

## Computer Architecture



Preliminaries

Digital Arithmetic and Logic

An Overview of Computer Architecture

MIPS/ARM Hardware

MIPS/ARM Software

MIPS/ARM Processor

Introduction to Advanced Topics

# Preliminaries

---

- ▶ We take a *rapid* glimpse at the Learning Outcomes of CO1104.
- ▶ We need some Preliminary Knowledge which we introduce over the next few slides . . .

## A Glimpse at the Module CO1104 Learning Outcomes

- ▶ Solve problems in computer arithmetic and logic (since processors perform arithmetic and logic).
  - ▶ Explain how simple computer processors (cores) work:  
processor = hardware + software.
  - ▶ Describe real digital circuit hardware and real software (MIPS/ARM) that is executed on the hardware; solve problems.
  - ▶ Explain how complex processors (cores) work; solve problems.
  - ▶ Explain advanced topics such as correctness, pipelining, multicore processors and cache memory.
- 
- ▶ Let's look at the module outline ...
  - ▶ ...and then straight on to the Preliminaries.

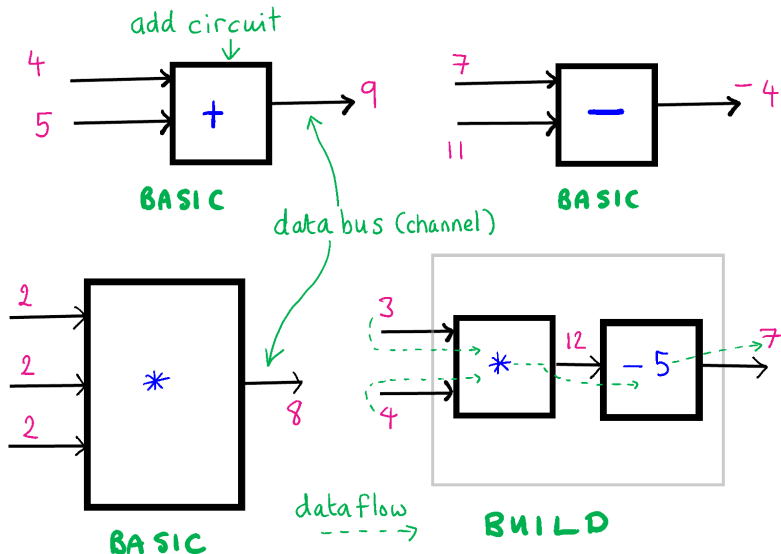
## Overview: Preliminary Knowledge for CO1104

**Hardware** processors execute computer programs. Programs consist of instructions (eg `add y, x, 3`) written in a special **language**. Instructions contain **variables**, and **data** (eg `3`) of certain **types** (eg `int`). Data is stored as **sequences** of **0**s and **1**s. Data moves around on **busses**. There is a **set** of instructions for the processor.

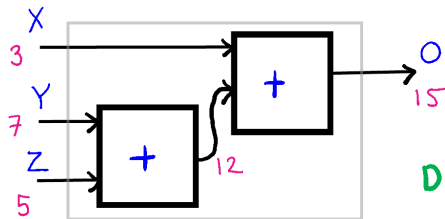
After studying this section you should be able to

- ▶ Draw pictures of basic **hardware** and **data busses**.
- ▶ Explain briefly what a **language** is.
- ▶ Give examples of **sets** and **types**.
- ▶ Give a definition of a **program variable** and give simple examples.
- ▶ Define a **sequence**, give examples, and make use of **sequence notation**

## Example: Elementary Hardware and Data Busses



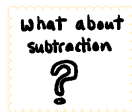
## Example: Elementary Hardware and Data Busses



DIFFERENT CIRCUITS  
but

SAME I/O BEHAVIOUR

$$O = X + Y + Z$$



## Example: A Datapath for add $y, x, 3$

To execute (run) this instruction we need some computer memory (a register file) with a couple of addresses, an adding unit, and some data busses. These items form a datapath.

Suppose  $x = 5$ . After execution, the memory state has  $y = 8$  and  $x = 5$ .

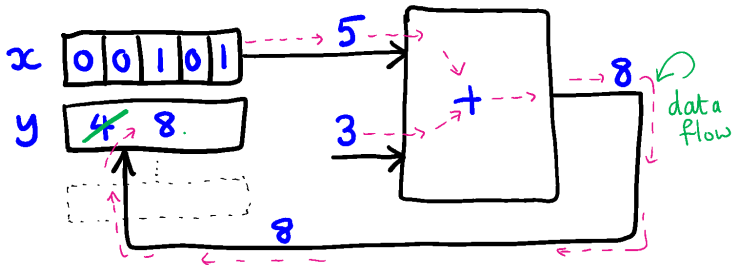
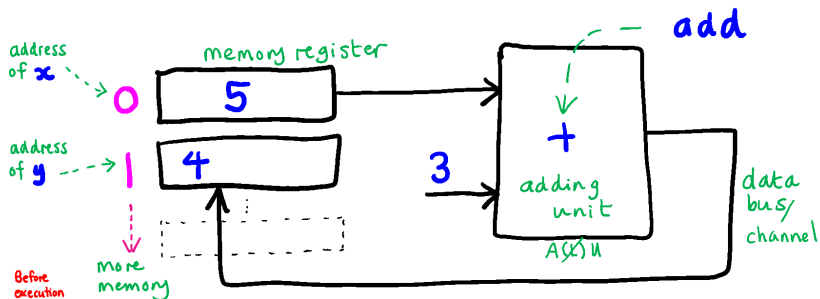
The data 3, 5, 8 is stored in memory using binary numbers. The variables  $x, y$  are assigned addresses in memory. LEC

- ▶ Let's look at the pictures on Slide 9.
- ▶ After that, we look in more detail at certain topics ...

**This module is mainly about understanding datapaths in great depth!**



## Example: A Datapath for add $y, x, 3$



- ▶ **Syntax** is “symbolic notation” (3 or `///` or *iii*).
- ▶ **Semantics** is the “meaning” of the syntax (three).
- ▶ These are different syntactic **representations** of three.
- ▶ A **language** consists of syntax and semantics.
- ▶ The syntax consists of
  - ▶ an **alphabet** (/);
  - ▶ **rules** for constructing “words” from alphabet (`/`, `///`, `////`, `/////`, ...)
- ▶ In a **programming** language the “words” are *program instructions*. The alphabet and rules appear in a programming language definition (`print(3)`).
- ▶ The semantics of a program is (a description of) what happens when the program instructions execute (`3` appears on the display).

- ▶ It is useful to work with collections of **similar kinds of data**.
- ▶ Mathematicians use **sets**; Informaticians use **types**.
- ▶ A **set** is a collection of items, eg **names**:  $\{ tom, harry, anna \}$ .
- ▶  $\mathbb{N} \stackrel{\text{def}}{=} \{ 0, 1, 2, \dots \}$  set of **natural numbers**.
- ▶  $\mathbb{Z} \stackrel{\text{def}}{=} \{ \dots, -2, -1, 0, 1, 2, \dots \}$  set of **integers**.
- ▶ Each member of a set is called an **element**.
- ▶  $tom \in \{ tom, harry, anna \}$  and  $reiko \notin \{ tom, harry, anna \}$ .
- ▶ Note that

$$\{ tom, harry, anna \} = \{ harry, anna, tom \} = \dots$$

The order of the elements does not matter.  
Elements cannot be repeated.

A computer processor implements a set of instructions  $\{ \textit{add}, \textit{sub}, \textit{mult}, \textit{div} \}$ . What is the size of the set?

- ▶ A **type** in a programming language plays the role of a set in mathematics. E.g. `int` is a type, and `3` is an element of this type. `456` is another element. We say “`3` is of type `int`”.
- ▶ Roughly speaking a type in a programming language can be understood as a set (e.g. the set of integers) but with restrictions (integers up to a certain size only).
- ▶ When programming, you might write a **type declaration**

`n :: int`      or      `int n`.

`n` is called a **variable**: a letter (or word, eg `age`) with an unknown value. The type declaration means that `n` is an integer.

- ▶ This is a *broad* subject! Variables can be *bound*, *constant*, *final*, *free*, *logical*, *mathematical*, *meta*, *program*, *schematic*, *static*, . . . . It takes a lot of experience to fully understand the different kinds.

A **program variable** has a **memory address**; and its **value** is stored at the address.

Suppose **savings** is a variable. As a “program involving **savings** executes” *the value of the variable may change* (eg the program computes my *interest* and so **savings** increases).

- ▶ In a programming language, items of code are usually written as a sequence of characters (eg `temp`, `age`, `calcInterest`) called **identifiers**.
- ▶ Identifiers are used to name methods, functions, class names etc etc. ...and of course variables.
- ▶ When code *executes*, each identifier is *assigned* an **address**: the address (a number) specifies a location in computer memory. LEC
- ▶ Later on the identifier is *assigned* an **initial value**: this is the content of the memory location. LEC
- ▶ `int varone ; int vartwo ; varone = 4 ; vartwo = varone`

- ▶ In a programming language, items of code are usually written as a sequence of characters (eg `temp`, `age`, `calcInterest`) called **identifiers**.
- ▶ Identifiers are used to name methods, functions, class names etc etc. ... and of course variables.
- ▶ When code *executes*, each identifier is *assigned* an **address**: the address (a number) specifies a location in computer memory. LEC
- ▶ Later on the identifier is *assigned* an **initial value**: this is the content of the memory location. LEC
- ▶ `int varone ; int vartwo ; varone = 4 ; vartwo = varone`

`varone`

*software* ↔ *hardware*

**AddressOf** `varone`



- ▶ In a programming language, items of code are usually written as a sequence of characters (eg `temp`, `age`, `calcInterest`) called **identifiers**.
- ▶ Identifiers are used to name methods, functions, class names etc etc. ... and of course variables.
- ▶ When code *executes*, each identifier is *assigned* an **address**: the address (a number) specifies a location in computer memory. LEC
- ▶ Later on the identifier is *assigned* an **initial value**: this is the content of the memory location. LEC
- ▶ `int varone ; int vartwo ; varone = 4 ; vartwo = varone`





- ▶ In a programming language, items of code are usually written as a sequence of characters (eg `temp`, `age`, `calcInterest`) called **identifiers**.
- ▶ Identifiers are used to name methods, functions, class names etc etc. ... and of course variables.
- ▶ When code *executes*, each identifier is *assigned* an **address**: the address (a number) specifies a location in computer memory. LEC
- ▶ Later on the identifier is *assigned* an **initial value**: this is the content of the memory location. LEC
- ▶ `int varone ; int vartwo ; varone = 4 ; vartwo = varone`



- ▶ In a programming language, items of code are usually written as a sequence of characters (eg `temp`, `age`, `calcInterest`) called **identifiers**.
- ▶ Identifiers are used to name methods, functions, class names etc etc. ... and of course variables.
- ▶ When code *executes*, each identifier is *assigned* an **address**: the address (a number) specifies a location in computer memory. LEC
- ▶ Later on the identifier is *assigned* an **initial value**: this is the content of the memory location. LEC
- ▶ `int varone ; int vartwo ; varone = 4 ; vartwo = varone`



We will use the sets of **denary**, **binary** and **hexary digits**:

- ▶  $Den \stackrel{\text{def}}{=} \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- ▶  $Bin \stackrel{\text{def}}{=} \{0, 1\}$
- ▶  $Hex \stackrel{\text{def}}{=} \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$

Writing “let  $d$  be a denary digit” is the same as writing “let  $d \in Den$ ”.

- ▶ **0101101** is a **sequence** of binary digits.
- ▶ What's the point? Computers *store numbers* as sequences of binary digits!
- ▶ **14AB2F33DD** a sequence of hexary digits.
- ▶ **1678** is both a sequence of denary digits, and of hexary digits.
  - ▶ We write **1678<sub>d</sub>** and **1678<sub>h</sub>** to distinguish.

- ▶ Note  $1678_d$  is different from  $6817_d$ .
- ▶ The digits in the sequence are **ordered**: 1st digit, 2nd digit, etc:

1st	2nd	3rd	4th
1	6	7	8

1st	2nd	3rd	4th
6	8	1	7

- ▶ If  $s = 6817$  sometimes we will *need to compute the digit in a given position*
  - ▶  $s_{1st} = 6$ ,  $s_{2nd} = 8$ ,  $s_{3rd} = 1$ ,  $s_{4th} = 7$ ; or ...
- ▶ ... we could use a different notation
  - ▶  $s[1st] = 6$ ,  $s[2nd] = 8$ ,  $s[3rd] = 1$ ,  $s[4th] = 7$ .
- ▶ The second example is written in **array notation**. Mostly we use **sequence notation** as in the first example.

## Sequences

- We could choose a different order:

<i>4th</i>	<i>3rd</i>	<i>2nd</i>	<i>1st</i>
6	8	1	7

- We could use a single number to specify the order:

4	3	2	1
6	8	1	7

What about the notation below? What digit is in position **0**? And position **3**?

3	2	1	0
6	8	1	7

## Sequences

- We could choose a different order:

<i>4th</i>	<i>3rd</i>	<i>2nd</i>	<i>1st</i>
6	8	1	7

- We could use a single number to specify the order:

4	3	2	1
6	8	1	7

7 and 6 are in position 0 and 3. The **length** of the sequence is 4.

3	2	1	0
6	8	1	7

## Some Useful Facts about Numbers and Notation

- The powers of **2** (*memorise!*):

<b>1</b>	<b>2</b>	<b>4</b>	<b>8</b>	<b>16</b>	<b>32</b>	<b>64</b>	<b>128</b>	<b>256</b>	<b>512</b>	<b>1024</b>
$2^0$	$2^1$	$2^2$	$2^3$	$2^4$	$2^5$	$2^6$	$2^7$	$2^8$	$2^9$	$2^{10}$

- $2^4 * 2^3 = (2 * 2 * 2 * 2) * (2 * 2 * 2) = 2^7 = 2^{4+3}$ .
- $2^n * 2^m = 2^{n+m}$  for  $n, m \in \mathbb{N}$ . (*understand!*)
- $(2^3)^4 = (2^3) * (2^3) * (2^3) * (2^3) = 2^{12} = 2^{3*4}$ .
- $(2^n)^m = 2^{n*m}$  for  $n, m \in \mathbb{N}$ .
- $3 * 4 = 4 + 4 + 4$ : **3 quantities** of **4**.
- $1 * n = n$ : **1 quantity** of  **$n$** . See Slide 30.
- $0 * n = 0$ : **0 quantities** of  **$n$** .

- ▶  $\leq$  means **less than or equal to**:
  - ▶  $4 \leq 6$  and  $12 \leq 12$ , but not  $6 \leq 4$ .
- ▶  $<$  means **less than but not equal to**
  - ▶  $4 < 6$  but not  $12 < 12$  and not  $6 < 4$ .
- ▶  $\geq$  means **greater than or equal to**.
- ▶  $>$  means **greater than but not equal to**.
- ▶ Remark:  $12 \leq 12$  is *true* and  $6 < 4$  is *false*.
- ▶ The integers are an **ordered** set: Given any  $x, y \in \mathbb{Z}$ , either  $x = y$ ,  $x < y$  or  $y < x$ . We can draw **number line** pictures. LEC
- ▶ Given the set of integers  $\{-5, 77, -3, 25, -345\}$  the **largest integer** is  $77$  and the **smallest** is  $-345$ .



Here is a simple program with a **for loop**:

```
for  $i = 4$  to  $9$  {print  $i * 1.01$ ;} end
```

What happens when it runs?

- ▶ We say  $i$  **ranges** from  $4$  up to  $9$ .
- ▶ Note that  $4$  is the *smallest* value of  $i$ , and  $9$  the *largest*.
- ▶ For any value of  $i$  we have  $4 \leq i \leq 9$ . We also say  $i$  **lies between**  $4$  and  $9$  (inclusively).

## Some Useful Facts about Numbers and Notation

- ▶ Saying that  $z \in \mathbb{Z}$  has a sign of  $+$  is the same as saying  $z > 0$  or that  $z$  is positive.
- ▶ Saying that  $z \in \mathbb{Z}$  has a sign of  $-$  is the same as saying  $z < 0$  or  $z$  is negative.
- ▶ If  $z = 0$  then  $z$  has no sign.

- ▶ Notice that

$$\dots -4 < -3 < -2 < -1 < 0 < 1 < 2 < 3 < 4 \dots$$

- ▶ We can replace  $<$  with  $\leq$ . I'll often say *smaller than* in place of *less than*.

Recall Slide 14. See also `equals.py`. In coding,

- ▶  $x = 4$  usually means “change the current value of  $x$  to 4”.
- ▶ Thus “=” **updates** the value of  $x$ . Sometimes  $=$  is written as  $:=$ .
- ▶ If now  $y = x$  usually this means “change the current value of  $y$  to the current value of  $x$ ”.
- ▶  $3 == 4$  is a statement that 3 and 4 are equal: it's *false*.
- ▶  $4 == 4$  is a statement that 4 and 4 are equal: it's *true*.
- ▶  $x + 8 == 3 * 4$  is true. Note  $z = 6$  implies  $z == 6$  is true.
- ▶ An **expression** is usually a small piece of code, often made of variables, and operations like addition.
- ▶ Updates:  $\langle \text{variable} \rangle = \langle \text{expression} \rangle$ .
- ▶ Equalities:  $\langle \text{expression} \rangle == \langle \text{expression} \rangle$ .

In Mathematics, and informal discussions,

- ▶ If we say “Let  $x = 4$ ” usually this means “We assert the value of  $x$  is equal to 4”.
- ▶ It's a bit like a computer update  $x := 4$  and a guarantee that  $x == 4$  is true.
- ▶ So now let  $x = 4, y = 8, z = 8$ . Then “since  $x = 4$  and  $y = 8$  we have  $x + y = 12$ ” is a true statement.
- ▶ Sometimes I will write  $x \stackrel{\text{def}}{=} 4$  to indicate I am *defining* the value of  $x$  for the first time. Note  $x \stackrel{\text{def}}{=} 4$  implies  $x = 4$  is true.
- ▶ We also say “Set  $x = 4$ ”.

A **byte** is a sequence of eight binary digits. The following prefixes have the given values:

Kibi (Ki)	$2^{10}$	Kilo (K)	$10^3$
Mebi (Mi)	$2^{20}$	Mega (M)	$10^6$
Gibi (Gi)	$2^{30}$	Giga (G)	$10^9$
Tebi (Ti)	$2^{40}$	Tera (T)	$10^{12}$

- ▶ A Kibibyte of data is a sequence of  $2^{10} * 8$  binary digits.
- ▶ A Kilobyte of data is a sequence of  $10^3 * 8$  binary digits.

How many digits, expressed in millions or billions, are there in a gigabyte of data?

A **byte** is a sequence of eight binary digits. The following prefixes have the given values:

Kibi (Ki)	$2^{10}$	Kilo (K)	$10^3$
Mebi (Mi)	$2^{20}$	Mega (M)	$10^6$
Gibi (Gi)	$2^{30}$	Giga (G)	$10^9$
Tebi (Ti)	$2^{40}$	Tera (T)	$10^{12}$

- ▶ A Kibibyte of data is a sequence of  $2^{10} * 8$  binary digits.
- ▶ A Kilobyte of data is a sequence of  $10^3 * 8$  binary digits.

How many digits, expressed in millions or billions, are there in a gigabyte of data? 8,000,000,000 ie 8 billion.

# Digital Arithmetic and Logic

---

- ▶ We study how natural numbers and integers are stored in computers.
- ▶ We move on to look at methods for computing addition and subtraction.
- ▶ There is an introduction to hardware for computing arithmetic and logic: the Arithmetic Logic Unit.
- ▶ We also introduce propositional logic.

## Overview: The Natural Numbers

Here is some **syntax** for the **natural numbers**

$$\mathbb{N} = \{\text{zero}, \text{one}, \text{two}, \text{three}, \text{four}, \dots, \text{twelve}, \dots, \}$$

*Counting* provides the **semantics**.

The number *twelve* can be written as **1100** (**binary**), **12** (**denary**) and **C** (**hexary**). Computers use all three **representations**. There are **4** binary numbers with **2** digits: **00**, **01**, **10**, **11** where

$$00_b = 0_d, 01_b = 1_d, 10_b = 2_d, 11_b = 3_d.$$

Of these, the smallest number ***n*** in denary is **0** and the largest **3**:

$$0 \leq n \leq 3$$



## Overview: The Natural Numbers

---

After studying this section you should be able to

- ▶ Give examples of **binary**, **denary** and **hexary** numbers.
- ▶ Use **sequence notation** (see Slide 16) for binary numbers.
- ▶ Convert **binary** to **denary**, **denary** to **binary**, and **hexary** to **denary**.
- ▶ State how many binary numbers with  $k$  digits there are, ...
- ▶ ... and state (using a simple formula) the largest and smallest denary numbers corresponding to the  $k$ -digit binary numbers.

- ▶ A **binary/denary/hexary number** is a *sequence* of binary/denary/hexary digits.
- ▶ A  **$k$ -digit** number is one with a digit sequence of length  $k$ .
- ▶ We can write down the natural numbers using
  - ▶ denary numbers; or
  - ▶ binary numbers; or
  - ▶ hexary numbers.
- ▶ E.g. *thirteen*: **13**, **1101**, and  **$D$** .
- ▶ We have **three** representations of *thirteen*.

What is the value of  $k$  for **1101**?

## Representing the Natural Numbers

- ▶ A **binary/denary/hexary number** is a *sequence* of binary/denary/hexary digits.
- ▶ A  **$k$ -digit** number is one with a digit sequence of length  $k$ .
- ▶ We can write down the natural numbers using
  - ▶ denary numbers; or
  - ▶ binary numbers; or
  - ▶ hexary numbers.
- ▶ E.g. *thirteen*: **13**, **1101**, and  **$D$** .
- ▶ We have **three** representations of *thirteen*.

What is the value of  $k$  for **1101**?

**$k = 4$ .**

# Representing the Natural Numbers

N	Denary	Binary	Hexary
<i>zero</i>	0	000	0
<i>one</i>	1	001	1
<i>two</i>	2	010	2
<i>three</i>	3	011	3
<i>four</i>	4	100	4
<i>five</i>	5	101	5
<i>six</i>	6	110	6
<i>seven</i>	7	111	7
<i>eight</i>	8	1000	8
<i>nine</i>	9	1001	9
<i>ten</i>	10	1010	A
<i>eleven</i>	11	1011	B
<i>twelve</i>	12	1100	C
<i>thirteen</i>	13	1101	D
<i>fourteen</i>	14	1110	E
<i>fifteen</i>	15	1111	F
<i>sixteen</i>	16	10000	10
<i>seventeen</i>	17	10001	11
<i>eighteen</i>	18	10010	12
<i>nineteen</i>	19	10011	13
<i>twenty</i>	20	10100	14

N	Denary	Binary	Hexary
<i>thirtyfour</i>	34	100010	22
<i>ninetynine</i>	99	1100011	63
?	666	?	?
?	32	100000	20
?	64	1000000	40
?	128	10000000	80
?	256	100000000	100
?	512	1000000000	200
?	1024	10000000000	400
?	2048	100000000000	800
?	4096	1000000000000	1000
?	8192	$\overline{10}$	2000
?	16384	$\overline{10}$	4000

Note the sequence notation  $\vec{0}$ .

If a  $k$ -digit binary number represents  $n \in \mathbb{N}$ , then  $0 \leq n \leq 2^k - 1$ .

- ▶ We write  $\text{Bin}^k$  for the  $k$ -digit binary numbers. For example

$$\text{Bin}^3 = \{ 000, 001, 010, 011, 100, 101, 110, 111 \} \quad (\text{size } 2^3 = 8)$$

- ▶ We write  $\vec{d}$  to declare a variable binary number.
- ▶ The digits **0** and **1** are also referred to as **bits**.
- ▶ We will say “ $k$ -digit” and “ $k$ -bit” interchangeably.

- Given the 5-digit binary number  $\vec{d} \stackrel{\text{def}}{=} 10110$ , each digit is in a certain **position**  $i$  (see Slide 16):

$i$	4	3	2	1	0
$d_i$	1	0	1	1	0

The variable  $i$  ranges over the digit positions, from 4 on the left down to 0 on the right.

- Given  $\vec{d}$  we define  $d_i$  to be the **digit of  $\vec{d}$  in position  $i$** . If  $\vec{d} \stackrel{\text{def}}{=} 10110 \in \text{Bin}^5$  then  $d_0 = 0$  and  $d_2 = 1$  for example.
- The digits of a  $k$ -digit number are in positions  $k - 1$  down to 0.
- Sometimes it is useful to write  $0 \leq i \leq k - 1$ .

## Converting Binary into Denary

- The human method:

1	0	1	1	0
16	8	4	2	1

and  $10110_b = (16 + 4 + 2)_d = 22_d$ .

- The computer method: Given  $10110_b$ ,  $k = 5$ .  $i$  ranges from  $k - 1$  down to  $0$ .

