower* than a. Thus the address field will be regarded as a 2s-complement number. It will denote an **offset** from the current instruction, which can be positive or negative.

- In a real computer, the PC is updated to point to an instruction *one word on* (with address $a + 4$) BEFORE it is updated to point to the next instruction given by a branch. The address field therefore measures offsets from $a + 4$. In our example, the address field will actually denote 5 rather than 6, with $a' = (a + 4) + (4 * 5) = a + 24$. Note that if the offset is actually 0, the machine will next execute the instruction found at address $a + 4$, one after the current one!

Putting all this together, suppose that the branch instruction is executed and the conditional is true. The new value of the PC will be given by

$$\texttt{PC} := (!\texttt{PC} + 4) + (4 * \text{address})$$

Note that this is a form of indexed addressing, known as **PC indexed addressing**. Note also that in an actual machine, $!\texttt{PC} + 4$ means "digitwise addition of the 32 digit $!\texttt{PC}$ to $\vec{0}100^{bin}$" and that the 16 digit address field will be sign extended to 32 digits.

Exercises 5.3.3

(i) Draw some diagrams of branch instructions in memory, and make sure you really understand the details given above.

(ii) What is the offset range given by the address? (Care!)

5.4 Compiling Real Programs

This section is non-examinable. See Hennessy and Patterson for more details.

Programs in a high level language are usually first translated into assembler. Such a translation program is called a **compiler**. Of course, the assembly code is then assembled into machine code to produce a program which can be run on the actual microarchitecture. We often refer (incorrectly) to this combined process as **compilation**. The machine code is then executed at **run time**.

The process of compilation (high level to assembly) is *very* complex, and we could devote whole modules to the topic. However, we do want to give a flavour of some of the key ideas. We do this by example.

- Every program variable will be associated with a MIPS register.
- Every program construct (such as a conditional test, while loop, etc) will be associated with a particular form of MIPS program (eg a branch or jump).
- Every program function will have its arguments stored in *argument* registers.
- Every program function will have its return variables stored in *saved* registers.
- Every program function will have its return expressions stored in *value* registers.
- The MIPS code for a function call must push those registers labelled as "preserved on call" onto the stack, and hence save their values across the call.

5.4.1 Compiling A Conditional

The JAVA code is

```
if (x==y) f = 2;
else      f = u + v;
```

We will associate the variables with saved registers as follows:

$$\mathtt{x} \mapsto \mathtt{\$s1} \quad \mathtt{y} \mapsto \mathtt{\$s2} \quad \mathtt{f} \mapsto \mathtt{\$s3} \quad \mathtt{u} \mapsto \mathtt{\$s4} \quad \mathtt{v} \mapsto \mathtt{\$s5}$$

The MIPS code is

```
       bne $s1, $s2, Else
       addi $s3, $zero, 2
       j Exit
Else : add $s3, $s4, $s5
Exit :
```

5.4.2 Compiling a While Loop and Array

The Java code is

```
while (A[i]==k) i = i+j;
```

Suppose that

$$\mathtt{i} \mapsto \mathtt{\$s3} \quad \mathtt{j} \mapsto \mathtt{\$s4} \quad \mathtt{k} \mapsto \mathtt{\$s5}$$

and that the assembler stores the base address of the array in `$s6`.

The MIPS code is

```
Loop : add $t1, $s3, $s3
       add $t1, $t1, $t1
       add $t1, $t1, $s6
       lw $t0, 0, $t1
       bne $t0, $s5, Exit
       add $s3, $s3, $s4
       j Loop
Exit :
```

5.4.3 Compiling A Simple Function

The JAVA code is

```
int simple (int x, int y)
      {
        int f;
        f = (x + y) + 8;
        return f;
      }
```

We will associate the variables with saved registers as follows:

$$\mathtt{x} \mapsto \mathtt{\$a0} \quad \mathtt{y} \mapsto \mathtt{\$a1} \quad \mathtt{f} \mapsto \mathtt{\$s0}$$

The MIPS code is

```
Simple :  subi $sp, $sp, 12
          sw $s0, 8, $sp
          sw $a0, 4, $sp
          sw $a1, 0, $sp
          add $t0, $a0, $a1
          addi $s0, $t0, 8
          add $v0, $s0, $zero
          lw $a1, 0, $sp
          lw $a0, 4, $sp
          lw $s0, 8, $sp
          addi $sp, $sp, 12
          jr $ra
Exit :
```

5.5 Further Examples

Examples 5.5.1

(i) Suppose that

$$\begin{aligned} R_2 &= 00000011.10111001.00000000.11110000 \\ R_3 &= 10000001.10101010.11111111.00001111 \\ k &= -32753^{\,dec} \end{aligned}$$

Give the contents of R_1 after the execution of

1. `add` R_1, R_2, R_3
2. `and` R_1, R_2, R_3
3. `addi` R_1, R_2, k
4. `andi` R_1, R_2, k
5. `ori` R_1, R_2, k

Answer hints (you should give more details):

1.

$$\begin{array}{r} 00000011.10111001.00000000.11110000 \\ 10000001.10101010.11111111.00001111 \\ \hline 10000101.01100011.11111111.11111111 \end{array}$$

2.

$$\begin{array}{r} 00000011.10111001.00000000.11110000 \\ 10000001.10101010.11111111.00001111 \\ \hline 00000001.10101000.00000000.00000000 \end{array}$$

3.

$$\begin{array}{r}00000011.10111001.00000000.11110000\\ 11111111.11111111.10000000.00001111\\ \hline 00000011.10111000.10000000.11111111\end{array}$$

4.

$$\begin{array}{r}00000011.10111001.00000000.11110000\\ 00000000.00000000.10000000.00001111\\ \hline 00000000.00000000.00000000.00000000\end{array}$$

5.

$$\begin{array}{r}00000011.10111001.00000000.11110000\\ 00000000.00000000.10000000.00001111\\ \hline 00000011.10111001.10000000.11111111\end{array}$$

(ii) The memory byte location with address 15 contains 10001001. The MIPS instruction `multi` R_1, R_2, k has semantics $R_1 := !R_2 * k$. The register `$s0` contains

00000000.00000001.00000000.00000000

Write a MIPS program which uses just one `lb` instruction, together with your choice of *arithmetic* category instructions *and no other category of instruction*, such that at the end of the program run `$t0` contains

00000000.00000000.10001010.00000000

Answer hints (you should give more details):

```
lb $t0, 15($zero)
multi $t0, $t0, 2^8
add $t0, $t0, $s0
addi $t1, $zero, 1
multi $t1, $t1, 2^8
add $t0, $t0, $t1
```

(iii) Suppose that `$a0` contains a positive integer $p \geq 2$. State in one simple but *precise* sentence what the contents of `$v0` is after the following MIPS program has been executed.

```
begin :   addi $t0, $zero, 0
          addi $t1, $zero, 0
repeat :  slt $t2, $a0, $t1
          bne $t2, $zero, finish
          add $t0, $t0, $t1
          addi $t1, $t1, 2
          j repeat
finish :  add $v0, $t0, $zero
```

Now explain carefully and in *detail* how you came up with your answer.

Answer hints: `$v0` will contain the sum of the first !`$a0` *DIV* 2 even numbers.

Registers `$t0` and `$t1` are set to 0. There is a loop. Register `$t1` is incremented in steps of 2, and thus records the even numbers. Register `$t0` keeps a running total, being incremented by the value of `$t1` which contains the next even number each time the loop repeats. At the end of the first loop, registers `$t0` and `$t1` contain 0 and 2. At the end of the second loop, registers `$t0` and `$t1` contain 2 and 4. Then $2+4$ and 6. At the end of the nth loop they will hold the sum of the first $n-1$ positive even numbers and $2*n$. We break out of the loop when $2*n$ is first strictly greater than p, by setting the flag `$t2` to be 1, and branching to finish. But $2*n > p \geq 2*n-2 \Longrightarrow (n-1) = p \; DIV \; 2$ Thus the first $p \; DIV \; 2$ even numbers will be summed.

6

The Micro Architecture Level

6.1 Introduction

In this chapter we shall show how to use some simple circuits to build a small CPU. There are many different ways of building a CPU. We refer to the overall design of the CPU as its **architecture**. **Micro** reminds us that we are talking about the CPU design, and not, for example, the overall computer design. We shall show how to develop a micro architecture, that is, a CPU design, for a very small subset of the MIPS ISA. This consists of

- The R-format instructions `add`, `sub`, `and`, `or`, and `slt`; and
- the I-format instructions `lw`, `sw`, and `beq`.

