Chapter 1

- Introduce some mathematics which we will use throughout the module.

Overview: Simple Mathematics

- Syntax and Semantics—words and their meaning.
- Sets—collections of things.
- Functions—data in and answers out.

Syntax and Semantics

- **Syntax** refers to symbols, or notation.
- **Semantics** refers to “meaning”.
- **Denotation** is the relation between some syntax and its semantics.

Languages

- A **language** consists of syntax and semantics.
- The syntax consists of
  - **alphabet**;
  - **rules** for writing “words” from alphabet.
- In a **programming** language the “words” are *program instructions*. Rules given in the programming manual.
- The semantics of a program is what happens when the program runs.

Sets

- A **set** is *an unordered collection of objects in which any object can appear at most once*. Eg $\{$‘a‘, ‘g‘, ‘r‘, ‘t‘$\}$.
- The objects in a set are called **elements**.
- ‘a‘ $\in \{$‘a‘, ‘g‘, ‘r‘, ‘t‘$\}$ and $4 \notin \{$‘a‘, ‘g‘, ‘r‘, ‘t‘$\}$.
- $\mathbb{N} \overset{\text{def}}{=} \{0, 1, 2, \ldots\}$ set of **natural numbers**.
- $\mathbb{Z} \overset{\text{def}}{=} \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ set of **integers**.

- The notation $e \in A$ means "e is an element in A".

- A is a set, but its elements are not known; neither do we know what e actually is.

- A and e are **variables**—unknown *quantities*.

- When programming, you might write

$$\texttt{n:int} \qquad \text{or} \qquad \texttt{int n}.$$

to indicate that n is an integer, with an unknown value.

- Note that

$$n \in \mathbb{Z}$$

means the same thing—and is "standard" in mathematics.

Sequences

- $\{1,6,7,8\}$ is the same as $\{6,8,1,7\}$.
- BUT 1678 is different from 6817 . . .
- . . . each is a *sequence* of digits.
- $\mathbb{D} \overset{\text{def}}{=} \{0,1,2,3,4,5,6,7,8,9\}$.
- Set of all sequences of length 2 is

$$\mathbb{D}^2 \overset{\text{def}}{=} \{00,01,02,\ldots,09,10,11,\ldots,99\}$$

- The set of all sequences of elements of A of length 4 is denoted by A^4.

- If A is $\mathbb{D}$, then for example $1678 \in \mathbb{D}^4$.

- If u, v, w, x are four (variable) elements of A, then $uvwx$ denotes a single typical element of A^4, ie $uvwx \in A^4$.

- We typically write the variables as a_3, a_2, a_1, a_0, so that

$$a_3a_2a_1a_0 \in A^4$$

Makes "position" of each variable clear; a_2 in position 2.

- A^n is the set of **sequences** of elements of A of length n.

- A typical element of A^n looks like $a_{n-1}a_{n-2}\ldots a_0$, where each $a_i \in A$.

- We refer to variable a_i, where i is the position 0, 1, or $\ldots n-1$.

- We sometimes call a_i a **component** of the sequence.

- We sometimes write $\vec{a} = a_{n-1}\ldots a_1a_0$.

Functions

- A *function* is a machine which takes inputs and produces outputs.

- For example function s where $3 \mapsto 9$ and $-2 \mapsto 4$.

- To *define* a function, we give
  - A set of inputs (eg $\mathbb{Z}$);
  - a set containing all of the outputs (eg $\mathbb{Z}$);
  - an output (eg z^2) for every input (eg z).

- $G\colon \{1,2,3,5\} \to \{3,6,11,27,99\}$

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

- We can also define G by giving the recipe

$$G(i) \overset{\text{def}}{=} i^2+2 \quad ; \quad G(i) \text{ is the output from input } i$$

- Note that

1. For each input from $\{1,2,3,5\}$ there **is** an output in $\{3,6,11,27,99\}$.
2. For each input there is one, and only one, output.

- We define a **function** $f:A \to B$ by

  - **either** giving a formula $f(a)$ where a is any input from A, with each output $f(a)$ in B;
  - **or** drawing arrows $a \mapsto b$, *one* for each $a \in A$ (and no others), with $b \in B$;
  - **or** writing down a set of input/output pairs (a, b), *exactly one* pair for each $a \in A$, with $b \in B$.

- We write $f(a)$ or $f\ a$ for this unique $b \in B$. For example, $G(3) = 3^2 + 2 = 11$.

- We say a is the **argument** of the function.

- Also say that $f(a)$ is the **output**, for **input** a.

- Consider the following set g of input/output pairs

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

This is not a function $\{\,3,8,9\,\} \to \{\,4,7,8,10,300\,\}$. Why?

- Let $A \overset{\text{def}}{=} \{\,3,4,5\,\}$ and $B \overset{\text{def}}{=} \{\,4,7\,\}$ and

$$f \overset{\text{def}}{=} \{\,(3,4),(4,7),(5,9)\,\}$$

Then $f{:}A \to B$ is not a function from A to B. Why?

- $\mathbb{B} = \mathbb{B}^1 = \{0,1\}$ is the set of binary numbers with 1 digit.
- $\mathbb{B}^2 = \{00,01,10,11\}$ is the set of binary numbers with 2 digits.
- $\mathbb{B}^3 =$? is the set of binary numbers with 3 digits.
- $\mathbb{B}^n$ is the set of binary numbers with n digits.
- We shall study functions $f{:}\mathbb{B}^n \to \mathbb{B}^m$ in CO1016.
- Such a function is an $n \mid m$-**ary function.**
- Computer circuits implement functions on binary numbers.

Top Level Computer Organisation

- What is a Computer?
- The Central Processing Unit [CPU] (calculating)
- Memory (storing)
- Input and Output (data in and data out)

What is a Computer?

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

Software (Programs)

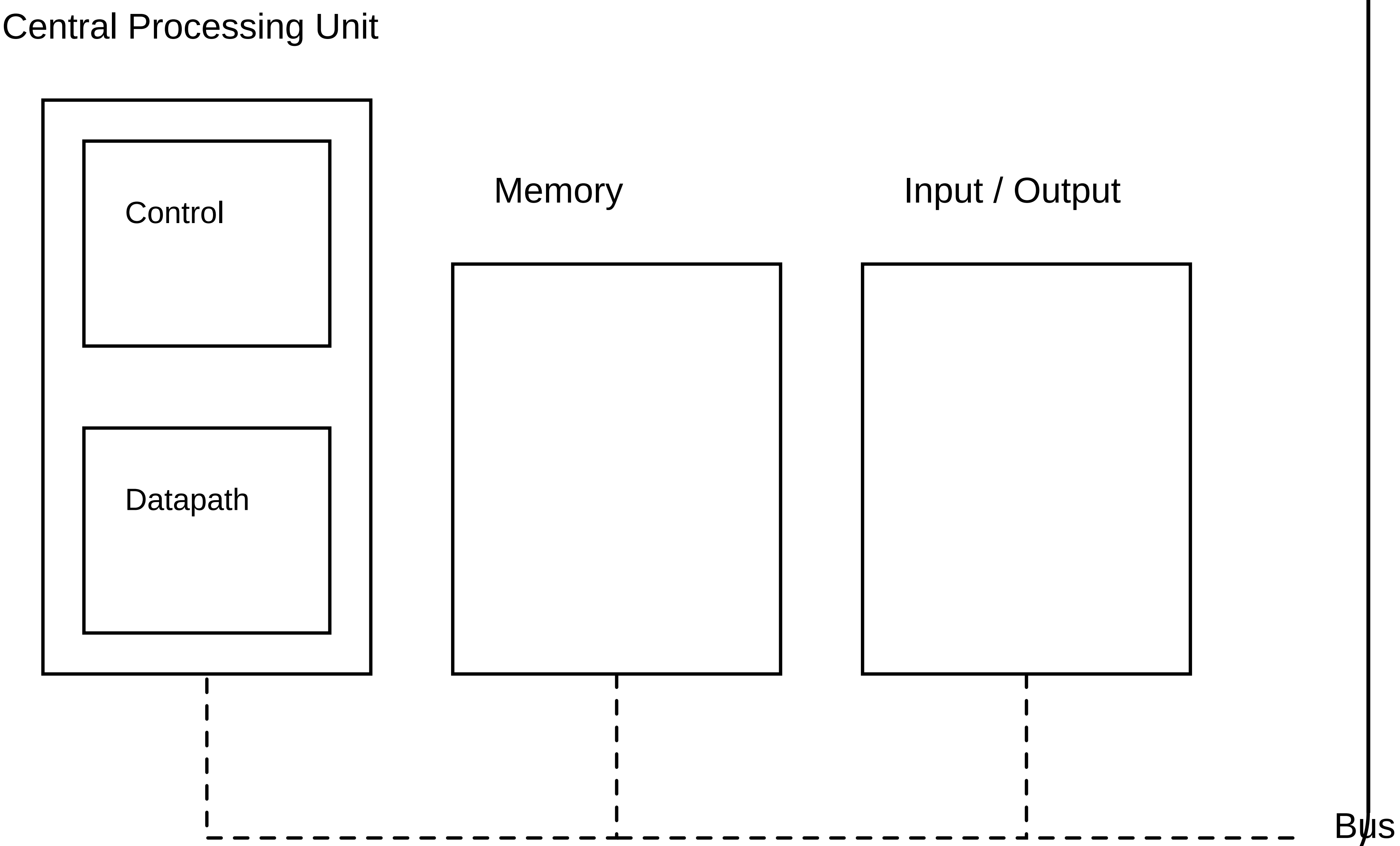
```
        a := 0;
        c := 4;
L       a := a+c;
        c := c-1;
        if c >= 1 then L;
        exit;
```

- An **algorithm** is a set of instructions or “recipe”.
- A **computer program (software)** is an implementation of an algorithm.
- Programs can be written in **assembly language** (the syntax).
- A program is a list of **assembly instructions**.
- Each instruction (eg $\mathtt{a} := 0$) has a *semantics*.

Hardware

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

How does a computer work?



Overview: The CPU

- How programs run on a computer.
- Basic details of a Central Processing Unit
  - Datapath: The part that *calculates*.
  - Control: The part that *organzises* the calculations which are described by a program.

Central Processing Unit

- Computers work by performing simple arithmetic. That’s it!
- Graphics made from small dots of light. Position and colour of a dots changes to produce “moving” images.
- The changes are calculated using complex arithmetic—but complex calculations reduce to simple arithmetic.

- A program (list of instructions) will be stored in main memory.

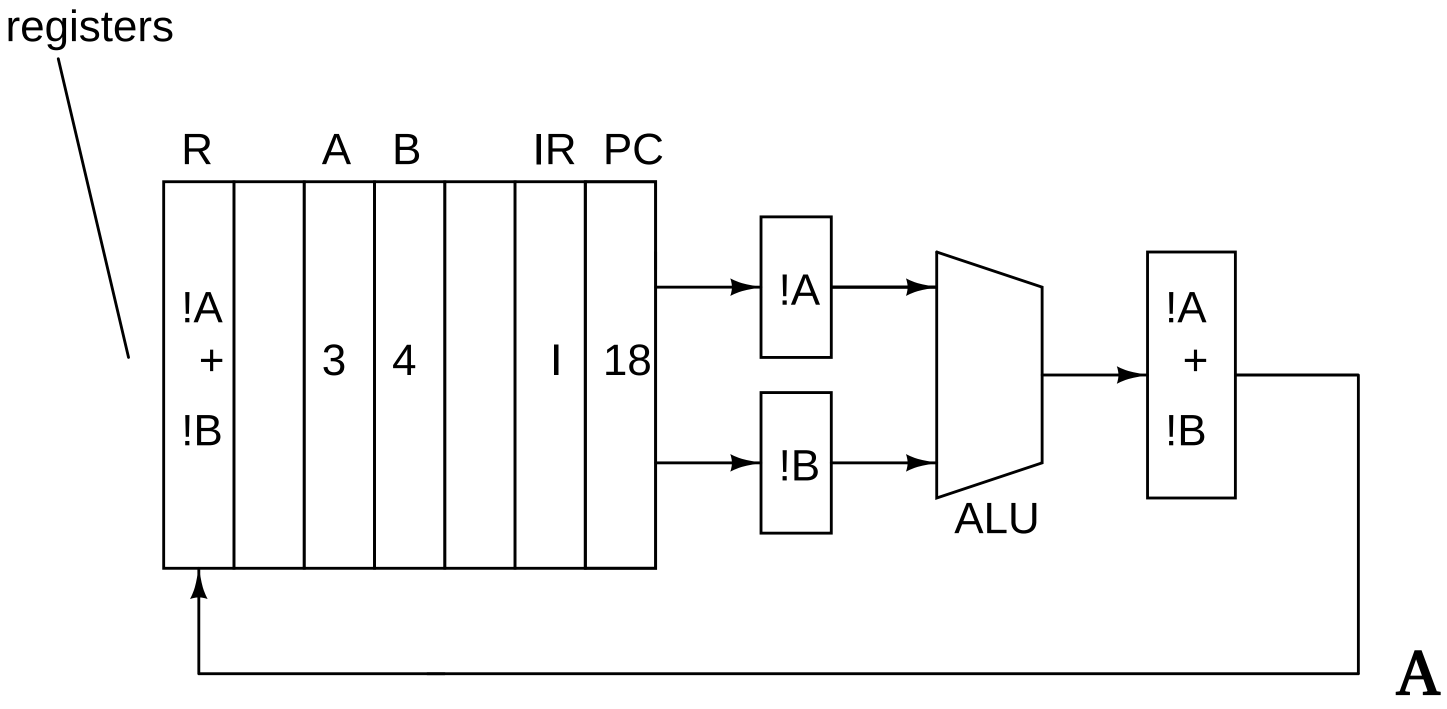
- Program execution: **Control** *copies* (“fetches”) each instruction into the CPU. Fetched instructions are *stored* in the **Instruction Register (IR)**

- There are many other registers. Each is computer memory; they store instructions and more besides . . . .

- Each instruction is executed by the **datapath.**

- The datapath is “like” a calculator . . .

- The *syntax* (assembly language) of our example instruction I is `add` R, A, B.
- The *semantics* is to pass contents of A and B to the ALU, calculate the sum, then store the result in R.
- Computer stores instruction as a sequence of 0s and 1s, eg 00111.0001.0011.0100. Called **machine language**.



Datapath after execution of I = `add` R, A, B

IR $\boxed{00111.0001.0011.0100}$

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

- **Fetch** an instruction from main memory—copy it into the IR;
- **Decode** the instruction—eg set the ALU to $+$;
- (**update** PC to point to next instruction;)
- **Execute** the instruction; and
- repeat the FDE-cycle.

Overview: Memory

- A broad understanding of how data is stored in computer memory.
- Knowledge of the different kinds of memory.
- Understanding of the technical details of how memory works.
- Know one key reason why we can build fast and cheap and big PCs.

Memory Characteristics

- **Memory** is divided into **primary memory** (**inboard memory**) and **secondary memory**.

1. **CPU memory**, where data is stored within the CPU, either in registers or in a cache.
2. **Main memory**, where data is stored inside other computer circuits, known as **memory chips**.
3. **Outboard memory** often located on the computer, such as **magnetic discs** and so on.
4. **Off-line memory** such as **magnetic tape**.

The Memory Hierarchy

- As you move down the **memory list**:
  - decreasing cost per unit of storage;
  - increasing capacity;
  - increasing access time;
  - decreasing access of memory by CPU.