$i$	4	3	2	1	0
$d_i$	1	0	1	1	0
$2^i$	16	8	4	2	1
$d_i * 2^i$	16	0	4	2	0

See Slide 18.

and sum the final row  $10110_b = (16 + 0 + 4 + 2 + 0)_d = 22_d$ .

For  $i = k - 1 = 4$  down to  $0$ : Compute  $d_i * 2^i$  and add these values up.

## Converting Binary and Hexary into Denary

Suppose that  $\vec{d} \in \text{Bin}^k$ . The digits  $d_i$  are in positions  $i = k - 1$  down to  $i = 0$ .

The **computer method** (formula/program):

$$\vec{d}_b \stackrel{\text{def}}{=} (d_{k-1} * 2^{k-1} + \dots + d_i * 2^i + \dots + d_0 * 2^0)_d$$

$$\vec{d}_h \stackrel{\text{def}}{=} (d_{k-1} * 16^{k-1} + \dots + d_i * 16^i + \dots + d_0 * 16^0)_d$$

We call  $d_{k-1}$  the **most significant** digit, and  $d_0$  the **least significant**.

Can you apply this to  $\vec{d} = 10110$ ? What is  $k$ ? See the visualiser ...

\* Also ... can you write a tiny computer program with a for-loop for  $i$ ,  $v$  for  $d_i * 2^i$  and  $s$  for the sum? Update  $s$  for each value of  $i$ .



## Examples: Converting Binary to Denary and Conversely

- ▶ On slide 31 we gave a method for converting binary to denary.
- ▶ For converting denary to binary there is also a method. We will only give an example of the method in use.

Can you convert  $101_b$  and  $B1A_h$  to denary? Have a go ...

Can you convert  $179_d$  to binary? I'll do this!

Exercise: What is the (expected) **formula** for converting denary to denary?! What does the conversion “do”? \*More difficult: what about binary to binary? And denary to binary?

## Examples: Converting Binary to Denary and Conversely

$i$	2	1	0
$d_i$	1	0	1
$2^i$	4	2	1
$d_i * 2^i$	4	0	1

$i$	2	1	0
$d_i$	$B$	1	$A$
$16^i$	$16^2$	$16^1$	$16^0$
$d_i * 16^i$	$11 * 256$	$1 * 16$	$10 * 1$

and so

$$101_b = (d_2 * 2^2 + d_1 * 2^1 + d_0 * 2^0)_d = (4 + 0 + 1)_d = 5_d$$

$$B1A_h = (d_2 * 16^2 + d_1 * 16^1 + d_0 * 16^0)_d$$

Powers of 2 are 1, 2, 4, 8, 16, 32, 64, 128, 256, and  $128 \leq 179 < 256$ .

$$\begin{array}{cccccccc} 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{array}$$

$$179 - 128 = 51, 51 - 32 = 19, 19 - 16 = 3, 3 - 2 = 1.$$

## Examples: Converting Binary to Denary and Conversely

$i$	2	1	0
$d_i$	1	0	1
$2^i$	4	2	1
$d_i * 2^i$	4	0	1

$i$	2	1	0
$d_i$	<b>B</b>	<b>1</b>	<b>A</b>
$16^i$	$16^2$	$16^1$	$16^0$
$d_i * 16^i$	$11 * 256$	$1 * 16$	$10 * 1$

and so

$$101_b = (d_2 * 2^2 + d_1 * 2^1 + d_0 * 2^0)_d = (4 + 0 + 1)_d = 5_d$$

$$B1A_h = (\mathbf{B} * 16^2 + \mathbf{1} * 16^1 + \mathbf{A} * 16^0)_d$$

Powers of 2 are 1, 2, 4, 8, 16, 32, 64, 128, 256, and  $128 \leq 179 < 256$ .

$$\begin{array}{cccccccc} 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{array}$$

$$179 - 128 = 51, 51 - 32 = 19, 19 - 16 = 3, 3 - 2 = 1.$$

## Examples: Converting Binary to Denary and Conversely

$i$	2	1	0
$d_i$	1	0	1
$2^i$	4	2	1
$d_i * 2^i$	4	0	1

$i$	2	1	0
$d_i$	$B$	1	$A$
$16^i$	$16^2$	$16^1$	$16^0$
$d_i * 16^i$	<b>11</b> * 256	<b>1</b> * 16	<b>10</b> * 1

and so

$$101_b = (d_2 * 2^2 + d_1 * 2^1 + d_0 * 2^0)_d = (4 + 0 + 1)_d = 5_d$$

$$B1A_h = (\mathbf{11} * 256 + \mathbf{1} * 16 + \mathbf{10} * 1)_d$$

Powers of 2 are 1, 2, 4, 8, 16, 32, 64, 128, 256, and  $128 \leq 179 < 256$ .

$$\begin{array}{cccccccc} 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{array}$$

$$179 - 128 = 51, 51 - 32 = 19, 19 - 16 = 3, 3 - 2 = 1.$$

## Examples: Converting Binary to Denary and Conversely

$i$	2	1	0
$d_i$	1	0	1
$2^i$	4	2	1
$d_i * 2^i$	4	0	1

$i$	2	1	0
$d_i$	$B$	1	$A$
$16^i$	$16^2$	$16^1$	$16^0$
$d_i * 16^i$	$11 * 256$	$1 * 16$	$10 * 1$

and so

$$101_b = (d_2 * 2^2 + d_1 * 2^1 + d_0 * 2^0)_d = (4 + 0 + 1)_d = 5_d$$

$$B1A_h = (2816 + 16 + 10)_d = 2842_d$$

Powers of 2 are 1, 2, 4, 8, 16, 32, 64, 128, 256, and  $128 \leq 179 < 256$ .

$$\begin{array}{cccccccc} 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{array}$$

$$179 - 128 = 51, 51 - 32 = 19, 19 - 16 = 3, 3 - 2 = 1.$$

## Examples: Converting Binary to Denary and Conversely

Powers of 2 are 1, 2, 4, 8, 16, 32, 64, 128, 256, and  $128 \leq 179 < 256$ .

128	64	32	16	8	4	2	1
1	0	1	1	0	0	1	1

	128	64	32	16	8	4	2	1
Answer:	1	0	1	1	0	0	1	1

$179 - 128 = 51$ ,  $51 - 32 = 19$ ,  $19 - 16 = 3$ ,  $3 - 2 = 1$ .

## \* Counting the $k$ -digit Binary Numbers \*

---

- ▶ How many  $k$ -digit binary numbers are there?
- ▶ 1-digit: 0, 1 — so 2 of them.
- ▶ 2-digit: 00, 01, 10, 11 — so 4 of them. A count of 4.
- ▶ 3-digit: 000, 001, 010, 011, 100, 101, 110, 111 — so 8 of them.

## \* Counting the $k$ -digit Binary Numbers \*

- ▶ How many  $k$ -digit binary numbers are there?
- ▶ 1-digit: 0, 1 — so 2 of them.
- ▶ 2-digit: 00, 01, 10, 11 — so 4 of them. A count of 4.
- ▶ 3-digit: 000, 001, 010, 011, 100, 101, 110, 111 — so 8 of them.

Suppose we already have all the binary numbers with  $k$  digits, and there are  $c$  of them. How many  $k + 1$  digit numbers are there?



## \* Counting the $k$ -digit Binary Numbers \*

- ▶ How many  $k$ -digit binary numbers are there?
- ▶ 1-digit: 0, 1 — so 2 of them.
- ▶ 2-digit: 00, 01, 10, 11 — so 4 of them. A count of 4.
- ▶ 3-digit: 000, 001, 010, 011, 100, 101, 110, 111 — so 8 of them.

Suppose we already have all the binary numbers with  $k$  digits, and there are  $c$  of them. How many  $k + 1$  digit numbers are there?

Suppose that  $\vec{b}$  ranges over all  $k$ -digit binary numbers—a count of  $c$ . Then the binary numbers  $0\vec{b}$  and  $1\vec{b}$  must range over all the  $(k + 1)$ -digit binary numbers (why?). If so, the count of  $(k + 1)$ -digit binary numbers must be  $2 * c$ . Informally: each extra digit doubles the count  $c \mapsto 2 * c$ .

This reasoning can be made precise with *mathematical induction*.

- ▶ 1-digit: 0, 1 — so 2 of them.
- ▶ 2-digit: 00, 01, 10, 11 — so 4 of them.
- ▶ 3-digit: 000, 001, 010, 011, 100, 101, 110, 111 — so 8 of them.

Slide 34 shows that each time we gain a digit, the count doubles:

- ▶ 1-digit:  $2 = 2^1$  of them.
- ▶ 2-digit:  $2 * 2^1 = 2^2$  of them.
- ▶ 3-digit:  $2 * 2^2 = 2^3$  of them. . . . .

### SUMMARY:

There are  $2^k$  binary numbers of length  $k$ , that is,  $k$ -digit numbers.

## Overview: The Integers

Here is some **syntax** for the **integers**

$$\mathbb{N} = \{\dots, \text{minus two}, \text{minus one}, \text{zero}, \text{one}, \text{two}, \text{three}, \text{four}, \dots, \}$$

The number *minus twelve* can be written as **10100** (**signed binary**) and **-12** (**denary**); two different representations. There are **8** signed binary numbers with **3** digits: **000**, **001**, **010**, **011**, **100**, **101**, **110**, **111**.

Of these, the smallest number **z** in denary is **-4** and the largest **3**:

$$\underline{-4} \leq z \leq \underline{3}.$$

## Overview: The Integers

After studying this section you should be able to

- ▶ Give examples of **signed binary** and **denary** numbers.
- ▶ Use **sequence notation** (see Slide 16) for signed binary numbers.
- ▶ Convert **signed binary** to **denary** and **denary** to **signed binary**.
- ▶ State how many signed binary numbers with  $k$ -digits there are, ...
- ▶ ... and give the largest and smallest corresponding denary numbers.

**IMPORTANT:** The **syntax** of signed and unsigned binary numbers is the same; the **semantics** in  $\mathbb{Z}$  is **different**.

- ▶ See Slide 38.
- ▶ We can represent the integers using **signed** denary.
- ▶ The denary representations are now of the form  $\pm d_{k-1} \dots d_0$  where each  $d_i \in \text{Den}$ . We sometimes omit the  $+$ .
- ▶ The binary representations **do not have** signs! They are just sequences of binary digits. But we refer to **signed** binary.
- ▶ For example  $100100_s = -28_d$ .
- ▶ If we refer to an integer  $z \in \mathbb{Z}$  we will normally write  $z$  down using denary notation.

How many 4-digit binary numbers are there?

## Representing the Integers

$\mathbb{Z}$	Denary	Binary
<i>minustwentyfive</i>	-25	100111
<i>minuseight</i>	-8	1000
<i>minusseven</i>	-7	1001
<i>minussix</i>	-6	1010
<i>minusfive</i>	-5	1011
<i>minusfour</i>	-4	1100
<i>minusthree</i>	-3	1101
<i>minustwo</i>	-2	1110
<i>minusone</i>	-1	1111
<i>minuszero</i>	0	0000

$\mathbb{Z}$	Denary	Binary
<i>one</i>	1	0001
<i>two</i>	2	0010
<i>three</i>	3	0011
<i>four</i>	4	0100
<i>five</i>	5	0101
<i>six</i>	6	0110
<i>seven</i>	7	0111
<i>twentyfive</i>	25	011001

If  $k$  digits represent  $z \in \mathbb{Z}$  then  $-2^{k-1} \leq z \leq 2^{k-1} - 1$  (+).

Does (+) match with the table?

## Converting Signed Binary into Denary

The denary conversion for **signed** binary numbers:

$$\vec{d}_s = (d_{k-1} * (-2^{k-1}) + d_{k-2} * 2^{k-2} + \dots + d_0 * 2^0)_d$$

Convert  $100111_s$  and  $011001_s$  into denary.

$i$	5	4	3	2	1	0
$d_i$	1	0	0	1	1	1
$2^i$	-32	16	8	4	2	1
$d_i * 2^i$	-32	0	0	4	2	1

$i$	5	4	3	2	1	0
$d_i$	0	1	1	0	0	1
$2^i$	-32	16	8	4	2	1
$d_i * 2^i$	0	16	8	0	0	1

$$100111_s = -32 + 4 + 2 + 1_d = -25_d$$

$$011001_s = 16 + 8 + 1_d = 25_d$$

## Converting Denary into Signed Binary

What 7-digit signed binary number represents  $-38_d$ ?

Note that  $k - 1 = 7 - 1 = 6$ ; the highest power of 2 is  $2^6 = 64$ . In signed binary a 1 digit in position 6 has value  $-2^6 = -64$ .

$-64$	32	16	8	4	2	1
1	0	1	1	0	1	0

Note that  $-38 = -64 + x$  so  $x = 26$ . Now convert 26 into unsigned binary:  $26 - 16 = 10$ ,  $10 - 8 = 2$ .



## Converting Denary into Signed Binary

You may find it useful to note that

$$\vec{d}_s = d_{k-1} * (-2^{k-1}) + \underbrace{(d_{k-2}d_{k-1} \dots d_1d_0)_b}_{\geq 0}$$

What 8-digit signed binary number represents  $-38_d$ ?

The answer is **11011010**. Why? Can you *deduce this* from the equation above?

## Overview: Addition, The Sign Digit and Subtraction

How does a computer work out  $6 + 7$  or  $3 - 9$ ?

- ▶ The denary numbers are converted to signed binary numbers:
  - ▶  $00110 + 00111$  and  $00011 - 01001$
- ▶ The resulting binary numbers are added or subtracted:
  - ▶  $01101$  and  $11010$
- ▶ The binary answer is converted to denary.
  - ▶  $13$  and  $-6$

## Overview: Addition, The Sign Digit and Subtraction

---

After studying this section you should be able to

- ▶ Add binary numbers.
- ▶ Explain what an ALU is, and draw pictures that include example inputs and outputs. Give examples of correct and incorrect ALU results, and solve simple problems about correctness.
- ▶ Be able to swap signs,  $+9$  to  $-9$ , but in binary.
- ▶ Subtract binary numbers (by swapping signs and adding:  $3 - 9 = 3 + (-9)$ ).
- ▶ Be able to explain what zero extension and sign extension are, and compute given examples.

## Addition in Denary

- ▶ Suppose  $a_2a_1a_0_d = 927_d$  and  $b_2b_1b_0_d = 436_d$ . The sum is  $s_3s_2s_1s_0_d = 1363_d$ ; notice the **digit in position 3** is non-zero.
- ▶ At school you learned **digitwise (digit-by-digit)** addition: you add up the digits in position  $i$  and write down a sum digit in position  $i$  and place a carry into position  $i + 1$ . Eg  $7 + 6 + 0 = 13$ ; carry **1** goes in position  $i + 1 = 0 + 1 = 1$ . Note that  $c_0 \stackrel{\text{def}}{=} 0$ .

$i$	$i = 3$	$i = 2$	$i = 1$	$i = 0$
$\vec{a}$		9	2	7
$\vec{b}$		4	3	6
$\vec{c}$	1	0	1	0
$\vec{s}$	1	3	6	3

The recipe for computing  $\vec{c}$  and  $\vec{s}$  from  $\vec{a}$  and  $\vec{b}$  is called the **digitwise algorithm (DA)** or **digit-by-digit algorithm**.

## (Human) Addition in Binary

- Suppose that  $a_2a_1a_0_b = 101_b$  and  $b_2b_1b_0_b = 111_b$ . Then the sum is  $s_3s_2s_1s_0_b = 1100_b$ .
- Note that  $d + 0 = 0 + d = d$  for any digit  $d \in \text{Bin}$ , and

$$1_b = 1_d = 01_b \quad 1_b + 1_b = 2_d = 10_b \quad 1_b + 1_b + 1_b = 11_b$$

- The digitwise algorithm (DA) also works in binary:

$i$	$i = 3$	$i = 2$	$i = 1$	$i = 0$
$\vec{a}$		1	0	1
$\vec{b}$		1	1	1
$\vec{c}$	1	1	1	$c_0 \stackrel{\text{def}}{=} 0$
$\vec{s}$	1	1	0	0

Hmm. What does “also works in binary” *really* mean?

## ALU – Arithmetic Logic Unit

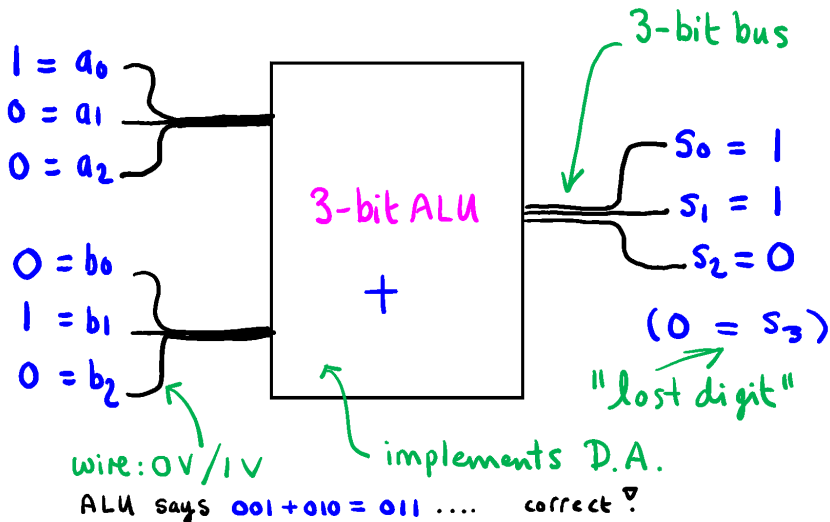
- ▶ A processor contains a  $k$ -bit ALU; it can add up two  $k$ -digit binary numbers. The two numbers are input into the ALU, and a  $k$ -digit result is output!!. An ALU can also perform subtraction and other **operations**.
- ▶ Inputs are also referred to as **operands**; and output the **result**.
- ▶ Note: ALUs provide a *hardware implementation* of the DA.

In the 3-digit example on slide 45 the binary operands are  $001_b = 1_d$  and  $010_b = 2_d$ . The DA outputs  $0011$  and the ALU outputs  $011$ ; since  $011_b = 3_d$  the ALU is *correct*.

In the 3-digit example on slide 46 the binary operands are  $101$  and  $111$ . The DA outputs  $1100$  but the ALU outputs  $100$ ; since  $100_b = 4_d \neq 12_d$  the ALU is *incorrect*.

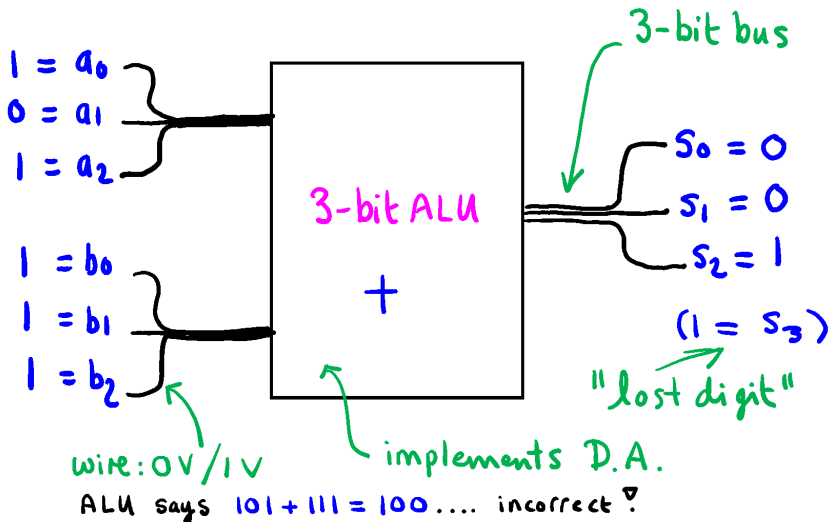
## 3-Bit ALU with Correct Unsigned Addition

Example with  $k = 3$



## 3-Bit ALU with Incorrect Unsigned Addition

Example with  $k = 3$





## Addition Examples

---

- ▶ Compute  $1110 + 0110$  giving all carry and sum digits.
- ▶ Compute  $1010 + 0101$  giving only the sum digits.
- ▶ Compute  $010001 + 110010$  giving both the 7-bit carry and sum, and then stating the 6-bit ALU result clearly.

## Addition Examples

- ▶ Compute  $1110 + 0110$  giving all carry and sum digits.
- ▶ Compute  $1010 + 0101$  giving only the sum digits.
- ▶ Compute  $010001 + 110010$  giving both the 7-bit carry and sum, and then stating the 6-bit ALU result clearly.

- ▶  $\vec{c} = 11100$  and  $\vec{s} = 10100$ .
- ▶  $\vec{s} = 1111$ .
- ▶  $\vec{c} = 1100000$  and  $\vec{s} = 1000011$ .

The DA result is  $1000011$ . The ALU result is  $000011$ .

In general:

Take binary  $k$ -digit operands  $a_{k-1} \dots a_1 a_0$  and  $b_{k-1} \dots b_1 b_0$

- ▶ If the DA result is  $s_k s_{k-1} \dots s_1 s_0$  ( $k + 1$ -digits), then
- ▶ the ALU result is  $s_{k-1} \dots s_1 s_0$  ( $k$ -digits).
- ▶ We say that the most significant digit  $s_k$  has been *lost*.

In the next section we always work with signed binary.

- ▶ Looking back at slide 38, what do you notice about the  $+/-$  signs of (denary) integers, compared to the signed binary numbers?
- ▶ Which of **101**, **01**, **0111**, **11100** equal positive denary numbers? Which are negative?

- ▶ Looking back at slide 38, what do you notice about the  $+/-$  signs of (denary) integers, compared to the signed binary numbers?
- ▶ Which of **101**, **01**, **0111**, **11100** equal positive denary numbers? Which are negative?

$$\begin{array}{lcl} \mathbf{101}_s & = & -3_d < 0 \\ \mathbf{01}_s & = & +1_d \geq 0 \end{array}$$

$$\begin{array}{lcl} \mathbf{11100}_s & = & -4_d < 0 \\ \mathbf{0111}_s & = & +7_d \geq 0 \end{array}$$

## The Sign Digit

$$\begin{array}{lcl} \textcolor{red}{1}0\textcolor{blue}{1}_s & = & -3_d < 0 \\ 0\textcolor{blue}{1}_s & = & +1_d \geq 0 \end{array}$$

$$\begin{array}{lcl} \textcolor{red}{1}1100_s & = & -4_d < 0 \\ 0\textcolor{blue}{1}11_s & = & +7_d \geq 0 \end{array}$$

Suppose that  $\vec{d}$  is any  $k$ -digit signed binary number. Then

If  $d_{k-1}$ , the most significant digit, is  $0$ , then  $\vec{d}_s \geq 0$  ie  $\vec{d}_s$  is either  $0$  or positive.

If  $d_{k-1}$ , the most significant digit, is  $1$ , then  $\vec{d}_s < 0$ , ie  $\vec{d}_s$  is negative.

## Negating Integers and Binary Numbers

- What about subtraction? In  $\mathbb{Z}$ , we know (eg) that  $23 - 20 = 23 - (+20) = 23 + (-20) = 3$ . So we can subtract by adding negations ...

If  $+\vec{d}_d$  is any non-zero denary integer, there is also an integer, the **negation**,  $-\vec{d}_d$  (and vice versa). We say that the *signs are swapped*.

- To **negate** a signed binary number we **flip** 0s and 1s, and add 1:

$$25_d = 011001_s \mapsto 100110_s \mapsto 100111_s = -25_d$$

The **complement** of **0** is **1** and of **1** is **0**. We write  $\bar{0} = 1$  and  $\bar{1} = 0$ . The **digitwise complement**  $\vec{d}$  means “flip 0s to 1s and 1s to 0s”. For example  $\vec{101010} = 010101$ . So, provided  $\vec{d} \neq 10$

**negation** of  $\vec{d}$  equals the ALU addition  $\vec{d} + 1$

## Subtraction in Binary

- ▶ Suppose that  $a_2a_1a_0 \stackrel{\text{def}}{=} 101$  and  $b_2b_1b_0 \stackrel{\text{def}}{=} 110$ . Then the denary subtraction is  $-3 - (-2) = -1$ .
- ▶ The negation of  $110$  is  $\overline{110} + 1 = 001 + 1 = 010$ .
- ▶ Now we work out the binary ALU result for  $101 + 010$

$\vec{a}$		1	0	1
$\vec{b}$	+	0	1	0
$\vec{s}$	(0)	1	1	1

- ▶ So the ALU result is  $111_s = (-4 + 2 + 1)_d = -1_d$ .

To calculate  $\vec{a} - \vec{b}$  calculate instead the sum  $\vec{a} + (\overline{\vec{b}} + 1)$ . Informally: swap 0s and 1s of  $\vec{b}$  and add 1. Then add  $\vec{a}$  to the result.



- ▶ Real memory locations are composed of **bits**. LEC
- ▶ Sometimes an unsigned binary number stored in a location needs to be copied into a larger location (more on why this is so later on).
- ▶ Eg start with **16** bits and copy into **32** bits.
- ▶ We fill the extra bits with zeros. This is called a **zero extension**.
- ▶ If the original number is  $\vec{b}$  we sometimes write the extended number as  $zx(\vec{b})$ .

