Chapter 1

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

Overview: Simple Mathematics

- Syntax and Semantics—words and 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 connection 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 states 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 ‘g‘ $\in$ { ‘a‘, ‘g‘, ‘r‘, ‘t‘ }.
- 4 $\notin$ { ‘a‘, ‘g‘, ‘r‘, ‘t‘ }.
- If A is any set, $e \in A$ means “e is an element in A”.
- Note e is a **variable** and ‘e‘ is a **letter** (**character**).

- $\mathbb{N} = \{0, 1, 2, \ldots\}$ set of **natural numbers**.
- $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ set of **integers**.
- If A is a set, $S \subseteq A$ means S is a set, and any element of S is an element of A. S is a **subset** of A.
- $$\{\text{‘g‘}, \text{‘h‘}, 7\} \subseteq \{7, 5, \text{‘g‘}, 3, \text{‘h‘}, \text{‘e‘}, \text{‘b‘}, \text{‘n‘}\} \qquad \mathbb{N} \subseteq \mathbb{Z}$$

Pairs and Cartesian Product

- Note that $\{1,2\} = \{2,1\}$.
- A **pair** is anything of the form (a,b).
- a is the **1st-component** and b the **2nd-component**.
- Note that $(1,2) = (1,2)$ and $(2,1) \neq (1,2)$ and $(3,4) \neq (6,1)$. In general

$$(a,b) = (a',b') \qquad \text{iff} \qquad a = a' \text{ and } b = b'$$

- $\{\, roy, pat \,\} \times \{\, 22, 33 \,\} =$

$$\{\, (roy, 22), (roy, 33), (pat, 22), (pat, 33) \,\}.$$

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

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

- $\mathbb{B} \stackrel{\text{def}}{=} \{\, 0, 1 \,\}$, the set of **binary digits**. What is $\mathbb{B} \times \mathbb{B}$?

- $(1,0,1) \in \mathbb{B} \times \mathbb{B} \times \mathbb{B}$.
- What is $\mathbb{B} \times \mathbb{B} \times \mathbb{B}$?
- We write A^2 instead of $A \times A$.
- So

$$\mathbb{B}^2 = \{\ (b,b')\ \mid\ b \in \mathbb{B}, b' \in \mathbb{B}\ \}$$

- We can generalise to n**-ary product** A^n.

- A typical element of A^n looks like $(a_{n-1}, a_{n-2}, \ldots, a_0)$. Such an element is called an n**-tuple**.

- We call each a_i a **component** of the n-tuple.

- We shall sometimes refer to an n-tuple in A^n as a **sequence** of elements from A, and write $a_{n-1}a_{n-2}\ldots a_0$ instead of $(a_{n-1}, a_{n-2}, \ldots, a_0)$.

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 of outputs (eg $\mathbb{N}$);
  - a set of input/output pairs (eg $s = \{\ (z, z^2)\ \mid\ z \in \mathbb{Z}\ \}$).

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

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

- Note that

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

- We can also define G by giving the recipe

$$G(i) \stackrel{\text{def}}{=} i^2 + 2$$

This is how we define functions in programming.

- For sets A and B we say f is a **function** iff
  - $f \subseteq A \times B$, and
  - for each $a \in A$ there is a unique $b \in B$ for which $(a,b) \in f$.
- We write $f(a)$ or $f\ a$ for this unique $b \in B$. For example, $G(3) = 3^2 + 2 = 11$. The a is the **argument**.
- We call A the **source** of the function f, and B the **target** of f. Write $f{:}A \to B$.

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

$$f \stackrel{\text{def}}{=} \{(3,4),(4,7),(5,4),(9,10)\}$$

Then $f\colon A \to B$ is not a function from A to B. Why?

- What about a computer which takes three numbers as "inputs" and gives two "outputs" consisting of the sum and product of the three integers?

- We can *model* the computer c as a function with source $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ and target $\mathbb{Z} \times \mathbb{Z}$, that is

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

- We call this function a $3 \mid 2$-ary function, indicating its input is a 3-tuple, and output a 2-tuple.
- A function $f\colon A^n \to A^m$ is an $n \mid m$**-ary function.**
- We shall study functions $f\colon \mathbb{B}^n \to \mathbb{B}^m$ in MC116.

Suppose that $f : A \times A \to A$ is any $2 \mid 1$-ary function. Given an input (a, a'), we have so far written $f(a, a')$ to denote the output.

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

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)

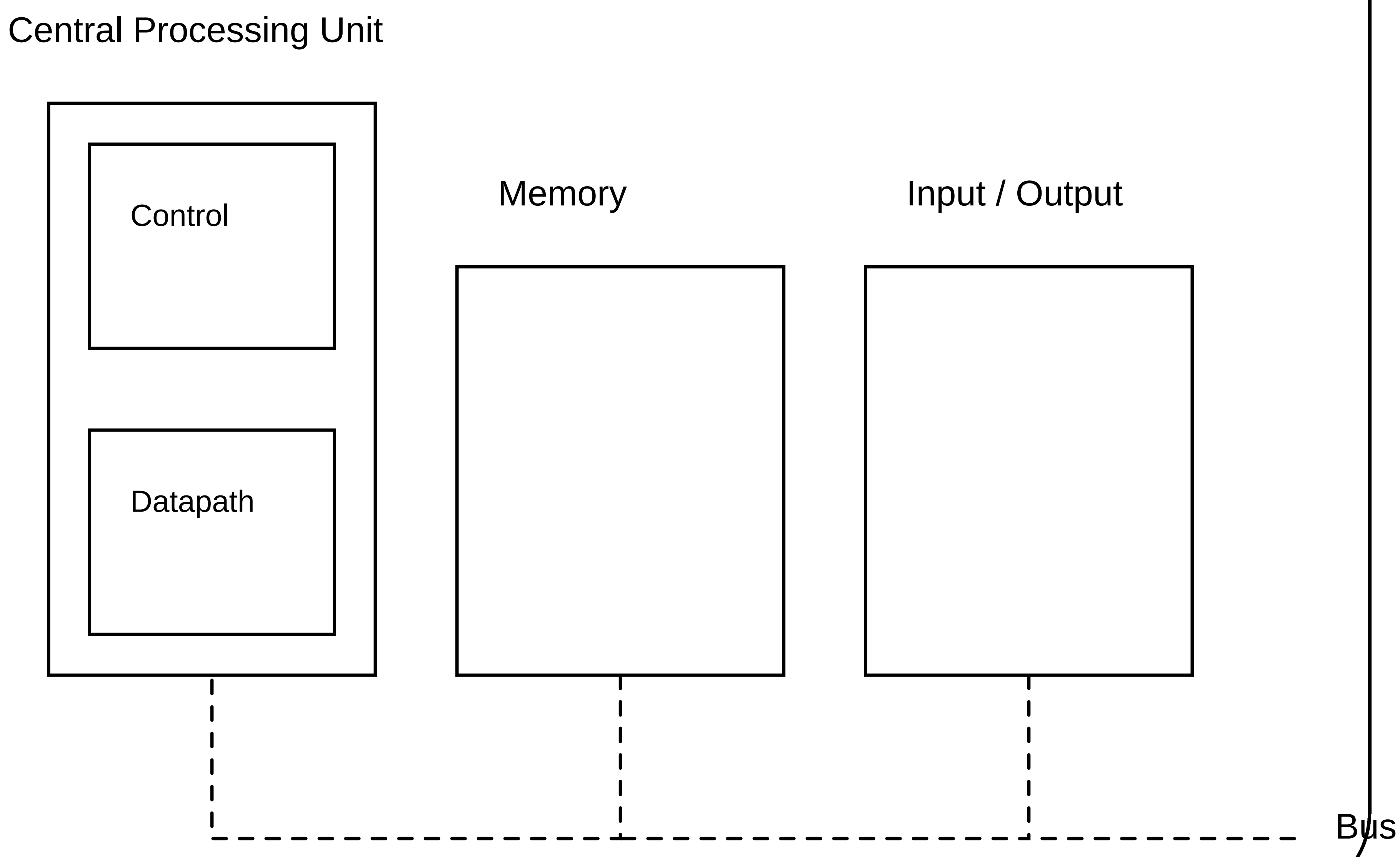
```
          a := 0;
          c := 4;
L         a := a+c;
          c := c-1;
          if c >= 1 then L;
          exit;
```