Recall the idea of a fetch-decode-execute (FDE) cycle, which is performed by the CPU, and in particular the datapath and the control unit. Figure 6.1 shows an abstract picture of a datapath. Note that there is more detail than in Chapter 2. Each thick black arrow denotes a bus. Note also that the busses indicated in the diagram have different widths—the diagram just gives a *rough* idea of the CPU we will soon construct. The rectangles (etc) denote electronic circuits. Data, represented by binary numbers, is carried by the busses as high and low voltages, 1 and 0. We sometimes refer to the data on a bus as a **signal**. The PC rectangle is indeed the program counter. Its contents give the address of the next instruction to be executed. Thus the bus joining the PC to `Instruction memory` is 32 bits wide. The address `!PC` is **passed** along the `Address` bus and into the `Instruction memory`. The next instruction (`I` say) is "fetched" from the `Instruction memory`. This means that its binary representation is passed along the 32 bit bus coming out of `Instruction memory`. Recall that the instruction is composed of *fields*. Each field is passed along a bus to a part of the data path. For example, the fields specifying register numbers are passed into the read-register busses `Register#` of the register file `Registers`. Note that as the read-register busses `Register#` are used to specify register numbers, they are therefore only 5 bits wide (there are 32 registers). The contents of these registers will be passed out of the two 32 bit read-data busses (the busses coming out of `Registers`) and into the ALU. The ALU performs any necessary computations, and any `Data memory` reads/writes take place. This completes the execution of the instruction.

Notice that the datapath is composed of combinational and sequential circuits. We shall call separate circuits, which perform a special task, **elements**. For example, there is a combinational ALU element. The sequential elements, such as `Data memory`, have a **state**, which is the contents of all of the memory locations at a given

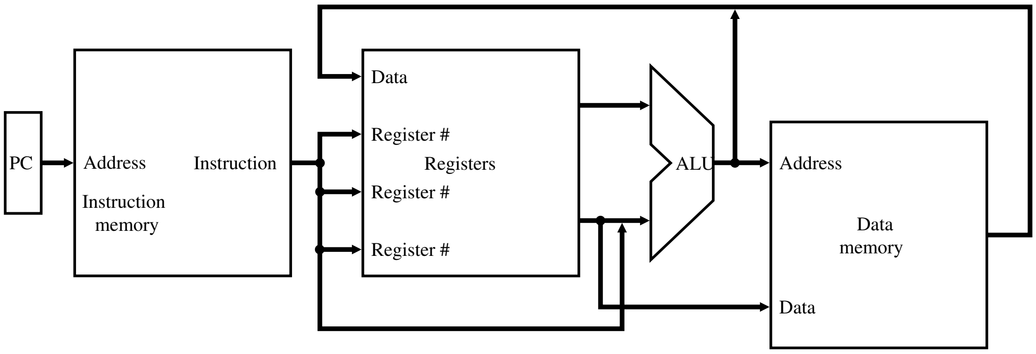


Figure 6.1: Abstract Single Cycle Datapath

time.

We will explain how to design a control unit later on. For the time being, just remember that it is used to "*perform*" the FDE cycle. In particular it will provide **settings** for the datapath elements. For example, if the instruction is `add`, control will send signals to the ALU to ensure that the addition operation is selected (see Section 4.4.6 of Chapter 4). We sometimes also say that the ALU has been **set**.

6.2 Timing and Performance

Before we build our CPU, we need to discuss time. Recall that a computer has a *clock* which is a circuit that generates regular pulses of a fixed length. These regular pulses are used to give a measure of time. Usually, the control unit will set up the datapath elements once during each cycle. Thus if a whole instruction can be executed with just one setting of the datapath, we call it a **single cycle** instruction. If the execution of an instruction requires the datapath to be set more than once, we call the instruction **multi cycle**. If a datapath can execute all instructions with a single setting for each, we call it a **single cycle datapath**. If a datapath has to execute at least some instructions with multiple settings, we call it a **multi cycle datapath**. *In this chapter we shall show how to build a CPU with both a single cycle and a multi cycle datapath.*

We refer to the number of clock cycles required to execute an instruction as the **cycles per instruction** or **CPI**, and write C_I for the CPI of instruction I. It is often the case that a particular class of instructions can be found which all have the same CPI. In particular, given a program P, the instructions which occur in P can be divided up into classes of instructions with identical CPI. We shall write $Cl_{CPI}(P)$ for the set of such classes, and we will refer to a typical class c. Given a program P, we write $|P|_c$ for the number of instructions in P of class c. We will also write C_c for the clock cycles required for any instruction in the class c, C_P for the total clock cycles required to execute the program P, and C_I for the total clock cycles required to execute the instruction I. Finally, the number of clock cycles required to execute a program P must be

$$\textit{Total Clock Cycles} = C_P = \sum_{c \in Cl_{CPI}(P)} |P|_c C_c$$

Time for an example.

Example 6.2.1 Suppose that data transfer instructions form a class called α requiring 5 cycles, arithmetic instructions a class called β requiring 4 cycles, and branch instructions a class called γ requiring 6 cycles. Given a program P

```
lw  ...
lw  ...
add  ...
sub  ...
bne  ...
sw  ...
```

then $Cl_{CPI}(P) = \{\alpha, \beta, \gamma\}$, and

Class c	C_c
α	5
β	4
γ	6

Class c	$\|P\|_c$
α	3
β	2
γ	1

Thus the number of clock cycles to execute P is

$$\textit{Total Clock Cycles} = C_P = (3 \times 5) + (2 \times 4) + (1 \times 6) = 29$$

We shall write π and f for a clock period (time for a complete clock cycle) and frequency. Of course

$$\pi = 1/f \qquad t_P = C_P \pi \qquad t_I = C_I \pi$$

follows from the definitions, where t_P is the time taken to execute P. The time taken to execute an instruction is known as **latency**. It also follows that the time taken to execute P is

$$t_P = \left(\sum_{c \in Cl_{CPI}(P)} |P|_c C_c \right) \pi$$

This allows us to formulate **Amdahl's law** which computes new execution times if we are able to speed up instructions. Suppose that a class c' of instructions can be speeded up by a factor su, so that they now require $C_{c'}/su$ cycles to execute. The new execution time will be

$$\textit{New}\, t_P = \left(\sum_{c \in Cl_{CPI}(P)\,;\, c \neq c'} |P|_c C_c \right) \pi + |P|_{c'} (C_{c'}/su) \pi$$

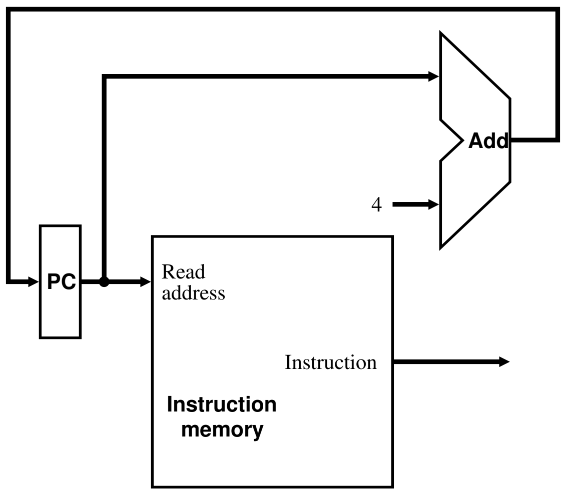


Figure 6.2: Fetching Instructions and Incrementing the PC

6.3 Datapath Components

In this section, we shall describe a number of components from which our datapaths can be built.

During the first part of the FDE-control cycle, we use the contents of the PC to provide an `Instruction memory` address. The instruction held at the word location given by the address will be sent to the rest of the datapath along the `Instruction` bus. At this stage of the FDE-control cycle, we update the PC to $!\texttt{PC}+4$ so that it points to the (next) program instruction which is one word along from the current instruction. The components in Figure 6.2 will implement this.