Let  $\vec{b} \stackrel{\text{def}}{=} 10001$ . If this is zero extended by **3** digits then the result is?

- ▶ Real memory locations are composed of **bits**. LEC
- ▶ Sometimes an unsigned binary number stored in a location needs to be copied into a larger location (more on why this is so later on).
- ▶ Eg start with **16** bits and copy into **32** bits.
- ▶ We fill the extra bits with zeros. This is called a **zero extension**.
- ▶ If the original number is  $\vec{b}$  we sometimes write the extended number as  $zx(\vec{b})$ .

Let  $\vec{b} \stackrel{\text{def}}{=} 10001$ . If this is zero extended by **3** digits then the result is?

$$zx(\vec{b}) = zx(10001) = 000.10001$$

- ▶ The same applies to signed binary numbers.
- ▶ Eg start with 16 bits and copy into 32 bits, extending by 16.
- ▶ We fill the extra bits with *the sign digit of the number to be copied*. So for a  $k$ -digit number  $b_{k-1}b_{k-2} \dots b_0$ , the extra bits contain  $b_{k-1}$ . This is called a **sign extension**.
- ▶ If  $\vec{b}$  is the original number we write the extended number as  $sx(\vec{b})$ .

Let  $\vec{b} \stackrel{\text{def}}{=} 10001$ . If this is sign extended by 3 digits then

$$sx(\vec{b}) = sx(10001) = 111.10001$$

There is a crucial property of sign extension:  $\vec{b}_s = sx(\vec{b})_s$ . For example  $10001_s = -16 + 1 = -128 + 64 + 32 + 16 + 1 = 111.10001_s$ .

## Overview: Logic and Logical Operations

A **circuit** is a collection of *conductors* through which *electric current* can flow. Circuits are *analogue* or *digital*: Very roughly speaking, inputs and outputs to analogue circuits are given using real numbers (continuous); and in the case of digital, are given using integers (discrete, represented in binary).

Logic is the study of **truth** (and **falsity**). We do calculations with the *logic values* true and false.

It is a fact that

Complex digital circuits (eg in your phone, computer, . . . ) can be built from very simple circuits called **gates**.

**Gates** *calculate* values using simple **logic**, and are *built* from transistors.

After studying this section you should be able to

- ▶ Explain why **propositional logic** is important, and what it is about.
- ▶ Define **propositions** involving **AND** and **OR** and **NEGATION**.
- ▶ Write down basic and complex **Truth Tables**.
- ▶ Draw simple pictures of **logic gates** and calculate **outputs** from given **inputs**.

- ▶ Logic is the study of statements that are either *true* or *false*. We call such statements **propositions**.
  - ▶ For example “it is raining” or  $0 \leq 3$  or  $7 < 7$ .
- ▶ We saw that the 3-digit binary numbers can represent natural numbers  $n$  from 0 to 7. Informally: variable  $n$  ranges between 0 and 7. Slightly more formally:  $0 \leq n$  and  $n \leq 7$ . And now as “program code”:

$$0 \leq n \quad \&\& \quad n \leq 7 \quad (P)$$

- ▶ If you give me a specific value for  $n$ , then
  - ▶  $0 \leq n$  is true, AND  $n \leq 7$  is true  $\implies P$  is true.
  - ▶  $0 \leq n$  is true, AND  $n \leq 7$  is true  $\longleftarrow P$  is true.

- ▶ We say a proposition ***P*** has a **value** of true or false.
- ▶ In general, if ***A*** and ***B*** are propositions, then

***A*** && ***B*** is true  $\iff$  ***A*** is true AND ***B*** is true



***A*** || ***B*** is true  $\iff$  ***A*** is true OR ***B*** is true

Can you see why this is so:

***A*** && ***B*** is false  $\iff$  ***A*** is false OR ***B*** is false

***A*** || ***B*** is false  $\iff$  ***A*** is false AND ***B*** is false

Computers typically use **0** for **false** and **1** for **true**.

It is very easy to compute the logic values of  $A \&\& B$  and also  $A || B$  for all possible logic values of  $A$  and  $B$  ...

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

$A$	$B$	$A \&\& B$
0	0	0
0	1	0
1	0	0
1	1	1

$A$	$B$	$A    B$	
0	0	0	$A    B$ is 1 $\implies$ A is 1 OR B is 1
0	1	1	$A    B$ is 1 $\iff$ A is 1 OR B is 1
1	0	1	$A    B$ is 1 $\iff$ A is 1 OR B is 1
1	1	1	$A    B$ is 1 $\iff$ A is 1 OR B is 1



- ▶ If  $A$  is a proposition, then  $\overline{A}$  is its **negation**. If  $A$  is true/false then  $\overline{A}$  is false/true.
  - ▶ Let  $A$  be “The sea is red”. Then  $\overline{A}$  is “The sea is not red”.
  - ▶ Let  $n \stackrel{\text{def}}{=} 4$  and  $m \stackrel{\text{def}}{=} 6$ . Then  $n = m$  is false. Further  $\overline{n = m}$  means  $n \neq m$  which is true.

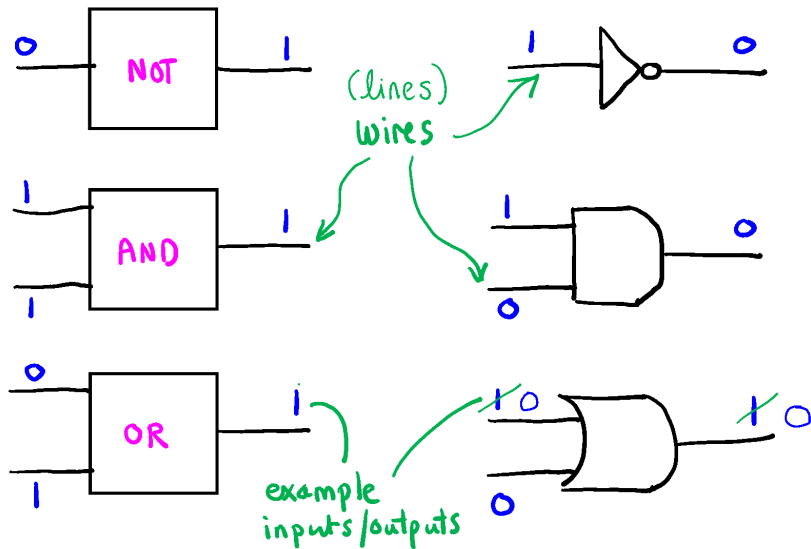
What is  $\overline{0 \leq 3}$ ? And  $\overline{7 < 7}$ ?

- ▶ If  $A$  is a proposition, then  $\overline{A}$  is its **negation**. If  $A$  is true/false then  $\overline{A}$  is false/true.
  - ▶ Let  $A$  be “The sea is red”. Then  $\overline{A}$  is “The sea is not red”.
  - ▶ Let  $n \stackrel{\text{def}}{=} 4$  and  $m \stackrel{\text{def}}{=} 6$ . Then  $n = m$  is false. Further  $\overline{n = m}$  means  $n \neq m$  which is true.

What is  $\overline{0 \leq 3}$ ? And  $\overline{7 < 7}$ ?

False ( $0 \leq 3$  is true). True ( $7 < 7$  is false).

# Logic Gates



Produce the truth table for  $R \stackrel{\text{def}}{=} (A \&\& \bar{B}) \parallel (C \parallel A)$ .

$A$	$B$	$C$	$\bar{B}$	$A \&\& \bar{B}$	$C \parallel A$	$R$
0	0	0	1	0	0	0
0	0	1	1	0	1	1
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	1	1	1	1
1	0	1	1	1	1	1
1	1	0	0	0	1	1
1	1	1	0	0	1	1

Note:  $ABC \in \text{Bin}^3$  so there should be  $2^3$  rows in the table.

Please write the rows in order from  $ABC_b = 0_d$  to  $ABC_b = 2^3 - 1_d$ .

- ▶ There are logical operations that are similar to  $+$  and  $-$ .
- ▶ They are **bitwise and**  $\&$  and **bitwise or**  $|$ . The formal definitions are on Slide 216; here we give only a couple of examples.
- ▶ These operations have practical uses but we omit them for now.

Let  $\vec{a} = 10110110$  and  $\vec{b} = 11110000$ . Then  $\vec{a} \& \vec{b} = 10110000$ .

$\vec{a}$	1	0	1	1	0	1	1	0
$\vec{b}$	1	1	1	1	0	0	0	0
$\vec{a} \& \vec{b}$	1 & 1 = 1	0	1	1	0	0	0	0

Sometimes we write

$$1010 \mid 0110 = (1 \mid 0)(0 \mid 1)(1 \mid 1)(0 \mid 0) = 1110$$

# An Overview of Computer Architecture

---

Computer = Hardware + Software

- ▶ We study examples of software which execute on a hardware processor.
- ▶ Next we give both a broad overview of memory, and the technical details required for an in-depth understanding.
- ▶ Finally we examine a simple processor in detail, especially the components called Datapath and Control.

## Overview: A Quick Tour of Processor Hardware and Software

Recall `add y, x, 3` on slide 8, 9. `x := 5; add y, x, 3` is another simple **software program**. It is a sequence of two **assembly instructions**. Each instruction **executes** on the digital **hardware** of a processor. Each instruction is stored in computer memory as a binary number known as a **machine instruction**.

After studying this section you should be able to

- ▶ Explain **processor hardware** in terms of simpler **hardware components**: the **Hardware Equation**.
- ▶ Recognize simple **programs** and **execute** them by completing a table of changes in the values of the variables.
- ▶ Explain in some detail what **assembly instructions** and **machine instructions** are, and give simple examples.

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

A **multi core processor** is (usually) a single physical integrated circuit that is built from a small number of individual processor circuits. These individual processors work together to give enhanced computational power. *multi* refers to “small number”. And *core* is an alternative name for an *individual processor*.

The term *Central Processing Unit (CPU)* is still used for a single core/single processor.



```

// program myAdd
// add the integers 0 up to 4
s := 0;
c := 1;
L   s := s+c;
    c := c+1;
    if c <= 4 then L;
    exit;
```

What is the value of `c` when we “first arrive” at the `if` statement?

variables specify memory areas  $v$  7

after executing  $v := 8$   $v$  8

8 overwrites 7

## Execution of myAdd

S		C	State Table	
0		1		
$S := S + C$		$C := C + 1$		
$0 + 1 = 1$		$2 = 1 + 1$	$C \leq 4$ TRUE	<div style="display: flex; align-items: center;"> <div style="margin-right: 10px;">             memory state updates ↓           </div> <div>exit:</div> </div>
3		3	$C \leq 4$ TRUE	
6		4	$C \leq 4$ TRUE	
10		5	$C \leq 4$ FALSE	

- ▶ An **algorithm** is a set of instructions or recipe for completing a task (eg *add up 0, 1, 2, 3, 4*).
- ▶ A **computer program** (eg `myAdd`) is an implementation of an algorithm, written in a programming language.
- ▶ There is a programming language known as **assembly**; a program consists of a sequence of **assembly instructions**.
- ▶ `myAdd` is an example of a such a program; each line of code is a single assembly instruction.
- ▶ Executing an assembly program means executing each instruction.
- ▶ Each instruction has a *semantics*: a precise description of exactly how the instruction executes.
- ▶ Each instruction is input to the processor, and the hardware executes the instruction (eg if `a := 0` is input, after execution the variable `a` contains `0`). On Slide 66 the value of `v` is **updated** from `7` to `8`.

- ▶ Each instruction can be physically stored on a computer as a binary number; this number is called a **machine instruction**.
- ▶ A sequence of machine instructions corresponding to an assembly program is a **machine language program** *MP*:

Usually *MP* is stored in the computer's main memory.

- ▶ We study the MIPS and ARM assembly and machine languages. MIPS was created in the 1980s but is still very much in use. ARM is much more recent and is found in a considerable majority of mobile devices.

## ASSEMBLY

S := 0

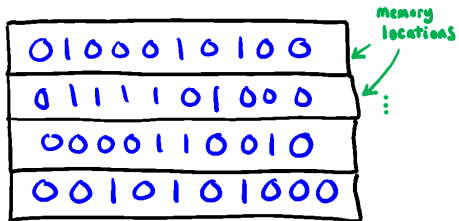
C := 1

S := S + C

C := C + 1

$$\begin{array}{l} I_1 \\ I_2 \\ I_3 \\ I_4 \\ I_5 \\ \vdots \end{array} \quad \left. \vphantom{\begin{array}{l} I_1 \\ I_2 \\ I_3 \\ I_4 \\ I_5 \\ \vdots \end{array}} \right\}$$

## MACHINE



We sometimes write this to indicate a general sequence of instructions –  
a program

Suppose you have a program *HP* written in a high-level language such as C, Java etc and stored on disk or flash memory. The steps involved in executing the program are:

- ▶ A **compiler** converts *HP* to an assembly program *AP* (on disk).
- ▶ An **assembler** converts *AP* into machine language ...
- ▶ ... and a **linker** produces a final machine language program *MP* by “linking in” code libraries.
- ▶ *MP* is **loaded** into main memory.
- ▶ Each instruction is then executed on the processor.

Note: this description is *over simplified* and does not properly describe how languages like Java execute on *virtual machines*.

- ▶ Suppose that our high-level program has been compiled down to a sequence of machine instructions, a machine language program *MP*, stored in memory.
- ▶ Roughly speaking, at any moment of time, a computer has a given **memory state**: the **memory** consists of locations where data is stored, and the **state** is a description of the data stored in each location.
- ▶ Each machine instruction *I* specifies how the computer's memory state should change: *I is input to the processor, I executes, and as a result the machine state changes.*
- ▶ We often say the state has been **updated**. See Slide 66, where the “state of variables *s* and *c* is recorded in a table”.

## Overview: Memory

A computer **memory** can be thought of as a collection of **locations** where one may store data. Each location has an **address**.

After studying this section you should be able to

- ▶ Summarise the basic ideas of computer **memory**, explain what **kinds** of memory technologies there are, and classify memory using the **memory hierarchy**.
- ▶ Explain the technical details of **basic memory**, such as **bits**, **bytes** and **words**, and the **Endian memory systems**.
- ▶ Solve simple problems, including drawing pictures of memory and giving your own examples.



- **Memory** comes in various kinds:
1. **Processor memory**, where data is stored within the processor itself. The smallest memory units are called *registers* and *cache*.
  2. **Main memory**, where data is stored inside computer circuits known as **memory chips**. The smallest memory units are called *cells*.
  3. **Outboard memory** often located on the computer, such as **magnetic hard discs** and so on.
  4. **External memory** such as **magnetic tape** or **cloud storage** or **flash drives**.

- ▶ As you move down the hierarchy, from 1 to 4:
  - ▶ decreasing cost per unit of memory;
  - ▶ increasing total capacity;
  - ▶ increasing access time;
  - ▶ decreasing access of memory directly by the processor.
- ▶ The design of powerful modern computing devices exploits the memory hierarchy in a crucial way.
- ▶ We return to it in the *cache memory* slides.

A processor has a memory of size **1024** bits; the whole memory costs **1/10**p. A **1**Gi-byte main memory chip costs **10**£. What is the cost per byte in each case? How many times more expensive is the processor memory (per byte)?

## The Memory Hierarchy: Unit Cost Example

A processor has a memory of size **1024** bits; the whole memory costs **1/10p**. A **1Gi**-byte main memory chip costs **10£**. What is the cost per byte in each case? How many times more expensive is the processor memory (per byte)?

The cost of **1** processor byte in pounds is

$$((1/1000)/1024) * 8 = (1/(1000 * 2^{10})) * 2^3 = 1/(1000 * 2^7)$$

The cost of **1** memory byte is  $10/2^{30}$ .

The cost ratio is  $1/(1000 * 2^7) \div 10/2^{30} = 2^{23}/10000 \approx 839$ .

*Informally, rounding to the most significant digit, the unit-cost of processor memory is 800 times the unit-cost of the main memory.*

- ▶ Extremely fast and small memory, such as processor storage, is made from **flip-flops (latches)**.

There are *four* key technologies for larger memories:

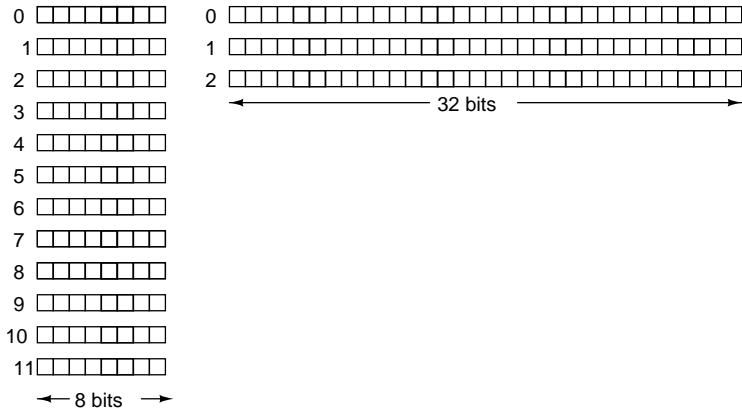
- ▶ **SRAM** (static random access) is very fast but correspondingly costly. Currently not used much.
- ▶ **DRAM** (dynamic random access) is an array of transistors and capacitors. A capacitor can store a small electrical charge, and a charge is interpreted as a binary digit. Each DRAM bit is physically simple, so is small and cheap. Being small gives a very high bit count per chip. Thus DRAM is often used for main memory, but it is also slower than SRAM. Used by mobile devices such as the iPad.

- ▶ **EEPROM** is erased by electrical voltage, and can be reprogrammed in place. EEPROMs are slower and smaller than D/SRAMS. **Flash** memory is a particular kind of EEPROM memory. Flash memory is, like DRAM, common on mobile devices. Note that writing data can wear bits out! Software solutions called **wear levelling** reduce the problem.
- ▶ **Disk Memory** is built from magnetic materials such as iron oxide. Magnetic fields can alter the state of these materials to store data.

- ▶ A computer can represent **0**s and **1**s as low and high voltages. A **bit** is a “digital circuit” that can “store” a **0** or **1**.
- ▶  $k$  bits can be joined together to make a fundamental memory *location* that can store a  $k$ -bit binary number.
  - ▶ In the processor these are called **registers**.
  - ▶ In a memory chip, **cells**.
- ▶ Memory locations have **addresses**  $a \in \mathbb{N}$ . We refer to “location  $a$ ” or “cell  $a$ ” ...
- ▶ Given an address  $a$  of a memory location one can find its **contents** (a  $k$ -bit binary number).

Memory chip	$\leftrightarrow$	Filing cabinet
Address of a location	$\leftrightarrow$	Draw label
Content of a location	$\leftrightarrow$	Documents in the draw

## Memory: Bits and Locations





- ▶ A memory with  $n$  locations will have addresses  $0$  to  $n - 1$ .
- ▶ If there are  $t$  bits in total, we call it an  **$t$ -bit memory** (we saw two **96-bit** memories).
- ▶ A standard **8-bit** main memory cell is called a **byte location**.
- ▶ The content is an **8-digit** binary number called a **byte**.
- ▶ A *group of consecutive* cells is called a **word location**.  
(Eg cells **4**, **5**, **6**, **7**)
- ▶ We soon define the *content* of a word location, called a **word**.
- ▶ We informally refer to both physical locations, and the binary content, as bytes and words.

The **address of a word location** is the *smallest* of its cell addresses; equivalently, the address of the first cell in the group.

- ▶ When data is input to a cell it is said to be **written** or **stored**. The data that was initially stored has now been **overwritten**.
- ▶ When data is output from a cell it is said to be **read**. Note that reading data does not usually alter the contents of a cell.
- ▶ A cell typically has two other kinds of input other than data. Each is called a **control signal**. One is **Read Enable** and the other **Write Enable**.
- ▶ Each signal has a value of **0** or **1**:

<b>RE</b> = Read Enable	<b>0</b>	Data cannot be read	"reading off"
<b>RE</b> = Read Enable	<b>1</b>	Data can be read	"reading on"
<b>WE</b> = Write Enable	<b>0</b>	Data cannot be written	"writing off"
<b>WE</b> = Write Enable	<b>1</b>	Data can be written	"writing on"

## Organizing Memory: Endian Systems

A number can be large, so it may not fit inside a cell. But it may “fit into a group of cells”. Suppose there is a word location composed of 2-bit cells, with addresses 3, 4, 5.

How can I store 011110? LEC

Suppose the cells contain 10, 01, 11 respectively, what is the content of the word location? LEC

See the in-lecture notes . . . LEC

Now more realistic questions.

Extracting the content of a word location (reading):

- ▶ Given a word location of four consecutive bytes, what is the **32**-bit MIPS binary number stored there?

Placing content in a word location (storing/writing):

- ▶ Given a **64**-bit number how *exactly* is the number stored in an ARM device with (double) word locations, each of eight bytes (writing)?

Let's look at some examples. MIPS first; then a summary; then ARM.

## Example: The Content of a Word Location

A word location consists of four byte locations with addresses 4,5,6,7. These contain the bytes 11111111, 00000000, 11110000, 00001111. What is the 32-bit MIPS word contained in the word location (reading)?

- ▶ 11111111.00000000.11110000.00001111 ?
- ▶ 11111111.11110000.00001111.00000000 ?
- ▶ 00001111.11110000.00000000.11111111 ?
- ▶ 01101110.00101001.01111101.11000000 (16 1s and 16 0s)?

## Example: The Content of a Word Location

A word location consists of four byte locations with addresses 4,5,6,7. These contain the bytes 11111111, 00000000, 11110000, 00001111. What is the **32**-bit MIPS word contained in the word location (reading)?

- ▶ 11111111.00000000.11110000.00001111 ? **Big Endian** (bigger)
- ▶ 11111111.11110000.00001111.00000000 ? **X**
- ▶ 00001111.11110000.00000000.11111111 ? **Little Endian**
- ▶ 01101110.00101001.01111101.11000000 (16 1s and 16 0s)? **X**

In **Big Endian** (**Little Endian**) memory the content of a word location is specified by **concatenating the individual cell contents** in **increasing (decreasing)** address order.

- ▶ The addresses get ***BIG**ger* in Big Endian.
- ▶ The addresses get ***LITTLE**r* in Little Endian.
- ▶ Both MIPS and ARM make use of Big Endian memory
- ▶ In CO1104 MIPS main memory has **8**-bit cells (ie bytes) and **32**-bit words locations.
- ▶ ARM works with **8**-bit cells and **64**-bit word locations: these are called **double words**.

Recall *most* and *least* significant digits (Slide 31)

most significant			...			least significant
leftmost			...			rightmost

Question: A user enters the number **1073774721** into a **64**-bit ARM mobile phone. If the compiler allocates byte addresses 8, 9, 10, 11, 12, 13, 14, 15, what data is stored in each byte (writing)? Hint: the number is  $1 + 2^7 + 2^{15} + 2^{30}$ .



- ▶ The number is 1000000.00000000.10000000.10000001, and in 64-bits is
- ▶ 00000000.00000000.00000000.00000000...  
...01000000.00000000.10000000.10000001
- ▶ In Big Endian the byte addresses increase, so the most significant byte 00000000 is stored in byte location 8, and so on with 10000000 in byte 14 and 10000001 in byte 15.

## Overview: A Simple Processor (Simple Core)

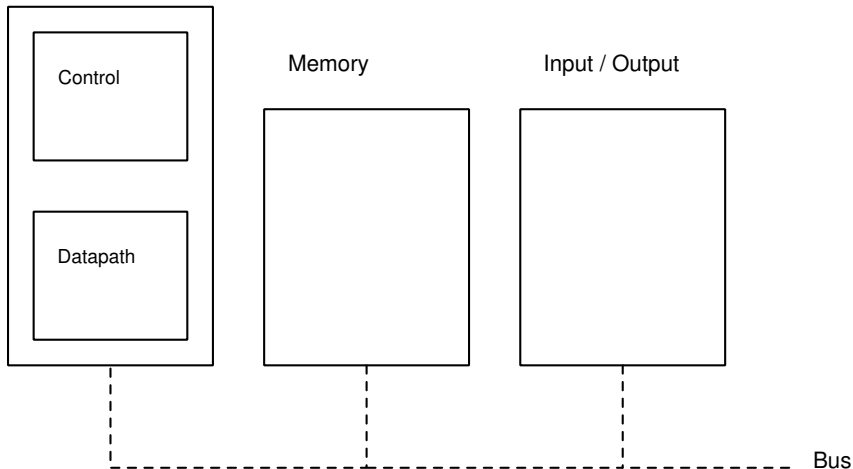
A program  $P$  is executed on a processor by executing each instruction  $I$  one by one. We have already seen the outline of a **Datapath** on which the execution takes place (Slide 8). There is also a **Control Unit** which organises the execution: it outputs **control signals**.