- 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 `a` $:= 0$) has a *semantics*.

Hardware

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

How does a computer work?



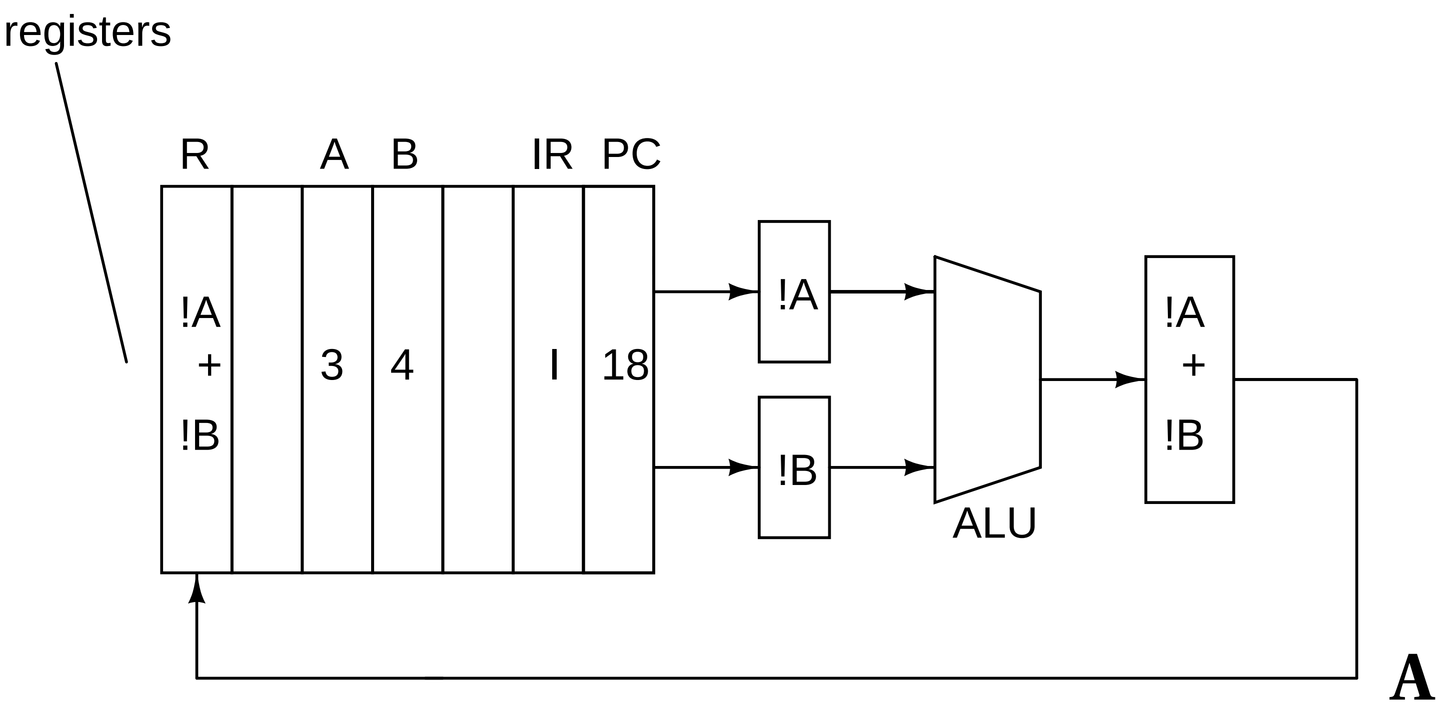
Overview: The CPU

- 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.

- Programs describe calculations performed by CPU.
- A program will be stored in main memory.
- Program execution: **control** “fetches” the program instructions executing each one on the **datapath**.
- **Fetch instruction** means *copy* from main memory to the CPU. Fetched instructions *stored* in **registers**.
- Registers are computer memory and have **contents**.
- Note the **Instruction Register (IR)**, **Program Counter (PC)**, and **general registers**.
- The datapath is “like” a calculator …



Datapath

- The *syntax* (assembly language) of our example instruction 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**.

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—work out what it does;
- (**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 why we can build fast and cheap and big PCs.

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)