6.3.1 R-Format

These have the form *name* R_1, R_2, R_3. Having fetched an instruction and updated the PC, we now consider circuits which will execute R-format instructions. A suitable datapath is given in Figure 6.3. The machine instruction is passed along the 32-bit wide `Instruction` bus. It will be useful to write $\texttt{I}[31-0]$ to denote the 32-digit instruction. Recall that the machine code instruction has binary fields for the numbers of the two source registers, and the number of the destination register. These fields are passed into a register file, by using some of the lines from the 32-bit wide `Instruction` bus. For example, the numbers for the source registers R_2 and R_3 are carried on the fields $\texttt{I}[25-21]$ and $\texttt{I}[20-16]$. These fields are passed along the 5 bit `Read register 1` and `Read register 2` busses, which are made from lines 25 to 21, and lines 20 to 16, of the `Instruction` bus. The destination register number is passed

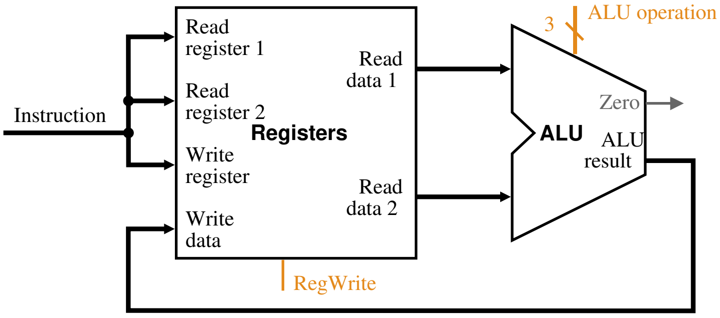


Figure 6.3: A Datapath for R-Format Instructions

along the `Write register` bus. Note that although these busses are only 5 bits wide, this is not indicated in the figure. The source operands (ie the contents of the source registers) now appear on the 32 bit `Read data 1` and `Read data 2` busses, and are thus read into an ALU which performs arithmetic or logic. Finally, the result of the ALU is written into the destination register by passing the result along the 32-bit `ALU result` bus and thus into the `Write data` bus of the register file. (Control will pass signals along the ALU control bus `ALU Operation` to select the correct ALU operation. The 3 bit `ALU Operation` consists of the 1 bit `Bnegate` line and 2 bit `Operation` bus.)

6.3.2 Load and Store

Next we look at load and store instructions which take the form

$$\texttt{lw}\ R_1,\ k(R_2) \qquad \text{and} \qquad \texttt{sw}\ R_1,\ k(R_2)$$

A suitable datapath is given in Figure 6.4. Recall the semantics of these two instructions:

$$R_1 := !W[sx(k)+!R_2] \qquad W[sx(k)+!R_2] := !R_1$$

Both compute a `Data memory` address a which is given informally by adding the offset k to the base address given by the contents of R_2. More precisely, as we saw in Chapter 5, a is the 32-bit word given by $!R_2 + sx(k)$. The datapath elements are set up to calculate this address. We examine `lw` in detail. The source register (R_2) number $\texttt{I}[25-21]$ is passed into `Read register 1`, and the 16-bit representation of k which is $\texttt{I}[15-0]$ is passed into the `Sign extend` circuit. The contents $!R_2$ of the source register are passed via `Read data 1` into the ALU, along with $sx(k)$. The control selects the ALU operation of addition. Thus $a = !R_2 + sx(k)$ is fed into `Address`. This causes the `Data memory` to pass the contents of the word location at a, that is $!W[a]$, into `Read data` and hence into the register file's `Write data` bus. The destination register (R_1)

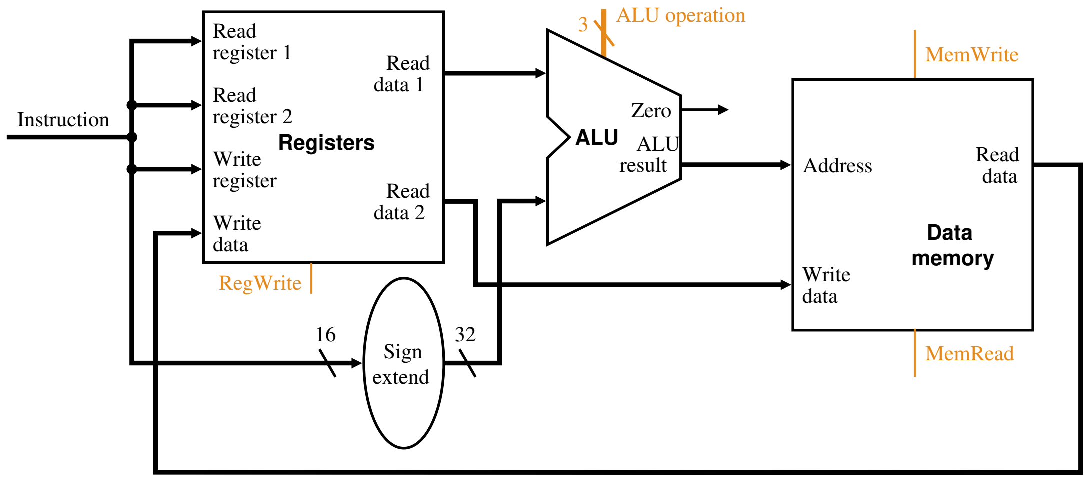


Figure 6.4: A Datapath for Load and Store Instructions

number `I`$[20-16]$ will already have been passed into `Write register`, and so $!W[a]$ will be written in R_1 as required. (The `RegWrite` control line, which is a write enable line, will be set high by control, to ensure the write can take place.)

Exercise 6.3.1 For store instructions, write down the (semantic) values which will be carried on the various Datapath busses during passage of a clock cycle, just as we have done for load instructions.

6.3.3 Branches

The "branch on equal" to instruction has the general form `beq` R_1, R_2, L. Recall the semantics

$$\text{if } !R_1 = !R_2 \text{ then goto } L$$

To execute it, first we calculate if $!R_1 = !R_2$ is true or false. Notice that this is the same as checking if $!R_1 - !R_2$ is zero or not. This can be done using the ALU. The control unit will select ALU subtraction. Recall (page 51) that the ALU has an output line `Zero` which "tests for zero". This will carry 1 if the subtraction is zero, and carry 0 if not—and hence indicate to the computer if $!R_1 = !R_2$ is true or false.

If the register contents are equal, the Datapath does not need to do any further work; the PC will need to be updated so that the next sequential instruction is executed, but that has *already been done* by another part of the Datapath (which we have already looked at).

Otherwise, we need to calculate a value for the PC, namely the address of the instruction to branch to. Before going on, read Chapter 5 subsection 5.3.4 again. We

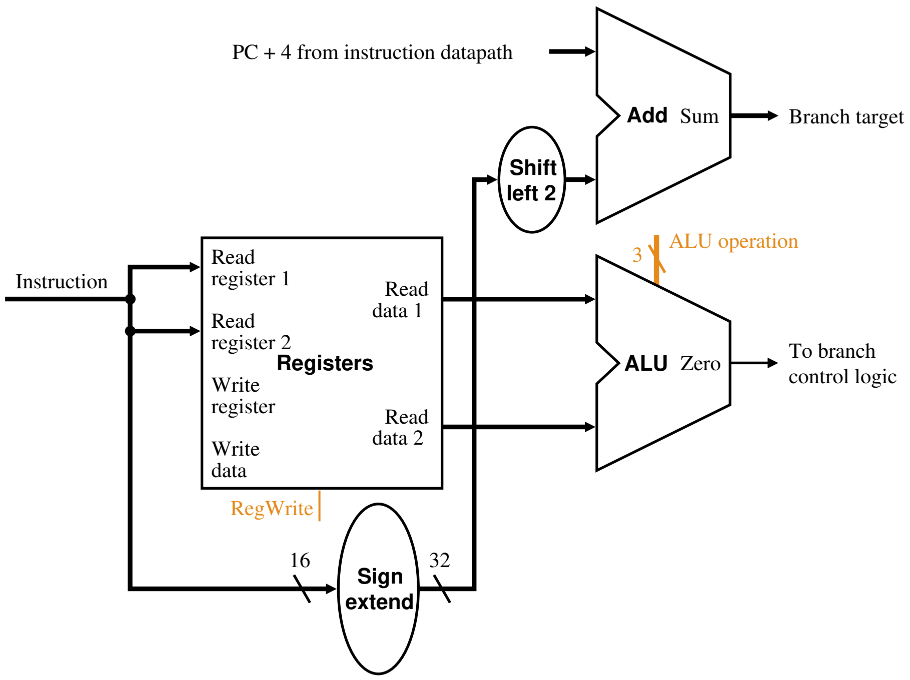


Figure 6.5: A Datapath for the Branch if Equal Instruction

see that the new address is given by

$$(!\mathtt{PC}+4)+(4*\mathtt{I}[15-0])$$

where $\mathtt{I}[15-0]$ is the machine code arising from L. In the circuits, the 16-digit $\mathtt{I}[15-0]$ must be sign extended to 32 digits. Also, we can multiply by 4 by shifting digits two places left; see page 39. Thus the address is given by the digitwise sum $(!\mathtt{PC}+4)+(sx(\mathtt{I}[15-0])\ll 2)$. A suitable datapath is given in Figure 6.5.

Exercise 6.3.2 Write down the values which will carried on the various Datapath busses during the instruction execution.