After studying this section you should be able to

- ▶ State what the **Fetch Decode Execute** cycle is, and explain it, including drawing pictures and giving examples.
- ▶ Explain at an abstract level how programs  $P$  and instructions  $I$  execute; define the **IR** and **PC**.
- ▶ **IMPORTANT**: Draw a detailed picture of a **Simple Datapath**, explain what all the components are, and solve unseen problems. Problems may include showing how simple instructions execute, and calculating execution times. **The Datapath underpins CO1104.**

# How does a computer work?

Central Processing Unit = CPU = Single Core = Processor



- ▶ Recall that *P* is a sequence of machine instructions stored in main memory: for example  $l_0, l_1, \dots, l_{16}, l_{18}, \dots$
- ▶ The instructions will be executed one by one using a **fetch-execute (FE)** cycle.

**Fetch** Each instruction is copied into a **register** in the processor. This is called the **Instruction Register (IR)** and it stores individual machine instructions.

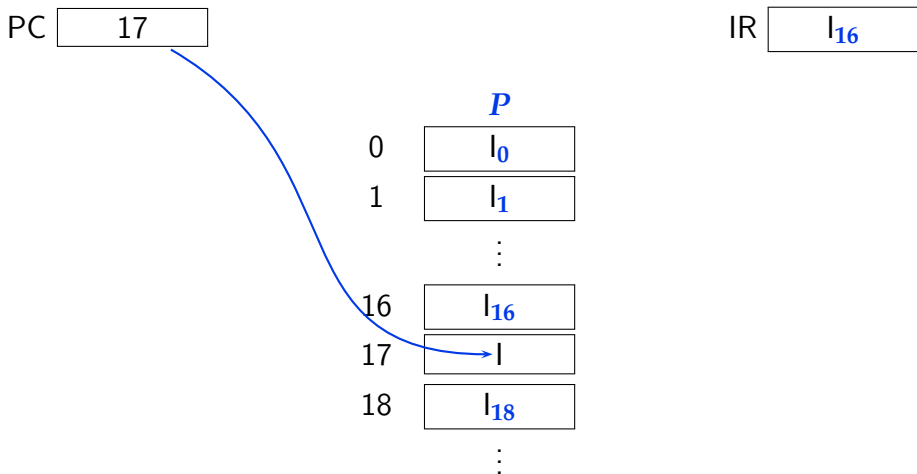
**Execute** The instruction is executed.

The FE-cycle repeats until all instructions in *P* are executed.

- ▶ How does the processor *find* the instructions?
- ▶ A processor register called the **Program Counter (PC)** stores the **address** in main memory of the **next** instruction to be executed. The PC is an example of a **pointer**.

## Executing a Machine Language Program *P*

A main memory (with program *P*) and IR/PC state, *during* execution of *I*<sub>16</sub>. The PC “points” to next instruction to be executed.



## Executing a Machine Language Program *P*

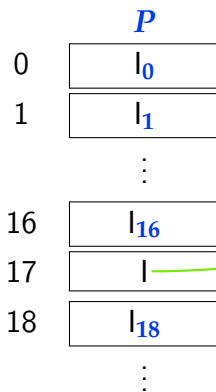
The state just after *I* is fetched, ie copied, into the IR. Execution of *I* begins ...

PC 

17
----

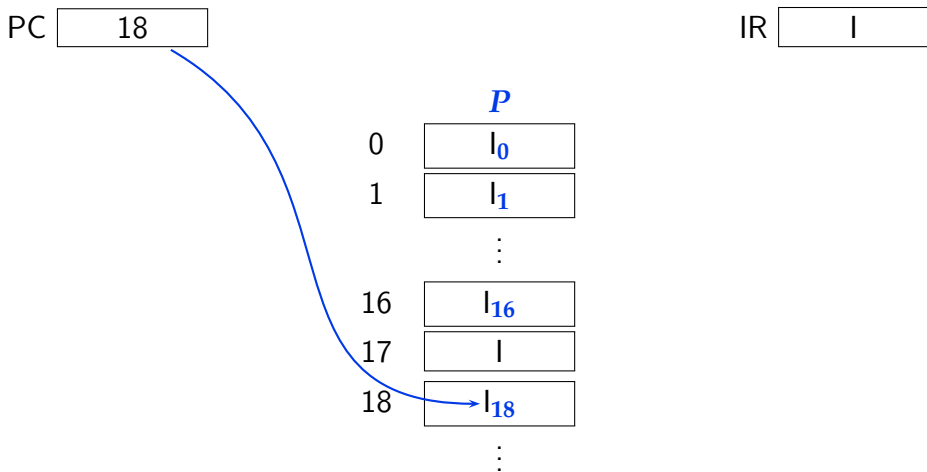
IR 

<i>I</i>
----------



## Executing a Machine Language Program *P*

...during execution of *I* the PC is updated to point to the next instruction to be executed. *I determines the new PC value.*

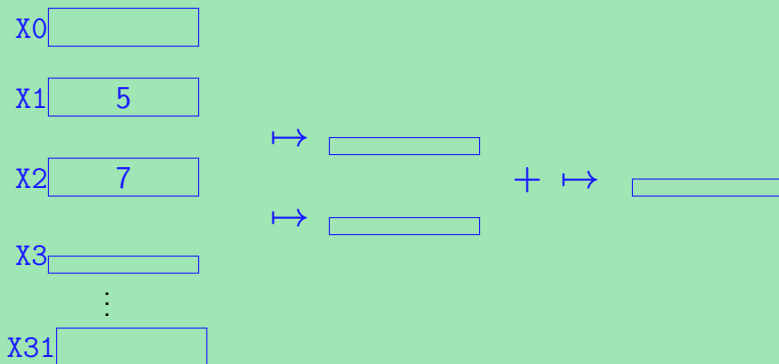


- ▶ Now we study how an individual instruction is executed.
- ▶ We look at an example: `ADD X3, X1, X2`.
- ▶ Let's take a high-level (abstract) view of the execution on a Datapath ...
- ▶ ...and then explain the Datapath details.



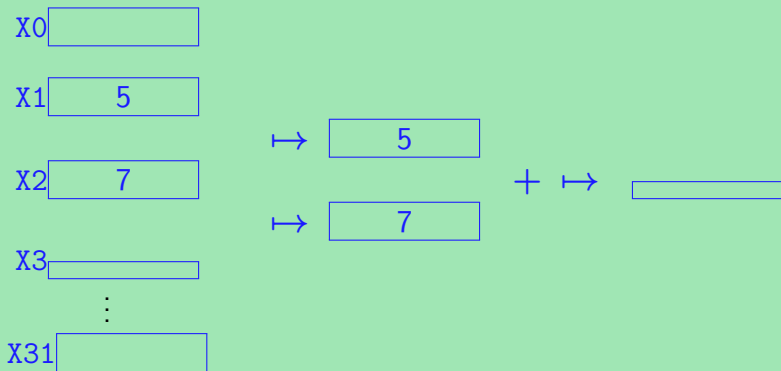
What are the execution stages of `ADD X3, X1, X2`?

Abstract view of a Datapath: This is the state before execution.



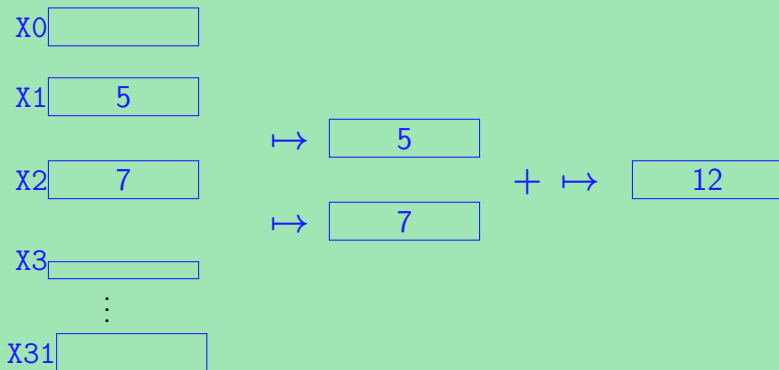
What are the execution stages of `ADD X3, X1, X2`?

Abstract view of a Datapath:



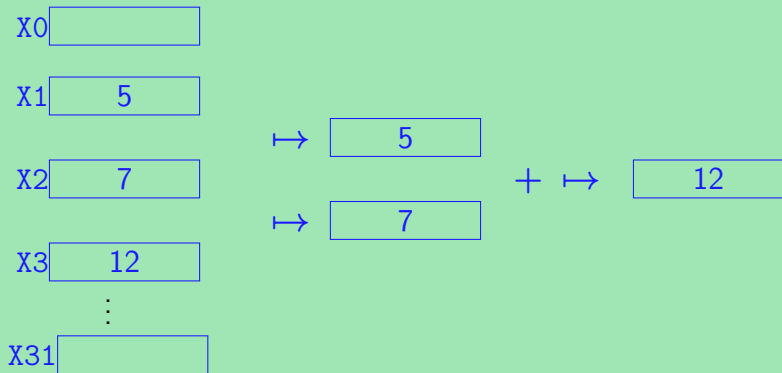
What are the execution stages of `ADD X3, X1, X2`?

Abstract view of a Datapath:



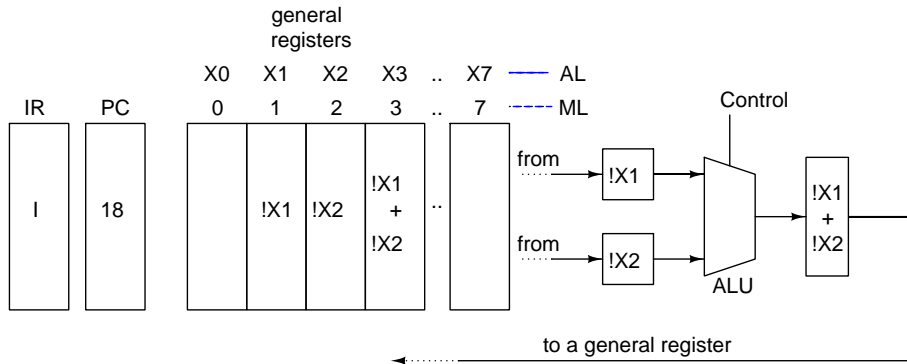
What are the execution stages of `ADD X3, X1, X2`?

Abstract view of a Datapath:



Now compare to Slide 8, and the detailed Datapath on Slide 93.

## Details of A Simple Datapath



**A Datapath after execution of  $I = \text{ADD } X3, X1, X2$**

IR 1000.011.001.010 (binary)

8.3.1.2 (denary)

- ▶ **General (Purpose) Registers** store general data such as numbers.
- ▶ X1 and X2 and X3 are registers. Here they contain 5, 7 and 12.
- ▶ In a real device each register has a **fixed content size** of typically 32 or 64 bits, and an **address** (see Slide 78).
- ▶ We refer to the **content** of X1 by writing !X1. E.g.

!X1 = 00000000.00000000.00000000.00000101      !X1<sub>s</sub> = 5

- ▶ We refer to the **address** of X1 by writing #X1. E.g.

#X1 = 001      #X1<sub>b</sub> = 1

**IMPORTANT!** Look back at Slide 8. We said that instructions contain variables. This is not quite true. They actually contain registers! A high-level language does have variables, and a compiler maps each variable to a chosen register.

- ▶ **Busses** are channels along which binary data is transmitted. They are composed of **parallel wires** and each wire has a high or low (1 or 0) voltage. The number of wires is called the **bus width**.
- ▶ The **Arithmetic Logic Unit (ALU)** performs arithmetic and logic operations (e.g. **+**). It has two **input registers** and one **output-register**. It also has a **Control** bus that is used to select the operation: The *Control Unit* (not drawn on Slide 93) sends out a control signal to the Control bus to make the selection.

LEC A **32-bit** ALU has two **32-bit** input-busses and one **32-bit** output-bus. Two **32-bit** operands are stored in the input registers and then sent along the input busses in parallel. The ALU computes the operation specified by Control. When an operation finishes, the **32-bit** result is transmitted along the output-bus and stored in the output-register.

- ▶ The *syntax* of *I* (in *assembly language*) is **ADD** **X3**, X1, X2.
- ▶ Recall Slide 90. The *machine language* for *I* is fetched from main memory into the IR. Then the content of the IR is

$$\underbrace{1000}_{+} . \quad \underbrace{011}_{\text{address of X3}} . \quad \underbrace{001}_{\text{address of X1}} . \quad \underbrace{010}_{\text{address of X2}} .$$

- ▶ The machine code *supplies the register addresses* so the processor is able to **locate** the three general registers. (See Slide 119—the address bus is omitted from Slide 93 to save space).
- ▶ In this example **X3**, X1 and **X2** have denary addresses **3**, 1 and **2**.
- ▶ It is a special feature of ARM that register **X<sub>n</sub>** has address ***n***.



- ▶ **1000** is sent to the Control Unit, which then sends out a signal to the ALU Control bus which makes the ALU do addition. LEC

The (machine language of) instruction *I* is **fetched** (i.e. copied) from Main Memory to the IR. The use of the machine language addresses to locate the registers, and to set the ALU to do **+**, is called **Decoding** (this is when the Control Unit is used). The final steps (below) make up the **Execution** stage.

- ▶ *I* is now executed (as in Slide 92):
  - ▶ The contents of **X1** and **X2** are copied to the ALU input registers ( $!X1 \stackrel{\text{def}}{=} 5_d$  and  $!X2 \stackrel{\text{def}}{=} 7_d$ ).
  - ▶ The contents of the ALU input registers are sent to the ALU, added, and the result copied into the output register.
  - ▶ The result is finally copied into **X3** ( $!X3 = 5 + 7 = 12$ ).

The *Control Unit* performs a sequence of steps, called the **fetch-decode-execute (FDE)** cycle.

**Fetch** Using the PC *main memory* address, the instruction at that address is fetched (copied) into the IR. This is called the **current** instruction.

**Decode** Registers are located using their addresses (the addresses are parts of the machine instruction). The ALU is set to perform an operation, such as addition (the ALU control value is also part of the machine instruction).

**Execute** The instruction is executed. The PC is up-dated (the current instruction is used to determine the address of the next instruction: we learn later on how this address is computed.)

The FDE-cycle repeats until all instructions in *P* are executed.

# Processor Hardware

---

- ▶ A hardware digital circuit can be thought of as a black box with binary numbers as inputs and outputs.
- ▶ These circuits are built from logic gates, themselves built from transistors. There is an electrical power supply, and currents and voltages may be measured within the circuits.
- ▶ We take a look at the hardware circuits required to build a small MIPS/ARM processor:
- ▶ ALU, multiplexor, clock, bit, location, register file and memory chip. (Note: in your independent study time, look up decoder.)

- ▶ We look at two kinds of hardware circuit.
  - ▶ **combinational** and **state**
- ▶ Think of a circuit as having  $n$  input wires and  $m$  output wires: Overall a binary number with  $n$ -digits goes in, and one with  $m$ -digits comes out.
- ▶ In the *combinational* case, suppose you give me any input  $\vec{i}$ . There may not be an output; but if there is (say  $\vec{o}$ ) it is *unique* in the following sense: If at any other time you give input  $\vec{i}$  then the circuit gives the *same* output  $\vec{o}$ . Thus combinational circuits implement *partial functions*.
- ▶ In the *state* case, at different times there may be different outputs  $\vec{o}$  and  $\vec{o}'$  for the same input  $\vec{i}$ .

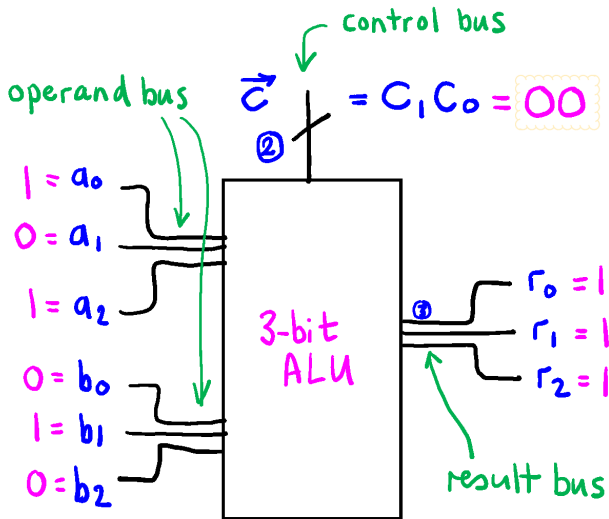
Can you give one example of each? We will see both as we progress.

We study **ALUs** and **Multiplexors** in detail.

At the end of this section you should be able to

- ▶ Draw an **ALU** and solve simple problems involving inputs and outputs, including the control bus.
- ▶ Explain in detail how a **Multiplexor** works, draw pictures, solve problems involving inputs and outputs, including the control bus.
- ▶ Given a circuit diagram for a **Multiplexor**, trace **dataflow** around the circuit by computing with **gates** (see Slide 59).

# Arithmetic Logic Units



control settings

$C_1 C_0$	op
00	+
01	-
10	&
11	

$101 - 010 = 011$   
 $101 \& 010 = 000$   
 $101 | 010 = 111$

$$\vec{r} = \vec{a} + \vec{b}$$

$$111 = 101 + 010$$

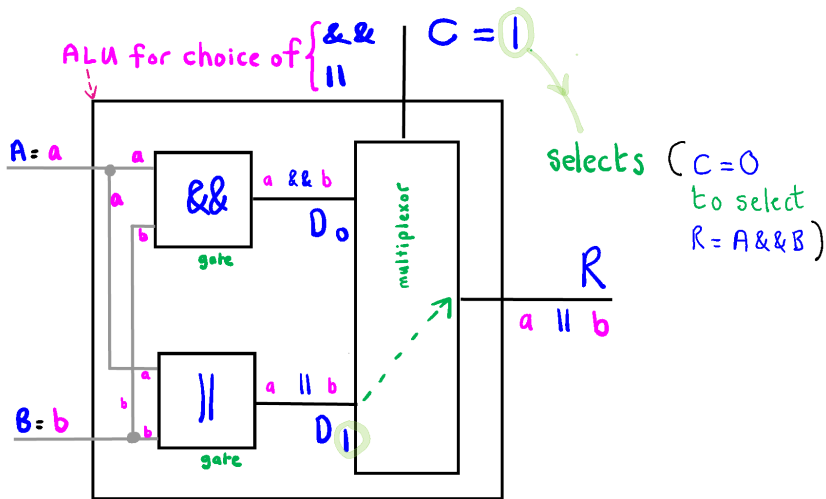
- ▶ Two  $k$ -bit data input busses (holding  $k$ -digit binary numbers  $\vec{a}, \vec{b}$ ).
- ▶ One  $k$ -bit output bus.
- ▶ One (input) control bus. Each control bus value (control signal) will select the ALU operation: this might be  $+$ ,  $-$ ,  $*$ ,  $\&$  or  $|$  (and so on).
- ▶ The result  $\vec{r}$ , given by the selected operation applied to the operands  $\vec{a}$  and  $\vec{b}$ , appears on the output bus.
- ▶ In this module MIPS uses a 32-bit ALU; the version of ARM in current mobile devices is typically 64-bit.

Some results, such as  $\vec{r} \stackrel{\text{def}}{=} \vec{a} + \vec{b}$ , may be wrong. Either  $\vec{r}$  is correct, or there is an overflow warning if incorrect. LEC

- ▶ Suppose an ALU has a particular operation selected.
- ▶ In an ALU there is one *basic* circuit for each operation. LEC
- ▶ These circuits are arranged in parallel: see the LEC example, and Slide 105.
- ▶ So the results of these circuits are available in parallel! All the results are sent to the data input busses of a multiplexor.
- ▶ The purpose of a **multiplexor** is to *select* the **required result**.
- ▶ Multiplexors are also called **data selectors**.

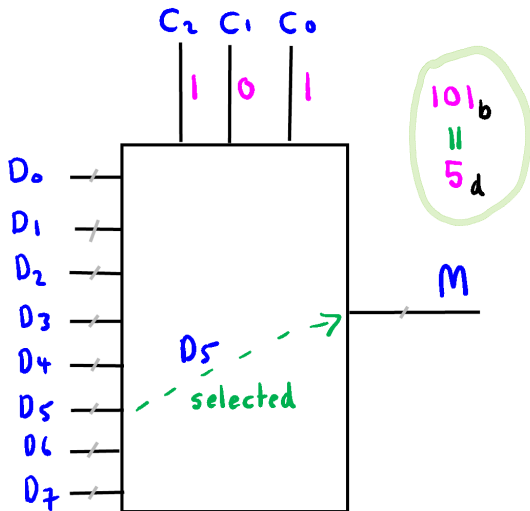


## Multiplexor (Building an ALU for $\&\&$ and $\|\|$ )



if  $C = 1$  then  $R = D_1 = A \|\| B$

# Multiplexor



$\vec{C}$	$M$
000	$D_0$
001	$D_1$
010	$D_2$
011	$D_3$
100	$D_4$
101	$D_5$
110	$D_6$
111	$D_7$

$$M = D_{C_2 C_1 C_0_b}$$

$D_i \in \text{Bin}^k$  (k-bit bus)

The output  $M$  is the data bus value  $M = D_{C_2C_1C_0b}$ .

- Example with control bus width 3 and data busses width 4:

$C_2$	$C_1$	$C_0$	$D_7$	$D_6$	$D_5$	$D_4$	$D_3$	$D_2$	$D_1$	$D_0$	$M$
0	0	1	—	—	1011	1001	1001	—	1111	—	1111
1	0	1	—	—	1011	1001	1001	—	1111	—	1011
?	?	?	—	—	1011	1001	1001	—	1111	—	1001

- In the example,  $101_b = (4 + 0 + 1)_d = 5_d$ , so the output  $M$  is  $D_5 = 1011$ .

If  $M = 1001$  and  $D_3 = D_4 = 1001$ , then what might  $C_2C_1C_0$  equal?

The output  $M$  is the data bus value  $M = D_{C_2C_1C_0b}$ .

- Example with control bus width **3** and data busses width **4**:

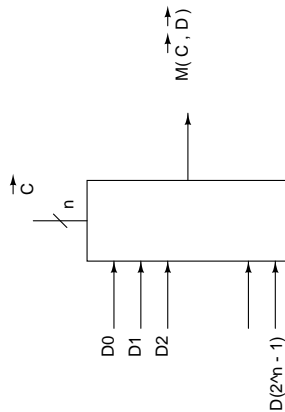
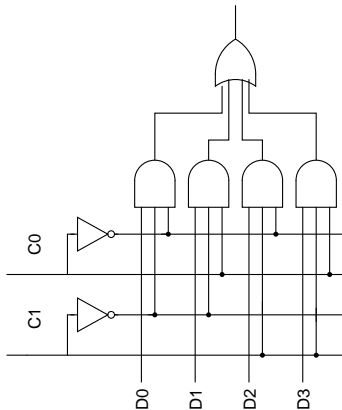
$C_2$	$C_1$	$C_0$	$D_7$	$D_6$	$D_5$	$D_4$	$D_3$	$D_2$	$D_1$	$D_0$	$M$
0	0	1	—	—	1011	1001	1001	—	1111	—	1111
1	0	1	—	—	1011	1001	1001	—	1111	—	1011
?	?	?	—	—	1011	1001	1001	—	1111	—	1001

- In the example,  $101_b = (4 + 0 + 1)_d = 5_d$ , so the output  $M$  is  $D_5 = 1011$ .

If  $M = 1001$  and  $D_3 = D_4 = 1001$ , then what might  $C_2C_1C_0$  equal?

$C_2C_1C_0_b = 3_d$  or  $= 4_d$ . Hence either **011** or **100**.

# Multiplexor Digital Circuit



### Summary:

- ▶ A **multiplexor** (usually) has
  - ▶  $n$  **control** input lines  $C_i$ , and
  - ▶  $2^n$  **data** input busses  $D_i$  and **1** **data** output bus.
- ▶ We refer to the control lines taken together as the **control bus** of width  $n$ .
- ▶ The data busses are (usually) all of the same width, say  $k$ -bits.

If there are  $2$  control lines, why are there  $2^2 = 4$  data busses?

### Summary:

- ▶ A **multiplexor** (usually) has
  - ▶  $n$  **control** input lines  $C_i$ , and
  - ▶  $2^n$  **data** input busses  $D_i$  and **1 data** output bus.
- ▶ We refer to the control lines taken together as the **control bus** of width  $n$ .
- ▶ The data busses are (usually) all of the same width, say  $k$ -bits.

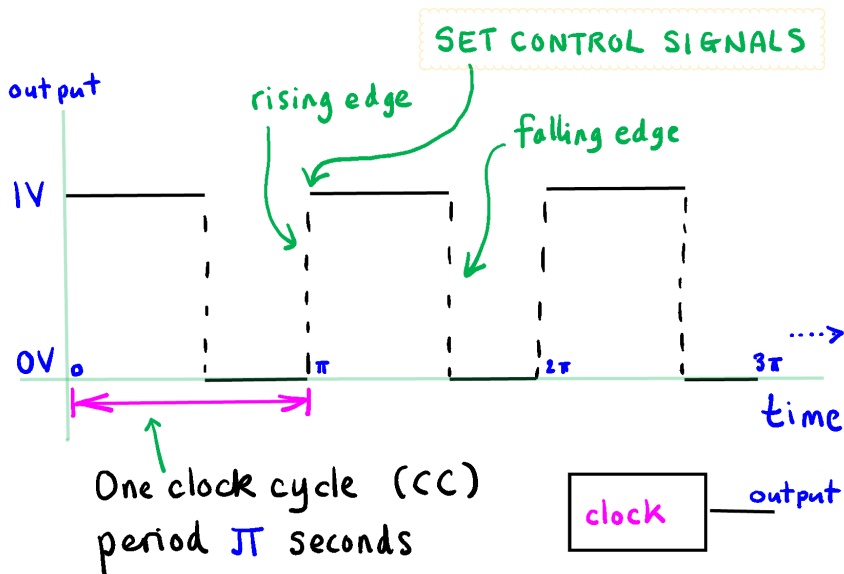
If there are 2 control lines, why are there  $2^2 = 4$  data busses?