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

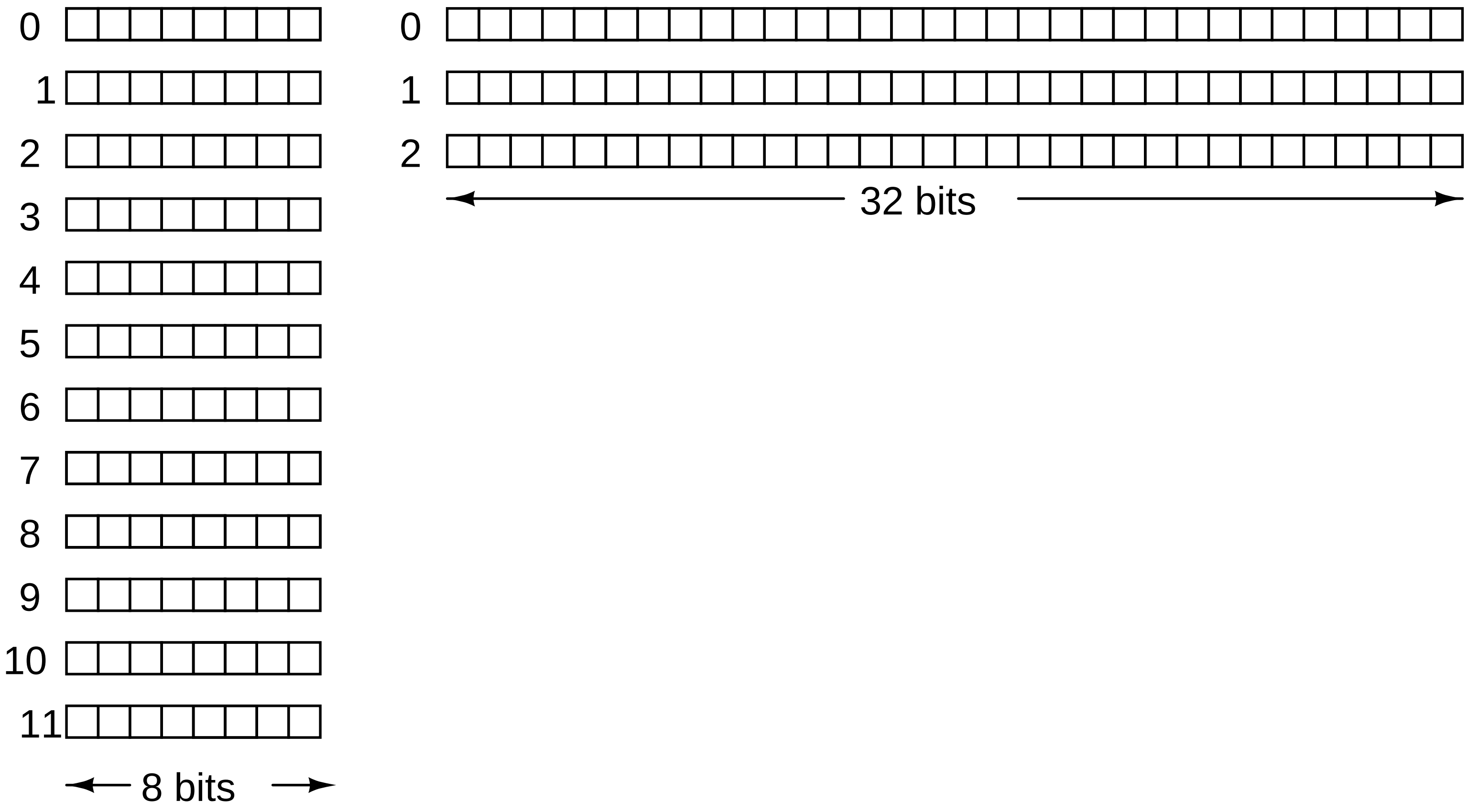
- As you move down the **memory list**:
  - decreasing cost per unit of storage;
  - increasing capacity;
  - increasing access time;
  - decreasing access of memory by CPU.

- Can use hierarchy to build a computer:
  - *fast*
  - *large capacity memory*
  - *good price*
- We use a **cache**.
- Current data and program instructions appear in **clusters—locality principle**.

Cache Memory

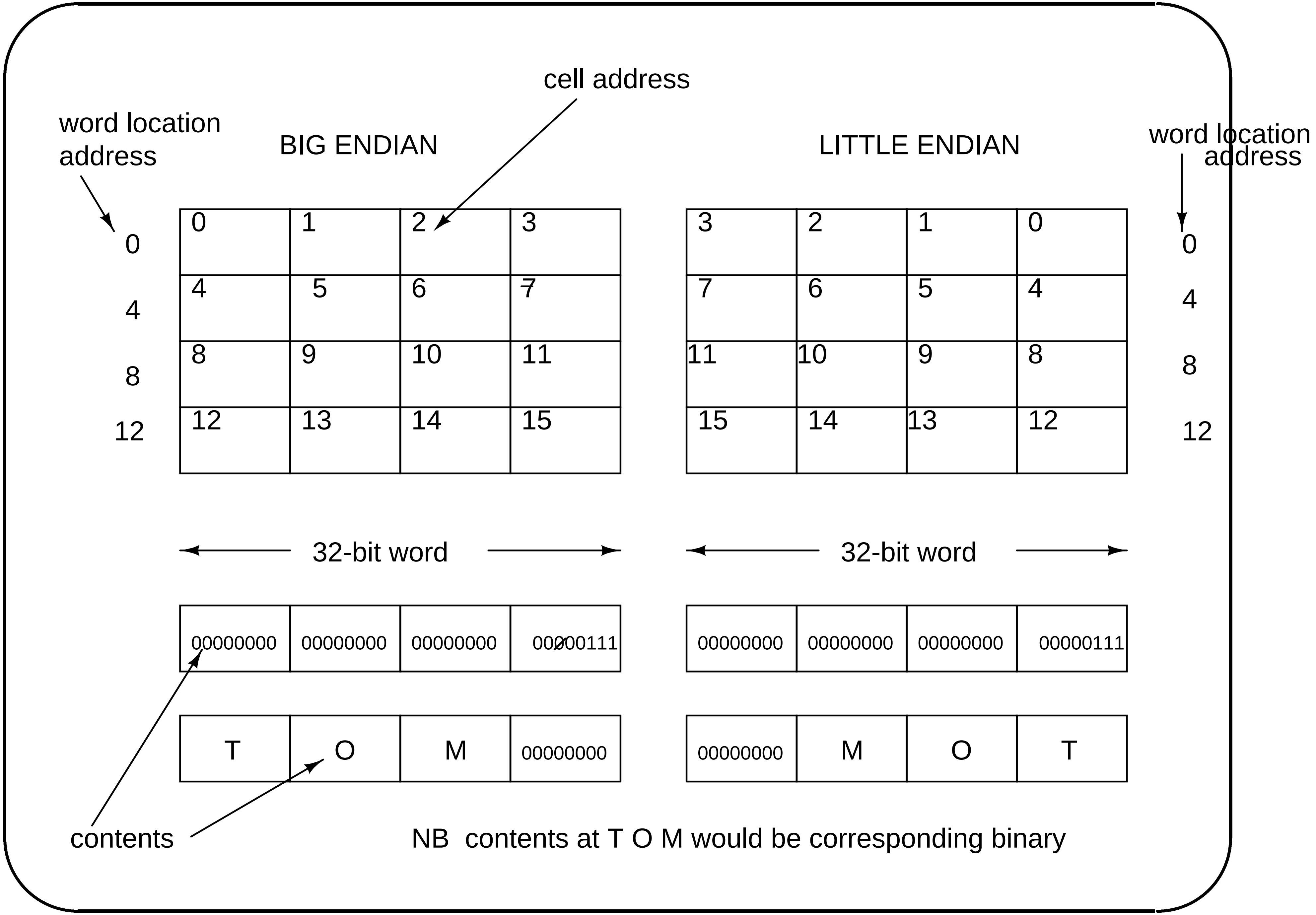
- **Cache** is fast memory found near, or on, the CPU.
- Fetch cycle: control will check if instruction in cache. If so, it will be fetched from cache to CPU.
- If not, control will fetch the (current) instruction's cluster from main memory into the cache, and place the current instruction into the CPU.