6.4 A Single-Cycle Datapath with Control

6.4.1 Building the Datapath

We now combine the separate datapaths into a single datapath that can implement all of the instructions given in Section 6.1. The required datapath is given in Figure 6.6. This datapath is essentially a direct conjunction of the datapath components, except for the addition of three multiplexors and their control lines, `PCSrc`, `ALUSrc` and

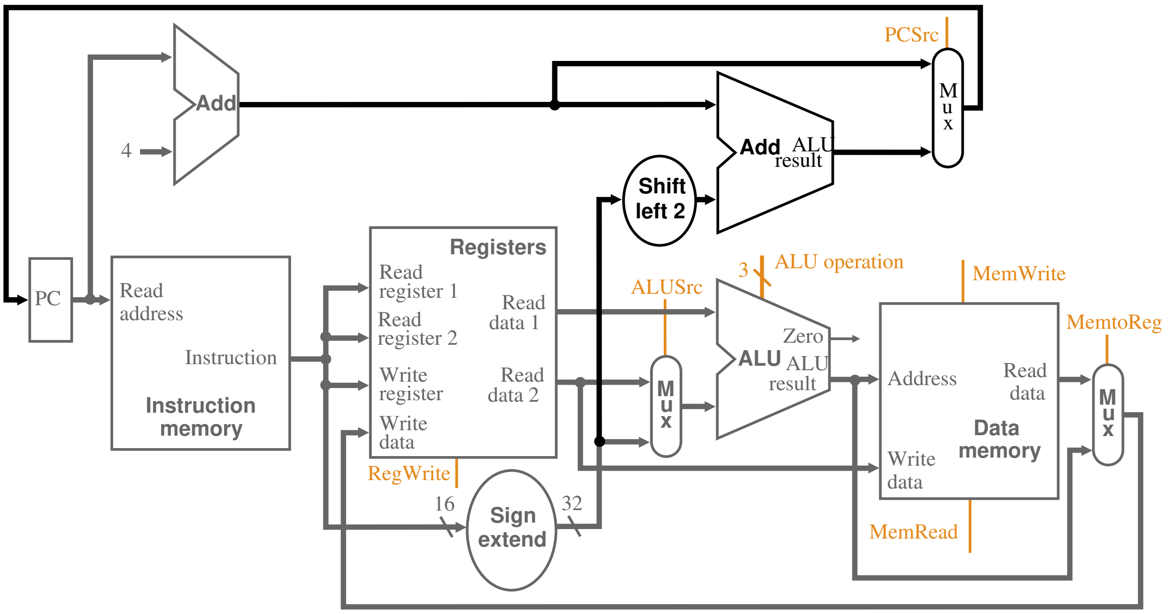


Figure 6.6: A Single Cycle Datapath for MIPS

`MemtoReg`. The new datapath provides two possible *sources* for the data to be written into the PC, second ALU source, and Register File respectively, selected by the multiplexors. The multiplexors have their control lines set by the CPU control unit. Our single cycle datapath only requires a single use of the control unit for each instruction. For any given instruction, the control unit will set the multiplexor control lines to ensure the correct data is written into the PC, second ALU source, and Register File.

- `PCSrc`: If the instruction being executed is R-format, or data transfer, the next instruction will be found at address $!\mathtt{PC}+4$. In such cases, `PCSrc` will be set to 0 to ensure that $!\mathtt{PC}+4$ is sent to the PC. If a branch instruction `beq` is executed, and the conditional is true, then `PCSrc` will be set to 1 to ensure that $(!\mathtt{PC}+4)+(sx(\mathtt{I}[15-0]) \ll 2)$ is sent to the PC.

- `ALUSrc`: the second ALU source comes from the `Read data 2` bus in the case of R-format and `beq` instructions. The second ALU source comes from the `Sign extend` element in the case of both `lw` and `sw` instructions.

- `MemtoReg`: *<u>Exercise:</u> Explain the two `Write data` sources for the Register file, and their selection using `MemtoReg`.*

6.4.2 Building Control

We now explain how the single cycle control unit is built. First, look at the diagram in Figure 6.7 which shows our single cycle CPU. You can see the Single Cycle Datapath with control units added. By definition, each instruction will execute in a single clock cycle. This means that we can set all of the control lines, once and for all, just prior to the execution of each instruction. This will ensure that when the instruction is passed into the datapath, the datapath elements are set correctly. The elements which have control lines/busses are

- the multiplexors and memory elements, and
- the ALU.

There are two separate control units, **Control** and **ALU Control**. This is because the settings for the multiplexors and memory elements are determined completely by the instruction opcode `I`$[31-26]$ and can have their own control unit. However, the setting of the ALU also requires the instruction's funct field `I`$[5-0]$, and hence the ALU has a separate control unit.

We consider how each of these elements is constructed. First note that each control unit is a simple combinational circuit. Thus, to construct each one, all we need is a truth table giving the input components and output components. The situation at the moment is

Unit	Inputs	Outputs
`Control`	`I`$[31-26]$	settings for multiplexors and memories
`ALU Control`	`I`$[31-26]$ and `I`$[5-0]$	settings for the ALU

However, we can simplify matters slightly, by reducing the number of inputs into the ALU control unit. The opcode `I`$[31-26]$ must be fed into `Control`. However, for our small selection of instructions there are only *four* different opcodes, for `lw`, `sw`, `beq` and any R-format instruction. `beq` requires subtraction. For any R-format instruction it is the funct field `I`$[5-0]$ which selects the ALU operation. Moreover, both `lw` and `sw` require ALU addition. So the opcodes only specify $4-1=3$ different situations. Thus instead of feeding `I`$[31-26]$ directly into `ALU Control`, we use `Control` to feed simpler output signals to `ALU Control`. We only require $4-1=3$ signals, and these will be passed to `ALU Control` along a 2-bit bus `ALUop`. The effects of these signals, and the signal values, are given below.

Instruction Opcode	Effect on `ALU Control`	`ALUOp1`	`ALUOp0`
`lw`	`ALU Control` sets ALU to +	0	0
`sw`	`ALU Control` sets ALU to +	0	0
`beq`	`ALU Control` sets ALU to -	0	1
R-Format	`ALU Control` sets ALU using funct	1	0

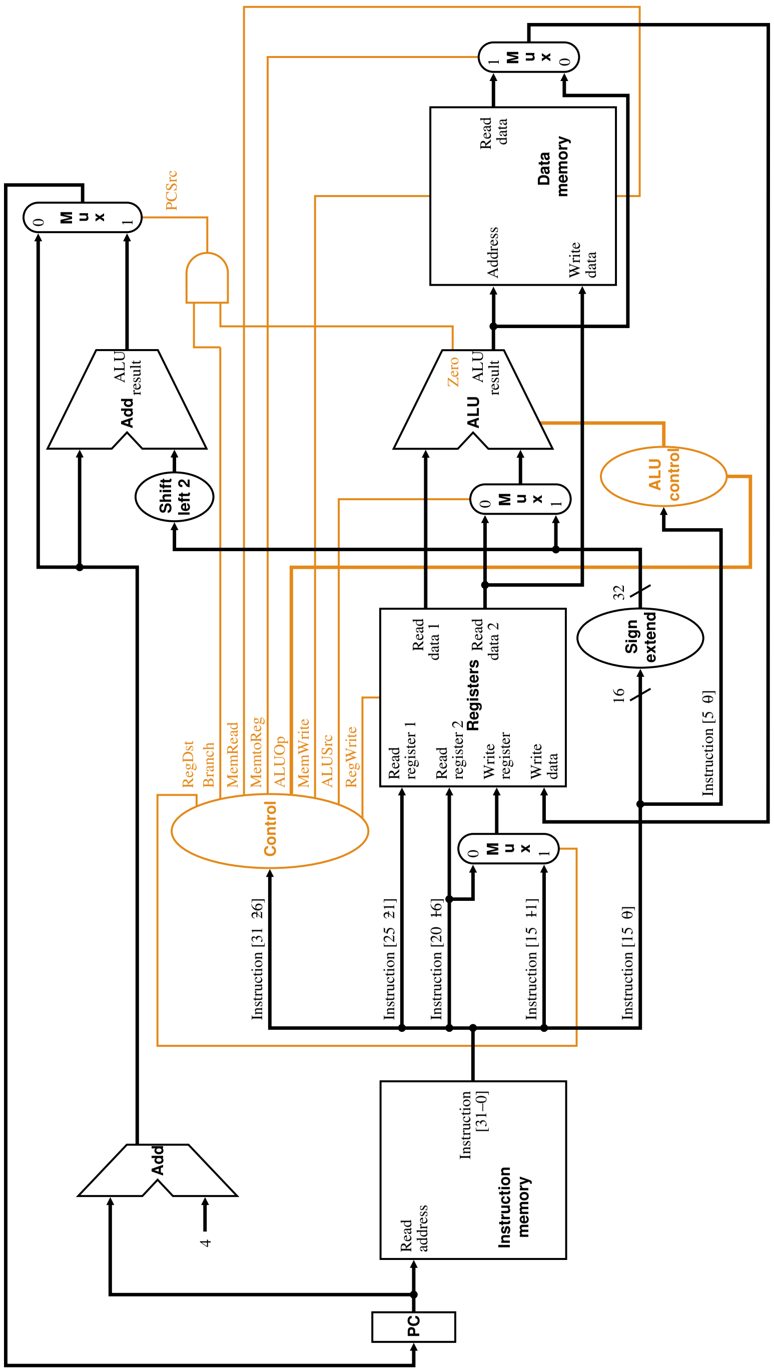


Figure 6.7: A Single Cycle Datapath with Control Units

The situation now is

Unit	Inputs	Outputs
`Control`	`I`$[31-26]$	settings for multiplexors and memories
`ALU Control`	`ALUOp0`, `ALUOp1` and `I`$[5-0]$	settings for the ALU

Let's now build `Control`, and then `ALU-Control`, by constructing their truth tables.