If there are 2 control lines, the control bus can supply  $2^2$  binary numbers (see Slide 35); each number selects one data bus.

- ▶ In the processor, the timing of various processes and tasks is crucial. A *clock* circuit provides this timing.
- ▶ A **high pulse** is a voltage of **1** which lasts for a fixed time; ditto **low pulse**.
- ▶ A **clock** is a circuit which generates a series of high then low pulses, repeatedly.
- ▶ The total time taken for a high then low pulse is called the **clock cycle time** or **period**.
- ▶ The start time of a high clock pulse is called a **rising edge**; the **falling edge** is when the high pulse ends.



## \*\* Clock \*\*

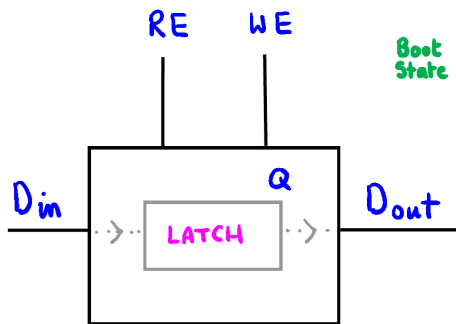


## Overview: State Circuits

We study **bits**, **memory locations**, and **address/data busses** in detail. We use these to make up some computer memory: **register files** and **memory chips**. (The set of *general registers* in the simple Datapath form a *register file*.)

At the end of this section you should be able to

- ▶ Draw detailed labelled pictures of **bits**, **cells/registers**, and **register files** (often starting at a **Boot State**).
- ▶ Recall and explain all of the input/output busses of these memory circuits, and the internal details.
- ▶ Solve simple (new) problems involving state changes to these memory circuits.
- ▶ Give state changes either by showing dataflow on a picture, or by giving updates such as  **$WD = 1001$** .



LATCH built from flip-flop

Boot State

RE	WE	Din	Q	Dout
0	0	0	0	—
0	0	1	0	—
0	1	1	1	—
0	0	1	1	—
1	0	1	1	1
0	0	1	1	—
0	0	0	1	—
0	1	0	0	—
0	0	0	0	—
1	0	0	0	0

red state changes  $\Rightarrow$  green state changes  
 CAUSE EFFECT

- ▶ Recall Slide 81.
- ▶ A bit has three **input** lines (two control, one data) and one **output** line (data). Inside is a **latch**; it stores a single bit,  $Q$ .
- ▶ If  $RE = 0$  we say **reading is off**; if  $RE = 1$  we say **reading is on**.
- ▶ If  $WE = 0$  we say **writing is off**; if  $WE = 1$  we say **writing is on**.
- ▶ To write/store a binary digit  $b$ : put  $b$  on the  $D_{in}$  line. Set  $RE$  to 0,  $WE$  to 1. Then the value of  $Q$  is set to  $b$ .
- ▶ To read a binary digit  $b$  (already stored): set  $RE$  to 1,  $WE$  to 0 (and  $b$ , read from  $Q$ , appears on the  $D_{out}$  line).

Superfast memory is built from *latches*. Look up both *latches* and *flip-flops* in further information sources:

- ▶ Our *data input line* **Din** corresponds to latch line **D**.
- ▶ Our **RE** line corresponds to latch line **C** (also called the *clock line*).
- ▶ Our *data output* **Dout** line corresponds to latch line **Q**.
- ▶ The full name for a latch is a *clocked CD-latch*.



- ▶ A  $k$ -bit location is composed of  $k$  single bits.
- ▶ There is a  $k$ -bit **data input bus**; and a  $k$ -bit **data output bus**.
- ▶ The location has one **RE** line and one **WE** line ...
- ▶ ... the corresponding lines from each of the bits are joined together:
  - ▶ INFORMALLY, the location is readable if each bit is readable at the same time.
  - ▶ FORMALLY, if the **RE** line is set to 1, each of the individual bits is made readable simultaneously by having its own **RE** line set to 1, so that the  $k$ -bit binary number stored in the  $k$  bits is output.

## Address and Data Busses (Reading)

- ▶ Recall that a register (or cell) has an *address*.
- ▶ If we want to read data from a register, we supply its address, on an **address bus, RR**, to *locate (point to)* the register.
- ▶ The register contents are output using a **data bus, RD**:  
(\*\* See Slide 125. LEC )

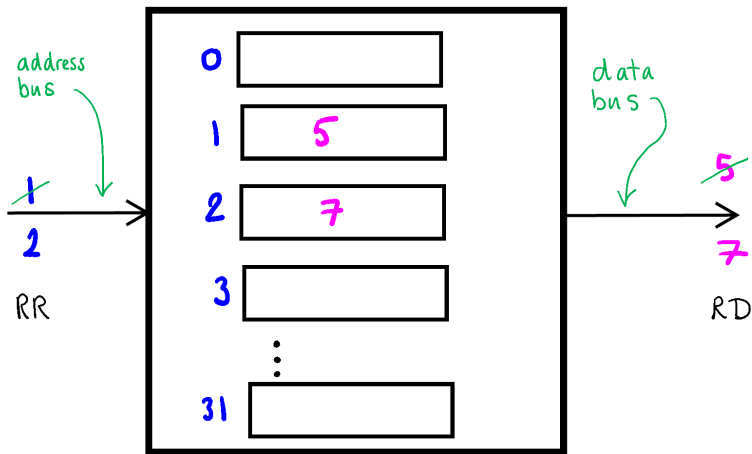
If there  $n$  registers, what is the width of the address bus? (Hint: if the bus width is  $k$ , how many registers can we address?)

Can you draw a picture similar to Slide 119 but for *writing*?

**WARNING:** I think a better name for the bus that supplies the address of a register to be read is **Read Address (RA)**. However, each register has a *unique* address and so the bus name **RR** makes sense. Since Hennessy and Patterson use **RR** I stick with it.



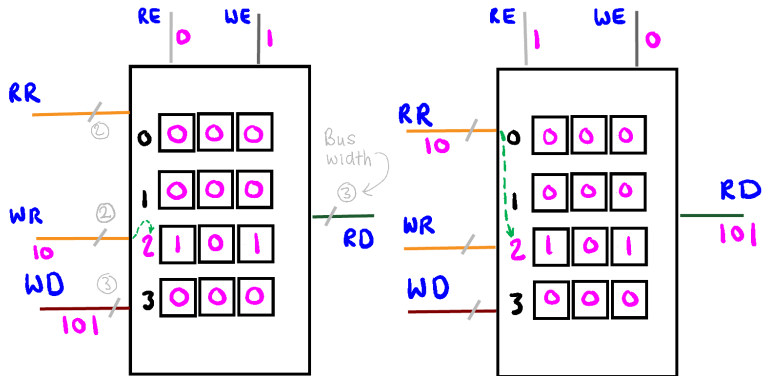
## Example: Address and Data Busses (Reading)



register file (main memory similar)

address comes from a machine instruction ; data goes to <sup>ALU</sup> or main memory

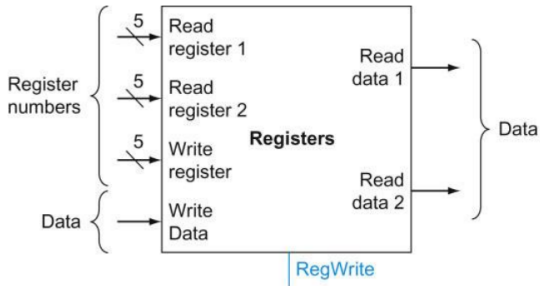
## Example: Register Files



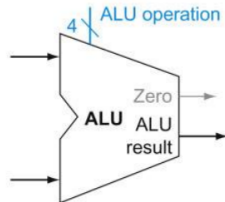
- **WR** and **RR** are "address busses" – they locate a register for writing/reading.  $10_b = 2_d$
- circuit detail omitted

- ▶ A *register file* is a collection of general registers found in a processor (recall our datapath). Data copied from main memory into a processor is stored in a register.
- ▶ A **register file** consists of
  - ▶  $k$ -bit registers;
  - ▶ **read-register** busses (**RR**) and **read-data** (**RD**) busses (in our processor there are 2 of each);
  - ▶ **write-register** (**WR**) busses and **write-data** (**WD**) busses (in our processor there is 1 of each); and
  - ▶ **read/write enable** lines **RE** and **WE**.
- ▶ The **WR** bus supplies the address of the register to be written to. The data to be written is on the **WD** bus.
- ▶ The **RR** bus supplies the address of the register to be read from. The data is read out on the **RD** bus.

# Register Files: In the MIPS/ARM Processor

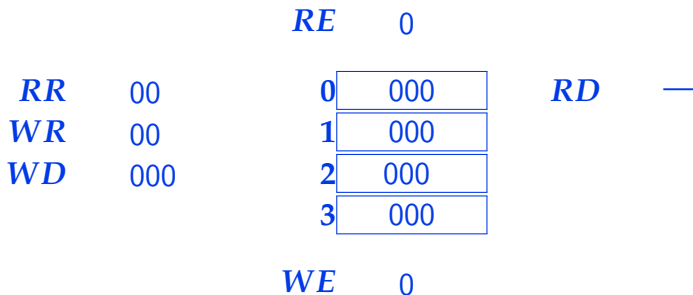


a. Registers



b. ALU

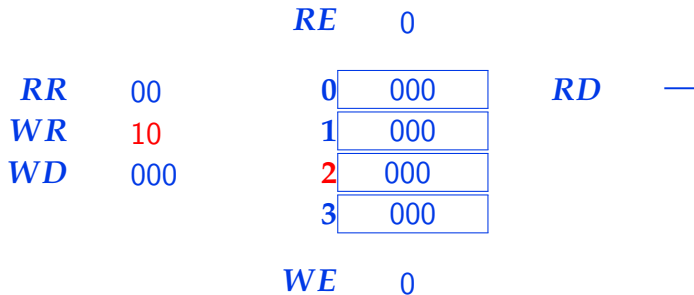
**RR** and **WR** are 2-bit input busses; **RD** is a 3-bit output bus; **WD** is a 3-bit input bus. **Boot State:**



**RR** and **WR**: *address (number)* of a register.

**WD**: data to be written (stored). **RD**: data to be read.

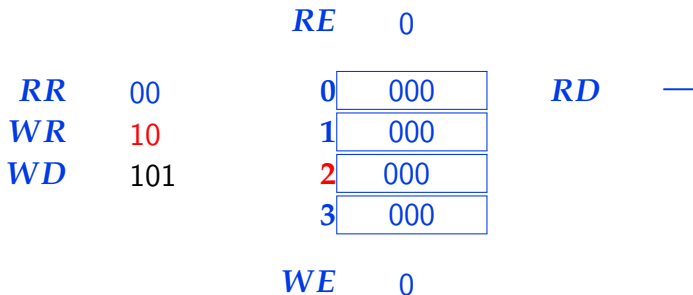
**RR** and **WR** are 2-bit input busses; **RD** is a 3-bit output bus; **WD** is a 3-bit input bus.



**RR** and **WR**: *address (number)* of a register.

**WD**: data to be written (stored). **RD**: data to be read.

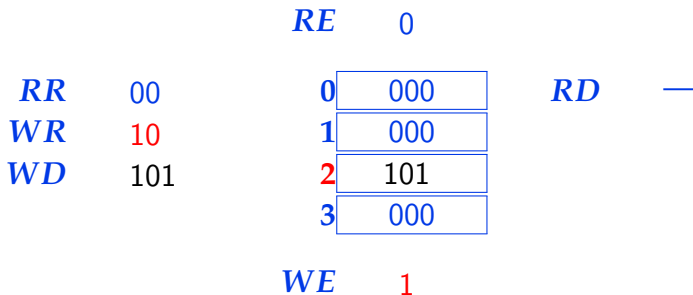
**RR** and **WR** are 2-bit input busses; **RD** is a 3-bit output bus; **WD** is a 3-bit input bus.



**RR** and **WR**: *address (number)* of a register.

**WD**: data to be written (stored). **RD**: data to be read.

**RR** and **WR** are **2**-bit input busses; **RD** is a **3**-bit output bus; **WD** is a **3**-bit input bus.

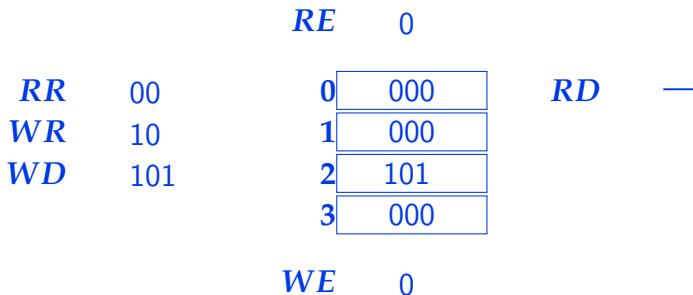


**RR** and **WR**: *address (number)* of a register.

**WD**: data to be written (stored). **RD**: data to be read.



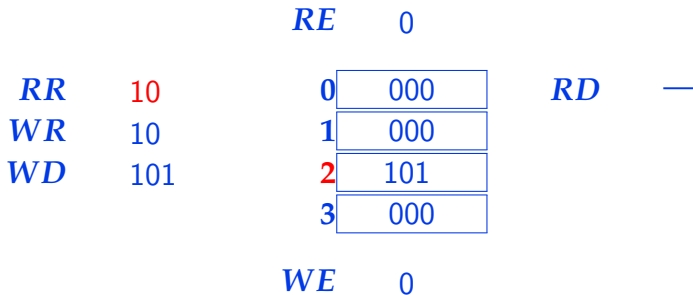
**RR** and **WR** are **2**-bit input busses; **RD** is a **3**-bit output bus; **WD** is a **3**-bit input bus.



**RR** and **WR**: *address (number)* of a register.

**WD**: data to be written (stored). **RD**: data to be read.

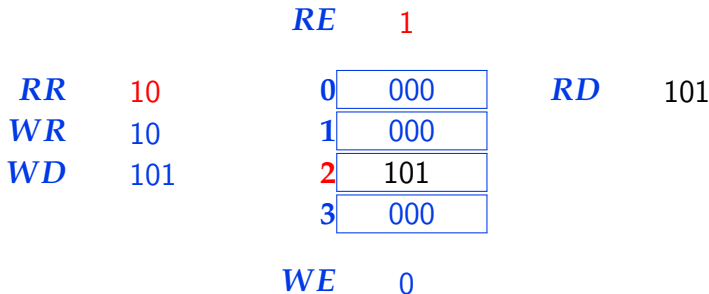
**RR** and **WR** are **2**-bit input busses; **RD** is a **3**-bit output bus; **WD** is a **3**-bit input bus.



**RR** and **WR**: *address (number)* of a register.

**WD**: data to be written (stored). **RD**: data to be read.

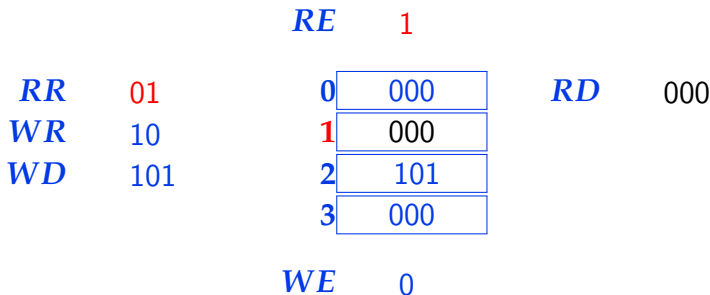
**RR** and **WR** are **2**-bit input busses; **RD** is a **3**-bit output bus; **WD** is a **3**-bit input bus.



**RR** and **WR**: *address (number)* of a register.

**WD**: data to be written (stored). **RD**: data to be read.

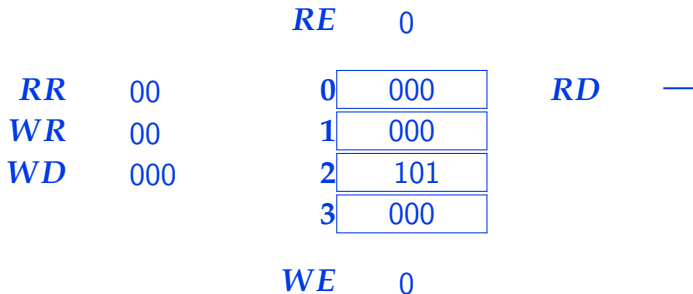
**RR** and **WR** are 2-bit input busses; **RD** is a 3-bit output bus; **WD** is a 3-bit input bus.



**RR** and **WR**: *address (number)* of a register.

**WD**: data to be written (stored). **RD**: data to be read.

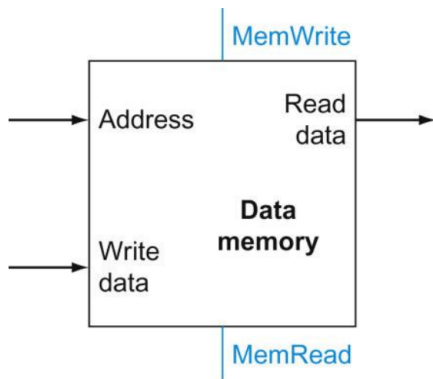
**RR** and **WR** are **2**-bit input busses; **RD** is a **3**-bit output bus; **WD** is a **3**-bit input bus.



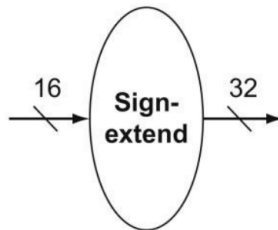
**RR** and **WR**: *address (number)* of a register.

**WD**: data to be written (stored). **RD**: data to be read.

# Memory Chip

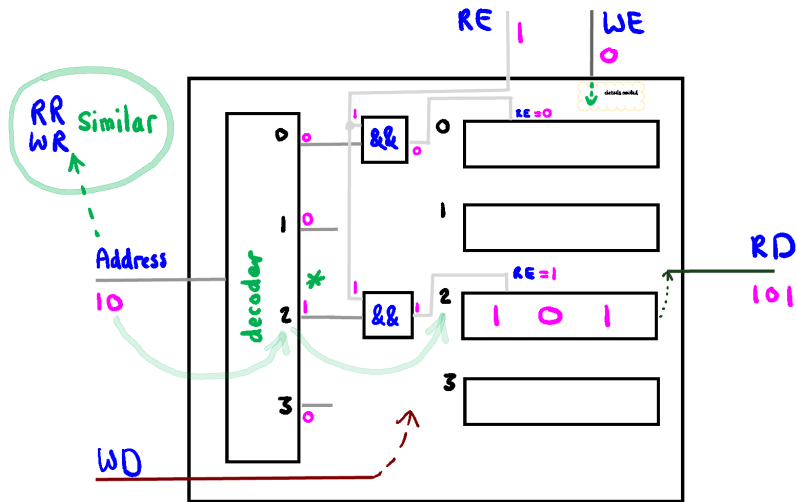


a. Data memory unit



b. Sign extension unit

## \*\* Memory Chip: Implementing Addressing in Hardware \*\*



- decoder takes Address and makes one output \*, others 0.  $10_b = 2_d$

# Processor Software (for MIPS/ARM)

---

- ▶ We describe in detail a particular set of instructions executed by a particular processor.
- ▶ Study assembly language and machine language (code) syntax of each instruction, and also the semantics of each instruction.
- ▶ We give a method for translating assembly language into machine language.



## Overview: Introduction to MIPS and ARM

We explain what an ISA is. We give an abstract view of MIPS and ARM processors, and the kind of architecture they implement (load/store). We introduce the MIPS ISA through an example program.

After studying this section you should be able to

- ▶ Give the definition of an ISA.
- ▶ Give examples to illustrate a load/store architecture, and recognize some simple MIPS instructions.
- ▶ Explain why MIPS instructions are stored at word locations with addresses which are multiples of 4.
- ▶ Be able to give basic explanations of how simple MIPS programs execute.

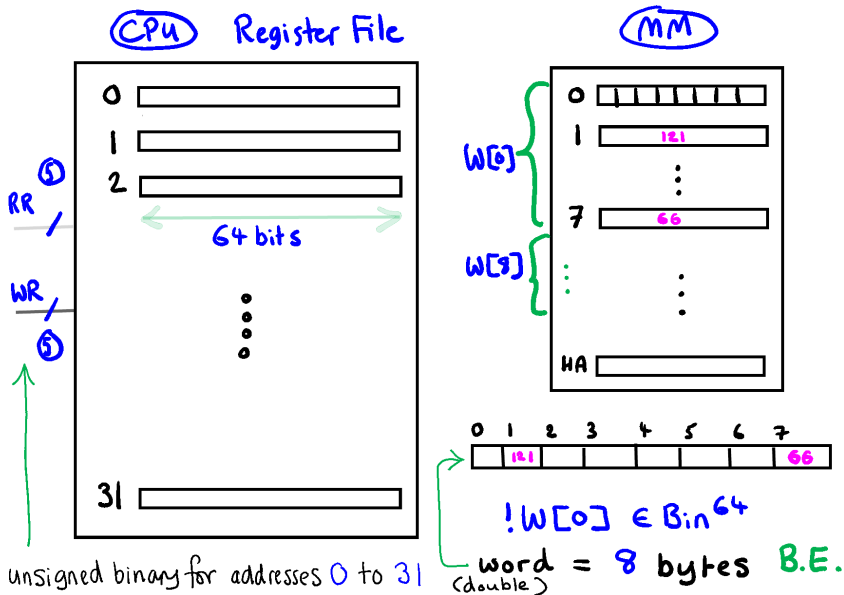
- ▶ This is an **Instruction Set Architecture**.
- ▶ One has a specified *set* of *instructions*, and a specified *processor*.
- ▶ Each instruction is very simple and has a *binary* form that will execute on the processor: a **machine** instruction written in **machine language**.
- ▶ Each instruction also has an **assembly** form such as

add  $R_1, R_2, R_3$

written in **assembly language**

- ▶ There is a very close *correspondence* with the processor hardware: **add** is executed on the ALU and each  $R_i$  is one of the registers in the processor. Each assembly *instruction* is “mirrored” in the *architecture* of the processor hardware.

# ARM Processor: The Register File



- ▶ Memory cells have size **8**; they are byte locations.
- ▶ Main memory and processor are both **64**-bit:
  - ▶ Main memory words are **64**-bit, composed of **8** bytes.
  - ▶ All registers have **64** bits.
  - ▶ Machine instructions are stored in **64**-bit word locations (and fetched to a **64**-bit instruction register, IR).
- ▶ Each register has an **assembly language symbol**. Eg **X4**.
- ▶ The Register File has **32** registers.
- ▶ Each register has an **address (number)**, ranging from **0<sub>d</sub>** to **31<sub>d</sub>**.
- ▶ The 5-digit unsigned binary representations of **0<sub>d</sub>** to **31<sub>d</sub>** are the machine language forms of the 32 register addresses: **00000**  
**11111**.

- ▶ Memory cells have size **8**; they are byte locations.
- ▶ Main memory and processor are both **32**-bit. This means:
- ▶ Main memory words are **32**-bit, composed of **4** cells.
- ▶ Machine instructions are stored in **32**-bit word locations.
- ▶ **THIS IS IMPORTANT!** If the first instruction of a program is stored in main memory word location **0**, the following instructions are stored at word locations **4**, **8**, **12**, and so on.
- ▶ The word locations use the **Big Endian** system. More on this later on.