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**.
- Groups of consecutive cells are called **word locations**.
- The **address of a byte location** is address of the corresponding 8-bit cell.
- 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**.


cell address
word location address
BIG ENDIAN
LITTLE ENDIAN
word location address
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
3 2 1 0
7 6 5 4
11 10 9 8
15 14 13 12
0
4
8
12
32-bit word
32-bit word
00000000 00000000 00000000 00000111
00000000 00000000 00000000 00000111
T O M 00000000
00000000 M O T
contents
NB contents at T O M would be corresponding binary

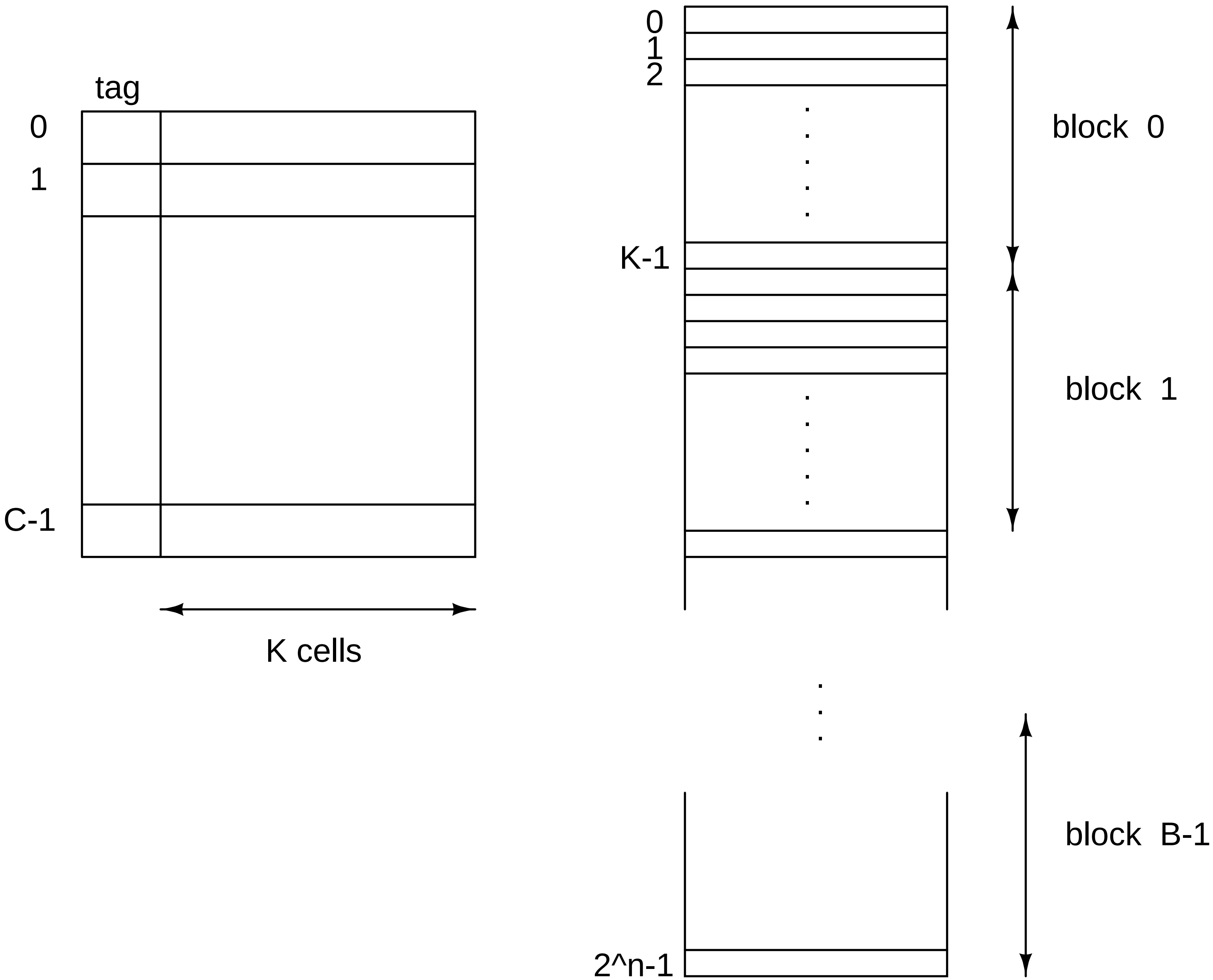
Organizing Memory - The Endian Systems

- How do we arrange byte locations into word locations?
- Byte locations 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**.
- Word locations often store integers and instructions.

- Integers are stored in word locations. If we store 7 as $\vec{0}111^{bin}$ then 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 an integer, known as a **character code**. Eg **ASCII**.
- Codes are stored in byte locations.
- 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


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.