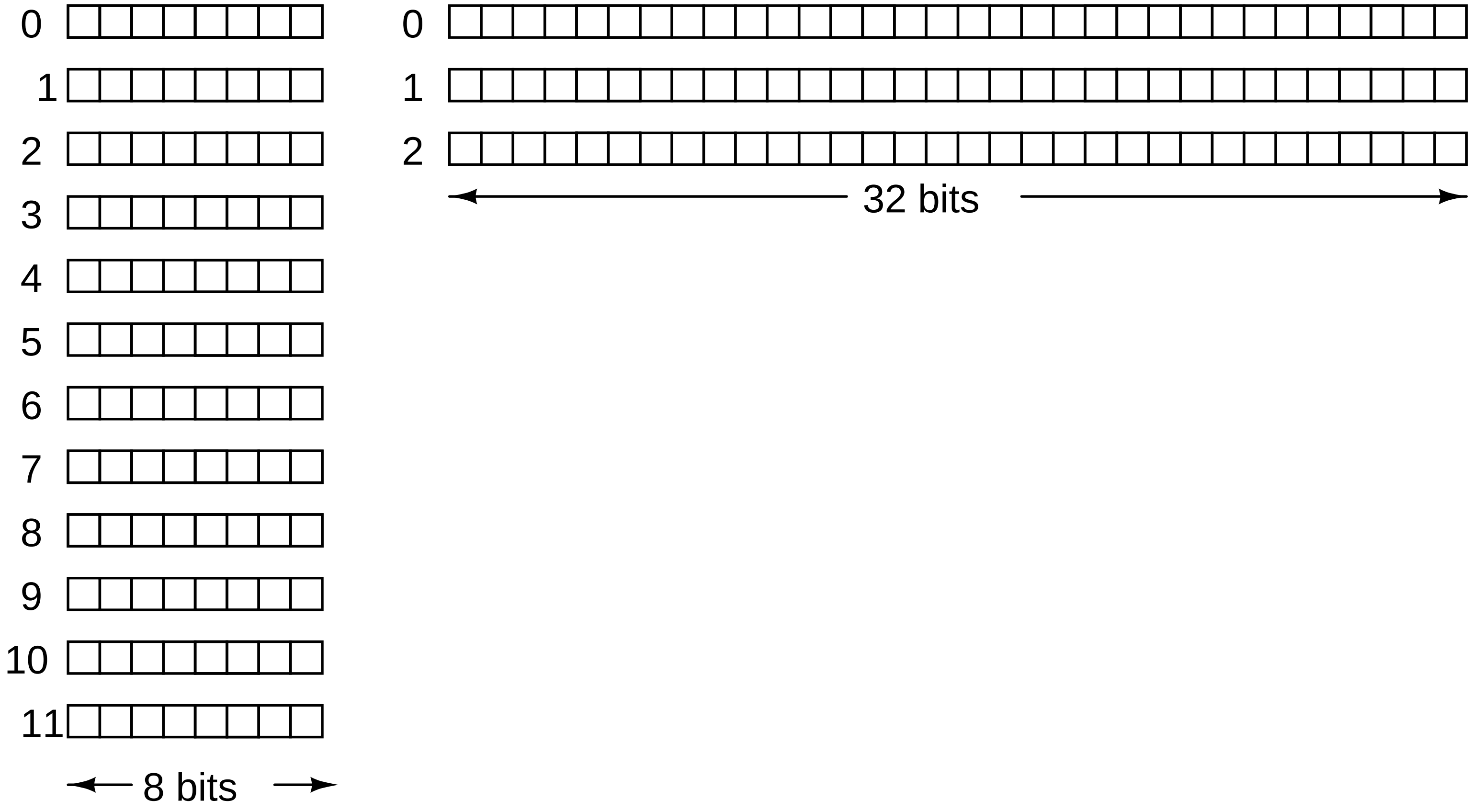
Bits

- Use the symbols 0 and 1 to represent information (data). Eg 1,2,3,4 represented by 1, 11, 111, 1111.
- A computer can represent 0s and 1s as low and high voltages. A **bit** is a “circuit” that can “store” a 0 or 1.
- We call 0 and 1 **binary digits**.

High Level view of (Main) Memory

- *storage facilities* (filing cabinets)
- *locations for storing objects* (draws)
  - Fundamental location is a *k*-**bit cell**.
- *addresses of locations* (draw labels)
  - Each cell has a numerical **address**.
- *contents of the locations* (documents)

Computer Memory Details

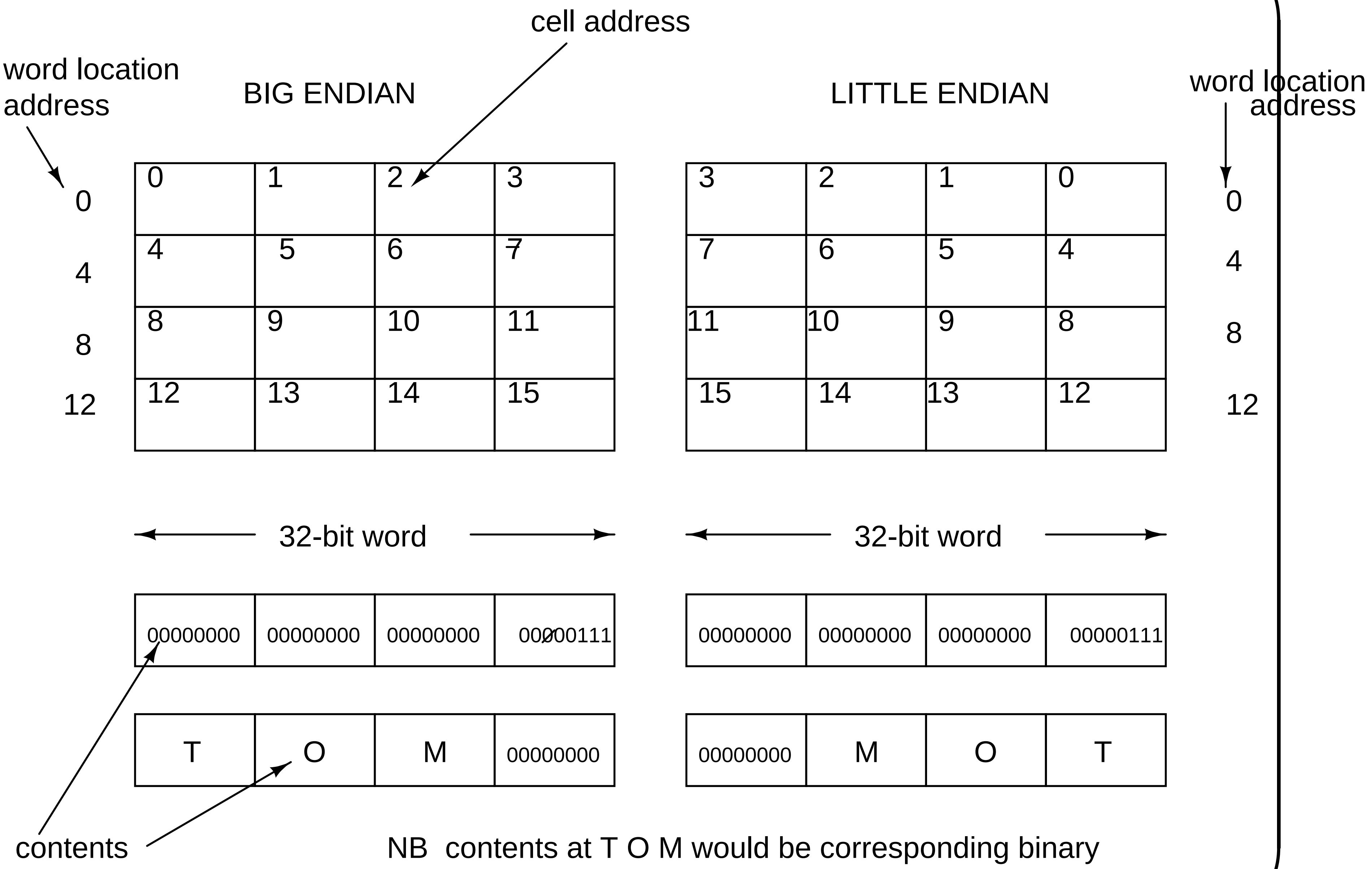


- A memory with n cells will have addresses 0 to $n-1$.
- If there are n bits in total, we call it an n**-bit memory**.
- The cell is the smallest *individually addressable* part of a memory.
- Any *cell* is composed of k bits. The binary digits contained in these bits are the **contents** of the cell.
- k-bit cell can hold 2^k different binary digit sequences. Why?

- Many computers have a standard 8-bit cell called a **byte location**.
- The **address of a byte location** is address of the corresponding 8-bit cell.
- Groups of consecutive cells are called **word locations**—used to store bigger data than will fit into a byte location.
- The **address of a word location** is the smallest address of its cells.
- The contents of a byte location are called a **byte**, and of a word location a **word**.

Organizing Memory - The Endian Systems

- How do we arrange cells (typically byte locations) into word locations? Why is the arrangement important?
- Cells in any word location can be addressed from "left to right", or "right to left".
- The left to right numbering is called **big endian**, and the right to left **little endian**.



cell address

BIG ENDIAN

word location address				
0	0	1	2	3
4	4	5	6	7
8	8	9	10	11
12	12	13	14	15

← 32-bit word →

00000000	00000000	00000000	00000111
T	O	M	00000000

LITTLE ENDIAN

				word location address
3	2	1	0	0
7	6	5	4	4
11	10	9	8	8
15	14	13	12	12

← 32-bit word →

00000000	00000000	00000000	00000111
00000000	M	O	T

contents

NB contents at T O M would be corresponding binary

- Integers are stored in word locations—they can be big! Store 2^{31} as $1\vec{0}^{\,\mathit{bin}} = 1000\ldots000^{\,\mathit{bin}}$—a 32-bit word.

- 7 will also be stored in a word location, as $\vec{0}111^{\,\mathit{bin}}$.

- The 1s will appear in byte locations 3 or 7 or 11 or 15 in the big endian system.

- ‘A‘ to ‘Z‘ will be represented by small integers, known as **character codes**. Eg **ASCII**—‘A‘ is stored as the binary for 65.

- Thus codes are stored in cells (byte locations)—check this!

- Problems can arise when data is transferred from a big endian (SPARC) to a little endian (Intel) computer …

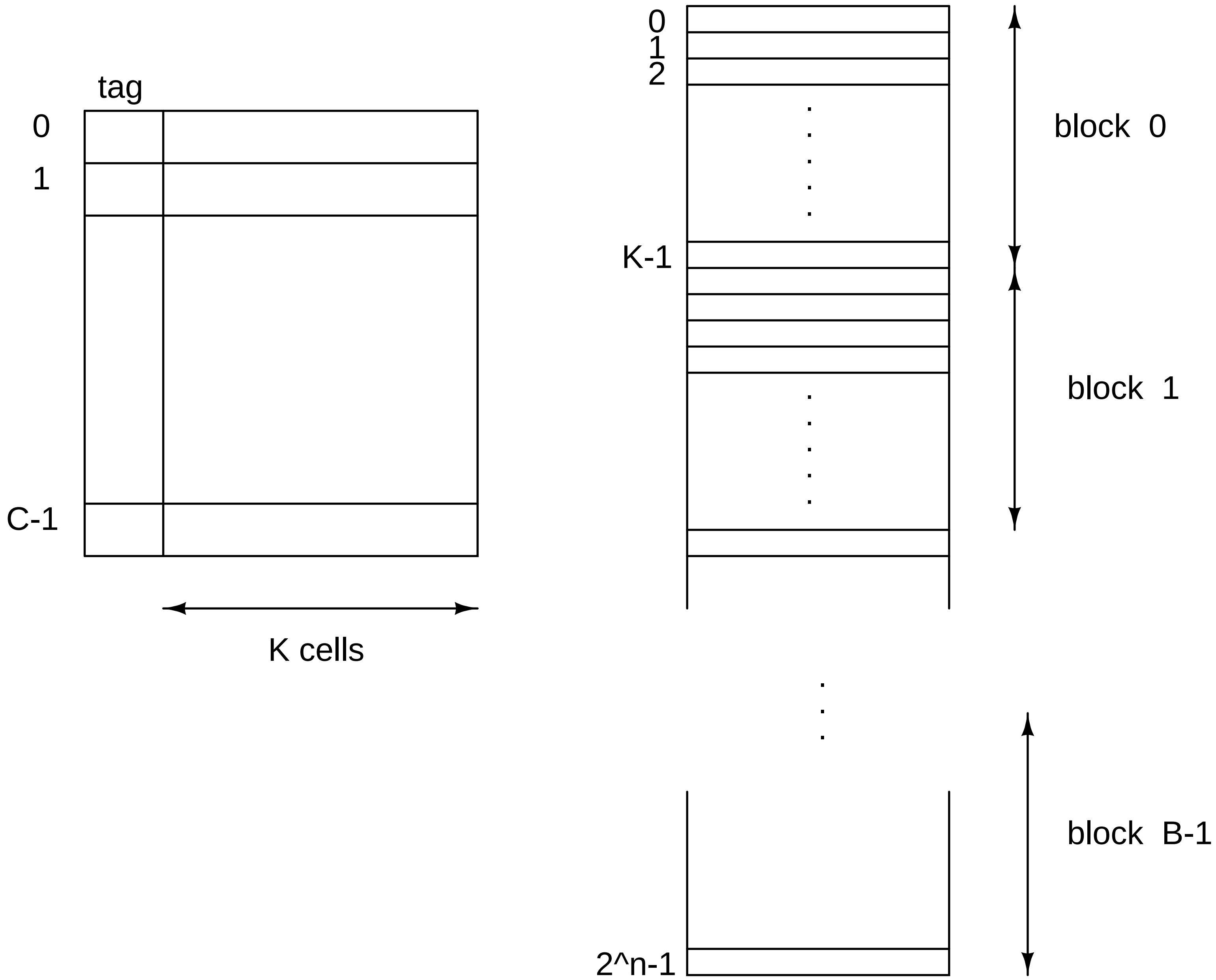
Physical Types of Primary Memory

Please read pages 20 to 22 which describe **Random Access Memory** (**RAM**), **Static** RAM (**SRAM**) etc etc

Cache Memory

- Can use hierarchy to build a modern computer:
  - *fast*
  - *large capacity memory*
  - *good price*
- This is achieved (in part) by using a **cache** . . .
- Given any instruction, it is likely that the next few instructions to be executed are nearby (*local*) in memory.
- This is called the **locality principle**.
- Such a small group of instructions is called a **cluster**.

- **Cache** is fast memory found near, or on, the CPU. It consists of **lines**.
- Each line stores (clusters of) instructions.
- Fetch cycle: control will check if the next instruction is in a cache line. If so, it will be fetched from cache to CPU.
- If not, control will fetch the next instruction's cluster from main memory into the cache, and place the next instruction into the CPU.
- We formalise the idea of a cluster as a **block**.


tag
0
1
C-1
K cells
0
1
2
K-1
2^n-1
block 0
block 1
block B-1

If instruction I is held in the cell with address a, then

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

Secondary Memory

- *Secondary Memory* Read Chapter 5 of Stallings. Additional information can be found on pages 69 to 88 of Tannenbaum. Non examinable.

Digital Arithmetic

- Decimal Number Systems (10 fingers)
- Binary Numbers in Computers (2 fingers)
- Binary Addition (adding with 2 fingers)
- 2s-Complement Numbers (dealing with negatives)
- Logical Operations (true and false)

Overview: Radix Number Systems

- How to represent integers in a computer as binary numbers.
- Learn about other representations.

Representations of numbers

- $\mathbb{N} \overset{\text{def}}{=} \{0, 1, 2, \ldots\}$ and $\mathbb{Z} \overset{\text{def}}{=} \{\ldots, -1, 0, 1, \ldots\}$.
- Recall *Roman* representation.
- A **representation** of $\mathbb{N}$ is given by
  - a set of symbols S
  - a (bijective) function $[\![-]\!]: S \to \mathbb{N}$
- A symbol s **denotes** the number $[\![s]\!] \in \mathbb{N}$.

Binary numbers

- *binary numbers* are a representation for integers.
- The **binary digits** are 0 or 1.
- A **binary number** is a *sequence of binary digits*, an element of $\mathbb{B}^k$, eg $10010^{\,bin} \in \mathbb{B}^5$.
- We often use d or d_i to stand for a binary digit.
- Write a k-digit binary number as

$$d_{k-1} \ldots d_0{}^{\,bin} \in \mathbb{B}^k$$

- $d_{k-1}\dots d_0{}^{\,bin}$ is a *sequence of digits*.
- It *denotes* an integer $[\![d_{k-1}\dots d_0{}^{\,bin}]\!]$.
-

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

- A digit d_i, 0 or 1, tells us how many 2^i we have, zero or one!