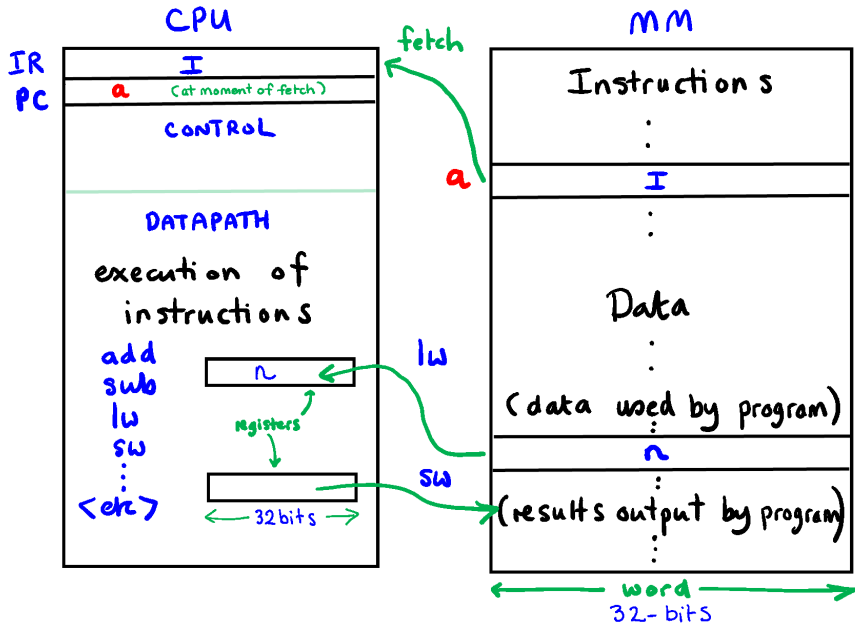
- ▶ All registers have **32** bits; instructions are fetched to a **32**-bit instruction register, IR.
- ▶ Each register has an **assembly language symbol**. Eg **\$s4**.
- ▶ The Register File has **32** registers.
- ▶ Each register has an **address (number)**, ranging from **0<sub>d</sub>** to **31<sub>d</sub>**.
- ▶ The 5-digit unsigned binary representations of **0<sub>d</sub>** to **31<sub>d</sub>** are the machine language forms of the 32 register addresses: **00000**  
**11111**.
- ▶ The addresses flow on the **Read register (RR)** and **Write register (WR)** busses (recall Slide 122).

What does a picture of the MIPS Processor Register File look like?

- ▶ Data copied into a register from main memory is **loaded**.
- ▶ Data copied into main memory from a register is **stored**.
- ▶ These forms of copying are **data transfers**.
- ▶ In some ISAs one may add a number stored in a register to a number found in main memory, storing the result in another register; but not so in MIPS and ARM.

Suppose some numbers are stored in main memory. We wish to sum them, then store the answer in main memory. So . . . each number must be *loaded* into a register; the answer is computed by the ALU and written to a register, and then the final sum can be *stored* back into main memory.

# MIPS and ARM: Load-Store Architectures, and Program Structure





- ▶ An *assembly program* is a sequence of **instructions**  $I$  or **labelled instructions**  $L : I$ .
- ▶ The  $PC$  *points to* the next instruction  $I$  to be executed; the  $PC$  is updated during execution of  $I$ .
- ▶ Most instructions are executed in memory word address order.
  - ▶ If currently  $PC = a$  then the  $PC$  is **updated** to contain the address of the instruction stored at the next physical word location in main memory:  $PC := a + 4$ .
- ▶ If the current instruction is a *branch*; such instructions contain a *proposition (test)* that computes to  $T$  or  $F$ .
  - ▶  $T$ : next instruction for execution is indicated by a **label**  $L$ .  $PC$  is **updated** to the address of the labelled instruction.  $PC := a'$ .
  - ▶  $F$ : next instruction for execution is at the next word location.

$$PC = a + 8$$

*main memory*   *Machine code*   *Assembly code*

$a$	Mins1	begin :	Ains1
$a + 4$	Mins2		Ains2
$a + 8$	Mins3		Ains3
$a' \stackrel{\text{def}}{=} a + 12$	Mins4	L :	Ains4
$a + 16$	Mins5		Ains5
$a + 20$	Mins6		BRAins6
$a + 24$	Mins7		exit :

*addresses*   *words*

*arrow points to current instruction*

$$PC = a + 12$$

*main memory*   *Machine code*   *Assembly code*

$a$	Mins1	begin :	Ains1
$a + 4$	Mins2		Ains2
$a + 8$	Mins3		Ains3
$a' \stackrel{\text{def}}{=} a + 12$	Mins4	L :	Ains4
$a + 16$	Mins5		Ains5
$a + 20$	Mins6		BRAins6
$a + 24$	Mins7		exit :

*addresses*   *words*

*arrow points to current instruction*

$$PC = a + 16$$

*main memory*   *Machine code*   *Assembly code*

$a$	Mins1	begin :	Ains1
$a + 4$	Mins2		Ains2
$a + 8$	Mins3		Ains3
$a' \stackrel{\text{def}}{=} a + 12$	Mins4	L :	Ains4
$a + 16$	Mins5		Ains5
$a + 20$	Mins6		BRAins6
$a + 24$	Mins7		exit :

*addresses*   *words*

*arrow points to current instruction*

$$PC = a + 20$$

*main memory*   *Machine code*   *Assembly code*

$a$	Mins1	begin :	Ains1
$a + 4$	Mins2		Ains2
$a + 8$	Mins3		Ains3
$a' \stackrel{\text{def}}{=} a + 12$	Mins4	L :	Ains4
$a + 16$	Mins5		Ains5
$a + 20$	Mins6		BRAins6
$a + 24$	Mins7		exit :

*addresses*   *words*

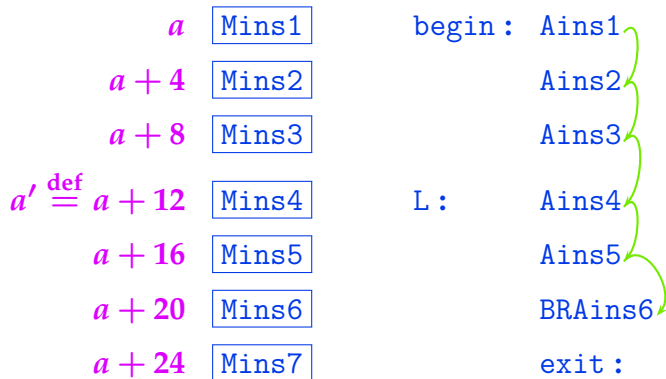
*arrow points to current instruction*

PC= $a'$



BRanchtest T

*main memory* Machine code    Assembly code

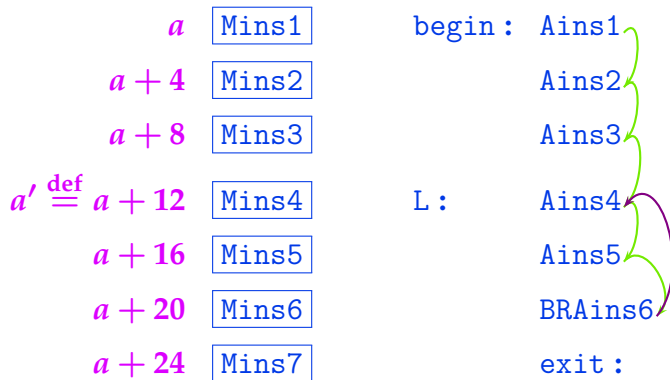


*addresses    words*

*arrow points to current instruction*

$$PC = a + 16$$

*main memory*   *Machine code*   *Assembly code*

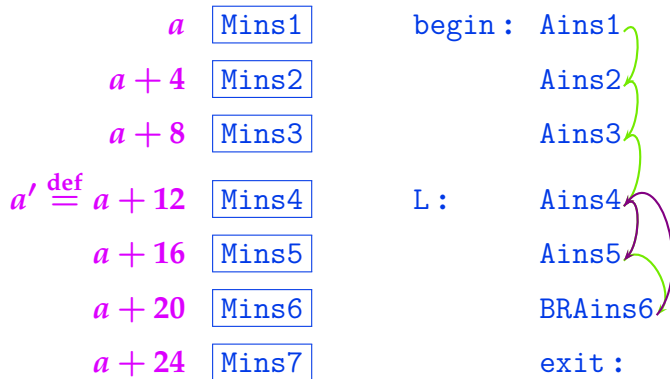


*addresses*   *words*

*arrow points to current instruction*

$$PC = a + 20$$

*main memory*   *Machine code*   *Assembly code*



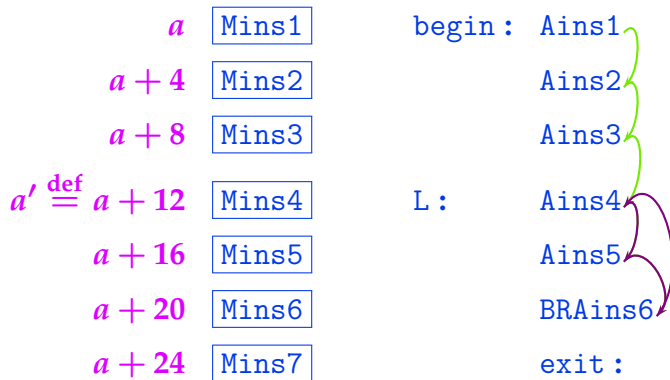
*addresses*   *words*

*arrow points to current instruction*

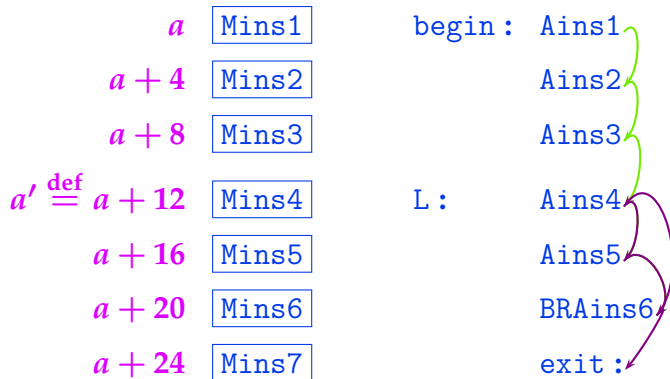


PC= $a + 24$ 

BRanchtest F

*main memory* Machine code Assembly code*addresses*    *words**arrow points to current instruction*

*main memory*   *Machine code*   *Assembly code*



*addresses*   *words*

*arrow points to current instruction*

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw  $s1, 64($zero)
exit:
```

\$s1	0	\$a0	0	\$t0	8
------	---	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	0	\$a0	0	\$t0	8
------	---	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	0	\$a0	1	\$t0	8
------	---	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw  $s1, 64($zero)
exit:
```

\$s1	0	\$a0	1	\$t0	8
------	---	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	1	\$a0	1	\$t0	8
------	---	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	1	\$a0	2	\$t0	8
------	---	------	---	------	---



## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	1	\$a0	2	\$t0	8
------	---	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1      3    \$a0      2    \$t0      8

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	3	\$a0	3	\$t0	8
------	---	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	3	\$a0	3	\$t0	8
------	---	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	6	\$a0	3	\$t0	8
------	---	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	6	\$a0	4	\$t0	8
------	---	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw  $s1, 64($zero)
exit:
```

\$s1	6	\$a0	4	\$t0	8
------	---	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	10	\$a0	4	\$t0	8
------	----	------	---	------	---



## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	10	\$a0	5	\$t0	8
------	----	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	10	\$a0	5	\$t0	8
------	----	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	15	\$a0	5	\$t0	8
------	----	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	15	\$a0	6	\$t0	8
------	----	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	15	\$a0	6	\$t0	8
------	----	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	21	\$a0	6	\$t0	8
------	----	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	21	\$a0	7	\$t0	8
------	----	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	21	\$a0	7	\$t0	8
------	----	------	---	------	---



## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	28	\$a0	7	\$t0	8
------	----	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero)
exit:
```

\$s1	28	\$a0	8	\$t0	8
------	----	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero) η
exit:
```

\$s1	28	\$a0	8	\$t0	8
------	----	------	---	------	---

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero) η
exit:
```

\$s1	28	\$a0	8	\$t0	8	MM: 64	<input type="text"/>
------	----	------	---	------	---	--------	----------------------

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw $s1, 64($zero) η
exit:
```

\$s1    28   \$a0    8   \$t0    8       | MM: 64 + 0

## An Example MIPS Program

Let's write a program to sum the first 8 numbers, from 0 to 7, and store the sum at main memory address 64: \$s1 stores the sum; \$a0 is a counter from 0 to 7 and \$t0 contains 8. Once executed !\$s1 should equal 28, and this should be stored into main memory word location 64.

```
begin:  addi $a0, $zero, 0
        addi $s1, $zero, 0
        addi $t0, $zero, 8
repeat: add $s1, $s1, $a0
        addi $a0, $a0, 1
        bne $a0, $t0, repeat
        sw  $s1, 64($zero)
exit:
```

\$s1	28	\$a0	8	\$t0	8	MM: 64	28
------	----	------	---	------	---	--------	----

We wish to write down the syntax and semantics of *all* core MIPS and ARM instructions. To do so we need some more notation and to learn about *addressing modes*.

After studying this section you should be able to

- ▶ Give examples of **assignment notation**, and make use of it in expressing semantics.
- ▶ Make use of *byte* and *word* **address notation**.
- ▶ Give examples of instruction **names**, **arguments**, **source data** and **destination data**.
- ▶ Define clearly and give examples of three different ways to locate data in MIPS and ARM: these are called **addressing modes**.

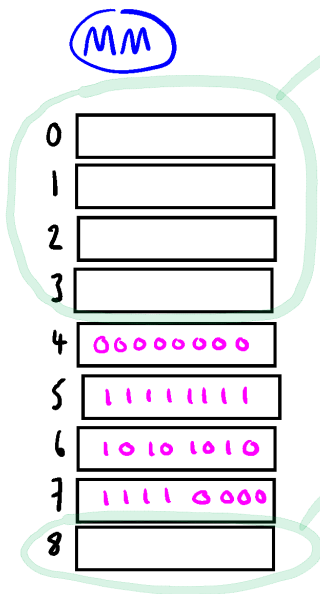
- ▶  $R$  and  $R_i$  each denote any of the MIPS registers. We may also use other capital letters for registers such as  $A, B, S, T \dots$
- ▶ An **assignment** takes the form  $R := \omega$  (eg  $\$t1 := \vec{0111}$ ) where  $\omega$  is a 32-bit binary word (ie  $\omega \in \text{Bin}^{32}$ ). We may write  $\omega$  in denary:  $R := 7$ .
- ▶  $R := 7$  means “register  $R$  is up-dated with (new) content 7.”
- ▶ We can then describe the semantics of the instruction `add  $R_1, R_2, R_3$`  as

$$R_1 := !R_2 + !R_3$$

where  $+$  is computed by a 32-bit ALU and  $!R_i \in \text{Bin}^{32}$ .



- ▶ We define an **address notation** for bytes and words.
- ▶  $B[a]$  is the byte location (MIPS cell) at main memory address  $a$ .
- ▶ We have a similar notation  $W[a]$  for word locations.
- ▶ Recall word locations are composed of 4 byte locations (cells), so their addresses are multiples of 4. (IMPORTANT!!)
- ▶ Program instructions are stored in consecutive word locations in main memory, for example 0, 4, 8, 12 ...
- ▶ MIPS is big endian format.



$W[0]$

Where is  $W[4]$ ?

The address is 4.

Four consecutive cells,  
 $a, a+1, a+2, a+3$ .

!  $\circ \circ a = 4$ . Cells have  
 addresses 4, 5, 6, 7.

BIG ENDIAN

$B[8]$

$!W[4] \in \text{Bin}^{32}$

||  
 ?

MM

0	10000000
1	✓
2	✓✓
3	✓✓✓
4	00000000
5	11111111
6	10101010
7	11110000
8	

BIG ENDIAN

!W[4] ∈ Bin<sup>32</sup>  
||

00000000.11111111.10101010.11110000

Copy  $2^3 + 2^5 + 1$  into W[0]:

10000000.00000000.10000000.00000001

0      1      2      3  
         ✓      ✓✓      ✓✓✓

addresses get bigger →

- ▶ *Addressing* refers to different ways in which data is *located*. If you know my address you can locate me.
- ▶ We often say that an address *points* to a location.
- ▶ Consider the instruction `add R1, R2, R3`
- ▶ It has one **name** (`add`) and three **arguments** (the registers).
- ▶ In the case of `add` the arguments tell us the *location* of the operands and of the result.
- ▶ In general the arguments of all instructions provide the locations for the data an instruction processes (**source** data) and the location of the result (**destination** data).

- ▶ In `addi $t0, $t1, 24` the number 24 is located by **immediate addressing**.
- ▶ 24 is “immediately available”: no need to “locate” the data!
- ▶ The semantics of the instruction is  $\$t0 := !\$t1 + 24$ .
- ▶ 24 is stored in binary; `+` means 32-bit ALU sum.
- ▶ Recall Slide 93. The instruction name, and each register, had a machine language form making up *part* of the 32-bit machine instruction.
- ▶ Constants like 24 are *stored as a part of the machine instruction; more on this later on.* LEC

Think about the 32-bit binary machine code. What length is the *part* of the code that's used for 24? Can you say anything about the length?

## Immediate Addressing

- ▶ In `addi $t0, $t1, 24` the number 24 is located by **immediate addressing**.
- ▶ 24 is “immediately available”: no need to “locate” the data!
- ▶ The semantics of the instruction is  $\$t0 := !\$t1 + 24$ .
- ▶ 24 is stored in binary; `+` means 32-bit ALU sum.
- ▶ Recall Slide 93. The instruction name, and each register, had a machine language form making up *part* of the 32-bit machine instruction.
- ▶ Constants like 24 are *stored as a part of the machine instruction; more on this later on.* LEC

*Must be  $\leq 32 - (5 + 5 + n)$  where  $n$  is the bit length of the machine instruction name and we need 5 bits each for the two register addresses.*

- ▶ In `add $t0, $t1, $t2` the operands (source data) are each located by **register addressing**.
- ▶ If the register is called *R*, the data is *!R*. Eg if

R 11001100

then the data is 11001100.

- ▶ The semantics of the instruction is

$\$t0 := !\$t1 + !\$t2$

## Example: Register Addressing

$\$t_0 := !\$t_1 + !\$t_2$

Let  $!\$t_1 = 011$   $!\$t_2 = 001$ .

Illustrate the execution of `add $t0 $t1, $t2`

$\$t_1$  011  
 $\$t_2$  001

$\$t_0$  100

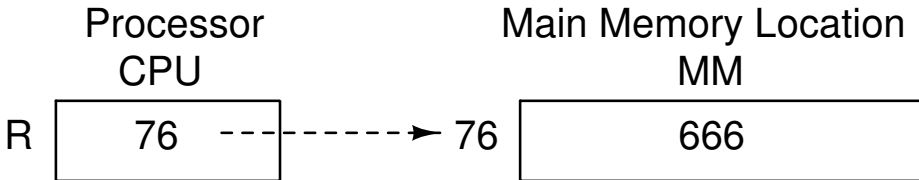
$$\begin{array}{r} 011 \\ + 001 \\ \hline 100 \end{array}$$

$!\$t_1 + !\$t_2$

$$\begin{array}{r} 101 \\ + 001 \\ \hline 110 \end{array}$$



- ▶ The data is *located in a main memory location*.
- ▶ A register  $R$  provides the *address*  $a$  of the location.
- ▶ The data is  $!W[a]$  or  $!B[a]$  where  $a \stackrel{\text{def}}{=} !R$ .

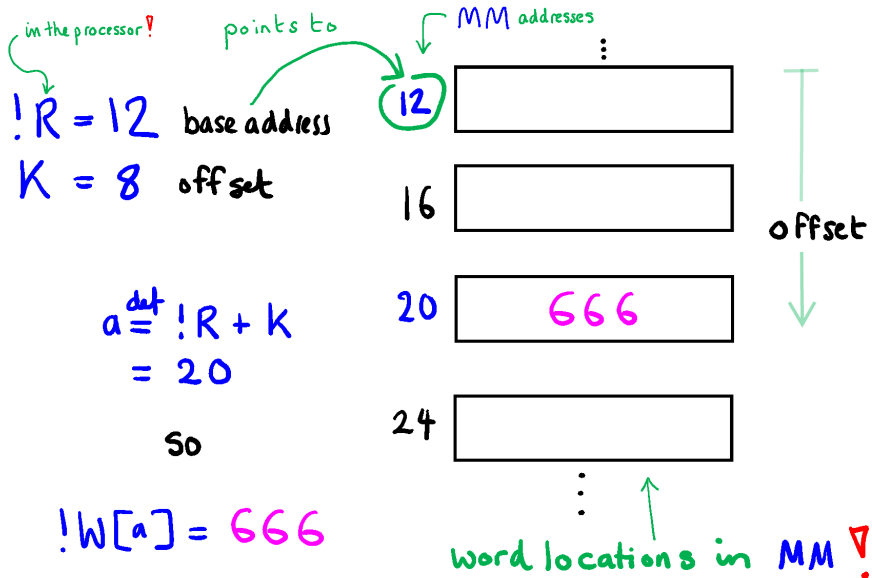


- ▶ In this case, we refer to  $R$  as a **pointer**. In the example  $a = 76$  and the data is **666**.
- ▶  $!W[76] = 666$  (if we are dealing with word locations; more on this later on).

Motivation: Locating a book on Computing in the library. Go to the Computing section (*base address*). Your books are nearby. Move a short distance, an *offset*, to find first book. Move another short distance to find second book. And so on ...

- ▶ We often need to load a sequence of words (eg numbers) from main memory, all “quite near” to a **base address**  $!R$ .
- ▶ Each word is offset from the base address by  $K$ .
- ▶ In  $1_w R', K(R)$  the source argument  $K(R)$  employs **indexed addressing**.
- ▶ The word to be loaded is at address  $a \stackrel{\text{def}}{=} !R + K$ ; rephrasing, the word is  $!W[a]$ .

# Indexed Addressing



We will learn assembly language (syntax) in detail for a set of core instructions. We look at execution semantics in depth.

After studying this section you should be able to

- ▶ Explain that the **core instructions** can be grouped into *categories* such as Arithmetic, Data Transfer and so on; a **category** is a set of instructions with similar execution semantics.
- ▶ Describe core assembly instructions giving both **SYNTAX** and **SEMANTICS** (detailed semantics need not be memorised).
- ▶ Given specific instructions, and either informal or formal semantics, **illustrate execution** behaviour by *calculating with data and by drawing pictures of main memory and processor registers*.

MIPS	ARM
add $R_1, R_2, R_3$	ADD $R_1, R_2, R_3$
sub $R_1, R_2, R_3$	SUB $R_1, R_2, R_3$
addi $R_1, R_2, K$	ADDI $R_1, R_2, K$
subi $R_1, R_2, K$	SUBI $R_1, R_2, K$
and $R_1, R_2, R_3$	AND $R_1, R_2, R_3$
or $R_1, R_2, R_3$	OR $R_1, R_2, R_3$
andi $R_1, R_2, K$	ANDI $R_1, R_2, K$
ori $R_1, R_2, K$	ORI $R_1, R_2, K$
lw $R_1, K(R_2)$	LDUR $R_1, [R_2, K]$
sw $R_1, K(R_2)$	STUR $R_1, [R_2, K]$
beq $R_1, R_2, L$	CBZ $R, L$
bne $R_1, R_2, L$	CBNZ $R, L$

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 $R_1, R_2, K_d$	$R_1 := !R_2 + sx(\vec{b})$ where $\vec{b}_s = K_d$
subi $R_1, R_2, K_d$	$R_1 := !R_2 + sx(\vec{b})$ where $\vec{b}_s = K_d$

**SYNTAX:** Each machine instruction is stored as a **32**-bit binary number. The denary number  $K_d$  must be stored as a *part* of those **32** digits, in fact the **rightmost 16** bits. These bits contain the signed number  $\vec{b}$ .

Note that **by definition** (and referring to slide 38)

$$2^{16-1} = -32768 \leq K_d \leq 2^{16-1} - 1 = +32767$$

**SEMANTICS:** **16** bit  $\vec{b}$  has to be *sign extended* to  $sx(\vec{b})$  before being input to the **32**-bit ALU (at execution time).

Let  $R_2 = 9_d$  and  $K = 5_d$ . Illustrate `addi`

$$9_d = 01001_s \quad \text{so} \quad R_2 = \vec{0}1001_s \in \text{Bin}^{32}$$

$$5_d = 0101_s$$

machine code for  $K$ : 16 digits:  $\vec{0}101_s \in \text{Bin}^{16}$

$$\text{sx}(\vec{0}101_s) = \vec{0}.\vec{0}101_s \in \text{Bin}^{32}$$

$$\begin{aligned} R_2 + \text{sx}(\vec{0}101_s) &= \vec{0}1001 + \vec{0}101 \\ &= \vec{0}1110 \in \text{Bin}^{32} \end{aligned}$$

sign  
extensions



## \*\* Logical Category Instructions \*\*

These instructions are similar to the `add` instruction. The logical operations are the *digitwise* `&` and `|`; see Slide 61 and Slide 216.

Syntax	Semantics
<code>and R<sub>1</sub>, R<sub>2</sub>, R<sub>3</sub></code>	$R_1 := !R_2 \ \& \ !R_3$
<code>or R<sub>1</sub>, R<sub>2</sub>, R<sub>3</sub></code>	$R_1 := !R_2 \   \ !R_3$
<code>andi R<sub>1</sub>, R<sub>2</sub>, K<sub>d</sub></code>	$R_1 := !R_2 \ \& \ zx(\vec{b}) \quad \text{where } \vec{b}_b = K_d$
<code>ori R<sub>1</sub>, R<sub>2</sub>, K<sub>d</sub></code>	$R_1 := !R_2 \   \ zx(\vec{b}) \quad \text{where } \vec{b}_b = K_d$

Note that by definition of MIPS (refer to slide 31)

$$0 \leq K_d \leq 2^{16} - 1 = +65535$$



LEC

The register `$t0` stores an RGB color with R and G and B in the lower three bytes. A graphics programmer wishes to update another register `$t1` so that it has the same B value as `$t0` along with G set to the maximum green value of `11111111` and R to minimum `00000000`.

Write some MIPS code to do this, making use of `andi` and `ori` instructions. *Hint: If  $x$  is 0 or 1, then*

- ▶  $x \&\& 1 = x$  and  $x \&\& 0 = 0$  (keep  $x$  or set to 0), and
- ▶  $x || 0 = x$  and  $x || 1 = 1$  (keep  $x$  or set to 1).

## \*\* Logical Category Instructions \*\*

We can use `&&` to keep the B bits, and set the R and G bits to  $\vec{0}$ . After this, we can use `||` to (again) keep the B bits, keep the R bits (which now equal  $\vec{0}$ ), and set the G bits to  $\vec{1}$ . Writing out the 32-bit words we have

!\$t0	=	00000000	rrrrrrrr	gggggggg	bbbbbbbb
&		00000000	00000000	00000000	11111111
<hr/>					
!\$t1	=	00000000	00000000	00000000	bbbbbbbb
		00000000	00000000	11111111	00000000
<hr/>					
		00000000	00000000	11111111	bbbbbbbb

Now the second row is  $zx(00000000.11111111)_b$  and  $00000000.11111111_b = 2^8 - 1_d = 255_d$ . And the fourth row is  $zx(11111111.00000000)_b$  and  $11111111.00000000_b = 255_d * 2^8_d = 65280_d$ .