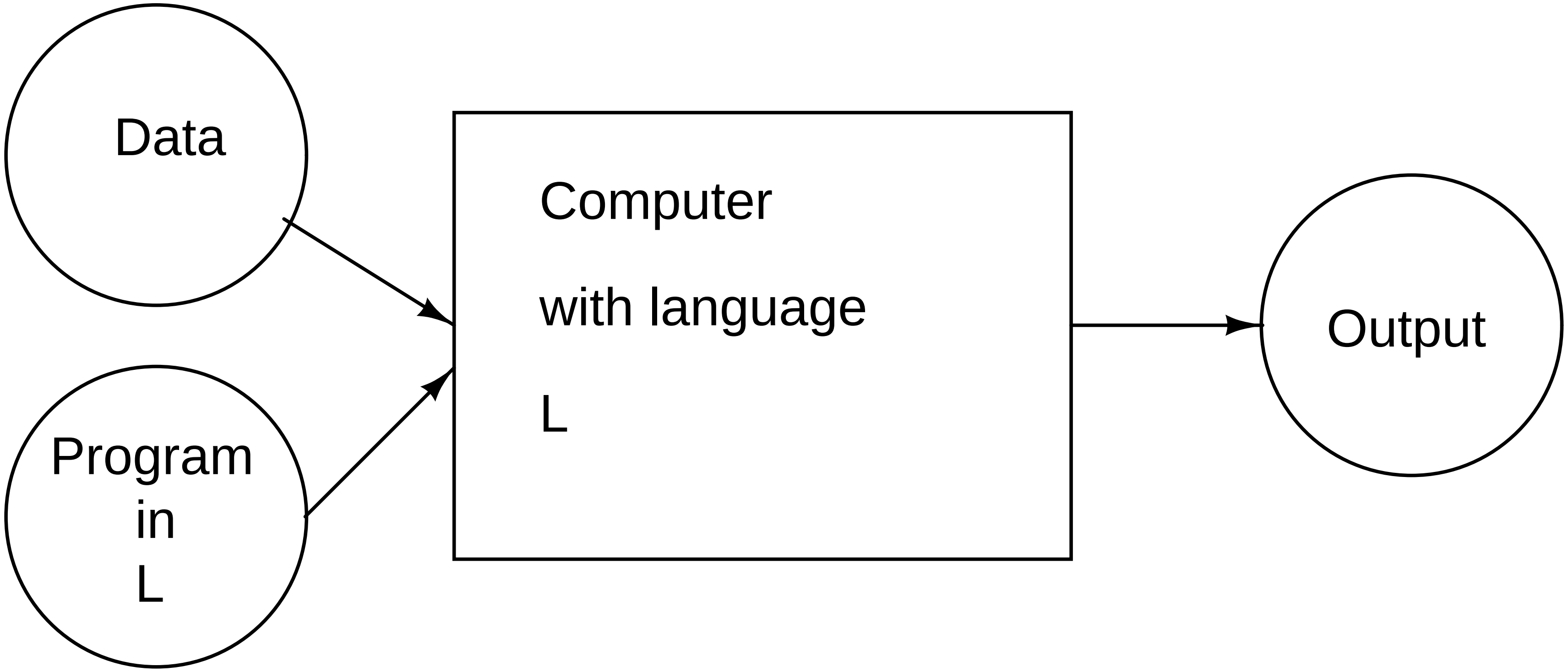
Overview: Translators and Interpreters

- A broad idea of how JAVA programs run.
- Know why we have very simple languages (assembler) and complex ones (JAVA).
- Understand about the range of software of a modern computer.

High and Low Level Languages

- Assembly language is **low level**. Each instruction can be executed by the CPU.
- JAVA is **high level**. One JAVA instruction is translated into many assembly language instructions.
- **Abstraction** refers to ignoring details. JAVA is more abstract than assembler.

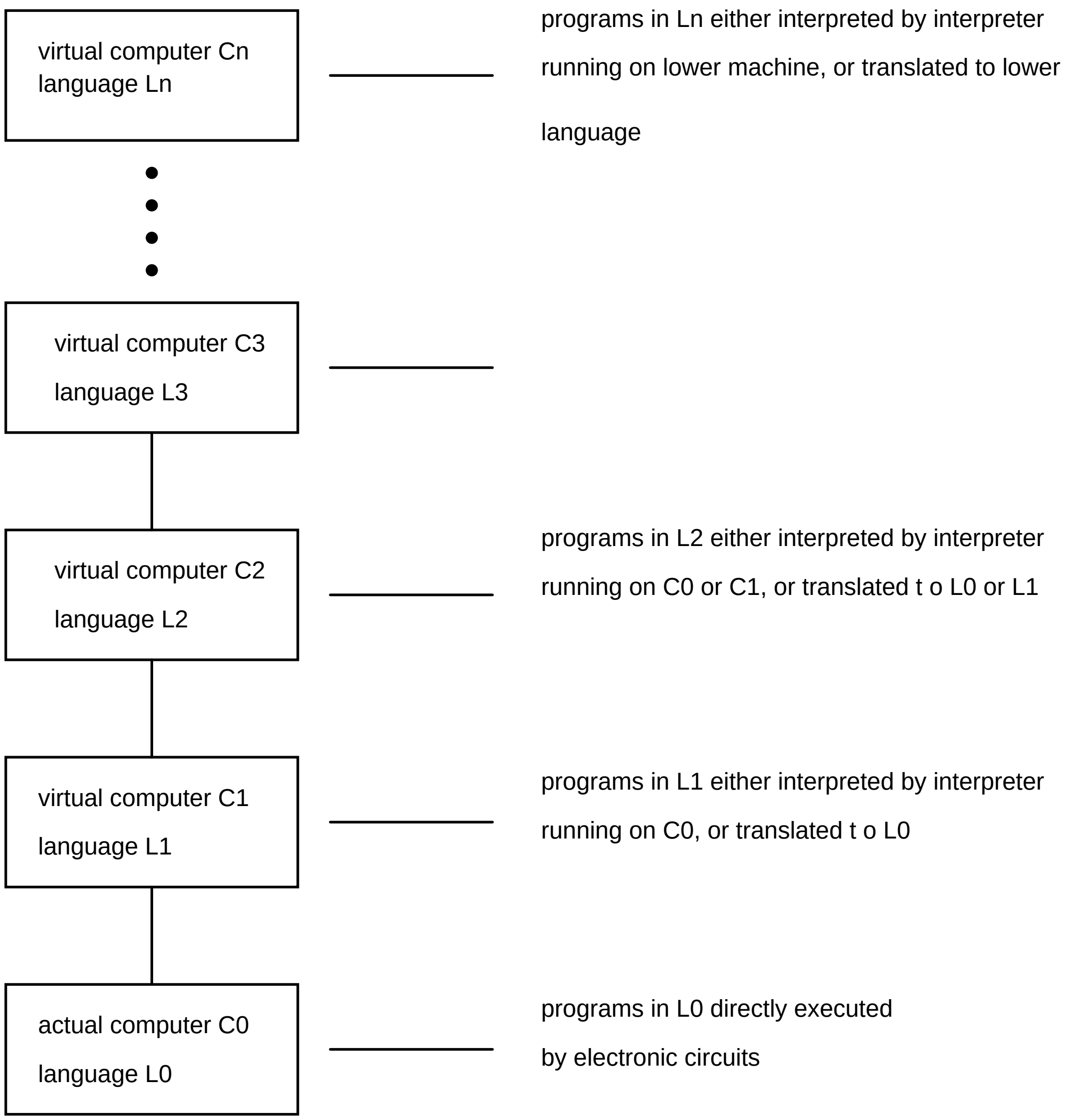
- Consider the program to add up the first ten numbers. Suppose we want to multiply.
- We could change $\mathtt{a} := \mathtt{a} + \mathtt{c}$ to $\mathtt{a} := \mathtt{a} * \mathtt{c}$.
- Multiplication is more abstract than addition …
- Multiplication is a **higher level** instruction than addition.
- The low level + can be executed by a CPU. The higher level $*$ must be **translated** into instructions only involving +.


Data
Program
in
L
Computer
with language
L
Output

Translation and Interpretation

Suppose computer C can be programmed in language L. We want to program in a more abstract L'.

- Given $P_{L'}$ written in L', replace each $P_{L'}$ instruction by an equivalent sequence of L instructions. This is a **translation** or **complilation**.
- Get P_L consisting entirely of L instructions.
- The translation is performed by an L program $Trans_L$ running on C, with $P_{L'}$ as data.
- P_L is then executed on C (**run time**).


virtual computer Cn
language Ln
programs in Ln either interpreted by interpreter running on lower machine, or translated to lower language
virtual computer C3
language L3
virtual computer C2
language L2
programs in L2 either interpreted by interpreter running on C0 or C1, or translated t o L0 or L1
virtual computer C1
language L1
programs in L1 either interpreted by interpreter running on C0, or translated t o L0
actual computer C0
language L0
programs in L0 directly executed by electronic circuits

Software/Language Levels

Read end of Chapter 2.