In general

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

Radix (Base)

- 10 and 2 are referred to as a **radix** or **base**.
- To represent a number in radix r, we take r different symbols. Each symbol is called a **digit**.
- Each digit denotes a number; we write $[\![d]\!]$ for the integer denoted by the digit d.
- A number is represented with radix r as $d_{k-1}\ldots d_0{}^r$, where

$$[\![d_{k-1}\ldots d_0{}^r]\!] \quad \overset{\text{def}}{=} \quad [\![d_{k-1}]\!] * r^{k-1} + [\![d_{k-2}]\!] * r^{k-2} + \ldots + [\![d_0]\!] * r^0$$

Hexadecimal

- The radix in hexadecimal is 16.
- Digits are $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$.
- We define $[\![0]\!] \overset{\text{def}}{=} 0$, $[\![1]\!] \overset{\text{def}}{=} 1$, $\ldots$ $[\![E]\!] \overset{\text{def}}{=} 14$, and $[\![F]\!] \overset{\text{def}}{=} 15$.
- Thus $B1A^{\mathit{hex}}$ represents the integer 2842: here we have $k = 3$ digits and B in position 2

$$[\![B1A^{\mathit{hex}}]\!] = [\![B]\!] * 16^2 + [\![1]\!] * 16^1 + [\![A]\!] * 16^0$$

Overview: Binary Numbers in Computers

- How to store binary numbers in computers.
- A few technical definitions.

Binary Numbers in Computers

- Store a binary number using bits.
- Must represent numbers within a finite range.
- In 4 bits, the largest number is given by $1111^{\,bin}$, that is

$$2^3 + 2^2 + 2^1 + 2^0 = 15 = 2^4 - 1$$

- If k bits represent a number n, then $0 \leq n \leq 2^k - 1$.
- We shall say the computer has k-**bit numbers** or equivalently k-**digit numbers**.

■ Write a k-bit/digit number as $d_{k-1}\ldots d_0{}^{bin}$ or $\vec{d}$.

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

■ If we have $k \overset{\text{def}}{=} 6$ bits, then the number 3 is represented by 000011^{bin}.

- We call 0 and 1 **complements** of each other, and write $\overline{0} \overset{\text{def}}{=} 1$ and $\overline{1} \overset{\text{def}}{=} 0$.

- If $\vec{d}^{\,bin} = d_{k-1} \ldots d_0{}^{bin}$ is a binary number, then its **digitwise complement** is defined to be $\overline{d_{k-1}} \ldots \overline{d_0}{}^{bin}$.

- We will sometimes denote the digitwise complement of $\vec{d}^{\,bin}$ by $\overline{\vec{d}^{\,bin}}$ or just simply $\overline{\vec{d}}$.

Overview: Binary Addition

- How does a machine add binary numbers?
- Is the machine always right?

Binary Addition

- In $a+b$, a and b are **operands** and $+$ the **operator**.

- Recall digitwise addition. If $d_2d_1d_0\,^{dec} = 927\,^{dec}$ and $d'_2d'_1d'_0\,^{dec} = 436\,^{dec}$, then the sum is

$$\begin{aligned}
s_3 &= (0+0+\boxed{1})\,MOD\,10 &= 1\\
s_2 &= (9+4+\boxed{0})\,MOD\,10 &= 3\\
s_1 &= (2+3+\boxed{1})\,MOD\,10 &= 6\\
s_0 &= (7+6+\boxed{0})\,MOD\,10 &= 3
\end{aligned}$$

- We describe formally the **digitwize algorithm** to compute $\vec{s}^{\,dec} \stackrel{\text{def}}{=} \vec{d}^{\,dec} + \vec{d'}^{\,dec}$.

- $carry_0 \stackrel{\text{def}}{=} 0$.

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

- To add binary numbers, change 10 to 2.

Check that $10^{\,bin} + 111^{\,bin} = 1001^{\,bin}$ by showing that the algorithm gives rise to the following table

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

Correctness

- Let $z = [\![\vec{d}^{\,bin}]\!]$, and $z' = [\![\vec{d'}^{\,bin}]\!]$.
- Digitwise algorithm calculates $\vec{s}^{\,bin}$ from $\vec{d}^{\,bin}$ and $\vec{d'}^{\,bin}$.
- The algorithm is **correct** if $[\![\vec{s}^{\,bin}]\!] = z + z'$.
- We say $z + z'$ is **correctly** represented by $\vec{s}^{\,dec}$.
- Thus a formal statement of correctness is

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

where $=$ is **test for equality** (written $==$ in Java).

Example of Correctness

- In fact the algorithm is always **correct**.
- A computer may not always be correct . . .

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

Overview: 2s-Complement Numbers

- How to represent any integer in a computer.
- How to add and subtract in a computer.
- Is the computer correct?

2s-Complement Numbers

- Must represent all *integers* in a computer.

- In a $k+1$-bit 2*s*-complement system, the bit in position k will be 0 for positive integers and zero, and 1 otherwise.

- Given a 2*s*-complement number $d_k \ldots d_0{}^{bin}$, the actual integer it represents is, by definition,

$$\llparenthesis d_k \ldots d_0{}^{bin} \rrparenthesis \stackrel{\text{def}}{=} -d_k * 2^k + [\![d_{k-1} \ldots d_0{}^{bin}]\!] = \\ -d_k * 2^k + (d_{k-1} * 2^{k-1} + d_{k-2} * 2^{k-2} + \ldots + d_0 * 2^0)$$

If $k = 5$, then

$$\llparenthesis 000111^{\,bin} \rrparenthesis = -0 * 2^5 + 7 = 7$$

but

$$\llparenthesis 100111^{\,bin} \rrparenthesis = -1 * 2^5 + 7 = -32 + 7 = -25$$

Note also that

$$25 = \llparenthesis 011001^{\,bin} \rrparenthesis.$$

- If m is any integer, its **negation** is defined to be $-1 * m$.

- Subtraction performed by adding the negation of an integer.

- The negation of the integer represented by $\vec{b}^{\,bin}$, is represented by the $k+1$ 2s-complement number number $\overline{\vec{b}^{\,bin}} + 1^{\,bin}$.

- Any integer z, to be representable with $k+1$ bits, must lie in the range below

$$-2^k \leq z \leq 2^k - 1$$

Overview: Correctness and Overflow

- We look at what happens when numbers get too big for a computer.
- We see that sometimes we can get unexpected, correct results.

Binary Numbers

- Recall correctness:

$$[\![\vec{s}^{\,bin}]\!] = [\![\vec{b}^{\,bin}]\!] + [\![\vec{b'}^{\,bin}]\!] \qquad (EQ?)$$

- Idea: *Given* $\vec{b}^{\,bin}$ and $\vec{b'}^{\,bin}$, compute $\vec{s}^{\,bin}$ using the Digitwise Alg. THEN check if EQ? is true or false.

- If true, algorithm (or computer) is correct.

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

- On paper: $[\![0111^{\,bin}]\!] = 7 = 3 + 4 = [\![011^{\,bin}]\!] + [\![100^{\,bin}]\!]$.
- 3-bits only: $[\![111^{\,bin}]\!] = 7 = 3 + 4 = [\![011^{\,bin}]\!] + [\![100^{\,bin}]\!]$.

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

- On paper: $\llbracket 1011^{\,bin} \rrbracket = 11 = 5 + 6 = \llbracket 101^{\,bin} \rrbracket + \llbracket 110^{\,bin} \rrbracket$
- 3 bits: $\llbracket 011^{\,bin} \rrbracket = 3 \neq 5 + 6 = \llbracket 101^{\,bin} \rrbracket + \llbracket 110^{\,bin} \rrbracket$

A 3-bit computer would give incorrect answer

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

- $[\![s_{k-1} \dots s_0 {}^{bin}]\!] = [\![b_{k-1} \dots b_0 {}^{bin}]\!] + [\![b'_{k-1} \dots b'_0 {}^{bin}]\!]$
- $0 \leq [\![b_{k-1} \dots b_0 {}^{bin}]\!] + [\![b'_{k-1} \dots b'_0 {}^{bin}]\!] \leq 2^k - 1$

both true or *both* false.

2s-Complement Numbers

Let $k = 2$, with 3 bits for $2s$-complement numbers. There may be a carry of 1, but with the *computer sum (result)* correct!!!

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

Here the computer sum is 101 with $k+1 = 3$ digits/bits. The carry in *position* $k+1 = 3$ is (1).

$$\llparenthesis 101 \rrparenthesis = -3 = \llparenthesis 111 \rrparenthesis + \llparenthesis 110 \rrparenthesis$$

Sign-bit Conditions

- Let P be “the sign bits of the two operands are complementary ($b_k = \overline{b'_k}$)”;
- and let Q be “the sign bits of the two operands are identical, and also the same as the sign bit of the computer sum ($b_k = b'_k = s_k$)”.

If $\boxed{P \text{ or } Q}$ is true, then the computer sum will be correct.

Result = Computer Sum = $k+1$-digit answer (excluding any carry into *position* $k+1$—see previous slide).

The following conditions are *equivalent* for 2s-complement correctness.

- $\llparenthesis s_k \ldots s_0{}^{bin} \rrparenthesis = \llparenthesis b_k \ldots b_0{}^{bin} \rrparenthesis + \llparenthesis b'_k \ldots b'_0{}^{bin} \rrparenthesis$
- $-2^k \leq \llparenthesis b_k \ldots b_0{}^{bin} \rrparenthesis + \llparenthesis b'_k \ldots b'_0{}^{bin} \rrparenthesis \leq 2^k - 1$
- P or Q is true.

Overview: Logical Operations

- We learn about simple logic, and some related functions.
- We shall see how these functions might be used by a computer.

AND and OR

- Typically use 1 to denote truth, and 0 to denote falsity.
- We shall soon use the logic functions $\boldsymbol{.} : \mathbb{B}^2 \to \mathbb{B}$ and $+_{or} : \mathbb{B}^2 \to \mathbb{B}$. Refer to these as **AND** and **OR**.
- We will sometimes write seqeunces $d_k \ldots d_0$ using brackets: $(d_k, \ldots, d_0)$. Call (d_1, d_0) a **pair** and (d_2, d_1, d_0) a **triple**.

- AND input pair (d,d') is mapped to an output written in infix notation $d \,\mathtt{.}\, d'$.

- $d \,\mathtt{.}\, d'$ is 1 precisely when both *d and d′* are 1, and otherwise $d \,\mathtt{.}\, d'$ is 0.

- OR input pair (d,d') is mapped to an output written in infix notation $d +_{or} d'$.

- $d +_{or} d'$ is 1 precisely when at least one of *d or d′* are 1, and otherwise $d +_{or} d'$ is 0.

Truth Tables

We give the functions $\mathbf{.}$ and $+_{or}$ by explicitly listing inputs and outputs in tables; in work on digital circuits we often use A, B and C to stand for binary digits 0 or 1. The tables give $(A,B) \mapsto A +_{or} B$ where $+_{or} : \mathbb{B}^2 \to \mathbb{B}$, and similarly $\mathbf{.}$.

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

A	B	$A \mathbf{.} B$
0	0	0
0	1	0
1	0	0
1	1	1

Zero Extension

- Sometimes, a binary number will be stored in k bits.
- We will then want to move the number into $k' + k$ bits, so that *its denotation is not changed.*
- We fill the extra bits with k' zeros. This is called a **zero extension**.
- We can model it as a function

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

where $\vec{0}$ denotes k' zeros.

- We will sometimes write the output as $zx(\vec{b})$.

Sign Extension

- To move a $k+1$ bit 2s-complement number $\delta\vec{b}$ into $k'+k+1$ bits copy the sign bit into the new k' positions. This is called a **sign extension**.

- We can model it as a function

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

where $\vec{\delta}$ means a sequence of k' δs.

- We will sometimes write the output as $sx(\delta\vec{b})$.

Digital Electronics

- Learn about circuits which perform calculations; and
- memory circuits that will store data.
- We learn how to design circuits, and how they work.

Overview: Switching Algebras and Basic Circuits

- We describe the *Switching Algebra* which gives a mathematical model of simple circuits.
- Then we show how to use such models to design and implement digital circuits.

The Switching Algebra

- The **switching algebra** consists of the following five things: $(\mathbb{B}, \mathbf{.}, +_{or}, \overline{}, 0, 1)$.
- The functions satisfy certain properties.

Property	$+_{or}$
Idempotent	$A +_{or} A = A$
Complement	$A +_{or} \overline{A} = 1$
Associative	$A +_{or} (B +_{or} C) = (A +_{or} B) +_{or} C$

- $\overline{A}\mathbf{.}B +_{or} C$ stands for $((\overline{A})\mathbf{.}B) +_{or} C$.

Implementing Functions by Circuits

- We implement $n \mid m$-ary functions over $\mathbb{B}$ as digital circuits. For example, $f\colon \mathbb{B}^3 \to \mathbb{B}^4$ and $(1,0,1) \mapsto (1,1,0,0)$,
- We shall represent such circuits with pictures.
- The horizontal **input lines** denote wires with a voltage.
- Each voltage indicates a binary digit, which will be one of the function’s input components.

- We sometimes *label* input and output lines, eg by A.

- We talk about the "input line" A to mean the physical wire into the circuit.

- The line A **carries** or **holds** a binary digit **value**.

- A **gate** is defined to be an electrical circuit which computes certain simple functions over $\mathbb{B}$.

- An **AND** gate computes **.**; there are other gates . . .

- There are also $m \mid 1$-ary versions of $\centerdot$ and $+_{or}$. If $m = 3$ then $\centerdot : \mathbb{B}^3 \to \mathbb{B}$ and we write $A \centerdot B \centerdot C$ for the output to input (A, B, C). The output $A \centerdot B \centerdot C = 1$ only when $A = 1$ and $B = 1$ and $C = 1$.

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

- We want to show how to build (implement) a circuit for an $n \mid m$-ary function.

- Each such function is equivalent to m different $n \mid 1$-ary functions …see pictures from the lecture.

- Thus we just demonstrate by example how to implement $n \mid 1$-ary functions.

- We start from a truth table …

Step 1: Sum-of-Products Example

We look at an example of a 2 | 1-ary function:

A	B	$g(A,B)$
0	0	1
0	1	0
1	0	0
1	1	1

We construct a **Sum-of-Products** expression . . .

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

Step 2: Implement S-o-P Example

- We have

$$g(A,B) = \overline{A} \cdot \overline{B} +_{or} A \cdot B$$

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

- It is easy to make a circuit to implement any $n \mid 1$-ary function given by a Sum-of-Products expression. See the notes and lectures.

Simplifying Logic Expressions

Circuits may not be optimal:

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

Overview: Combinational Circuits

- A circuit which implements an $n \mid m$-ary *function* is known as a **combinational** circuit.
- We shall look at specific, useful combinational circuits:
- Circuits for addition; for subtraction; for *joining circuits together* to build a CPU and so on.