```
andi $t1, $t0, 255d
ori  $t1, $t1, 65280d
```

Syntax	Semantics
$\text{lw } R_1, K_d(R_2)$	$R_1 := !W[!R_2 + sx(\vec{b})]$ where $\vec{b}_s = K_d$
$\text{sw } R_1, K_d(R_2)$	$W[!R_2 + sx(\vec{b})] := !R_1$ where $\vec{b}_s = K_d$

Note that the offset  $K_d$  might be negative. LEC

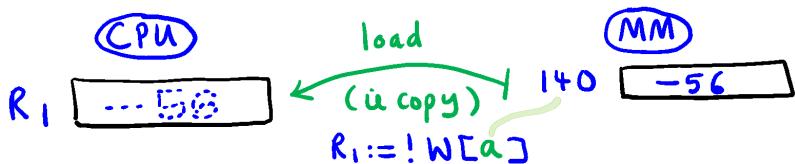
**SYNTAX:** Each machine instruction is stored as a 32-bit binary number. The denary number  $K_d$  must be stored as a *part* of those 32 digits, in fact the **rightmost 16** bits. These bits contain the signed number  $\vec{b}$ .

**SEMANTICS:** 16 bit  $\vec{b}$  has to be *sign extended* to  $sx(\vec{b})$  before being input to the 32-bit ALU (at execution time).

## Illustration of Execution: Data Transfer Instructions

Illustrate execution of  $lw R_1, K(R_2)$  when  $K_d = 12$ ,  $!R_2 = 128$ ,  $!W[140] = -56$  - using denary and binary.

*den* Suppose that  $K_d = \vec{b}_s$ . Then since  $sx(\vec{b})_s = K_d$  the calculation in denary of  $!R_2 + sx(\vec{b})_s$  is  $128 + 12 = 140$ . Hence:

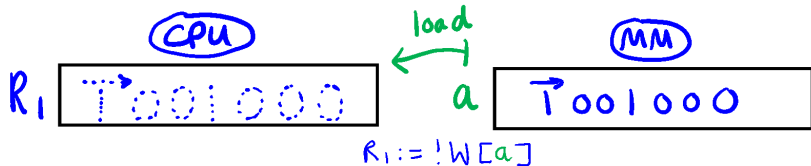


# Illustration of Execution: Data Transfer Instructions

(binary)  $K_d = 12_d = 8 + 4_d$  so  $12_d = \vec{0}1100_s \in \text{Bin}^{16}$   
 $!R_2 = \vec{0}10000000_s \in \text{Bin}^{32}$   
 $*sx(\vec{0}1100) = \vec{0}.\vec{0}1100 \in \text{Bin}^{32}$   
 $\therefore !R_2 + sx(K) = \vec{0}.\vec{0}10001100 \in \text{Bin}^{32}$   
 $\underbrace{\hspace{10em}}_a$

Also  $-56 = -64 + x \Rightarrow x = 8$ . So

why?  $-56_d = \vec{1}001000_s$  and hence  
 $= \vec{1}001000_s \in \text{Bin}^{32}$



```
L: ins1  
    ins2  
    ins3  
    ins4  
    bne $t0, $t2, L  
    ninsm
```

Syntax	Semantics
<code>beq <math>R_1</math>, <math>R_2</math>, L</code>	<b>if <math>!R_1 = !R_2</math> then goto L else goto ninsm</b>
<code>bne <math>R_1</math>, <math>R_2</math>, L</code>	<b>if <math>!R_1 \neq !R_2</math> then goto L else goto ninsm</b>

See Slide 135 onwards. If a branch test is true, then the PC is updated to point to `ins1`; if false to `ninsm`.

## Example MIPS Program

A Professor has *ns* students, each of whom has *3* marks. The marks are stored, one mark per word location, in main memory starting at word location *ma*. The Professor wants a program to produce the total for each student, placing the totals in main memory starting at address *ta*. Write a MIPS program to do this.

In practice the Professor will write a high level program (Java, Python, whatever ...) to read in the marks, produce totals, and output/store the totals. Compilation/interpretation will produce a machine language MIPS program, and after running it, a final memory state rather like that described in the question.

STEP 1 Understand the problem; draw pictures.

STEP 2 Write a sketch program solution.

STEP 3 Write a skeleton program.

STEP 4 Produce MIPS code.

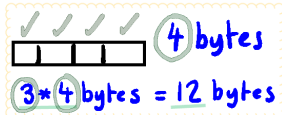
STEP 5 Test your solution.



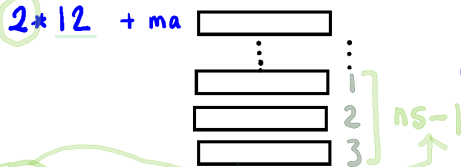
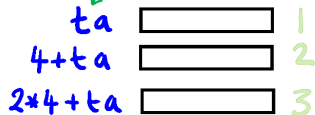
# Example MIPS Program

## ① UNDERSTAND PROBLEM

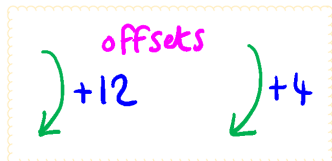
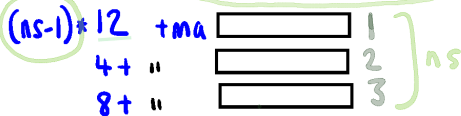
*cma will range over* *c will range over*



*cta will range over*



$(ns-1) * 4 + ma$  ns

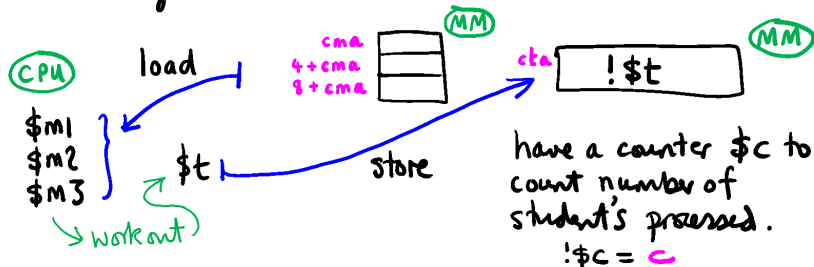


## ② SKETCH PROGRAM SOLUTION

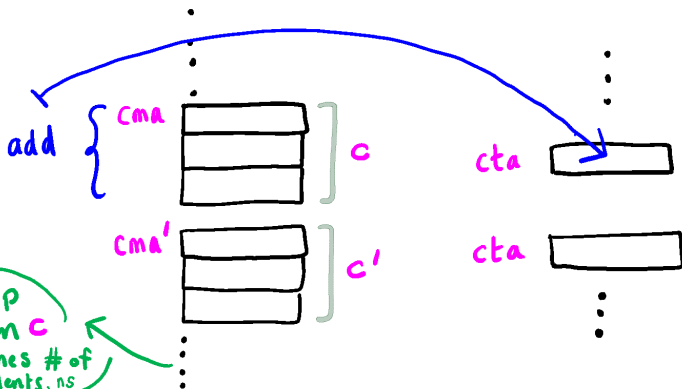


repeat for all students

Hang on! We must load the marks!



## Example MIPS Program



STOP  
When  $c$   
reaches # of  
students,  $ns$

To move from one student to next, we need to compute  $cma'$  and  $cta'$  from  $cma$  and  $cta$ . But we already did the work for this!  $c'$  is easy!

$$cma' = cma + 12 ; cta' = cta + 4 ; c' = c + 1$$

← offsets →

12 = 3 words  $\times$  4 bytes

## ③ SKELETON PROGRAM

add INITIALISE  $\$c$   
 addi  $\$t$   
 $\$cma$   
 $\$cta$

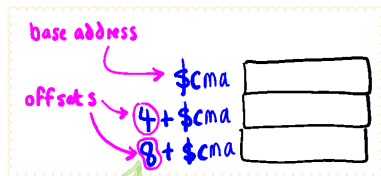
lw [ load marks into  $\$m1, \$m2, \$m3$   
 eg lw  $\$m3$  8( $\$cma$ )

add [ add up marks eg add  $\$t$   $\$t$   $\$m1$

store [ sw  $\$t$  0( $\$cta$ )

addi [  $\$cma = \$cma + 12$   
 $\$cta = \$cta + 4$   
 $\$c = \$c + 1$

bne [ branch to if  $\$c \neq \$ns$



// STEP 4

// setting up register values

```
addi $ns, $zero,  $ns_d$     // total number of students
addi $cma, $zero,  $ma_d$     // initial current mark address
addi $cta, $zero,  $ta_d$     // initial current total address
addi $c, $zero, 0          // initial student counter
addi $t, $zero, 0          // initial mark total
:
```

## Example MIPS Program