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

- $\mathbb{N} \stackrel{\text{def}}{=} \{0,1,2,\dots\}$ and $\mathbb{Z} \stackrel{\text{def}}{=} \{\dots,-1,0,1,\dots\}$.
- 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

$$d_{k-1}\dots d_0{}^{\,bin} \in \mathbb{B}^k = \underbrace{\mathbb{B}\times\mathbb{B}\times\dots\times\mathbb{B}}_{k}$$

- $d_{k-1}\dots d_0{}^{bin}$ 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 stands for 2^i. In general we have

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

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

$$[\![d_{k-1} \ldots d_0\,^r]\!] \quad \stackrel{\text{def}}{=} \quad \sum_{i=0}^{i=k-1} [\![d_i]\!] * r^i$$

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 $[\![0]\!] \stackrel{\text{def}}{=} 0$, $[\![1]\!] \stackrel{\text{def}}{=} 1$, $\ldots$ $[\![E]\!] \stackrel{\text{def}}{=} 14$, and $[\![F]\!] \stackrel{\text{def}}{=} 15$.
- Thus $B1A^{\,hex}$ represents the integer 2842:

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

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 $0 \le n \le 2^k - 1$.
- We shall say the computer has k-**bit numbers**.

- Write a k-bit 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 $k \stackrel{\mathrm{def}}{=} 6$ bits, then the number 3 is represented by 000011^{bin}.

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

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

- 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

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

- 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
$c\vec{arr}y^{\,bin}$	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

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

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

Example of Correctness

$\vec{d}^{\,bin}$		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 2s-complement system, the $k+1$th bit will be 0 for positive integers and zero, and 1 otherwise.

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

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

If $k = 5$, then

$$(\!|000111^{\,bin}|\!) = -0 * 2^5 + 7 = 7$$

but

$$(\!|100111^{\,bin}|\!) = -1 * 2^5 + 7 = -32 + 7 = -25$$

Note also that

$$25 = (\!|011001^{\,bin}|\!).$$

- 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, 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: $[\![1011^{\,bin}]\!] = 11 = 5 + 6 = [\![101^{\,bin}]\!] + [\![110^{\,bin}]\!]$
- 3 bits: $[\![011^{\,bin}]\!] = 3 \neq 5 + 6 = [\![101^{\,bin}]\!] + [\![110^{\,bin}]\!]$

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

$$(\!|101|\!) = -3 = (\!|111|\!) + (\!|110|\!)$$

Sign-bit Conditions

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

Result = Computer Sum = k-digit answer ($k+1$ “Lost”).

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

- $\llparenthesis s_k \ldots s_0 {}^{bin} \rrparenthesis = \llparenthesis b_k \ldots b_0 {}^{bin} \rrparenthesis + \llparenthesis b'_k \ldots b'_0 {}^{bin} \rrparenthesis$
- $-2^k \leq \llparenthesis b_k \ldots b_0 {}^{bin} \rrparenthesis + \llparenthesis b'_k \ldots b'_0 {}^{bin} \rrparenthesis \leq 2^k - 1$
- the (previous) sign-bit conditions hold.

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

- Use 1 to denote truth, and 0 to denote falsity.
- We shall soon use the logic functions $\centerdot : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ and $+_{or} : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$.
- Input (d, d') is mapped to $d \centerdot d'$.
- $d \centerdot d'$ is 1 precisely when both d *and* d' are 1, and otherwise $d \centerdot d'$ is 0.
- Some examples are $0 \centerdot 0 = 0$, $1 \centerdot 1 = 1$, $1 +_{or} 1 = 1$ and $1 +_{or} 0 = 1$.

There are *digitwise* versions of these functions

$$d_k \ldots d_0{}^{\,bin}\ AND\ d'_k \ldots d'_0{}^{\,bin} \stackrel{\mathrm{def}}{=} (d_k \centerdot d'_k) \ldots (d_0 \centerdot d'_0)^{\,bin}$$

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

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

Truth Tables

We give the functions $\mathbf{.}$ and $+_{or}$ by explicitly computing them in tables ...

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

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 $\mathbf{.}$; 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)

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

$\vdots$

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 ...
- 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 $2 \mid 1$-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

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

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

- 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.

Circuits may not be optimal:

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

- A $n \mid m$-ary function f gives rise to m different $n \mid 1$-ary functions $f_0, \ldots, f_{m-1}$.
- We showed how to implement each f_i.
- Thus we can implement f by combining the circuits for the f_i "in parallel".

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.