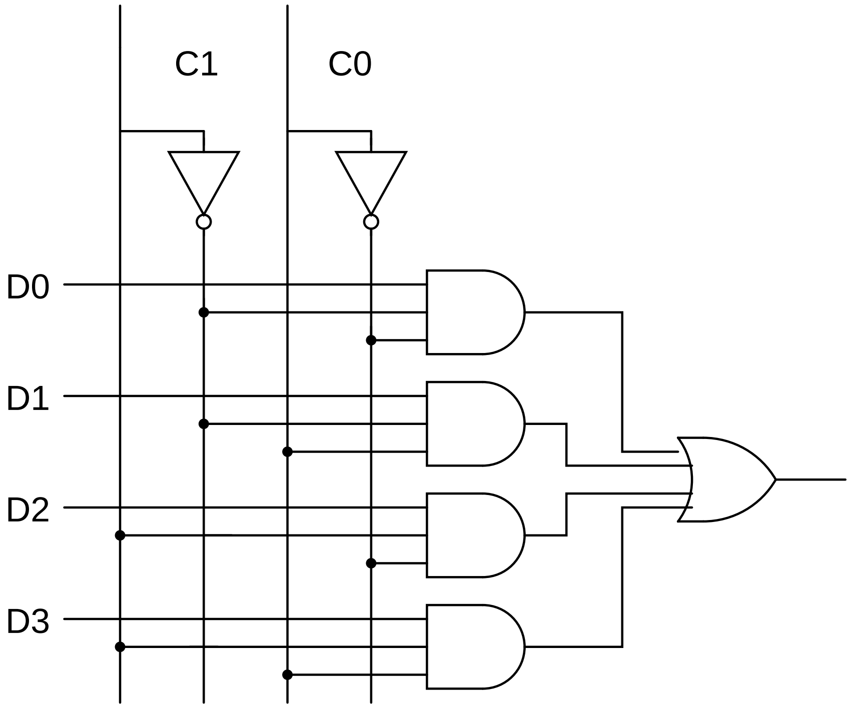
Multiplexor

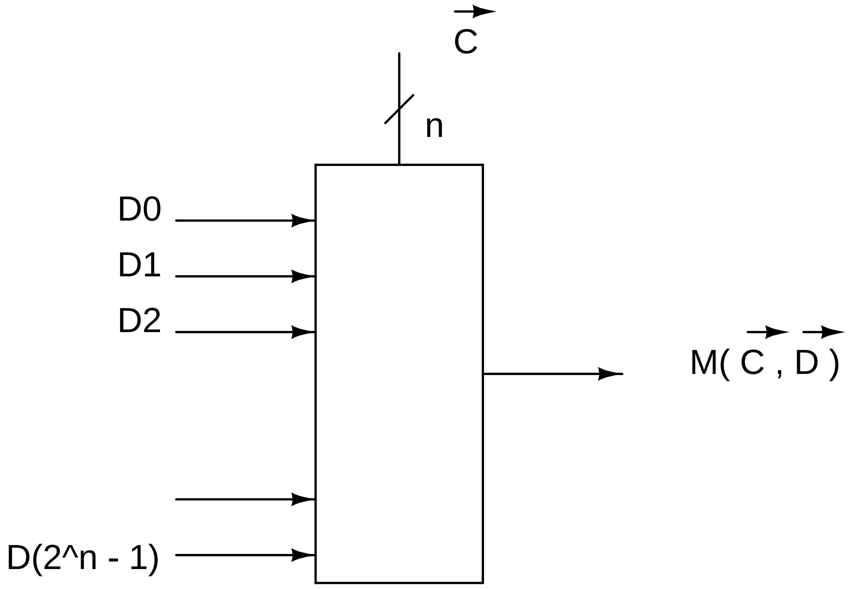
- A **multiplexor** has $n+2^n$ input lines which divide up into n **control** input lines and 2^n **data** input lines.

- Truth table when n is 3 ($d_i \in \mathbb{B}$)

C_2	C_1	C_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	M
1	0	1	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0	d_5

- Value of output is $D_{[\![C_2C_1C_0]\!]}$. Thus we write $M(\vec{C},\vec{D}) = D_{[\![C_2C_1C_0]\!]}$.


C1
C0
D0
D1
D2
D3


$\vec{C}$
n
D0
D1
D2
D(2^n - 1)
M($\vec{C}$, $\vec{D}$)

Decoders

- A **decoder** has n input lines and 2^n output lines.
- For each of the 2^n input tuples, one of the output lines is set to 1 and the others are set to 0.
- Usually output line $L_{[\![\vec{b}]\!]} = 1$ where $\vec{b}$ is the input tuple.
- Use a decoder to **enable/disable** one of many circuits.

Clocks

- In many digital circuits, the timing of various processes and tasks is crucial.
- A **pulse** is a voltage of 1 which lasts for a short time.
- A **clock** is a circuit which generates a series of pulses.
- Each pulse lasts for fixed time, and there is a fixed interval between pulses during which the voltage produced is 0.

- The production of such a high and low voltage is called a **clock cycle**.
- The total time taken for this is called the **clock cycle time** or **period**.
- The start time of a clock pulse is called a **rising edge**; the **falling edge** is when the pulse ends.

Adders

- We want a circuit (**full adder**) to perform addition of single binary digits.
- It will have three input lines for $carry_i$, and d_i and d'_i.
- It will have two output lines for s_i and $carry_{i+1}$.

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

Arithmetic Logical Units

- Two *Data Input k-bit busses* (each holding a k-digit binary number $\vec{a}$ and $\vec{b}$).

- One *Output k-bit bus* value (holds *result* $\vec{r}$, given by a selected function applied to the two k-bit integers).

- One *Control Input bus* which will be used to *select* the ALU function. **KEY IDEA.**

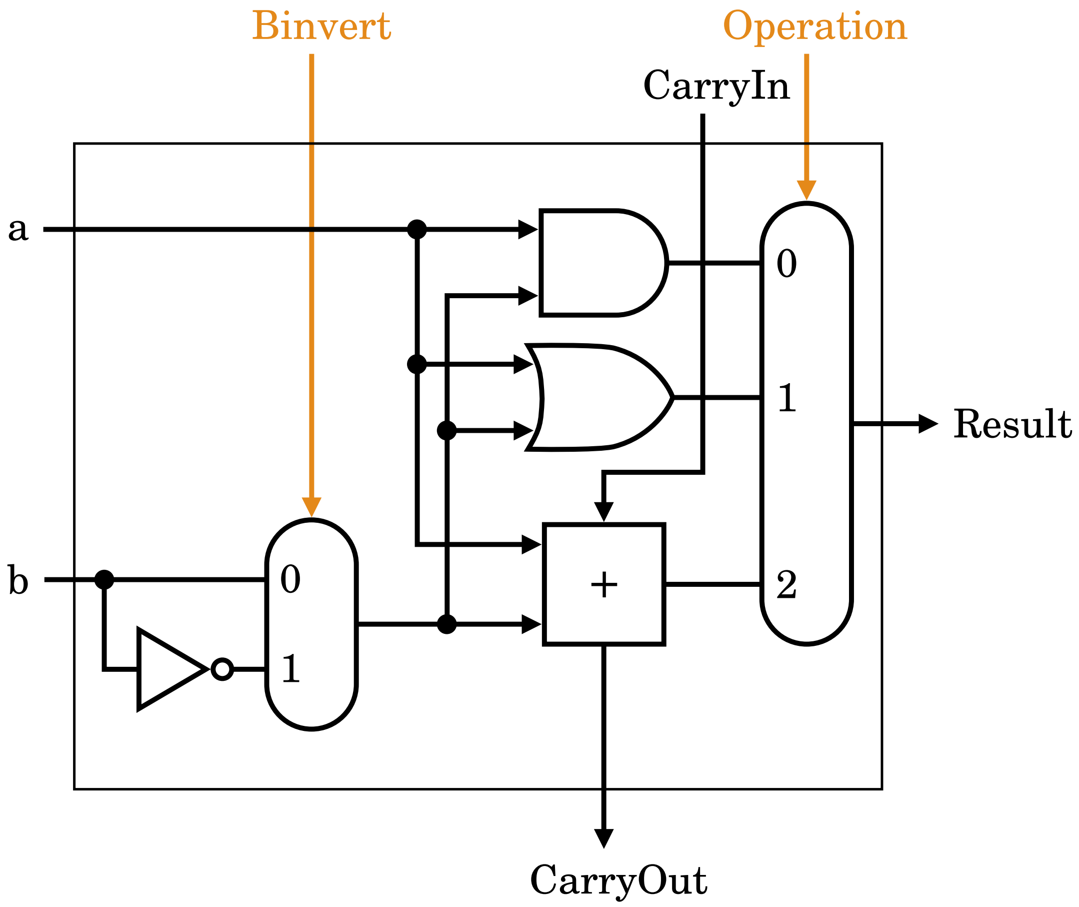
- We shall design an ALU where $k = 32$, with functions from Chapter 3, and ...

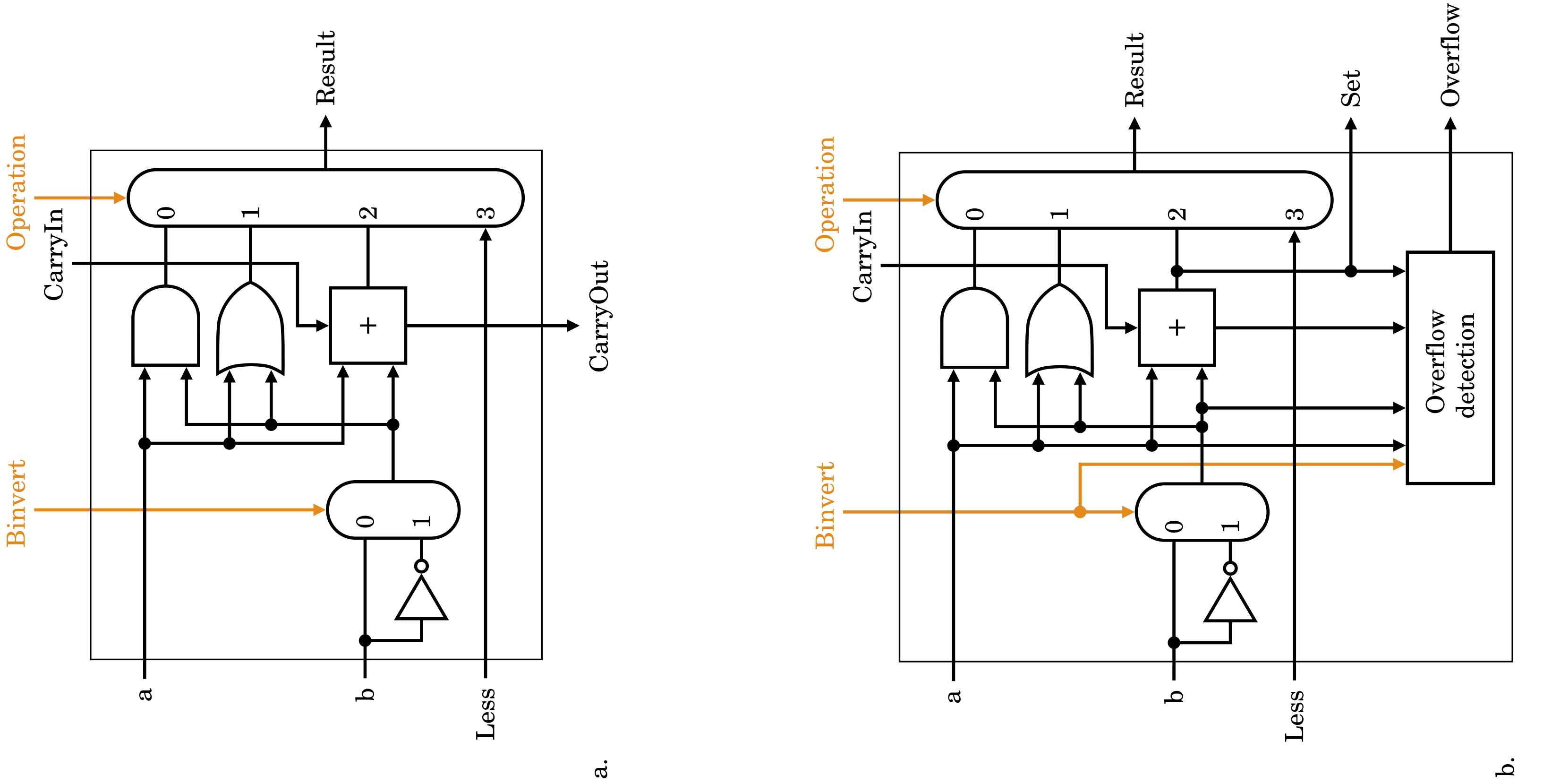
- the "set on less than" `slt` function which returns

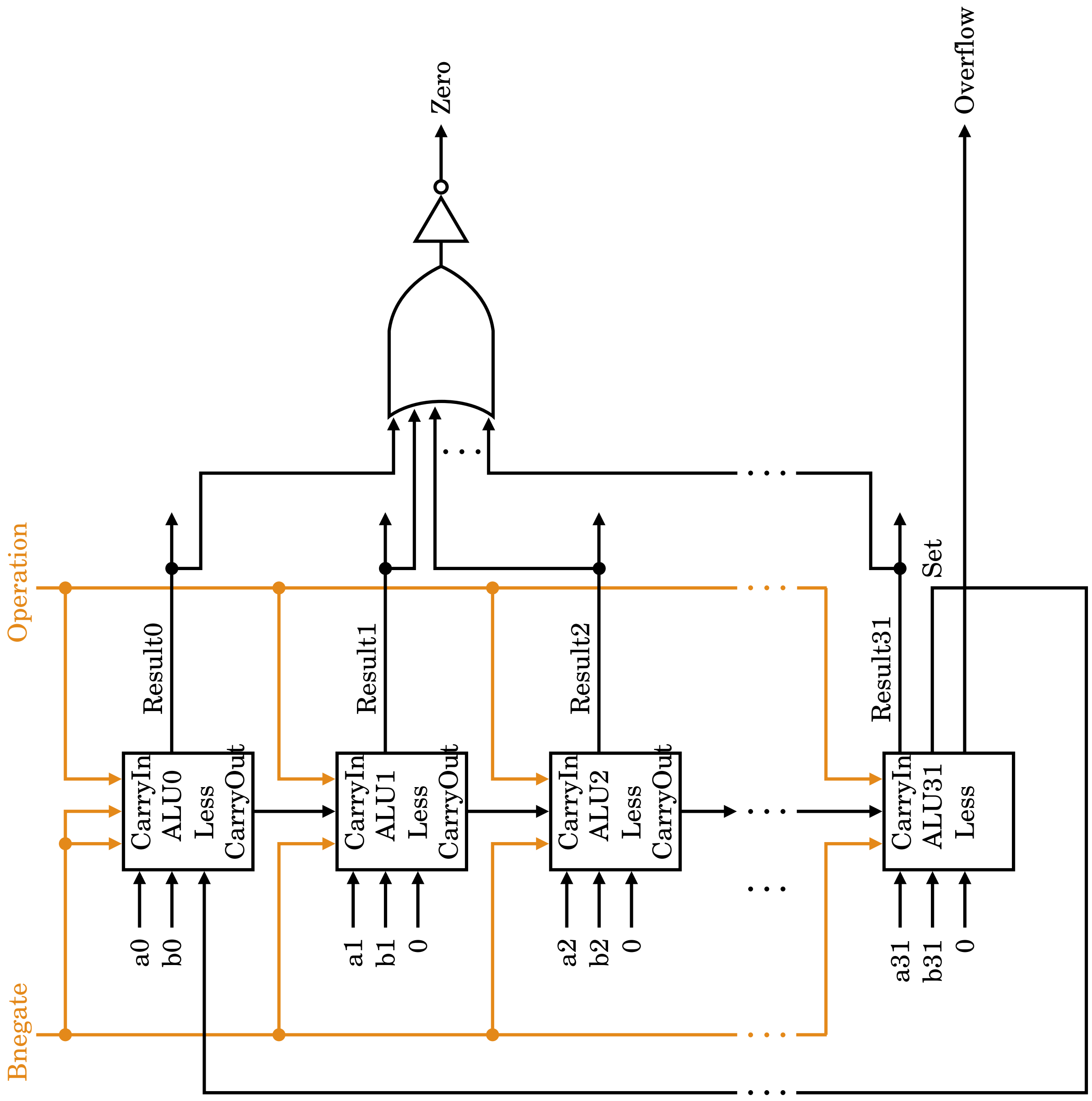
$$\begin{aligned} r_{31} = 0, \ldots, r_1 = 0, r_0 = 1 &\quad \text{if} \quad (\!|\vec{a}^{\,bin}|\!) < (\!|\vec{b}^{\,bin}|\!) \\ r_{31} = 0, \ldots, r_1 = 0, r_0 = 0 &\quad \text{if} \quad \text{otherwise} \end{aligned}$$

- A single bit output line `Zero` which returns 1 when *any* result is $\vec{0}$, and 0 otherwise.