Control The input tuple to the Control unit is the opcode field of the current instruction. The output tuple is given by the values on `RegDst`, `Branch` and so on. From the previous section we know the values of the `ALUOp` components. All other output lines are 1-bit. In Table 6.1 we record the effects which take place when these output lines have value 0 or 1. We can now write down the Control truth table, from which we can build the Control unit via a PLA. It appears in Table 6.2. An X means that the actual value does not matter. Here are some explanations.

- When executing a `lw` instruction, the contents of a main memory word location will be copied into the register file. Thus `RegWrite` must be 1 to enable the copying to take place.
- When executing a `sw` instruction, nothing should be written into the register file. Thus `RegWrite` must be 0 to disable writing to the file. And if writing is disabled, the data held on `Write data` does not matter. Hence it does not matter what `MemtoReg` holds.

Exercise 6.4.1 Justify the other entries in the table.

ALU Control This is now extremely simple. The input components come from the `ALUOp` bus and the funct field. The output should select the correct ALU operation, by giving the values of `Bnegate` and `Operation` which are the ALU's control lines. See Section 4.4.6 of Chapter 4 and Table 6.3. In Chapter 4, an example explains why the three control lines are set to 111 for `slt`.

6.5 A Multi-Cycle Datapath with Control

6.5.1 Building the Datapath

In a single cycle datapath, a complete instruction is executed during one clock cycle, and each datapath element is set just once by control. The values of control are only altered as we pass from one clock cycle to the next. If a complete instruction must be executed during this clock cycle, then all combinational circuits used by the

Signal Name	Effect when 0	Effect when 1
`RegDst`	The register destination number for the `Write register` comes from the rt field.	The register destination number for the `Write register` comes from the rd field.
`Branch`	The contents of `PC` can only be updated with !`PC` + 4	The contents of `PC` may be updated with a branch target; this only happens if `Zero` is high.
`RegWrite`	The register file cannot be written to.	The register specified by the value of the `Write register` input is written with the value of the `Write data` input.
`ALUSrc`	The second ALU operand is the value on `Read data 2`.	The second ALU input is the sign-extended, lower 16 bits of the instruction.
`PCSrc`	The contents of `PC` is updated with !`PC` + 4	The contents of `PC` is updated with a branch target.
`MemRead`	`Data memory` cannot be read from.	`Data memory` contents specified by `Address` input value are put on the `Read data` output.
`MemWrite`	`Data memory` cannot be read to.	`Data memory` contents specified by `Address` input value are replaced by the value on the `Write data input`.
`MemtoReg`	The value fed to the register `Write data` input comes from the ALU.	The value fed to the register `Write data` input comes from the `Data memory`.

Table 6.1: Single Cycle Control Signals

Instruction	Opcode field						RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp2
lw	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0
sw	1	0	1	0	1	1	X	1	X	0	0	1	0	0	0
beq	0	0	0	1	0	0	X	0	X	0	0	0	1	0	1
R-format	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0

Table 6.2: Control Unit Truth Table

Instruction (ALU operation)	ALUOp1	ALUOp2	Funct Field						ALU-Control Output		
lw (+)	0	0	X	X	X	X	X	X	0	1	0
sw (+)	0	0	X	X	X	X	X	X	0	1	0
beq (-)	0	1	X	X	X	X	X	X	1	1	0
add (+)	1	0	1	0	0	0	0	0	0	1	0
sub (-)	1	0	1	0	0	0	1	0	1	1	0
and (*AND*)	1	0	1	0	0	1	0	0	0	0	0
or (*OR*)	1	0	1	0	0	1	0	1	0	0	1
slt	1	0	1	0	1	0	1	0	1	1	1

Table 6.3: ALU-Control Unit Truth Table

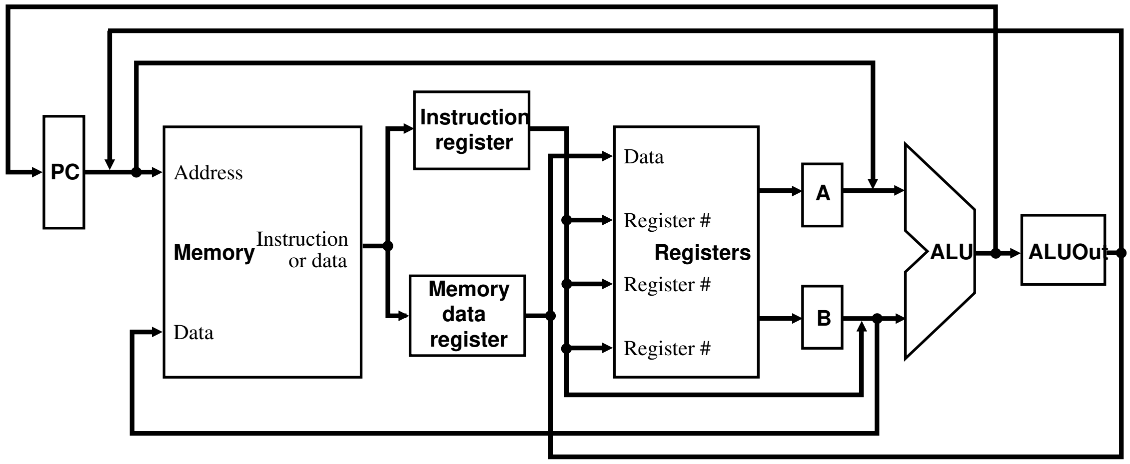


Figure 6.8: Abstract Multi Cycle Datapath

instruction must be set correctly at the start of the cycle. Thus, for example, if two arithmetic operations are to be performed during an instruction's execution, then we shall require two ALUs; we saw exactly this in our Single Cycle datapath (note the third ALU was used only to update the PC).

In a multicycle CPU, combinational elements can be used more than once per instruction execution, provided that each use happens on a different clock cycle. This ensures that when the element is used again, it has been correctly reset at the start of the new clock cycle. In Figure 6.8 we see an abstract view of a multicycle datapath. Note that just one ALU is used for all arithmetic operations, and that there is just one memory. In the multicycle datapath, a clock cycle will be sufficient for just one of the following

(i) a Memory data transfer (load or store);

(ii) a Register File access (two reads or one write);

(iii) an ALU operation.

In order to ensure that data is passed correctly between the memory, register file and ALU elements, whose control settings change on each clock cycle, we build in new registers between these elements. They can be seen on the CPU diagram. They are used to store data which will be used by the *current* instruction but on a later clock cycle. If they were not present, data might be corrupted when the control settings change. We sometimes refer to the `Instruction register` as `IR`, and `Memory data register` as `MDR`.

Now that we have looked at a basic outline for our multicycle datapath, we begin to build a multicycle datapath for the instructions on page 86. First, we look at a basic datapath for non-branching instructions (we will add in the control units later on). This is given in Figure 6.9. Note that some of the multiplexors are identical to those in the Single Cycle datapath. Some are extended versions of those in the Single Cycle datapath. Others are new.

(i) There is a new multiplexor before the `Memory Address` line. In the multicycle machine, there is just one memory which stores both instructions and data, whereas in the Single Cycle Datapath we had `Instruction` memory and `Data` memory. When a memory read takes place, we have to specify if the information read is in fact an instruction, or some other data. If an instruction is read, then its address must be supplied by the PC. Otherwise, if data is read, the address will come from the ALU. The choice is made by the multiplexor.

(ii) The single ALU now has to increment the PC by 4 and deal with sign-extended and shifted inputs. The ALU second source is connected to an extension of a previous multiplexor to make these choices.

Exercise 6.5.1 Look at the other multiplexors and make sure you understand what their roles are.