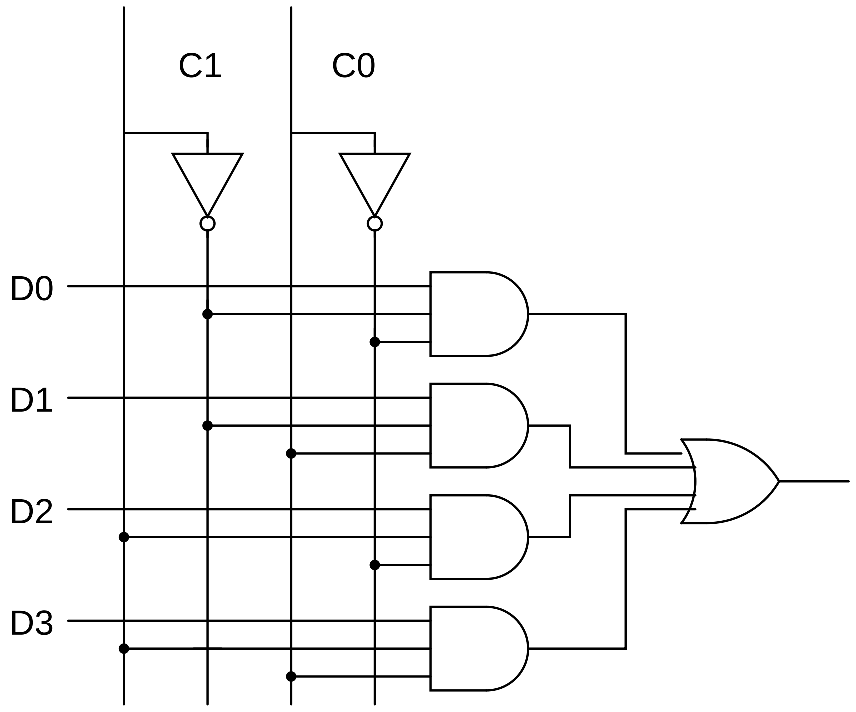
Multiplexor

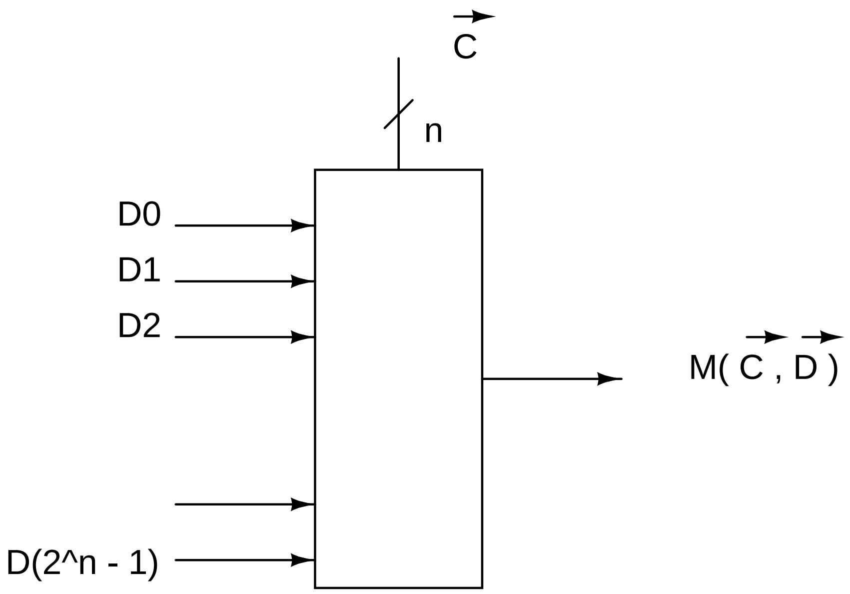
- A **multiplexor** has $n+2^n$ input lines which divide up into n **control** input lines and 2^n **data** input lines.

- 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


C
n
D0
D1
D2
D(2^n - 1)
M(C , 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.**

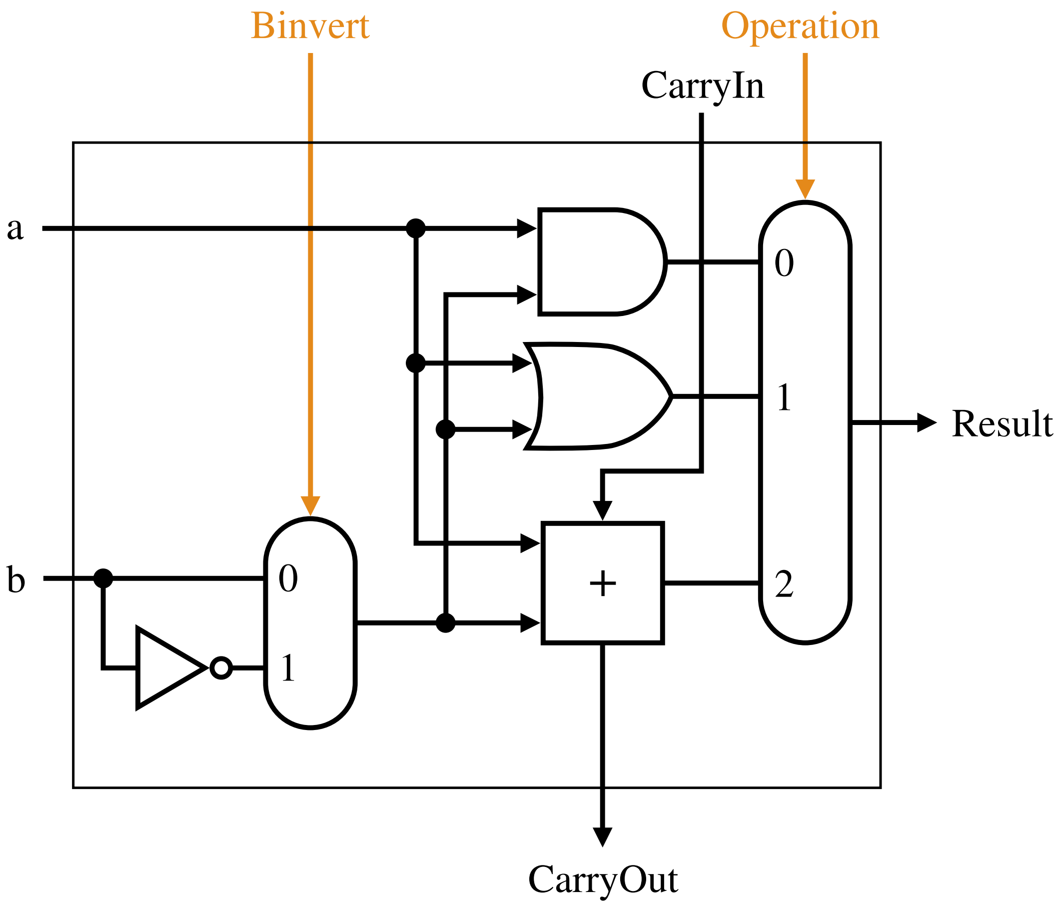
- We shall design an ALU where $k = 32$, with functions from Chapter 3, and …

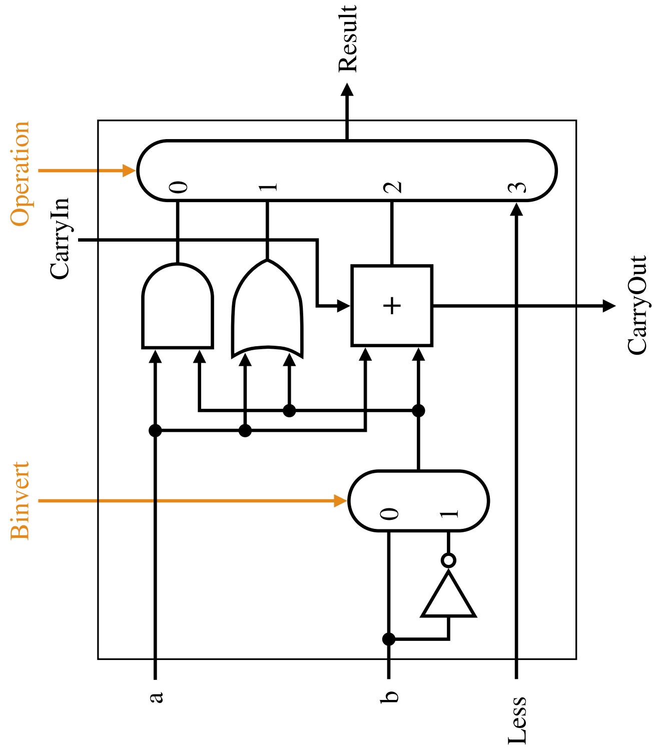
- the "set on less than" `slt` function which returns