- A single bit output line which will hold 1 when a function causes overflow, and will hold 0 otherwise.


Binvert
Operation
CarryIn
a
b
0
1
+
0
1
2
Result
CarryOut


Operation
CarryIn
Result
0
1
2
3
+
CarryOut
Binvert
0
1
a
b
Less
a.
Operation
CarryIn
Result
Set
Overflow
0
1
2
3
+
Overflow detection
Binvert
0
1
a
b
Less
b.


Zero
Overflow
Operation
Bnegate
Result0
Result1
Result2
Result31
Set
CarryIn
ALU0
Less
CarryOut
CarryIn
ALU1
Less
CarryOut
CarryIn
ALU2
Less
CarryOut
CarryIn
ALU31
Less
a0
b0
a1
b1
0
a2
b2
0
a31
b31
0

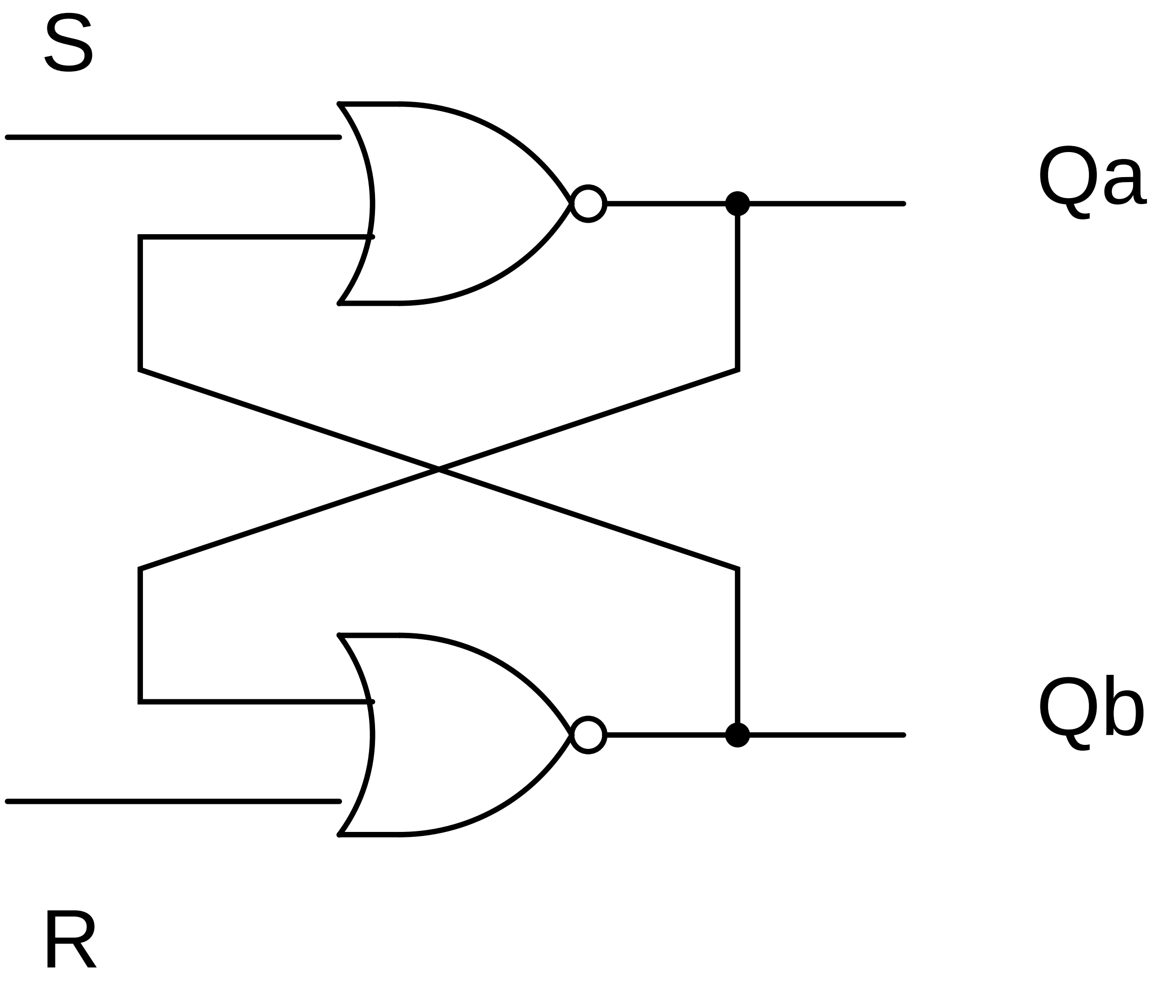
Overview: Sequential Circuits

- Sequential circuits may have different “ouputs” for the same “input”.
- (They implement *relations*.)
- These circuits can be used to build digital memory.

Sequential circuits

- Combinational circuits are functions.
- A **sequential** circuit can have many allowed outputs for a given input.
- Such circuits implement not functions, but *relations*.
- For such “allowed” input and output, the circuit is **stable**.
- Otherwise the circuit is **unstable**.
- We shall require a NOR gate to give examples ...

SR Latches



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

Stable States

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

Unstable States

S is the **set** line and R the **reset** line. Q_b is a signal which can be set (to 1) and reset (to 0). Q_b will be used as a 1 bit memory.

Clocked D Latches

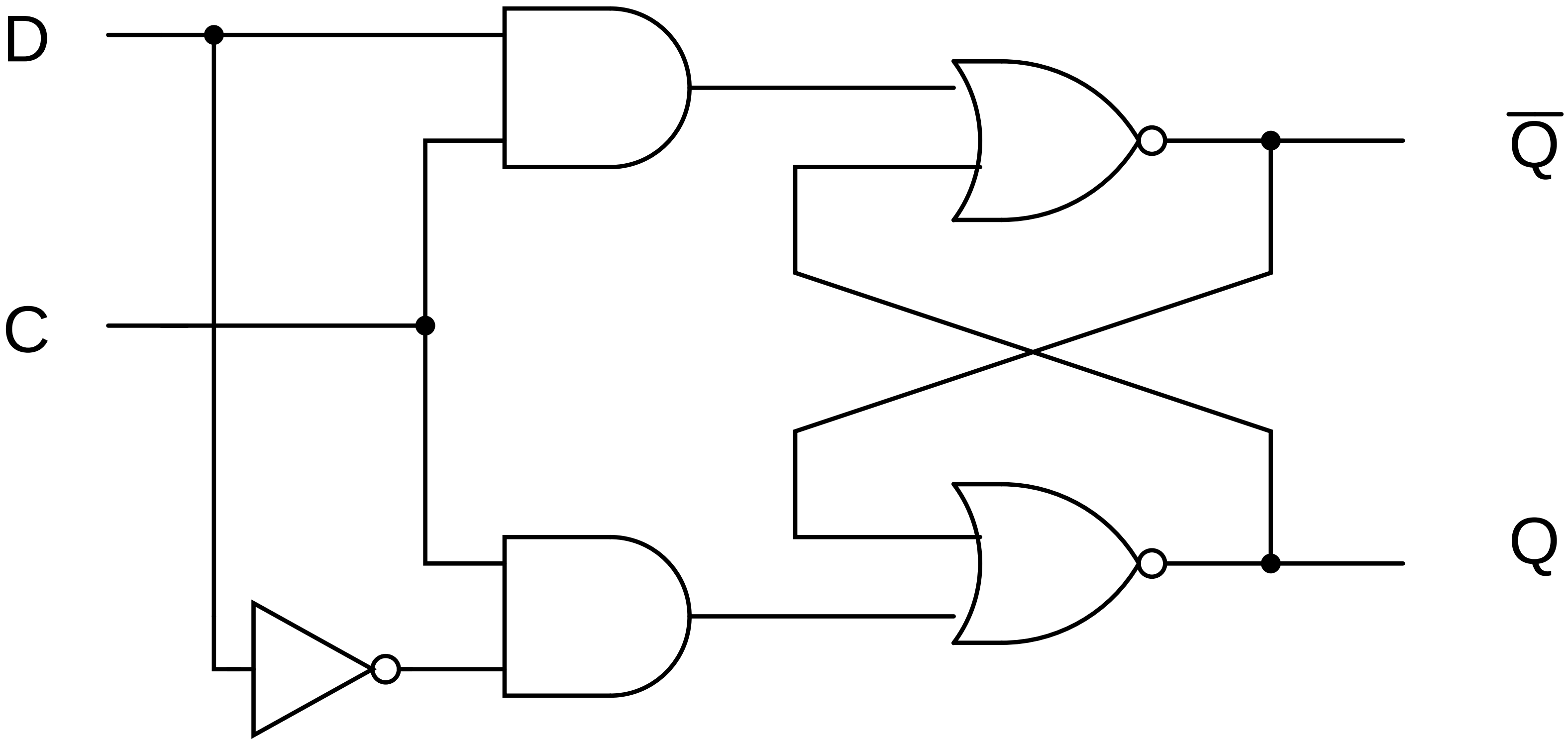
- A model of 1**-bit computer memory** is a pad-locked box storing 0s and 1s. The box has a glass window.

- *Note: The words **input/write/store** mean the same thing. The same goes for **output/read**.*

- **data line** *D* provides the data (0 or 1) which we *want* to store.

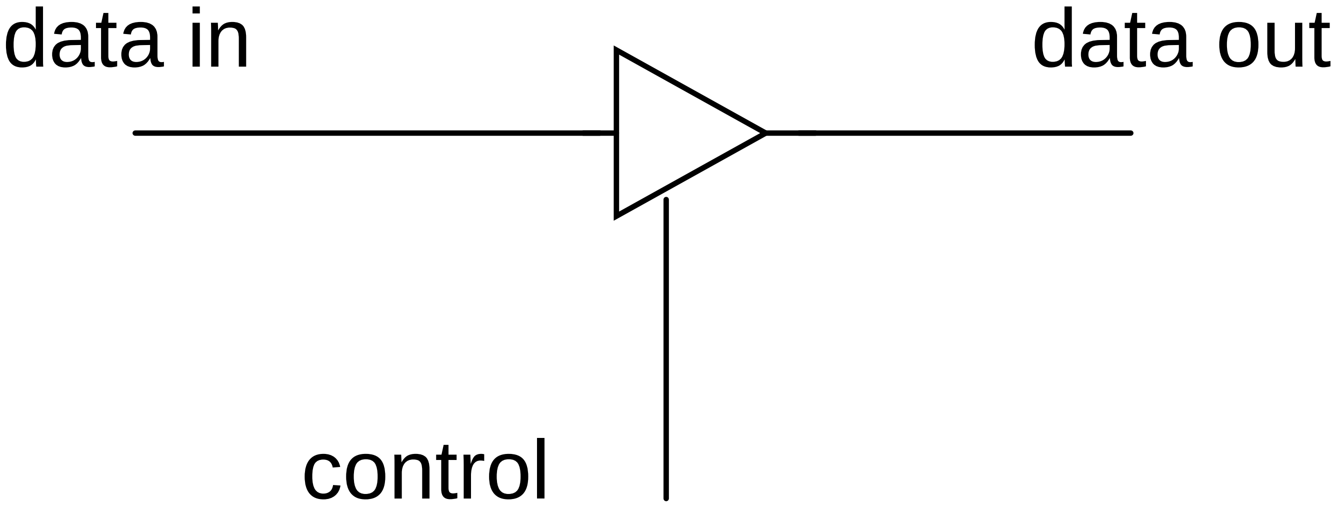
- **clock line** indicates when data can and cannot be stored—a **write enable** line.

- *Q* holds the stored data.


D
C
$\overline{Q}$
Q

Tri State Buffers

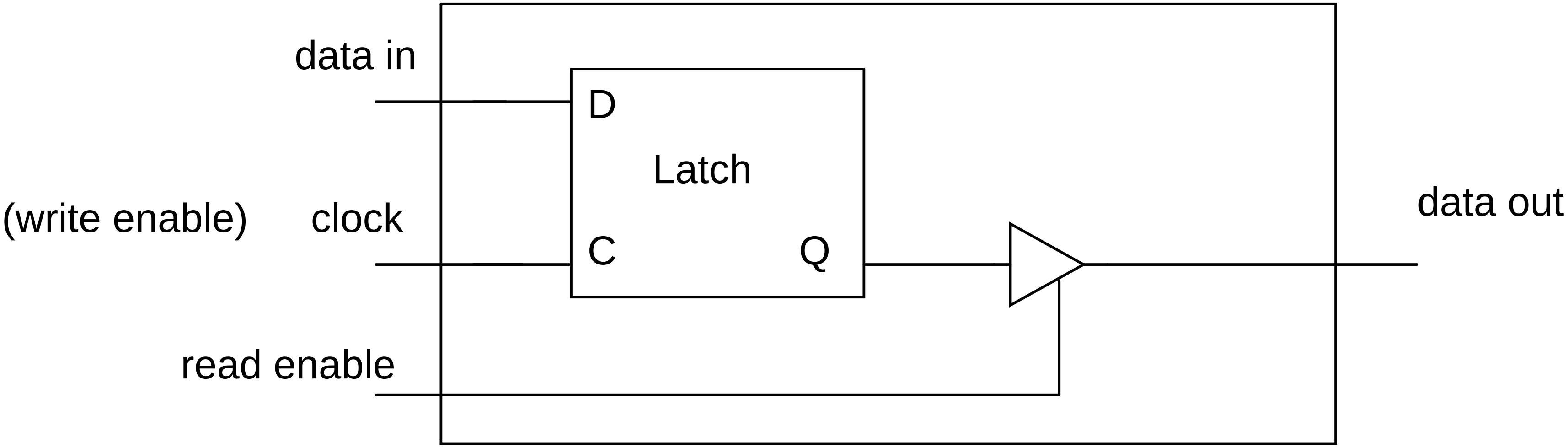
- A Tri State Buffer is an electronic switch.
- We use it to make a **read enable** line for a 1-bit memory.



Example

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

Answer:



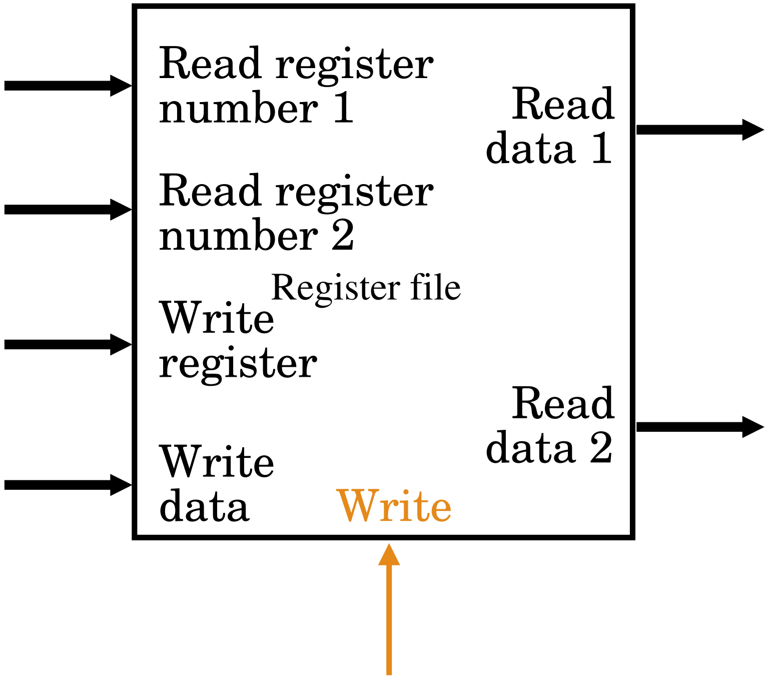
Register Files

■ A *k*-bit register is *k* 1-bit memories which have their clock lines wired together to form a **write** enable line.

■ A **register file** (for a CPU) consists of

- a set of *k*-bit registers;
- a set of **read-register** busses and a set of **read-data** busses;
- a set of **write-register** busses and a set of **write-data** busses; and
- a **write enable** line.