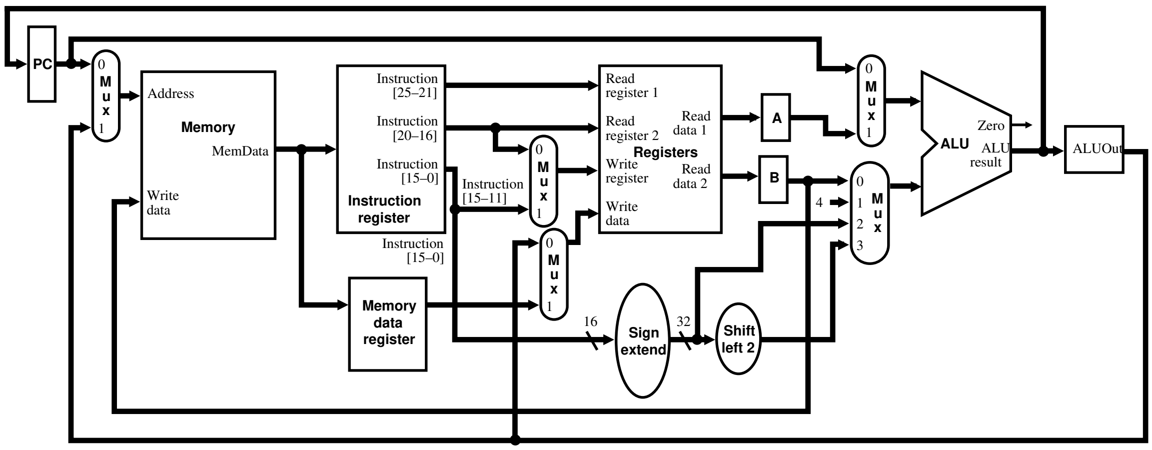
Finally, we can add in circuits for branch instructions, and also the new control unit and control lines. The final multicycle datapath, together with control lines, appears in Figure 6.10. Note that there are circuits to compute jump instructions. These will not be part of this course, and are certainly non-examinable. We will not say anything more here about the datapath. The way it works will become clearer as we study its control units. The principles are the same as for the Single Cycle Datapath.

6.5.2 Building Control

6.5.2.1 The Effects of Control Settings In order to decide on the outputs of our control unit, we need to decide on the individual effects of the various signals. The ideas are similar to the singlecycle datapath, so we will not say any more here, apart from listing the effects. The effects of setting the control signals are given in Table 6.4.

6.5.2.2 Execution Steps In the Multicycle Datapath, the execution of one instruction will be broken into a number of steps. Each step requires one clock cycle for its completion. (Previously, we had one instruction executing in one cycle. The control unit was used only once.) Each of these steps forms part of a Fetch Decode Execute (FDE) process. The first step (cycle) involves fetching an instruction. The second cycle decodes the instruction. The remaining cycles involve a sequence of execution steps which vary according to the instruction under consideration.

In order to describe the FDE process, we shall give the changes of state which occur in the CPU on each of the clock cycles. These appear in Table 6.5. Note that R denotes a register, and that $R[n]$ is register n.



97108/Patterson
Figure 05.31

Figure 6.9: A Multi Cycle Datapath for Non-Branching Instructions

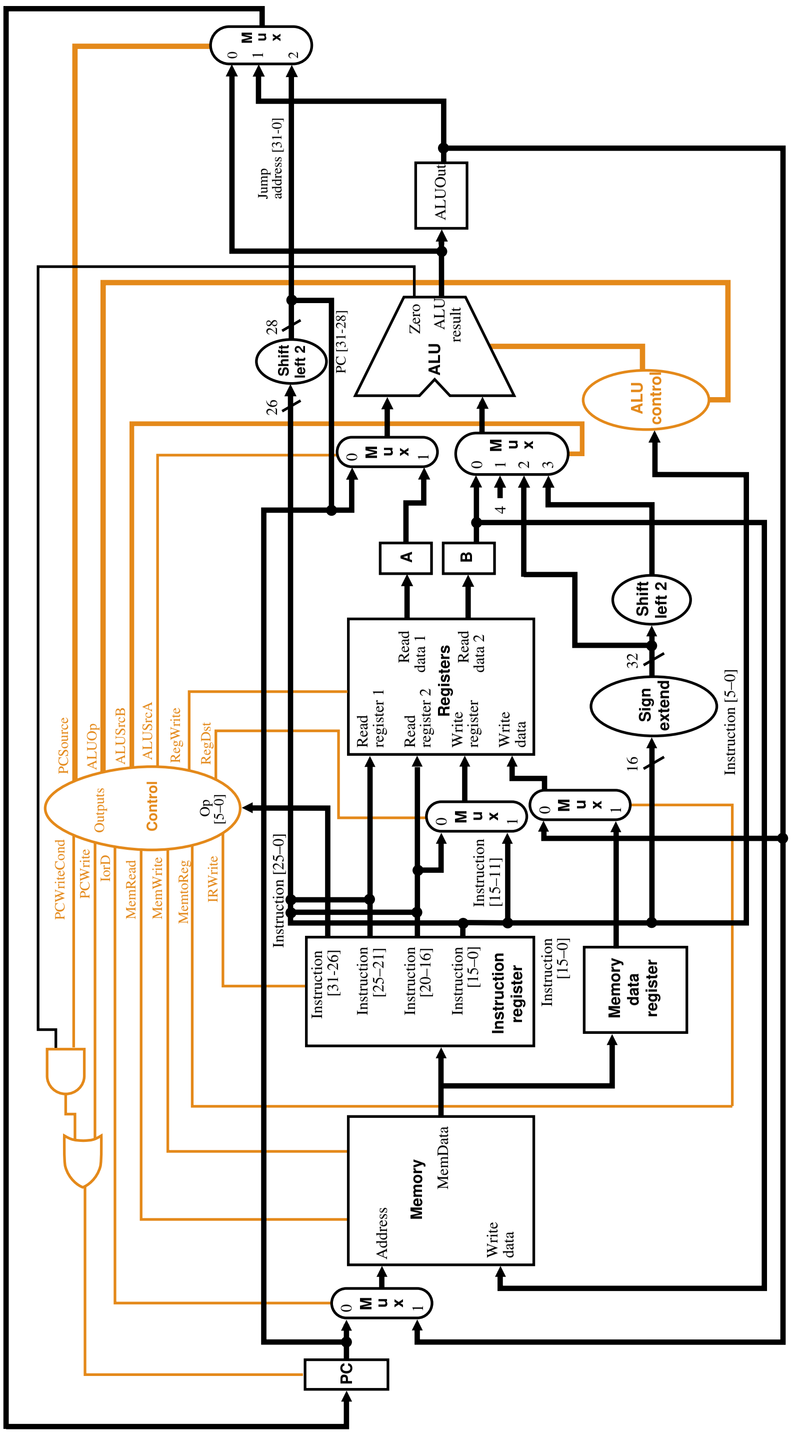


Figure 6.10: A Multicycle Datapath for MIPS

Signal Name	Effect when 0	Effect when 1
`PCWriteCond`	The writing of the PC is determined by `PCWrite`.	The PC is written if the `ALU Zero` line is 1.
`PCWrite`	The writing of the PC is determined by `PCWriteCond`.	The PC is written; the data source is controlled by `PCSource`
`IorD`	The `Memory` address is the value of the PC.	The `Memory` address is the value of `ALUOut`.
`MemRead`	`Memory` cannot be read from.	Contents of `Memory` location specified by `Address` input value are put on the `MemData` output.
`MemWrite`	`Memory` cannot be written to.	`Memory` location specified by `Address` input value is updated by the value on the `Write data` input.
`MemtoReg`	The value fed to the register `Write data` input comes from the `ALUOut` register.	The value fed to the register `Write data` input comes from the MDR.
`IRWrite`	The IR cannot be written to.	`Memory` output is written into the IR.
`ALUSrcA`	The first ALU operand is the contents of the PC.	The first ALU operand is the contents of the `A` register.
`RegWrite`	The register file cannot be written to.	The register specified by the value of the `Write register` input is updated with the value of the `Write data` input.
`RegDst`	The register destination number for the `Write register` comes from the rt field.	The register destination number for the `Write register` comes from the rd field.