```
⋮  
L: lw $m1, 0($cma) // load each current mark  
   lw $m2, 4($cma)  
   lw $m3, 8($cma)  
   add $t, $t, $m1 // total the three marks  
   add $t, $t, $m2  
   add $t, $t, $m3  
   sw $t, 0($cta) // store final total  
   addi $t, $zero, 0 // reset total to 0  
   addi $cma, $cma, 12 // next current mark address  
   addi $cta, $cta, 4 // next current total address  
   addi $c, $c, 1 // increase student counter  
   bne $c, $ns, L // repeat until all students counted
```

Undertake STEP 5: test the code by computing a table of register values.

We want to translate assembly instructions into machine language.

After studying this section you should be able to

- ▶ Read and understand problems that involve assembly instruction **formats**. Each format specifies the machine language version as a concatenation of binary numbers called **fields**.
- ▶ **Translate** any core MIPS assembly instruction into a **32-bit** machine language instruction, given a **format table**.
- ▶ **IMPORTANT**: Branch Instructions are treated as an Advanced Topic. See slide 194.

- ▶ Each **32**-bit machine instruction belongs to one of three **formats**. These are known as **R**, **I** and **J** formats (we do not cover **J**).
- ▶ Each format has a **field layout table**: This is a table specifying how the **32** bits are divided up into parts known as **fields**. The table gives each field a position, and length in bits.
  - ▶ Recall Slide 93:  
The instruction has four fields, of lengths **4**, **3**, **3**, **3**.
- ▶ The field layout describes how to immediately work out individual fields (though branch labels require additional work).



*name*  $R_1$ ,  $R_2$ ,  $R_3$  such as `add $t0, $s1, $s2`, and `sub, and, ...`

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
$\vec{0}$	$\#R_2$	$\#R_3$	$\#R_1$	$\vec{0}$	function field for <i>name</i>
$\vec{0}$	$\#\$s1$	$\#\$s2$	$\#\$t0$	$\vec{0}$	function field for <code>add</code>
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

- ▶ The opcode field (op) is always  $0_d$ . It determines the basic behaviour of the instruction: use the ALU.
- ▶ The function field (funct) determines the specific ALU operation. This can be looked up in the MIPS ISA documentation. LEC
- ▶ The shamt field is  $0_d$  in CO1104: we don't study shift amounts.

## I-Format Table

*name*  $R_1, R_2, K_d$  (see slides page 152 to 160 for  $\vec{b}$ )

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits
opcode for $\text{addi}, \text{subi} / \text{andi}, \text{ori}$	$\#R_2$	$\#R_1$	$\vec{b}$

*name*  $R_1, K_d(R_2)$

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits
opcode for $\text{lw}, \text{sw}$	$\#R_2$	$\#R_1$	$\vec{b}$

*name*  $R_1, R_2, L$  (see slide 176 for definition of  $\vec{b}$ )

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits
opcode for $\text{beq}, \text{bne}$	$\#R_1$	$\#R_2$	$\vec{b}$

- ▶ Look up the *format*. This identifies the *field layout* table.
- ▶ Using the instruction name, look up the denary *opcode*; and for R-format the denary *function* field too. Convert to 6-bit unsigned binary.
- ▶ Using the register symbols, look up each denary register address from a table; the corresponding 5-bit unsigned binary is the required field.
- ▶ Translate any constants into *signed/unsigned binary* with 16-digits to get the *address* field.
- ▶ For branch instructions translate the label into a 16-digit *address* field; see slide 176.

**NOTE:** The “address” field is not a terribly useful name . . .

## Register Address (Number) Table

---

Register		Usage
\$zero	0	contents always zero
\$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-\$t7	8-15	temporary (not preserved across call)
\$s0-\$s7	16-23	saved temporary (preserved across call)

## Example: Machine Language for `lw`

---

What is the machine code for `lw $s1, 8($t1)` ?

## Example: Machine Language for `lw`

What is the machine code for `lw $s1, 8($t1)` ?

- ▶ *name*  $R_1$ ,  $K(R_2)$  is I-format.
- ▶ Use look-up table for `lw` opcode: 35.
- ▶ Use register address table to find  $\#\$s1 = 17$  and for  $\#\$t1 = 9$ .

6 bits	5 bits	5 bits	16 bits
opcode for <code>lw</code>	$\#R_2$	$\#R_1$	$K_d$
$35_d$	$9_d$	$17_d$	$8_d$
$100011_b$	$01001_b$	$10001_b$	$00000000000001000_s$

► See Slide 194.

# A MIPS/ARM Processor

---

We will

- ▶ Build a simple datapath in which each ISA instruction is executed in one clock cycle.
- ▶ Study the Control Unit: Once per clock cycle the Control Unit sets all control signals (at decode stage) and the instruction can then execute.



We make a brief introduction. Then we look at the **datapath hardware** required for **PC updates**, executing **R-Format instructions**, and then executing **I-Format Load/Store Instructions**. We leave **I-Format Branch Instructions** to the final chapter.

After studying this section you should be able to

- ▶ Trace **dataflow** around any *given* **datapath** and **instruction** (see lectures, and videos). Note: you do not need to memorize the hardware diagrams; they will be *given* in tests and examinations.
- ▶ Solve simple questions about the execution of an instruction and the corresponding dataflow.

- ▶ The design/structure of the datapath is referred to as the **datapath architecture**.
- ▶ We develop a datapath architecture for
  - ▶ The R-format instructions `add`, `sub`, `and`, `or`,
  - ▶ the I-format instructions `lw`, `sw`, and `beq`.
- ▶ We do not consider `addi`, `subi`, `andi`, `ori`, or `bne` ... however you may like to think about these instructions later on.

To do this

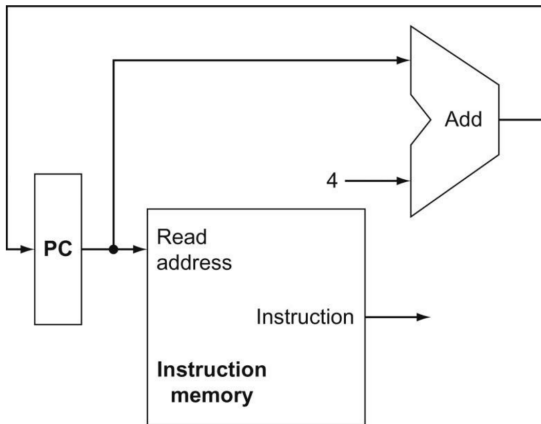
- ▶ We give hardware circuits for the execution of particular instructions . . .
- ▶ . . . and explain how these work by tracing the flow of data around the hardware as an instruction executes. LEC (See Slide 93.)
- ▶ We then combine these circuits to make a complete datapath.
- ▶ Finally we look at the control unit.

To do this

- ▶ We give hardware circuits for the execution of particular instructions . . .
- ▶ . . . and explain how these work by **tracing** the flow of data around the hardware as an instruction executes. LEC (See Slide 93.)
- ▶ We then combine these circuits to make a complete datapath.
- ▶ Finally we look at the control unit.

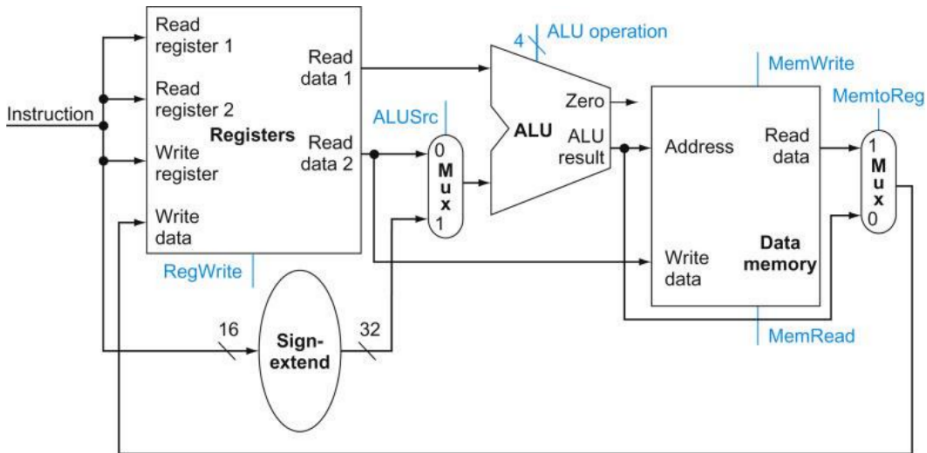
**NOTE:** the explanation of the dataflow is presented as follows. In the next few slides, you will find datapath hardware circuit diagrams, and each one is followed by a description of how data flows around the diagram. In the lectures I will annotate the diagrams with the dataflow, but you might like to try this for yourself before watching me do it.

## Datapath for Instruction Fetch and PC Update



- ▶  $a \stackrel{\text{def}}{=} !PC \in \text{Bin}^{32}$  is passed to Read address.
- ▶ The machine instruction  $I$  stored at address  $a$  is passed along the Instruction bus.
- ▶  $a$  is also input to the ALU and then the PC is up-dated to point to the instruction in the next word location  $PC := a + 4$ .

# Datapath for R-Format and Load/Store Instructions

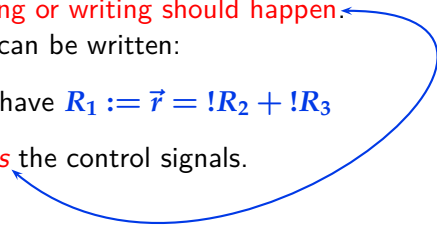


- These have the form *name*  $R_1, R_2, R_3$ .

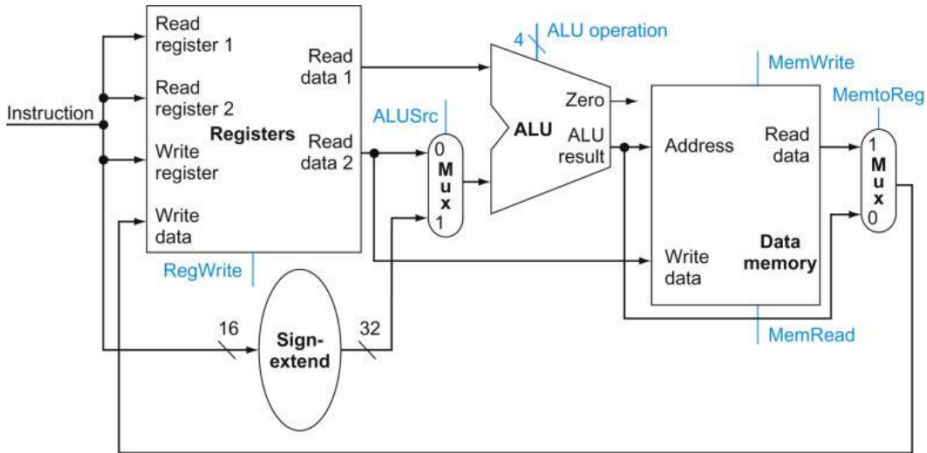
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
31-26	25-21	20-16	15-11	10-6	5-0
opcode = $\vec{0}$	$\#R_2$	$\#R_3$	$\#R_1$	$\vec{0}$	function field for <i>name</i>

- We examine `add  $R_1, R_2, R_3$`  with semantics  $R_1 := !R_2 + !R_3$ .
- Instruction `I[31 – 0]` passed along 32-bit wide `Instruction` bus.
- $\#R_2 = I[25 – 21]$  and  $\#R_3 = I[20 – 16]$  are passed into 5-bit `Read register 1` and `Read register 2`.
- $\#R_1 = I[15 – 11]$  is passed into `Write register`.
- In response the source operands  $!R_2$  and  $!R_3$  appear on the 32-bit `Read data 1` and `Read data 2`



- ▶ The *opcode* and *function field* are input to a Control Unit; this outputs control signals . . . LEC
  - ▶ . . . the signal on **ALUSrc** is 0 so that **!R<sub>3</sub>** is passed into the lower ALU input bus; and **!R<sub>2</sub>** flows into the upper ALU input bus.
  - ▶ Control selects **+** by sending a signal on **ALU Operation**; hence the 32-bit result  $\vec{r} \stackrel{\text{def}}{=} !R_2 + !R_3$  appears on the ALU output bus.
  - ▶ **MemtoReg** is set to 0 so that result  $\vec{r}$  is passed into **Write data**.
  - ▶ **MemWrite** and **MemRead** are set to 0 since no memory reading or writing should happen. **RegWrite** is set to 1 so the result can be written:
  - ▶ Since **Write register** = **#R<sub>1</sub>** we have  $R_1 := \vec{r} = !R_2 + !R_3$
  - ▶ Note that the semantics *determines* the control signals.
- 

# Datapath for R-Format and Load/Store Instructions



## Dataflow for Load/Store Instruction Execution

6 bits	5 bits	5 bits	16 bits
31-26	25-21	20-16	15-0
opcode for $\text{lw}$ , $\text{sw}$	$\#R_2$	$\#R_1$	$\vec{b}$

- ▶ We examine  $\text{lw } R_1, K_d(R_2)$ . Let  $\vec{b}_s = K_d$  where  $\vec{b} \in \text{Bin}^{16}$ .
- ▶ Let  $sx(\vec{b}) \in \text{Bin}^{32}$ .
- ▶ The semantics of  $\text{lw}$  is  $R_1 := !W[!R_2 + sx(\vec{b})]$ .
- ▶  $\#R_2 = I[25 - 21]$  is passed into **Read register1**
- ▶  $\#R_1 = I[20 - 16]$  is passed into **Write register**.
- ▶  $\vec{b} = I[15 - 0]$  is passed into **Sign extend**.
- ▶  $!R_2$  passed into **Read data1** and hence to ALU.

- ▶ Control sets `ALUSrc` to **1** so that  $sx(\vec{b})$  is also passed into the ALU.
- ▶ Control selects `+` using `ALU Operation`; hence  $\vec{a} \stackrel{\text{def}}{=} !R_2 + sx(\vec{b}) \in Bin^{32}$  is passed into Data memory Address.
- ▶ Control sets `MemRead` to **1** (and `MemWrite` to **0**), and so  $!W[\vec{a}]$  is passed into Read data. LEC
- ▶ `MemtoReg` is set to **1**; hence  $!W[\vec{a}]$  passed into register file Write data.
- ▶ `RegWrite` is set to **1**; and since  $\#R_1 = I[20 - 16]$  is in Write register we have  $R_1 := !W[\vec{a}]$ .
- ▶ Thus the execution has resulted in  $R_1 := !W[!R_2 + sx(\vec{b})]$  as required.

## \* Datapath for Branch if Equal Instruction \*

---

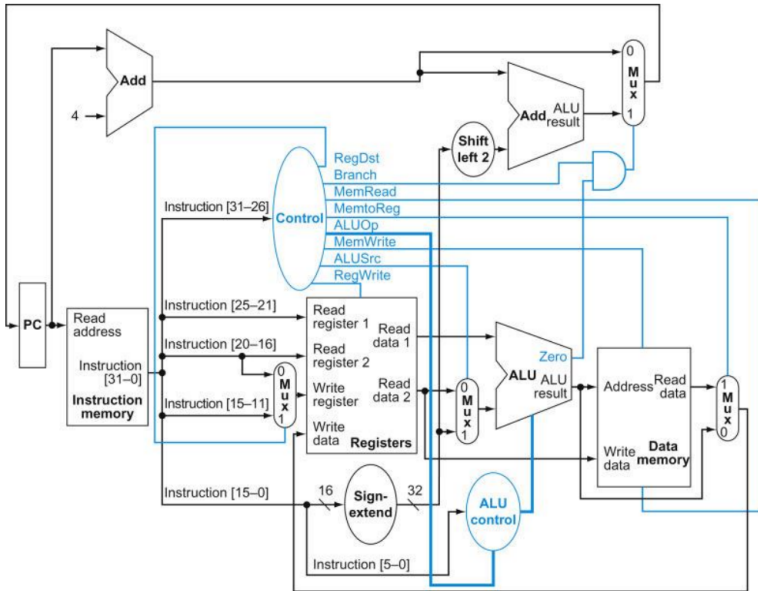
► See Slide 194.

- ▶ We assemble the datapath components into a single cycle datapath (each instruction takes one clock cycle) with Control:

### A MIPS Processor

- ▶ We briefly overview the Control Units.
- ▶ Note that branch instruction semantics are left as an Advanced Topic. Read those slides before thinking about their control.

# The MIPS Processor



## The Main Control Unit

The Control Unit outputs the control signal values we have already met. Note that **RegDst** is new and left as an exercise to understand.

							Signals to set multiplexors and read/write								
Inst	Opcode field = I[31-26]						RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp2
R-f't	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0
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

For branch instructions first see Slide 201 onwards. **Branch** is set to **1** for branch instructions and **0** otherwise: it is an exercise to work out the role of the AND gate in the Processor.



This is not for assessment: See Hennessy and Patterson for a discussion.

# Advanced Topics

---

We finish the module by studying the following topics:

- ▶ Arithmetic Correctness Examples and Conditions
- ▶ Branches: Machine Language and Processor Hardware

If there is time we will look briefly at:

- ▶ Predicate Logic and Bitwise Logic.
- ▶ Cache Memory (slides included)
- ▶ Pipelining Instructions (overview comments only)
- ▶ Multicore Processors (slides not included)
- ▶ GPUs (slides not included)

We explain in detail what it means for the DA to be correct for unsigned addition. We do the same for ALU signed addition. We look at examples, and provide formal tests for ALU signed addition to be correct.

After studying this section you should be able to

- ▶ Explain the general definitions of correctness for addition, and give examples. Work out what it means for other operations to be correct.
- ▶ Solve problems involving both DA (unsigned) and ALU (signed) correctness, including the detailed application of the Conditions for ALU Correctness.

## \* Digitwise Algorithm Correctness (Unsigned Binary) \*

- ▶ A computer user wants to add up  $5_d$  and  $7_d$ . They enter  $5_d + 7_d$  into the machine. The correct denary answer is  $12_d$ .
- ▶ Let's *assume* that the computer correctly converts the denary numbers to binary  $\vec{a} \stackrel{\text{def}}{=} 101$  and  $\vec{b} \stackrel{\text{def}}{=} 111$ .
- ▶ The DA calculates  $\vec{s} = 1100$  from  $\vec{a}$  and  $\vec{b}$ .
- ▶ The DA is **correct** **if**  $\vec{s}_b = 12_d$ .
- ▶ Let's also assume the computer converts  $\vec{s}$  to denary without error, and this result is output to the user. Let's check:  
 $\vec{s}_b = 1100_b = (8 + 4)_d = 12_d$ , correct!

In fact the DA *for unsigned binary* is *always correct*. Think about exactly what this means. Can you *prove* correctness? And what does “proving” correctness really mean?

Compute  $3 + 2$  using the binary DA. Is DA correct?

►  $3_d = 011_b$  and  $2_d = 010_b$

	0	1	1
+	0	1	0
0	1	0	1

► The correct denary answer is  $3_d + 2_d = 5_d$ .

► The DA binary result is  $0101_b$ . Converting to denary

$$0101_b = (4 + 1)_d = 5_d$$

and so the DA is correct.

Compute  $6 + 6$  using the binary DA. Is DA correct?

►  $6_d = 110_b$  and  $6_d = 110_b$

	1	1	0
+	1	1	0
1	1	0	0

► The DA result is  $1100_b = (8 + 4 + 0)_d = 12_d$ . Correct, since  $6_d + 6_d = 12_d$ .

## \* ALU Correctness Examples with 3-Digit Signed Binary \*

A 3-bit ALU adds  $-1$  and  $-2$ . Is the ALU correct? And for 3 and 2?

►  $-1_d = 111_s$  and  $-2_d = 110_s$

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

► The 3-bit ALU result is  $101_s$ .

► The correct denary result is  $-1_d + (-2_d) = -3_d$ .

► *NOTE ALU sign digit 1 is in position 2, so ...*

► ... the 3-bit ALU result in denary is  $(-4 + 1)_d = -3_d$ .

► Hence: the ALU is correct, *even though there is a lost digit (1)!*

## \* ALU Correctness Examples with 3-Digit Signed Binary \*

A 3-bit ALU adds  $-1$  and  $-2$ . Is the ALU correct? And for 3 and 2?

▶  $3_d = 011_s$  and  $2_d = 010_s$

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

▶ The 3-bit ALU result is  $101_s$ .

▶ The correct denary result is  $3_d + 2_d = 5_d$ .

▶ The 3-bit ALU result in denary is  $(-4 + 1)_d = -3_d$ .

▶ Hence: the ALU is incorrect; the  $-3_d$  "should" be  $5_d$ .

In which example has **overflow** occurred?



## \* ALU Correctness Examples with 3-Digit Signed Binary \*

A 3-bit ALU adds  $-1$  and  $-2$ . Is the ALU correct? And for 3 and 2?

►  $3_d = 011_s$  and  $2_d = 010_s$

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

► The 3-bit ALU result is  $101_s$ .

► The correct denary result is  $3_d + 2_d = 5_d$ .

► The 3-bit ALU result in denary is  $(-4 + 1)_d = -3_d$ .

► Hence: the ALU is incorrect; the  $-3_d$  “should” be  $5_d$ .

In which example has **overflow** occurred? Overflow is a synonym for *incorrect*!

We wish to add two denary integers  $z$  and  $z'$ . Let's assume they can be represented by two  $k$ -bit signed binary numbers  $\vec{a}$  and  $\vec{b}$ , which we input to an ALU. The ALU result  $\vec{s}$  is correct precisely when *either* of these *equivalent* conditions is true:

- ▶ The correct denary result  $z + z' = \vec{a}_s + \vec{b}_s$  can be represented by a  $k$ -bit signed binary number, that is,

$$-2^{k-1} \leq z + z' \leq 2^{k-1} - 1$$

- ▶ Either of the following conditions hold:
  - ▶ The sign bits of the two operands are different ( $a_{k-1} \neq b_{k-1}$ ).
  - ▶ The sign bits of the two operands and the sign bit of the ALU result are all the same ( $a_{k-1} = b_{k-1} = s_{k-1}$ ).

Do these conditions hold in the examples we have looked at?

We look at both the **machine language** of, and the **datapath hardware** required for executing, **Branch Instructions**.

After studying this section you should be able to

- ▶ Work out machine language for branches.
- ▶ Trace **dataflow** around any *given* **datapath** and **instruction** (see lectures, and videos). Note: you do not need to memorize the hardware diagrams; they will be *given* in tests and examinations.
- ▶ Solve simple questions about the execution of branches and the corresponding dataflow.

- ▶ What is the machine language for `beq`?
- ▶ Such instructions are I-format:

op	rs	rt	address
<i>6 bits</i>	<i>5 bits</i>	<i>5 bits</i>	<i>16 bits</i>
opcode for <code>beq</code>	<code>#R<sub>1</sub></code>	<code>#R<sub>2</sub></code>	<code>L</code>

- ▶ Looking up the opcode and register address is easy!
- ▶ We can use the semantics to determine the binary address field, provided we know the memory address of `beq`, and the address of the labelled instruction. Now ...
- ▶ `beq R1, R2, L` has semantics

**if `!R1 = !R2` then goto `L` else goto `ninsm`**

- ▶ `!R1 = !R2` TRUE then execute instruction labelled `L`
- ▶ `!R1 = !R2` FALSE then execute the next instruction in memory

- ▶ The address field (in denary) is the offset in words from the address of the *next instruction*  $I$  in memory (say  $a_d + 4$ ) to the address of the labelled instruction  $L : I'$  (say  $a'_d$ ).

$a_d$	beq $R_1, R_2, L$	20	beq $R_1, R_2, L$
$a_d + 4$	<span style="border: 1px solid black; padding: 2px;">I .....</span> 1 word	24	<span style="border: 1px solid black; padding: 2px;">I .....</span> 1 word
	$\vdots$ 2,3,4, ... words	28	<span style="border: 1px solid black; padding: 2px;">J .....</span> 2 words
	<span style="border: 1px solid black; padding: 2px;">.....</span> $ow_d$ words	32	<span style="border: 1px solid black; padding: 2px;">K .....</span> 3 words
$a'_d$	$L : I' \dots\dots$	36	$L : I' \dots\dots$

- ▶ In the example  $ow_d = 3_d$ . We convert to *signed* binary to obtain the machine address field:  $0000000000000011 \in Bin^{16}$ .
- ▶ Note that  $a'_d$  might be less than  $a_d$ : we saw examples where the  $L$  labelled instruction is stored in memory *before* beq. In such cases  $ow_d$  will be negative! What is  $ow_d$  if  $a'_d = a_d + 4$ ?

- ▶ Each word location consists of four byte locations (cells). So

offset in cells is  $4 * ow_d$  (eg  $4 * 3_d = 12_d$ )

- ▶ Thus

$$a'_d = (a_d + 4) + 4 * ow_d$$

- ▶ So if we know the addresses of the branch and labelled instructions,  $a_d$  and  $a'_d$ , we can work out the word offset. And from  $ow_d$  we can compute signed binary address field  $\vec{b} \in Bin^{16}$ .

Suppose that a labelled instruction is at address  $40_d$  and a branch at  $60_d$ . What is the binary address field?

► Thus

$$a'_d = (a_d + 4) + 4 * ow_d$$

The branch instruction is at address  $a_d = 60_d$  and so the next instruction in memory is at  $a_d + 4 = 60_d + 4_d = 64_d$ . The label is at  $a'_d = 40_d$ . So  $40_d = 64_d + 4 * ow_d$  and hence  $-24_d = 4 * ow_d$  implying that  $ow_d = -6_d$ . Since  $-6_d = 1010_s$  the binary address field is  $sx(1010) = 1111111111111010 \in Bin^{16}$ .

*Remark:* Suppose you happen to know the complete machine code for a branch instruction. So you know the address field  $\vec{b} \in Bin^{16}$ . You can trivially work out the offset  $ow_d = \vec{b}_s$ . And if you also know the memory address for the branch  $a_d$ , then you can compute the address of the labelled instruction  $a'_d$ .

- ▶ The last section explained how to work out the machine language for a branch instruction `beq` (with label  $L$ ) at address  $a$  when the instruction labelled  $L$  is at address  $a'$ .
- ▶ Suppose we know the machine language for a branch `beq  $R_1, R_2, L$` . What *exactly* happens at execution time?
- ▶ Once again `beq  $R_1, R_2, L$`  has semantics
  - ▶  $!R_1 = !R_2$  TRUE execute the instruction at address  $a'_d$
  - ▶  $!R_1 = !R_2$  FALSE execute the instruction at address  $a_d + 4$
- ▶ But it is the PC that *points* to the next instruction to be executed.
- ▶ So the execution of a branch instruction amounts to nothing other than up-dating the PC:  $PC := a'_d$  or  $PC := a_d$ .



- So the *PC* should be updated to

$$\begin{array}{ll} a'_d = (a_d + 4) + 4 * ow_d & \text{if test TRUE} \\ a_d + 4 & \text{if test FALSE} \end{array}$$

- Before execution of *beq* we have  $!PC = a$ . Therefore the execution semantics (in binary) is given by

$$\begin{array}{ll} PC := \underbrace{(!PC + 4) + (4 * sx(\vec{b}))}_{\text{Branch target (binary)}} & \text{if test TRUE} \\ PC := !PC + 4 & \text{if test FALSE} \end{array}$$

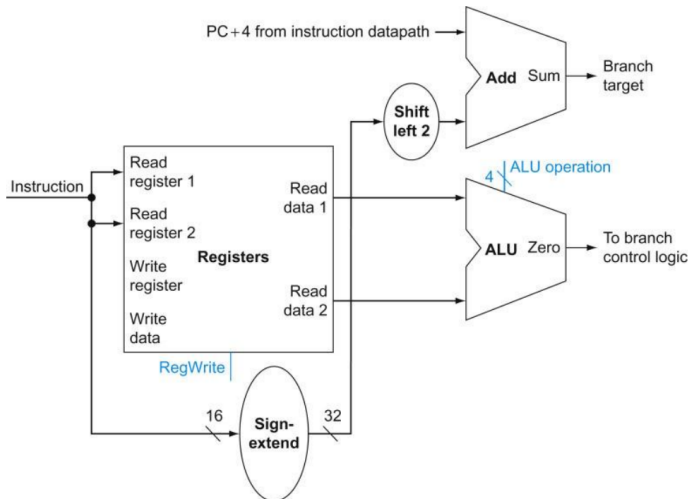
where  $\vec{b} \in \text{Bin}^{16}$  is the *beq* binary address field, and  $sx(\vec{b})$  is a sign extension to 32-bits. Note  $+$  and  $*$  are 32-bit ALU operations; and so 4 is short-hand for 32-digit binary  $\vec{0100}$ .

- ▶ We need to work out if  $!R_1 = !R_2$  is TRUE or FALSE ...
- ▶ The ALU has a (so-called) **test for zero** output  $\text{Zero} \in \text{Bin}$ . If the ALU result  $\vec{r}$  equals  $\vec{0}$ , then  $\text{Zero} = 1$ ; and if  $\vec{r}$  is not equal to  $\vec{0}$ , then  $\text{Zero} = 0$ .
- ▶ The ALU calculates  $\vec{r} \stackrel{\text{def}}{=} !R_1 - !R_2 \in \text{Bin}^{32}$ .
- ▶ Then we have

$!R_1 = !R_2$	$\vec{r} \stackrel{\text{def}}{=} !R_1 - !R_2$	Zero
TRUE	$= \vec{0}$	1
FALSE	$\neq \vec{0}$	0

- ▶  $\text{Zero}$  outputs from the ALU: TRUE/FALSE is computed as  $1/0$ .

## \* Datapath for Computing Branch target and TRUE/FALSE \*



**Branch target** is the name of the datapath bus returning the *address of the labelled instruction*, denoted in denary by  $a'_a$  previously. Hennessy and Patterson use this name. For this reason I refer to it myself.

## \* Dataflow for Computing Branch target and TRUE/FALSE \*

6 bits	5 bits	5 bits	16 bits
31-26	25-21	20-16	15-0
opcode for beq	$\#R_1$	$\#R_2$	$\vec{b}$

- ▶  $\#R_1 = I[25 - 21]$  and  $\#R_2 = I[20 - 16]$  are passed into 5-bit Read register 1 and Read register 2.
- ▶  $\vec{b} \in Bin^{16}$  goes into the sign extend unit, extended to 32 digits.
- ▶ In response the source operands  $!R_1$  and  $!R_2$  appear on the 32-bit Read data 1 and Read data 2
- ▶  $!R_1$  flows into the upper ALU input bus, and  $!R_2$  into the lower lower ALU input bus.
- ▶ The opcode is sent to a control unit; this generates a signal on ALUOperation to make the ALU do subtraction.

- ▶ Zero outputs from the ALU: TRUE/FALSE is computed as 1/0.
- ▶ Refer back to Slide 206. Notice that the inputs to the upper ALU make it compute

$$(!PC + 4) + (sx(\vec{b}) \ll 2)$$

which equals the binary Branch target ( $a'_d$  in denary)

$$(!PC + 4) + (4 * sx(\vec{b}))$$

- ▶ The Final Step is to update PC. We use Zero, Branch target and the current value  $!PC = a$  to do so ...



## \* Branch if Equal Instruction Execution (Final Step) \*

- ▶ The Zero bus is connected to PCSrc (so they are equal).
- ▶ The multiplexor ensures that PC is correctly updated:

$!R_1 = !R_2$	Zero	PCSrc	PC :=
TRUE	1	1	$(!PC + 4) + (4 * sx(\vec{b}))$
FALSE	0	0	$!PC + 4$

where we have seen on Slide 206 that

- ▶  $[(!PC + 4) + (4 * sx(\vec{b}))]_s = a'_d$  address of  $L : I'$
- ▶  $[!PC + 4]_s = a_d + 4$ , with  $a_d$  the address of `beq  $R_1, R_2, L$` .

## **\*\* Overview: Predicate and Bitwise Logic \*\***

We look at one simple example of predicate logic, negating  $\&\&$ , and connections with sets. We look at the definition of bitwise  $\&\&$  and  $||$ , and a simple application.

After studying this section you should be able to

- ▶ Explain what a **predicate** is. Learn more about **predicate logic** and connections with sets.
- ▶ Define bitwise  $\&\&$  and  $||$ , and compute simple examples. Solve simple problems.

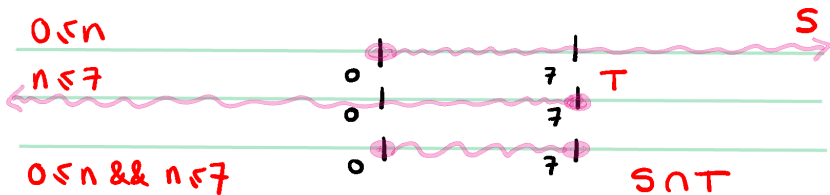


What is  $\overline{0 \leq n \ \&\& \ n \leq 7}$ ? Does it make sense to ask if this is true or false? For what values of  $n$  is this statement true?

Look at the pictures on Slide 215. By calculating the negation as an  $||$ , one sees that the predicate expression is true when  $n$  is strictly below  $0$  or strictly above  $7$ .

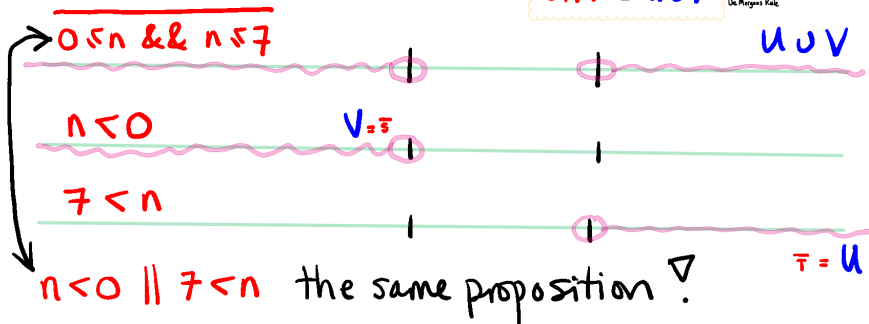
Notice the connections between logic's  $\&\&$ ,  $||$  and  $\overline{\phantom{x}}$ , and set theory's  $\cap$ ,  $\cup$  and complement  $\overline{\phantom{x}}$ .

# \*\* An Example Using Variables: Predicate Logic \*\*



$$\overline{S \cap T} = U \cup V$$

De Morgan's Rule



## \* Digitwise (Bit-by-bit) Logic \*

- ▶ I have a computer that works with 8-bit unsigned numbers, and I want to round some numbers to the four most significant digits:  $1011.0110 \mapsto 1011.0000$ . How can I do this? LEC
- ▶ Consider using  $\&\&$ , bit-by-bit (ie digitwise) in columns

$\vec{a}$	1	0	1	1	0	1	1	0
$\vec{b}$	1	1	1	1	0	0	0	0
$\vec{a} \& \vec{b}$	1 & 1 = 1	0	1	1	0	0	0	0

We choose  $\vec{b}$  so we can round up any value of  $\vec{a}$  to four digits.

We can define *digitwise*  $\&\&$  and  $||$

$$a_k \dots a_0 \& b_k \dots b_0 \stackrel{\text{def}}{=} (a_k \& b_k) \dots (a_0 \& b_0)$$

$$a_k \dots a_0 || b_k \dots b_0 \stackrel{\text{def}}{=} (a_k || b_k) \dots (a_0 || b_0)$$

## Overview: Cache Memory

---

- ▶ We explain the high level ideas behind cache memory.
- ▶ We then look at the technical details of a simple cache.

- ▶ The principles of the memory hierarchy can be used to help design modern computing devices that:
  - ▶ *are fast*
  - ▶ *have a large capacity memory*
  - ▶ *are a good price*
- ▶ This is achieved (in part) by using a **cache**. This is fast memory, of a moderate size, that is usually “located on, or near to, a processor board”.
- ▶ Just as a main memory is composed of cells, cache is composed of **lines**. There will be  $C$  in total.
- ▶ Lines have **numbers** from  $0$  to  $C - 1$ .

- ▶ We also make use of the **locality principle**:
- ▶ Given any instruction **I** that is about to be fetched, it is likely that the “next few instructions to be fetched” are physically nearby in memory (they are said to be “local”).
- ▶ We refer to such a memory area as a **block**, and sometimes to **I**s block. LEC

- ▶ The FDE cycle is modified in the presence of cache memory.
- ▶ Instructions to be fetched may already be stored in the cache:  
*Each line stores exactly one block of instructions.*
- ▶ **Modified FDE cycle:** control will check if the instruction **I** to be fetched is in a cache line. If so, **I** will be fetched from the cache to the IR in the CPU.
- ▶ If not, control will copy (fetch) the block containing **I** from main memory into a cache line, and then copy **I** from the cache line into the IR. To do so:
  - ▶ We need the address of **I**: it is **!PC** at fetch time.
  - ▶ Control will need to compute which block **I** is in (actually, the start and end address of the block).
  - ▶ It will also need to compute a line number for the block.
- ▶ **I** is now in the IR and can be decoded and executed.

Suppose that

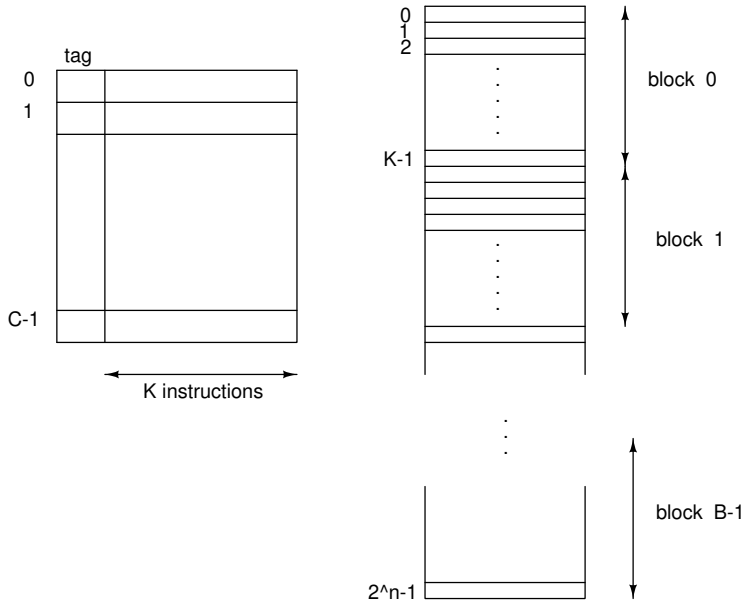
- ▶  $a \stackrel{\text{def}}{=} !\text{PC}$ .
- ▶ Instruction  $I$  is held in  $W[a]$  (where  $a$  is a multiple of 4).
- ▶ There are  $K$  instructions in each block (by design); blocks are numbered from 0 upwards.
- ▶ The cache has  $C$  lines (by design).

Then

- ▶  $I$  appears in block  $b \stackrel{\text{def}}{=} (a/4) \text{ DIV } K$ ;
- ▶ the first instruction in block  $b$  has address  $b * (4 * K)$ ;
- ▶ the last instruction has address  $(b + 1) * (4 * K) - 1$ ; and
- ▶ block  $b$  is copied into cache line  $l = b \text{ MOD } C$  with tag  $b$ .



# A Cache and a Main Memory



## Overview: Parallelism via Pipelining

---

- ▶ Instructions are executed on the processor in parallel: at any time, a small number of instructions are executing, each at a different stage of completion.
- ▶ The idea is that each individual processor circuit is executing a different stage/part of each instruction.
- ▶ A simple analogy: You have to clean your clothes, your Mum's and your Dad's (three instructions). There is a washing machine, a drier, and an ironing board (three "circuits").
- ▶ We start with Dad's clothes in the washing machine. Once complete, they move to the drier; and we add Mum's clothes to the washing machine.
- ▶ At the next stage, the washer has your clothes, the drier Mum's clothes, and Dad's clothes are being ironed.