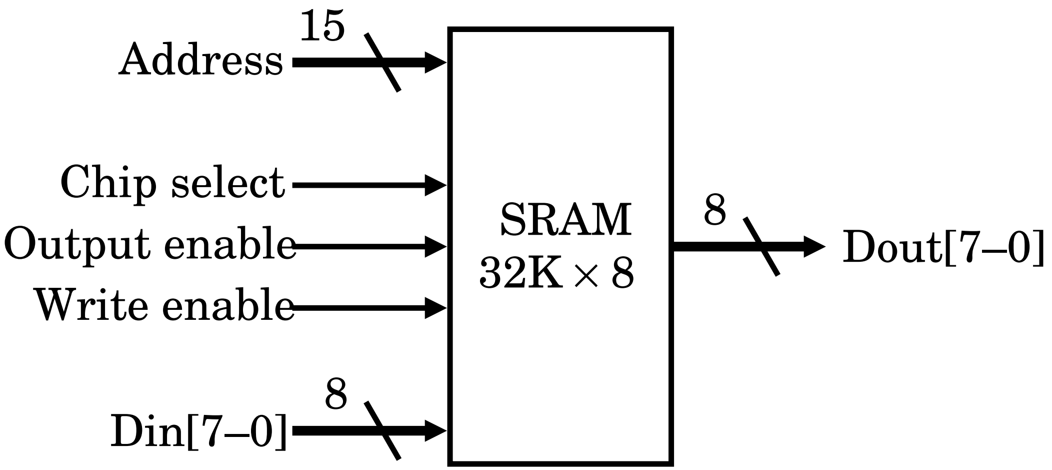
Reading from/ Writing to the Register File:



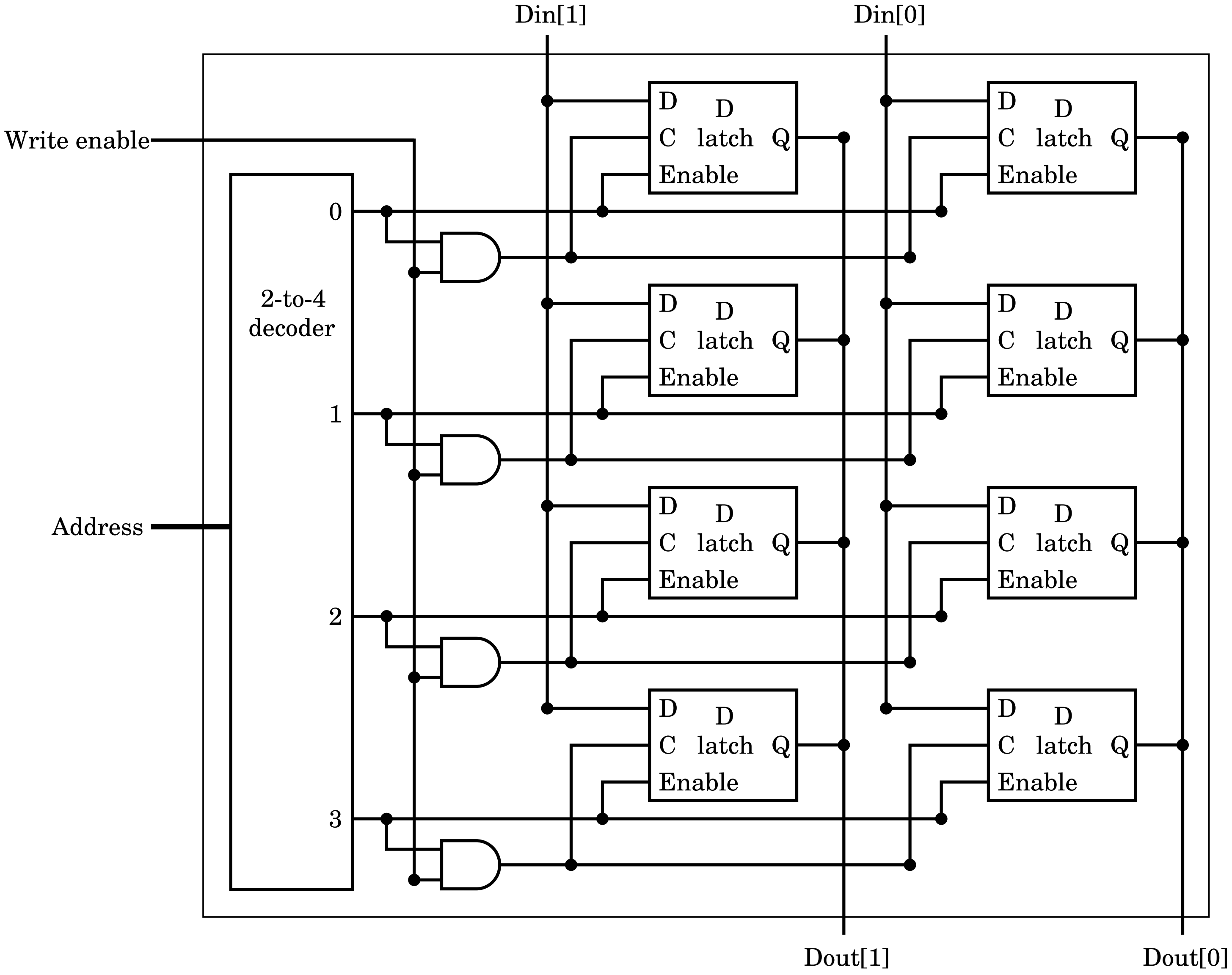
Building Static Random Access Memory

Computers count in binary.

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



- The number of bits in a cell is sometimes called the chip's **width**, and the number of cells its **height**.
- We talk about an $h \times w$-SRAM.


Din[1]
Din[0]
Write enable
Address
2-to-4
decoder
0
1
2
3
D
C
Enable
D
latch
Q
Dout[1]
Dout[0]

The Instruction Set Architecture Level

- We describe in detail the instructions (ISA) which can be executed by a processor . . .
- Study ISA assembly language.
- Study ISA machine language.
- Look at simple ISA programs.

Overview: Introducing the MIPS ISA

- Describe a MIPS processor.
- Explain the MIPS ISA language by example.

ISA and MIPS Basics

- Recall

```
            a := 0;  c := 4;
L           a := a+c;
            c := c-1;
            if c >= 1 then L;
```

- We introduce the **MIPS** ISA. It has instructions with syntax like `add $a, $a, $c`.
- Such instructions run on the *MIPS R2000* processor.
- This has a register file which contains 32 registers.

- MIPS R2000 provides a 32-bit computer:
  - CPU registers have 32 bits.
  - Instructions stored in 32-bit main memory word locations.
  - Main memory cells have 8 bits: byte locations.
- Each register is denoted by a special (assembly language) symbol. Eg `$s4`.

- We identify a register by its assembly symbol (eg `$s4`) or **register number** (CPU address), 0 to 31.
- The binary representations of 0 to 31 are the machine language register numbers for the 32 registers.
- *Note: The MIPS R2000 has 32 registers, and each register has 32 bits. Do not let this confuse you!*

- MIPS R2000 instructions have two forms, *machine language* and *assembly language*.
- MIPS ISA assembly example `add $t0`, `$t1`, `$t2`.
- There is a machine language form

–	`$t1`	`$t2`	`$t0`	–	*add*
000000	01001	01010	01000	00000	100000
6 bits	*5 bits*	*5 bits*	*5 bits*	*5 bits*	*6 bits*

- Each section is called a **field**.

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

Overview: MIPS Assembly Language

- You will understand the details of CPU registers.
- You will learn about different ways of reading/writing data between CPU and main memory.
- Learn some assembly instructions; SYNTAX and SEMANTICS.
- Show that the instructions can be grouped into *categories*.

Registers, Locations and Assignment

- R denotes any MIPS register.
- To describe ISA semantics we shall use **assignments**.
- An assignment takes the form $R := \omega$ (eg `$t1` $:= \vec{0}$)
- If the register has k bits, the word ω is a sequence of k binary digits.
- We will stick to $k = 32$.

- We will adopt a convention. We may write things like $R := 7$. The convention is that this will be shorthand for $R := \vec{0}111^{bin}$.

- We can describe the semantics of the instruction `add` R_1, R_2, R_3 as

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

- $!R_2+!R_3$ is given by (32 digit) digitwise sum.

- $B[a]$ is the byte location at main memory address a.
- We have a similar notation $W[a]$ for word locations.
- Program instructions stored in consecutive word locations in main memory.
- Recall cells are 8 bits. So addresses of word locations are multiples of 4. (IMPORTANT!!)
- The MIPS R2000 is big endian format.

Introduction to Addressing

- *Addressing* refers to the ways in which data is read/written.
- Data copied from main memory into a register is **loaded**.
- Data copied from a register into main memory is **stored**.
- These forms of copying are **data transfers**.

- Consider the two instructions

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

- We refer to `add` and `addi` as instruction **names**.

- The registers R_i and the integer 5 are called instruction **arguments**.

- The arguments tell us *where* data is read from or written (stored) to.

- R_2 and R_3 contain data to be (read and) added. We call them **source** arguments.
- R_1 specifies where the result is to be stored. We shall call it a **destination** argument.
- Finally, we refer to the numbers to be added as **operands**.

Immediate Addressing

- The data is specified as part of the instruction.
- For example 24 in `addi $t0`, `$t1`, 24.
- The semantics of the instruction is given by $\texttt{\$t0} := !\texttt{\$t1} + 24$.
- We sometimes refer to data (operands) given by immediate addressing as **constants**.
- $+$ means digitwize sum; 24 is stored in binary.

Register Addressing

- The data is given as the contents of a register.
- If the register is called *R*, the data is !*R*.
- `add` uses register addressing for its source arguments.
- The semantics of `add $t0`, `$t1`, `$t2` is

$$\texttt{\$t0} := !\texttt{\$t1} + !\texttt{\$t2}$$

Register Indirect Addressing

- The data specified by R is $!W[!R]$ or $!B[!R]$
- In this case, we refer to R as a **pointer**.

Indexed Addressing

- Often need to load a sequence of words from memory, all “quite near” to a **base** address k.
- An **offset** indicates how far data is located from the base address. The offset equals $!R$.
- Instructions have a source argument $k(R)$.
- The source data is given by the contents of $W[!R+k]$.

- k has denoted any integer.
- So $!R+k$ *means add the 32 digits of* $!R$ *to a 32 digit 2s-complement representation of* k.
- The machine code field for $k \in \mathbb{Z}$ is only *16 bits*!!
- A 16-bit representation of k will be copied into a 32-bit ALU. This will involve a sign extension.
- Note that

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

Arithmetic Category Instructions

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

See example …

Data Transfer Instructions

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

Conditional Instructions

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

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

Overview: MIPS Machine Language

- Explain how to represent the ISA assembly instructions inside the machine as binary digit sequences (machine code).
- Each digit sequence is made up of special parts, called fields. We give an algorithm for working out what the fields are.
- Look at how to translate branch labels into actual machine addressses.

Instruction Fields

- Given any MIPS instruction, what is the corresponding 32-bit machine language instruction?
- Each machine instruction belongs to one of three **formats**. These are know as **R**, **I** and **J** formats.
- Each format has a **field layout**, specifying how the 32 bits are divided up into sections known as **fields**.

R-Format

Here is a field layout for *name* R_1, R_2, R_3

	Fields					
	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	$\vec{0}$	*name*
	$\vec{0}$	`$s1`	`$s2`	`$t0`	$\vec{0}$	`add`
	000000	10001	10010	01000	00000	100000

- For R-format instructions the Opcode (op) is 0^{dec}.

- The Function field (funct) is derived from the instruction name `add`. There is a look-up table in the notes . . .

- The rs and rt fields specify the *numbers* of the source registers (17^{dec} and 18^{dec})

- The rd field, 8^{dec}, specifies the *number* of the destination register.

- Finally, the shamt field is ALWAYS set to 0^{dec} in CO1016.

I-Format

Here is a field layout for *name* $R_1,\ k(R_2)$

op	rs	rt	address
6 bits	*5 bits*	*5 bits*	*16 bits*

`addi`, `andi`, `ori`, `lw`, `sw`, `lb`, `lbu`, `sb`	R_2	R_1	k

Translating Assembly to Machine Language

- An **assembler** is a program that will translate assembly language into machine language.
- We need the field layouts we have just looked at; and
- a **field translation/look-up table** on page 78 of notes; and
- a **register number table**.

Register number table on page 79 of the notes.

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)
`$t0`	8	temporary (not preserved across call)
`$t1`	9	temporary (not preserved across call)
`$t2`	10	temporary (not preserved across call)
`$fp`	30	frame pointer (preserved across call)

Example

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 number table for `$s1`: 17 and for `$t1`: 9.

6 bits	*5 bits*	*5 bits*	*16 bits*
`lw`	R_2	R_1	k
100011	01001	10001	0000000000001000

Machine Code for Branches

- Recall `beq` R_1, R_2, L.
  - $!R_1 =!R_2$ TRUE: the *label* `L` *“points”* to the next instruction to be executed.
  - $!R_1 =!R_2$ FALSE: execute *next instruction in memory*.
- Such instructions are I-format:

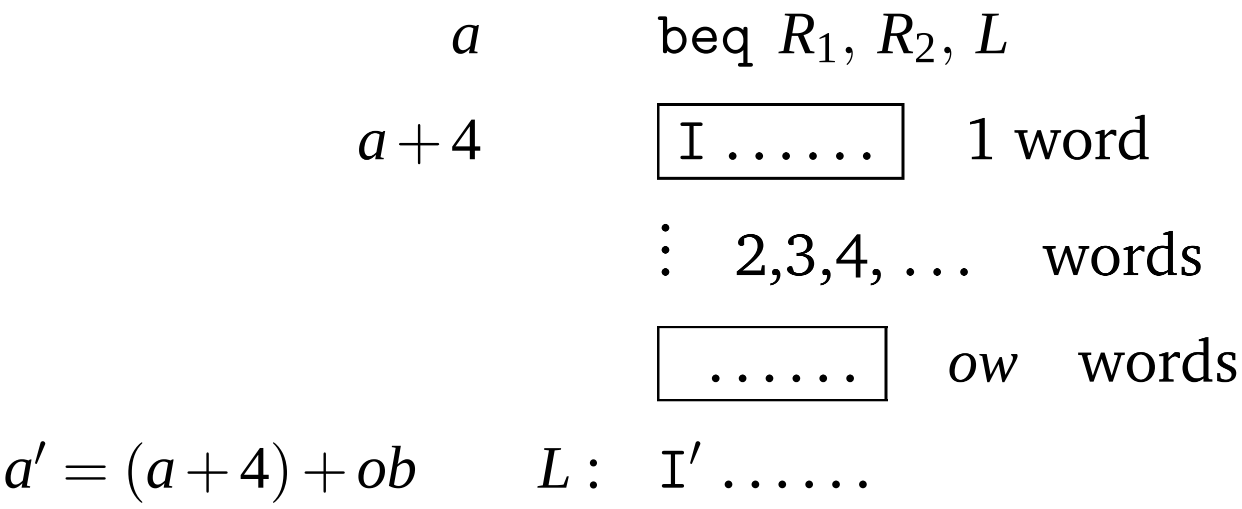
op	rs	rt	address
6 bits	*5 bits*	*5 bits*	*16 bits*
`beq`	R_1	R_2	L

- Looking up the opcode and register numbers is easy!
- The address field specifies the distance (offset) *ow* in <u>words</u> from the address of the *next instruction* `I` *in memory* to the address of the labelled **instruction** $\mathtt{I}'$

$$\begin{array}{rrll}
a & & \mathtt{beq}\ R_1,\, R_2,\, L & \\
a+4 & & \boxed{\mathtt{I}\ldots\ldots} & 1 \text{ word} \\
 & & \vdots \quad 2,3,4,\ldots & \text{words} \\
 & & \boxed{\ \ldots\ldots} & ow \quad \text{words} \\
a' & L: & \mathtt{I}'\ldots\ldots &
\end{array}$$

- To be precise $ow = \llparenthesis \text{address} \rrparenthesis \in \mathbb{Z}$ words. *Thus we can work out the address field.* See my example on the OHP … if $ow = 2 \in \mathbb{Z}$, address =00000000.00000010 $\in \mathbb{B}^{16}$.