Signal Name	Effect when 00	Effect when 01	Effect when 10	Effect when 11
`PCSource`	`ALUresult` sent to PC	`ALUOut` sent to PC	`Jump Address` sent to PC	N/A
`ALUOp`	ALU performs +	ALU performs -	ALU uses funct	N/A
`ALUSrcB`	*Exercise:*	*Exercise:*	*Exercise:*	*Exercise:*

Table 6.4: Effects of Multi Cycle Control Signals

Step	R-Format	Data Transfer	Branches
Fetch	$\mathtt{IR} := !W[!\mathtt{PC}]$ $\mathtt{PC} := !\mathtt{PC} + 4$		
Decode	$\mathtt{A} := !R[!\mathtt{IR}[25-21]]$ $\mathtt{B} := !R[!\mathtt{IR}[20-16]]$ $\mathtt{ALUOut} := !\mathtt{PC} + (sx(!\mathtt{IR}[15-0]) \ll 2)$		
Execution 1	$\mathtt{ALUOut} := !\mathtt{A}\ op !\mathtt{B}$	$\mathtt{ALUOut} :=$ $!\mathtt{A} + sx(!\mathtt{IR}[15-0])$	$!\mathtt{A} = !\mathtt{B} \Rightarrow \mathtt{PC} := !\mathtt{ALUOut}$
Execution 2	$R[!\mathtt{IR}[15-11]] := !\mathtt{ALUOut}$	$\mathtt{MDR} := !W[!\mathtt{ALUOut}]$ or $W[!\mathtt{ALUOut}] := !\mathtt{B}$	
Execution 3		$R[!\mathtt{IR}[20-16]] := !\mathtt{MDR}$	

Table 6.5: FDE Steps in Summary

Next we provide some more detailed explanation for each of the steps in the FDE cycle. You should read ahead, and then look back at the previous table which provides a useful summary. Recall that we have to work relative to the constraints of page 99. In the Fetch and Decode steps, we perform as many actions as possible which are applicable to all instructions. Then we perform further Execution steps, tailored to individual instructions.

(1) **Fetch** Two assignments are performed which change the state of the datapath by altering the contents of the IR and the PC. Broadly speaking, we fetch (copy) the current instruction into the IR, and update the PC to point to the next instruction. The control settings required are

Signal Name	Value
`PCWriteCond`	X
`PCWrite`	1
`IoD`	0
`MemRead`	1
`MemWrite`	0
`MemtoReg`	X
`IRWrite`	1

Signal Name	Value
`PCSource`	00
`ALUOp`	00
`ALUSrcB`	01
`ALUSrcA`	0
`RegWrite`	0
`RegDst`	X

We explain the choices for the settings

$\boxed{\texttt{IR} := !W[!\texttt{PC}]}$ We read an instruction from `Memory`, and store it in the IR. The (word) address for the instruction is given by the PC. Control does the following: The instruction is read from `Memory`, so we `MemRead` is set to 1. To store it, we set `IRWrite` = 1. The address for the instruction comes from the PC, so we set `IorD` to 0 to select the PC.

$\boxed{\texttt{PC} := !\texttt{PC} + 4}$ We also increment the PC by 4 during this step. Control does the following: We set `ALUSrcA` to 0 to send the PC contents to the ALU, `ALUSrcB` to 01 to send 4 to the ALU, and `ALUOp` to 00 to select ALU addition. We then store the sum back into the PC, so `PCSource` is set to 00, and of course `PCWrite` is set to 1.

In this cycle, `Memory` is not written to, so `MemWrite` is set to 0. `Registers` is also not written to, so `RegWrite` is set to 0. The settings of certain multiplexors do not matter during the fetch cycle, and can be set to either 0 or 1, which is denoted by X. For example, `MemtoReg` selects the data which would be written to the register file `Registers`. But as `RegWrite` is set to 0, this does not matter.

(2) **Decode** Having fetched the current instruction into the IR, and updated the PC, we have performed as many tasks as possible subject to the constraints of page 99. We once again perform as many tasks as possible which are applicable to all instructions. This cycle is called the *decode* cycle. The control settings required are

Signal Name	Value
`PCWriteCond`	0
`PCWrite`	0
`IoD`	X
`MemRead`	0
`MemWrite`	0
`MemtoReg`	X
`IRWrite`	0

Signal Name	Value
`PCSource`	X
`ALUOp`	00
`ALUSrcB`	11
`ALUSrcA`	0
`RegWrite`	0
`RegDst`	X

$\boxed{\texttt{A} := !R[!\texttt{IR}[25-21]]}$ and $\boxed{\texttt{B} := !R[!\texttt{IR}[20-16]]}$ We store the rs and rt fields of the current instruction in `A` and `B` repectively. The current instruction has been stored in the IR `IR`, and these fields are found in $!\texttt{IR}[25-21]$ and $!\texttt{IR}[20-16]$. This will happen essentially automatically after the Fetch step—no control signals need to be high. (Note that the register file does not have a read enable line: if it had, then this would need to be set to 1.)

$\boxed{\texttt{ALUOut} := !\texttt{PC} + (sx(!\texttt{IR}[15-0]) \ll 2)}$ We shall also compute a branch target address using the ALU and store the result in `ALUOut`—see page 80. This address will only be used if the current instruction is indeed `beq` and its conditional test is true. However, nothing is lost by computing the address on each decode cycle. Note that in this cycle the PC *has already been updated by 4 during the fetch cycle.* Thus the first operand used to compute the branch target address comes from the PC, so `ALUSrcA` is set to 0. The second operand comes from the output of the `Shift left 2` element; this requires `ALUSrcB` set to 11. Note that the sign extension has already been done, with the extended number fed into the `Shift left 2` element.

Exercise 6.5.2 Explain the remaining control settings in detail.

(3) **Execution 1** In this cycle, the control unit settings are different for each instruction category.

In the case of arithmetic and logical category instructions, the datapath will perform the ALU operation determined by the instruction's funct field. The result is stored in `ALUOut`. For data transfer instructions, the memory address is computed and stored in `ALUOut`. For branches, we test the operands for equality (stored in `A` and `B`). If they are equal, the branch address computed in the Decode step will be stored in the the PC. In fact this step is the final one for branch execution.

Arithmetic and Logical The control settings required are

Signal Name	Value
`PCWriteCond`	0
`PCWrite`	0
`IoD`	X
`MemRead`	0
`MemWrite`	0
`MemtoReg`	X
`IRWrite`	0

Signal Name	Value
`PCSource`	X
`ALUOp`	10
`ALUSrcB`	00
`ALUSrcA`	1
`RegWrite`	0
`RegDst`	X

$\boxed{\texttt{ALUOut} := !\texttt{A}\ op !\texttt{B}}$ The ALU operation is determined by the funct field of the instruction, and this occurs when `ALUOp` holds 10—the details are the same as those for the single cycle CPU. We have to pass the contents of `A` and `B` into the ALU. This will happen if `ALUSrcA` is set to 1, and `ALUSrcB` is set to 00. All other settings are 0.

Data Transfer The control settings required are

Signal Name	Value
`PCWriteCond`	0
`PCWrite`	0
`IoD`	X
`MemRead`	0
`MemWrite`	0
`MemtoReg`	X
`IRWrite`	0

Signal Name	Value
`PCSource`	X
`ALUOp`	00
`ALUSrcB`	10
`ALUSrcA`	1
`RegWrite`	0
`RegDst`	X

$\boxed{\texttt{ALUOut} := !\texttt{A} + sx(!\texttt{IR}[15-0])}$ The ALU must perform addition, so `ALUOp` is set to 00. The first operand comes from `A` so `ALUSrcA` is set to 1. The second operand comes from `Sign extend` so `ALUSrcB` is set to 10.

Exercise 6.5.3 Explain the remaining control settings.

Branches The control settings required are

Signal Name	Value
`PCWriteCond`	1
`PCWrite`	0
`IoD`	X
`MemRead`	0
`MemWrite`	0
`MemtoReg`	X
`IRWrite`	0

Signal Name	Value
`PCSource`	01
`ALUOp`	01
`ALUSrcB`	00
`ALUSrcA`	1
`RegWrite`	0
`RegDst`	X

Example 6.5.4 Explain the control settings in detail.

Answer: The datapath must perform $\boxed{!\texttt{A} = !\texttt{B} \Rightarrow \texttt{PC} := !\texttt{ALUOut}}$. In order to send the contents of `A` and `B` to the ALU, `ALUSrcA` must be set to 1, and `ALUSrcB` must be set to

00. Recall that the equality is found to be true or false by seeing if the difference of the numbers is zero. Thus the ALU computes `!A`−`!B` so `ALUOp` is set to 01. The PC has already been updated to point to the next instruction. Only if the subtraction yields 0 do we update the PC to a new branch address. We set `PCSource` to 01 so that the branch address is sent along the PC input bus. `PCWriteCond` is set to 1. This value is fed into an AND gate, along with `Zero`. If `Zero` holds 1 (which it will if `!A`−`!B` = 0) then the AND gate output will be 1, and the PC will be updated. If `Zero` holds 0 (which it will if `!A`−`!B` ≠ 0) then the AND gate output will be 0, and the PC will not be updated.

For a branch, the updating of the PC is conditional. Thus by setting `PCWrite` to 0, and feeding this into an OR gate, the PC control line's value is given by the `PCWriteCond` value (see above). The memory is neither read from or written to, and the register file and IR are not written to. Hence `MemRead`, `MemWrite`, `RegWrite` and `IRWrite` are all set to 0. A consequence is that the multiplexors which supply data to the memory and register file can have any setting X.