- What about the address a' of the labelled instruction?



a `beq` R_1, R_2, L

$a+4$ $\boxed{\texttt{I} \ldots\ldots}$ 1 word

$\vdots$ 2,3,4, … words

$\boxed{\ldots\ldots}$ ow words

$a' = (a+4) + ob$ L : `I`$'$ ……

- Addresses count cells (bytes); each word location consists of four cells. So

$$ob = 4 * ow = 4 * \llparenthesis \text{address} \rrparenthesis \in \mathbb{Z}$$

Branch Semantics at Run Time

- The last section explained how to work out the machine code for a branch instruction with label L by using the position of the instruction labelled L.
- Suppose we know the machine code for a branch. *What happens at run time?* Recall the SEMANTICS

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

- As `beq` R_1, R_2, L STARTS to execute, suppose the PC contains $a \in \mathbb{Z}$.

- The effect of execution is to specify the *next* instruction to be executed. So the *PC should be updated* to

$$\begin{array}{ll} a' = (a+4)+ob \in \mathbb{Z} & \text{if test TRUE} \\ a+4 \in \mathbb{Z} & \text{if test FALSE} \end{array}$$

- The new binary value (in $\mathbb{B}^{32}$) of the PC is given by

$$\begin{array}{llll} \mathtt{PC} & := & (!\mathtt{PC}+4)+(4*\text{address}) & \text{if test TRUE} \\ \mathtt{PC} & := & (!\mathtt{PC}+4) & \text{if test FALSE} \end{array}$$

Chapter 6

- Build a datapath in which each ISA instruction is executed in one clock cycle.
- Design control for this.

In notes; non-examinable:

- Build a datapath in which each ISA instruction requires many clock cycles for execution, but the amount of hardware is reduced.
- Design control for this.

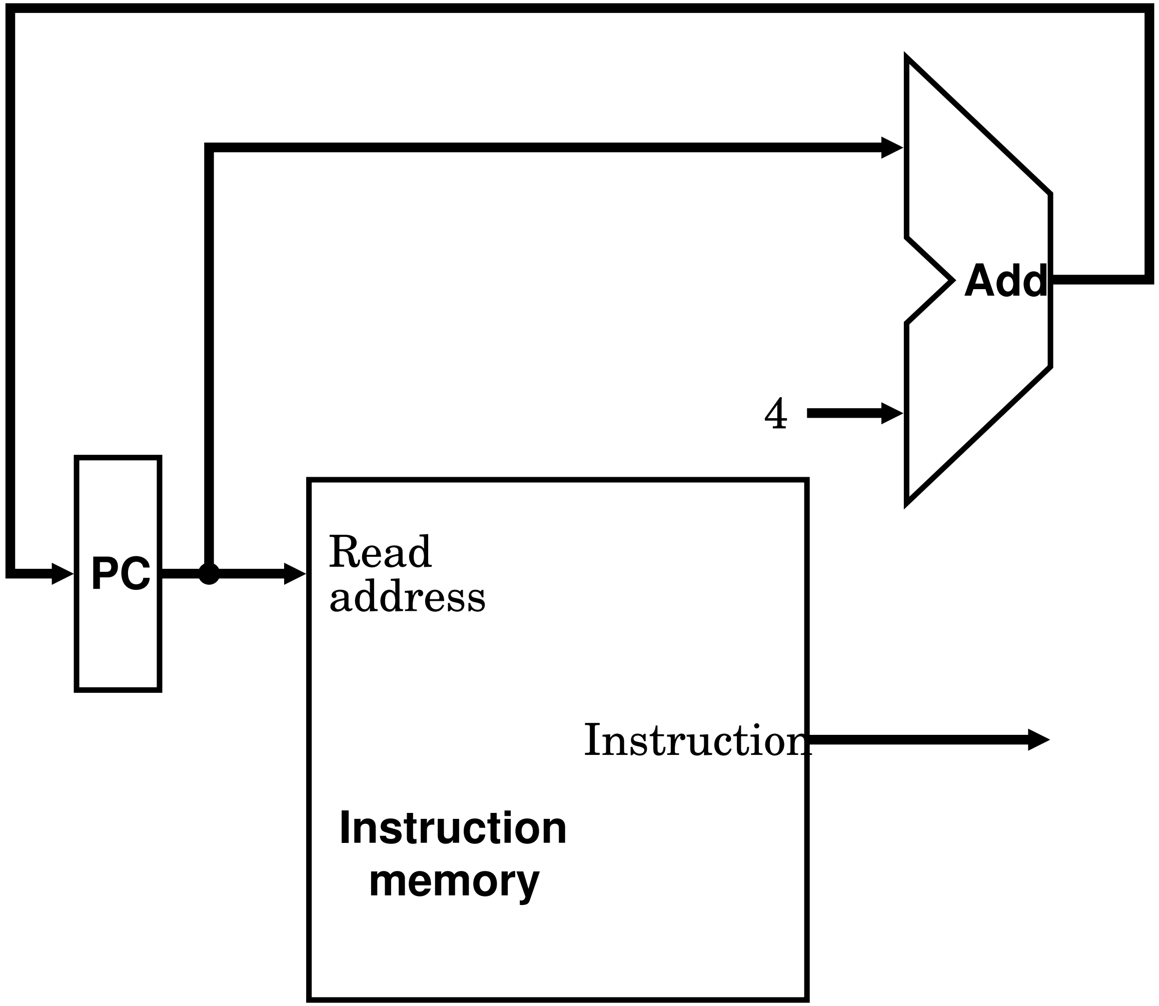
Introduction

- Show how to build a small CPU = datapath + control.
- Design of the CPU is called its **architecture**.
- **Micro** refers to CPU design, not overall computer design.
- We shall develop a micro architecture for
  - The R-format instructions `add`, `sub`, `and`, `or`, and `slt`;
  - the I-format instructions `lw`, `sw`, and `beq`.

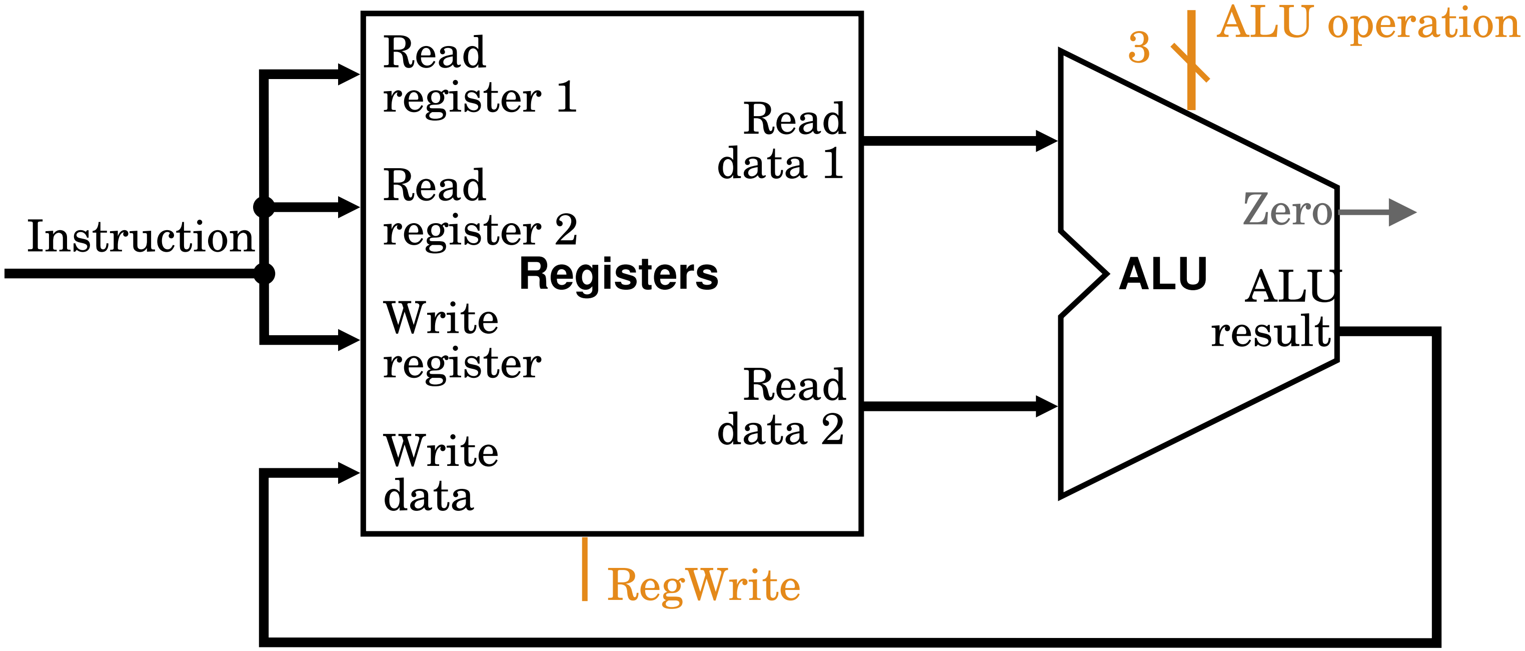
Overview: Datapath Components

- We give examples of the circuits in a datapath responsible for the execution of different categories of instruction.

Basic Instruction Fetch and PC Increment



R-Format Instructions



6 bits	*5 bits*	*5 bits*	*5 bits*	*5 bits*	*6 bits*
$\vec{0}$	$\#R_2$	$\#R_3$	$\#R_1$	$\vec{0}$	*name*

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

- Machine instruction $\mathtt{I}[31-0] \in \mathbb{B}^{32}$ passed along 32-bit wide `Instruction` bus; the fields are divided up . . .

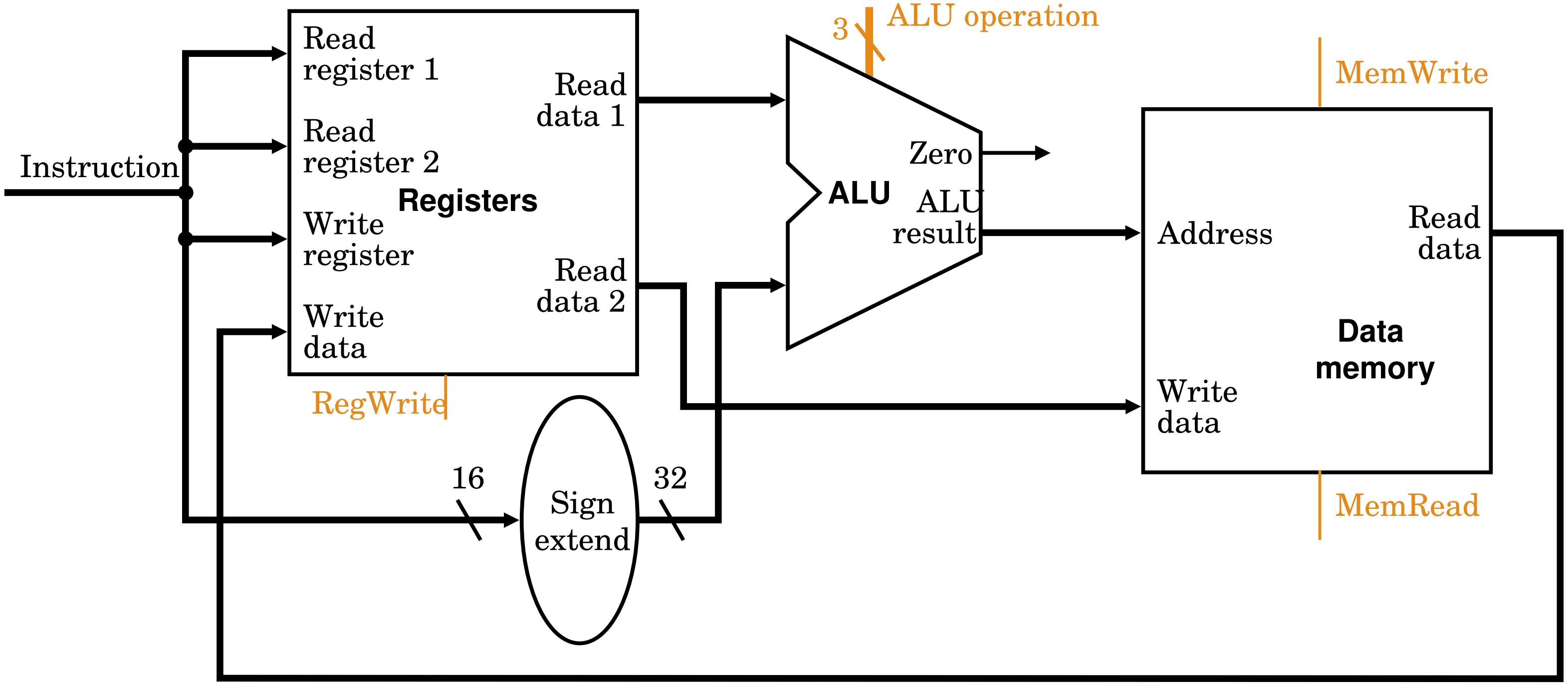
- The numbers of R_2 and R_3 ie $\mathtt{I}[25-21] \in \mathbb{B}^5$ and $\mathtt{I}[20-16] \in \mathbb{B}^5$ are passed along 5-bit `Read register 1` and `Read register 2`. These are source registers.

- Destination register number $\mathtt{I}[15-11] \in \mathbb{B}^5$ passed along 5 bit `Write register` bus.

- Source operands $!R_1$ and $!R_2$ appear on the 32-bit `Read data 1` and `Read data 2`

- ALU operation is set by control and result is written . . .

Load/Store Instructions



6 bits	5 bits	5 bits	16 bits
name	$\#R_2$	$\#R_1$	k

- We examine `lw` $R_1, k(R_2)$ where $R_1 := !W[!R_2 + sx(k)]$.
- The number of register R_2 ie $\mathtt{I}[25-21] \in \mathbb{B}^5$ goes to `Read register 1`
- 16-bit representation of k ie $\mathtt{I}[15-0] \in \mathbb{B}^{16}$ goes to `Sign extend`.
- Contents of R_2 go via `Read data 1` to ALU, along with sign extended k.
- `Data memory` gets `Address` $a \overset{\text{def}}{=} !R_2 + sx(k) \in \mathbb{B}^{32}$.
- `Data memory` sends the contents of the word at a via `Read data` into register file `Write data`.
- Register number of R_1 ie $\mathtt{I}[20-16] \in \mathbb{B}^5$ is in `Write register`, so $!W[a] \in \mathbb{B}^{32}$ is written in R_1.

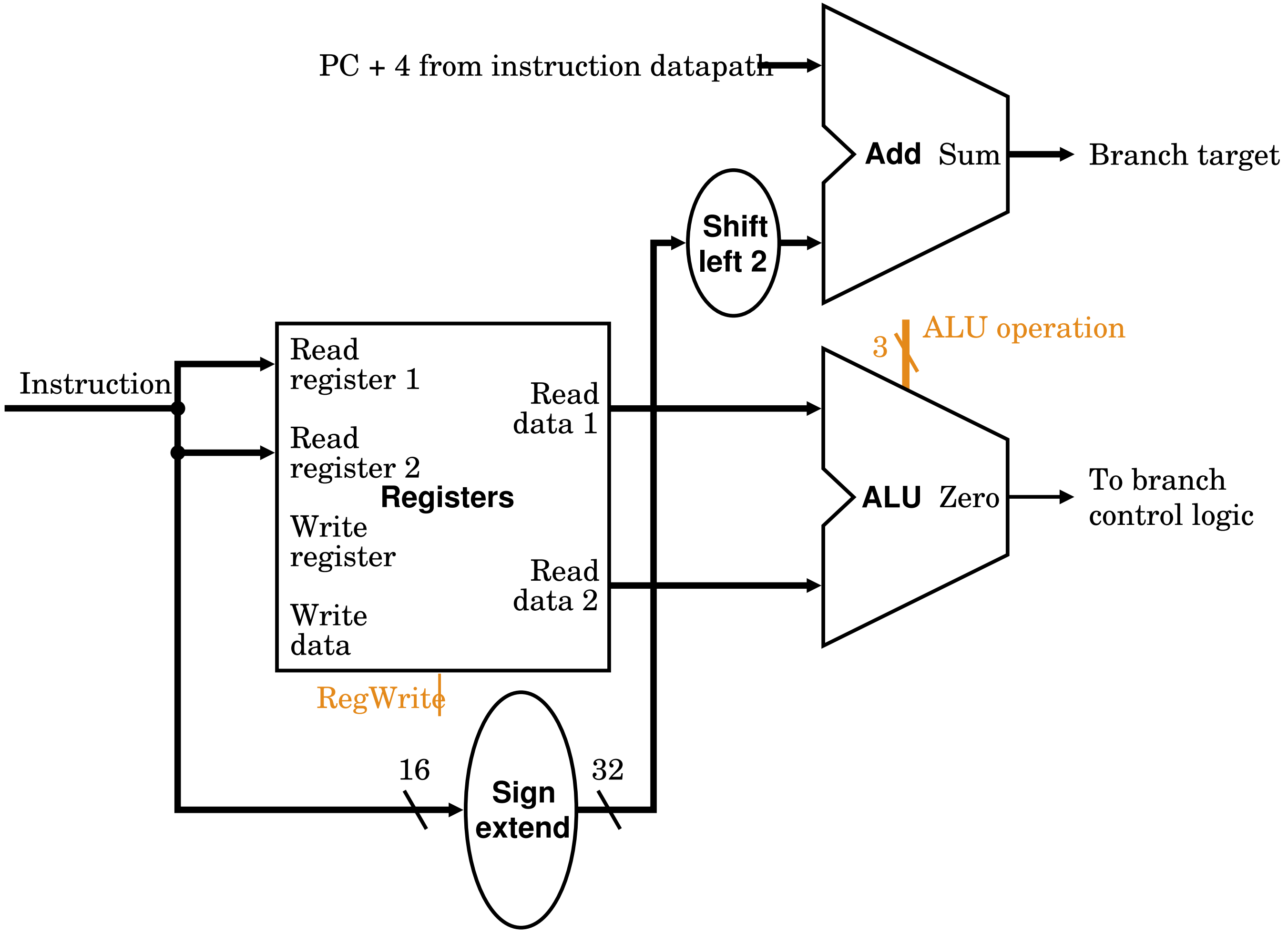
Branch if Equal Instruction

- Look at `beq` R_1, R_2, L with semantics

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

- Need to check if $!R_1 = !R_2$ is true or false ...
- ALU calculates $\vec{r} \overset{\text{def}}{=} !R_1 - !R_2$. `Zero` tests if $\vec{r} = \vec{0}$.

$!R_1 = !R_2$	$\vec{r} \overset{\text{def}}{=} !R_1 - !R_2$	`Zero`
True	$= \vec{0}$	1
False	$\neq \vec{0}$	0

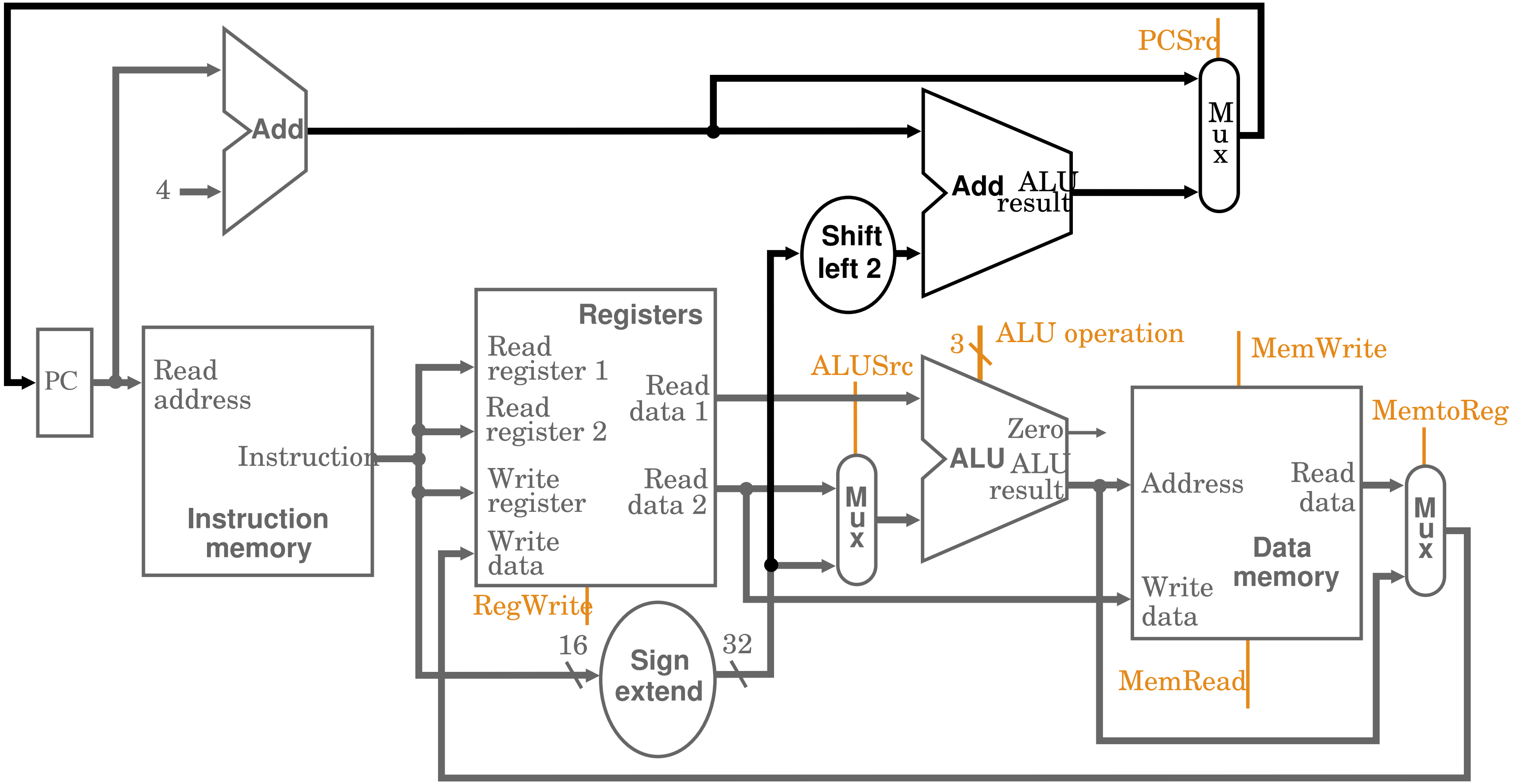


6 bits	*5 bits*	*5 bits*	*16 bits*
`beq`	#R_1	#R_2	*address*

- Executing `beq` updates the PC:

$$\begin{aligned} \texttt{PC} &:= (!\texttt{PC}+4)+(4*\text{address}) \quad \text{if} \quad \texttt{Zero}=1 \\ \texttt{PC} &:= (!\texttt{PC}+4) \quad \text{if} \quad \texttt{Zero}=0 \end{aligned}$$

$$\begin{array}{rl} a \overset{\text{def}}{=} (\!|!\texttt{PC}|\!) \in \mathbb{Z} & \boxed{\texttt{beq}\ R_1,\ R_2,\ L} \\ a+4 & \boxed{\texttt{I}\ \ldots\ldots} \quad \text{1 word} \\ & \vdots\ 2,\ldots,(\!|\text{address}|\!)\ \text{words} \\ a' = \underbrace{(a+4)+4*(\!|\text{address}|\!)}_{\text{Branch target}} \quad L: & \boxed{\texttt{I}'\ \ldots\ldots} \end{array}$$

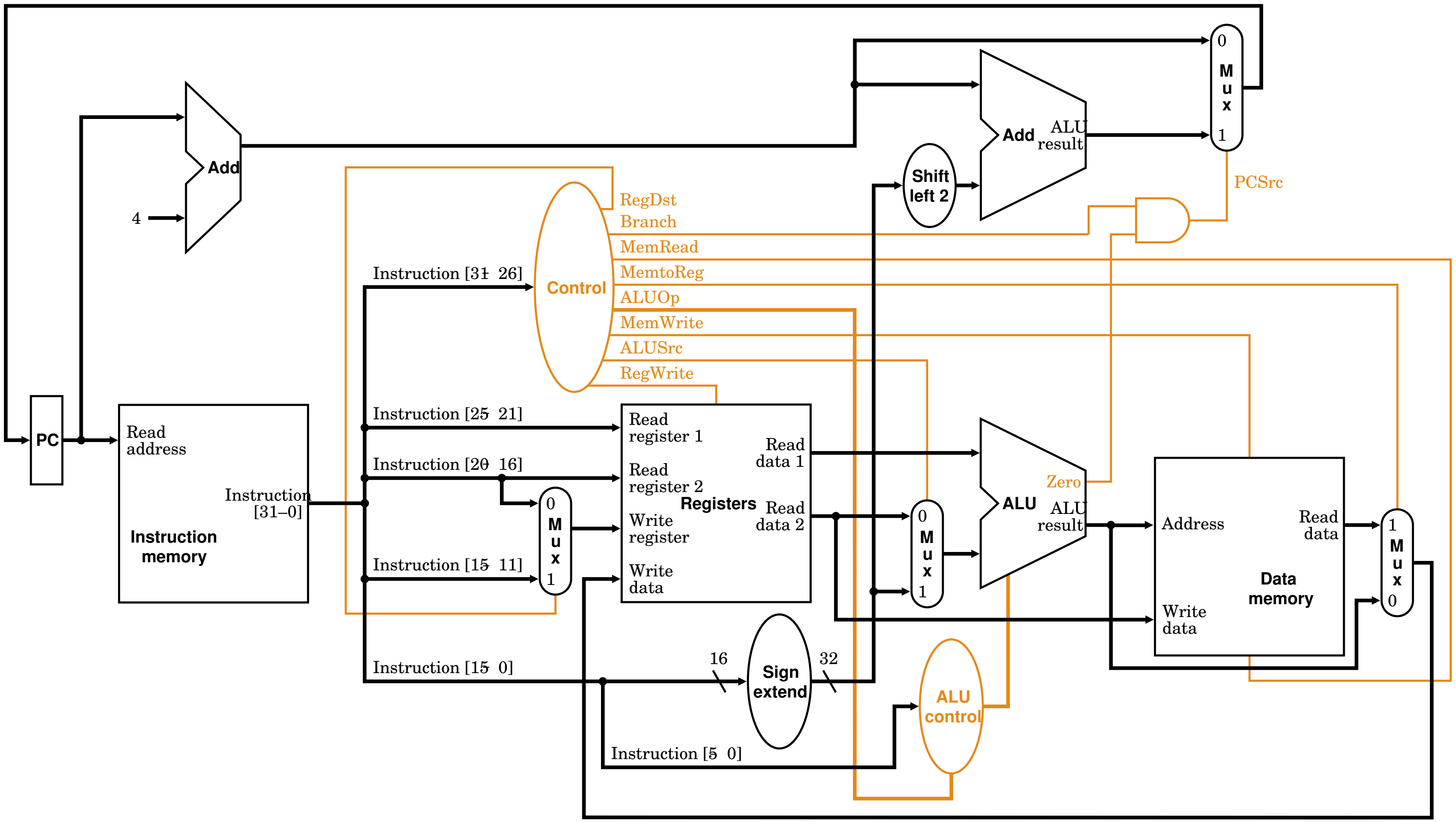

PCSrc
Add
4
M
u
x
Add
ALU
result
Shift
left 2
PC
Read
address
Instruction
Instruction
memory
Registers
Read
register 1
Read
register 2
Write
register
Write
data
Read
data 1
Read
data 2
RegWrite
16
Sign
extend
32
ALUSrc
M
u
x
3
ALU operation
Zero
ALU
ALU
result
MemWrite
Address
Read
data
Data
memory
Write
data
MemRead
MemtoReg
M
u
x

- There is a control `PCSrc` for a multiplexor. In fact (see later on) `PCSrc` is set equal to `Zero` when a branch instruction is executed.
  - So if `PCSrc`=`Zero`=0 then the PC *will—see circuit*—be updated with $\texttt{!PC} + 4$.
  - And if `PCSrc`=`Zero`=1 the PC is updated with $(\texttt{!PC} + 4) + (sx(\texttt{I}[15-0]) \ll 2) = (\texttt{!PC} + 4) + (4 * \text{address})$.
- Eg $1011 \ll 2 \overset{\text{def}}{=} 1011.00$ and

$$(\!|1011.00|\!) = -20 = 4 * -5 = 4 * (\!|1011|\!)$$

Overview: A Single-Cycle Datapath with Control

- We assemble the datapath components into a single cycle datapath (each instruction takes one clock cycle).
- We design a control unit.


PC
Add
4
Read address
Instruction [31–0]
Instruction memory
Instruction [31 26]
Control
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [25 21]
Instruction [20 16]
Instruction [15 11]
Instruction [15 0]
0
M
u
x
1
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
16
Sign extend
32
Instruction [5 0]
ALU control
Shift left 2
Add
ALU result
PCSrc
ALU
Zero
ALU result
Address
Read data
Write data
Data memory

Building Control

We decide on the *effects* on the datapath that actual control signals must have.

We then draw up truth tables of control signal values to ensure that various instructions are executed.

							Settings for control of multiplexors and memory read/write enable								
Inst	Opcode field = I[31-26]						RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp2
lw	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0
sw	1	0	1	0	1	1	X	1	X	0	0	1	0	0	0
beq	0	0	0	1	0	0	X	0	X	0	0	0	1	0	1
R-f’t	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0

Instruction (ALU operation)	ALUOp1	ALUOp2	Funct Field = `I`[5-0]						ALU-Control Output: sets ALU operations		
`lw/sw` (+)	0	0	X	X	X	X	X	X	0	1	0
`beq` (-)	0	1	X	X	X	X	X	X	1	1	0
`add` (+)	1	0	1	0	0	0	0	0	0	1	0
`sub` (-)	1	0	1	0	0	0	1	0	1	1	0
`or` (*OR*)	1	0	1	0	0	1	0	1	0	0	1
`slt` (Less)	1	0	1	0	1	0	1	0	1	1	1

Overview: Timing and Performance

- Explain the timing and performance of a processor.
- Give equations for processor performance.

Timing and Performance

- Recall that a *clock* generates pulses, each lasting for a fixed period of time.
- Clock period called a **cycle**.
- Control unit will set datapath elements, using their control busses (eg ALU to $+$), once per cycle.
- If an instruction can be executed in one clock cycle, we call it a **single cycle instruction**.

- If a datapath can execute each instruction in one cycle, we call it a **single cycle datapath**.
- If a datapath requires multiple settings to execute at least some instructions, we call it a **multi cycle datapath**.
- We look at timing issues for a multicycle processor.

An instruction requires a number of clock cycles to execute. Called the **cycles per instruction** or **CPI**. Write C_I for the CPI of instruction I.

A 6 instruction program P

Instruction I	class	cycles C_I
`lw` ...	α	4
`lw` ...	α	4
`add` ...	β	5
`sub` ...	β	5
`bne` ...	γ	6
`sw` ...	α	4

Write $|P|_c$ for the number of instructions in P of class c and C_c for the number of cylces to run any class c instruction.

Class c	$\|P\|_c$
α	3
β	2
γ	1

Class c	C_c
α	5
β	4
γ	6

Thus the number of clock cycles to execute P is

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

- Write $Cl_{CPI}(P)$ for the set of classes of instructions with identical CPI in a program P.
- $$\textit{Total Clock Cycles} = C_P \overset{\text{def}}{=} \sum_{c \in Cl_{CPI}(P)} |P|_c * C_c$$
- We shall write π for a clock period. Of course

$$t_P = C_P * \pi \qquad\qquad t_I = C_I * \pi$$

- It also follows that the time taken to execute P is

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