(4) **Execution 2** The control settings required are

Arithmetic and Logical The control settings required are

Signal Name	Value
`PCWriteCond`	0
`PCWrite`	0
`IoD`	X
`MemRead`	0
`MemWrite`	0
`MemtoReg`	0
`IRWrite`	0

Signal Name	Value
`PCSource`	X
`ALUOp`	X
`ALUSrcB`	X
`ALUSrcA`	X
`RegWrite`	1
`RegDst`	1

Data Transfer The control settings required for load word and store word are

Signal Name	Value
`PCWriteCond`	0
`PCWrite`	0
`IoD`	1
`MemRead`	1
`MemWrite`	0
`MemtoReg`	X
`IRWrite`	0
`PCSource`	X
`ALUOp`	X
`ALUSrcB`	X
`ALUSrcA`	X
`RegWrite`	0
`RegDst`	X

Signal Name	Value
`PCWriteCond`	0
`PCWrite`	0
`IoD`	1
`MemRead`	0
`MemWrite`	1
`MemtoReg`	X
`IRWrite`	0
`PCSource`	X
`ALUOp`	X
`ALUSrcB`	X
`ALUSrcA`	X
`RegWrite`	0
`RegDst`	X

Exercise 6.5.5 Explain the control settings in detail.

(5) **Execution 3** Only a load word instruction requires a third execution step.

Data Transfer The control settings required are

Signal Name	Value
PCWriteCond	
PCWrite	
IoD	
MemRead	
MemWrite	
MemtoReg	
IRWrite	

Signal Name	Value
PCSource	
ALUOp	
ALUSrcB	
ALUSrcA	
RegWrite	
RegDst	

Exercise 6.5.6 Fill in the control settings for Execution 3.

6.6 Microprogrammed Control

This section is non examinable.

We have now studied the steps of the FDE process which take place when certain MIPS instructions execute. For each step, we know the required output of the control units. Now we have to explain how to design a control unit so that for any given instruction, the correct step signals are produced on each clock cycle.

Let us recall Dr. Doitall from Chapter 2. A computer program was represented by the instruction leaflet that came with the bookcase. Dr. Doitall represented control. In this case, Dr. Doitall knew how to proceed with the bookcase building. He has to read an instruction, work out the tools needed, and then carry out the required task. He then goes on to the next instruction. This gives a model of the FDE process. We also saw in Section 2.3.4 that the FDE process can be considered as a simple algorithm which describes the various tasks which take place. The key idea here is that we implement algorithms as *software*. We actually write a program which will perform the FDE process. In order that we do not confuse this program with the "users" program in memory, we call it a **microprogram**. This will consist of a sequence of instructions which tell Control what values should appear on the output lines. To avoid confusion, we call these instructions **microinstructions**. Thus the sequence of MIPS execution steps given on page 105 corresponds (essentially) to the execution of the microinstructions, with each microinstruction setting the Control output lines for each FDE step. Note that in each Execution step, the control settings are different for each instruction category. These different settings will be implemented by branching within the microprogram.

A microprogram has a layout similar to a MIPS program, namely a sequence of labels and instructions. Also, each microinstruction consists of seven fields. Six of

Field Name	Field Function
ALU Control	Select ALU operation; output always to `ALUOut`
SRC1	Specify source of first ALU operand
SCR2	Specify source of second ALU operand
Register Control	Specify read or write, and write source
Memory	Specify read or write, with write source or read destination
PCWrite	Specify writing of the PC
Sequencing	Specify how to choose the next microinstruction

Table 6.6: Microinstruction Field Semantics

these are used to set the values of Control output, and the seventh provides a label indicating the next microinstruction to be executed. The field semantics are given in Table 6.6. The specific values of the fields, and the corresponding semantics appear in Table 6.7. The actual microprogram appears in Table 6.8.

Exercise 6.6.1 Try "running" the microprogram on paper, with a sample MIPS instruction, checking that the instruction is executed as you would expect.

An abstract picture of the implementation of the microprogram is given in Figure 6.11. The lectures will explain the details. Here, the microprogram will be implemented in hardware.

Field	Field Values	Semantic Comments
Label	Any string	Use for microcode sequencing
ALU-Control	Add	Clear
	Subt	Clear
	Func code	Use funct field to determine ALU operation
SRC1	PC	Clear
	A	Clear
SRC2	B	Clear
	4	Clear
	Extend	Use output of sign extend element
	Extshft	Use output of shift element
Register Control	Read	Use rs and rt fields of IR as Read register numbers. Send the Read data 1 and 2 ouputs to `A` and `B`
	Write ALU	Write the Register file. Write register is the rd field of `IR`; and Write data is contents of `ALUOut`
	Write MDR	Write the Register file. Write register is the rt field of `IR`; and Write data is contents of `MDR`
Memory	Read PC	Read memory with PC as address; write contents into IR
	Read ALU	Read memory with `ALUOut` as address; write contents into MDR
	Write ALU	Write memory with `ALUOut` as address; contents of B as data
PCWrite Control	ALU	Send ALU output to PC
	ALUOut-cond	If ALU Zero is asserted, write ALUOut into PC
	Jump address	Send jump address to PC
Sequencing	Seq	Move to the following instruction
	Fetch	Goto Fetch label
	Dispatch i	Use ROM dispatch tables to calculate next instruction

Table 6.7: Microprogram Field Values and Semantics

Table 6.8: The Microprogram

Label	ALU control	SRC1	SRC2	Reg control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
RFormat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

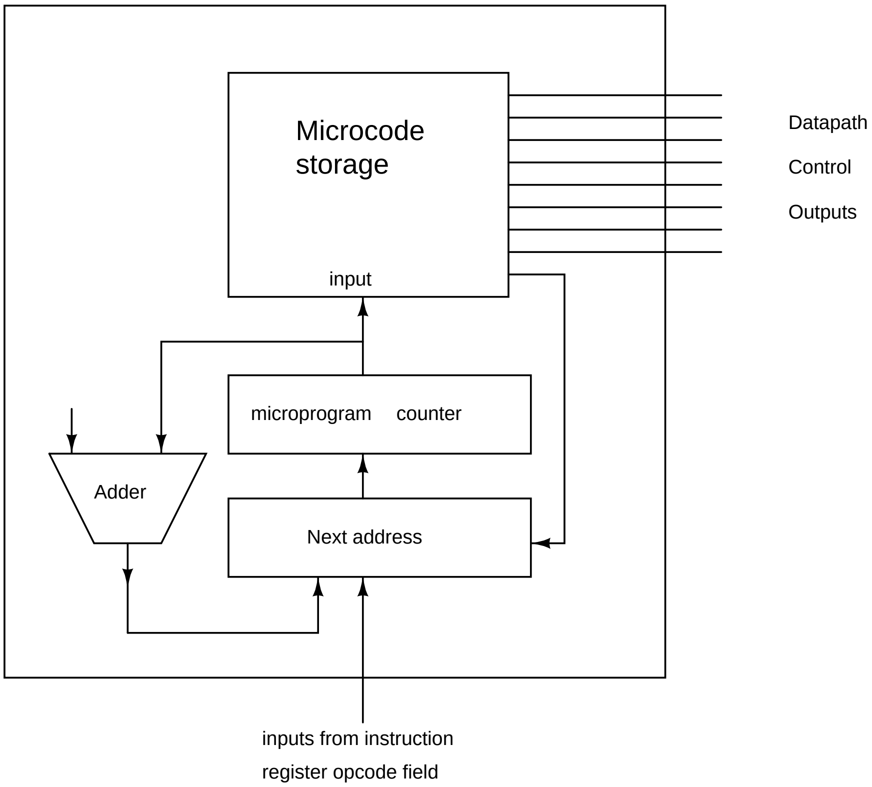


Figure 6.11: Abstract Microprogram Implementation

6.7 Further Examples

Examples 6.7.1

(i) Question 5 from January 2000 exam (see booklet of Exam Papers).

Answer hints:

1. The first three are trivial from the table. In iv, the Zero control line is used in `bne` which will fail.
2. (a) Input bus is 001000. Also

Instruction	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp2
addi	0	1	0	1	0	0	0	-	-

   (b) 00. Yes; the function field of lw is ignored by the ALU Control unit when ALUOp has 00.

3. (a)

	IM	RF	ALU	DM	RF	Time
add	y	y	y		y	4
sw	y	y	y	y		5
lw	y	y	y	y	y	6

   (b) $t = 20 * 1 * 4 + 10 * 1 * 5 + 10 * 1 * 6 = 190$.

   (c) $t = 20 * 1 * 3 + 10 * 2 * 3 + 10 * 2 * 3 = 180$.

   (d) $t = (20 * (1/2) * 3)/2 + 10 * 2 * 3 + 10 * (2/10) * 3 = 81$.

Subject Index