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

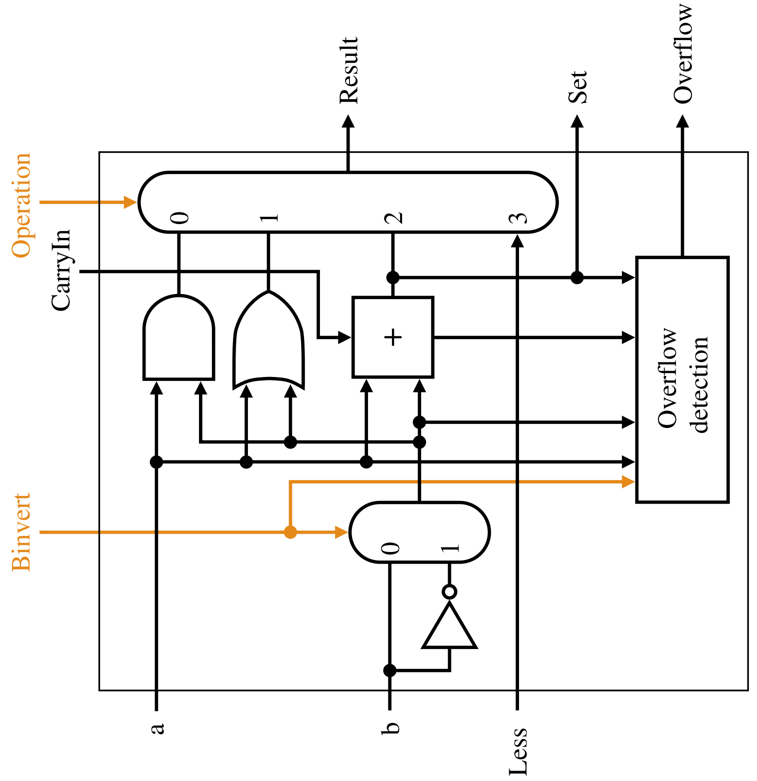
- A single bit output line `Zero` which returns 1 when *any* result is $\vec{0}$, and 0 otherwise.

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

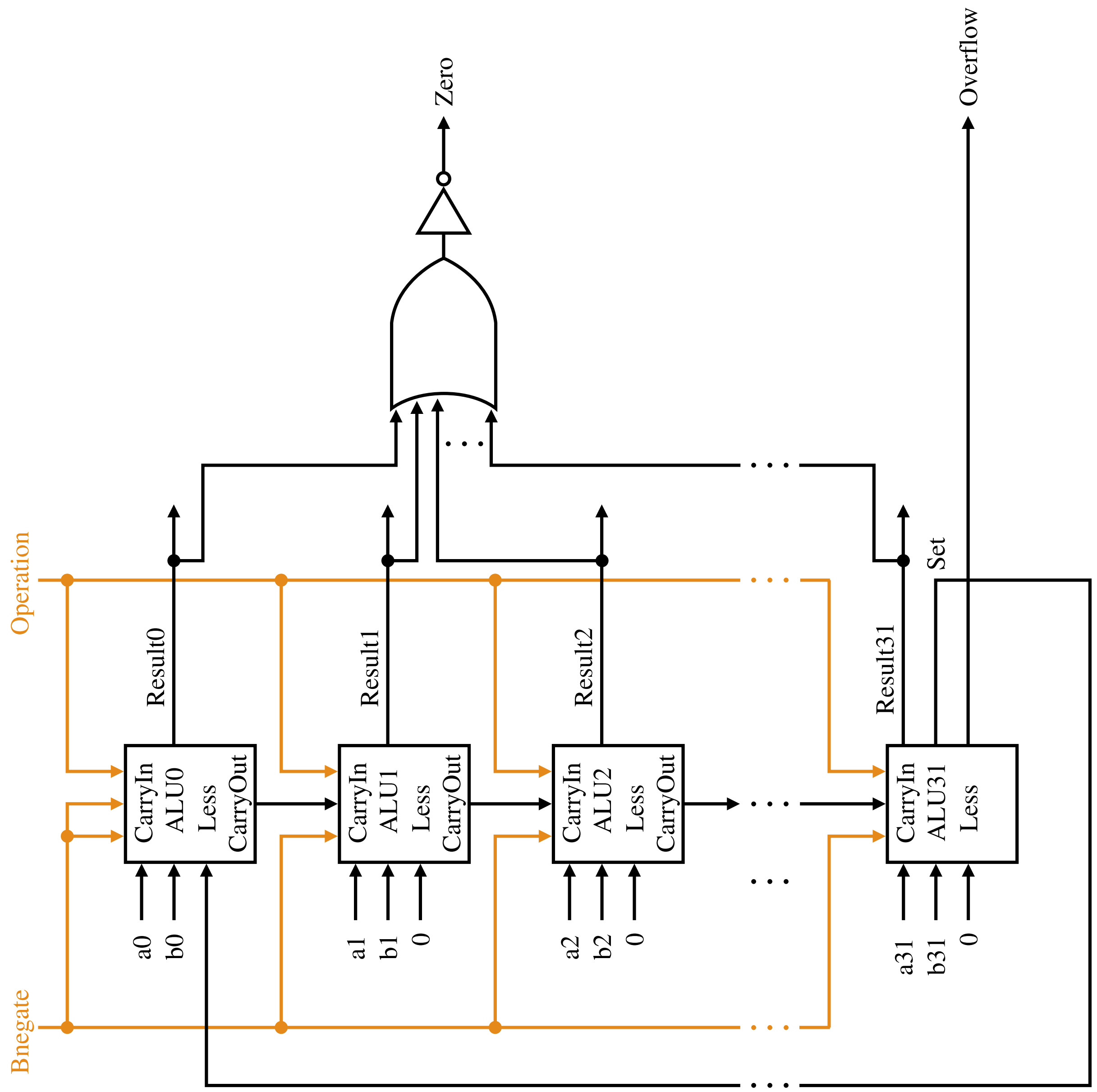

Binvert
Operation
CarryIn
a
b
0
1
+
0
1
2
Result
CarryOut


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
Set
Operation
Bnegate
Result0
Result1
Result2
Result31
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

Example

Explain how to make the big ALU select the `slt` *function, and how the* `slt` *function is computed.*

Answer:

- $(\!|\vec{a}^{\,bin}|\!) < (\!|\vec{b}^{\,bin}|\!)$ iff $(\!|\vec{a}^{\,bin}|\!) + (\!|\overline{\vec{b}^{\,bin}} + 1^{\,bin}|\!) < 0.$
- If no overflow occurs, by correctness $(\!|\vec{a}^{\,bin} + \overline{\vec{b}^{\,bin}} + 1^{\,bin}|\!) < 0.$
- Set `Bnegate` to 1, and `Operation` to 11. So `Result1` to `Result31` are all 0.

- Value of `Result0` comes from the `Set` line of ALU31.
- By examining ALU31, we see this will hold digit 31 of $\vec{a}^{\,bin} + \overline{\vec{b}^{\,bin}} + 1^{\,bin}$. This is the sign digit.
- So it will be 1 precisely when $(\!|\vec{a}^{\,bin} + \overline{\vec{b}}^{\,bin} + 1^{\,bin}|\!) < 0$ and will be 0 if not.

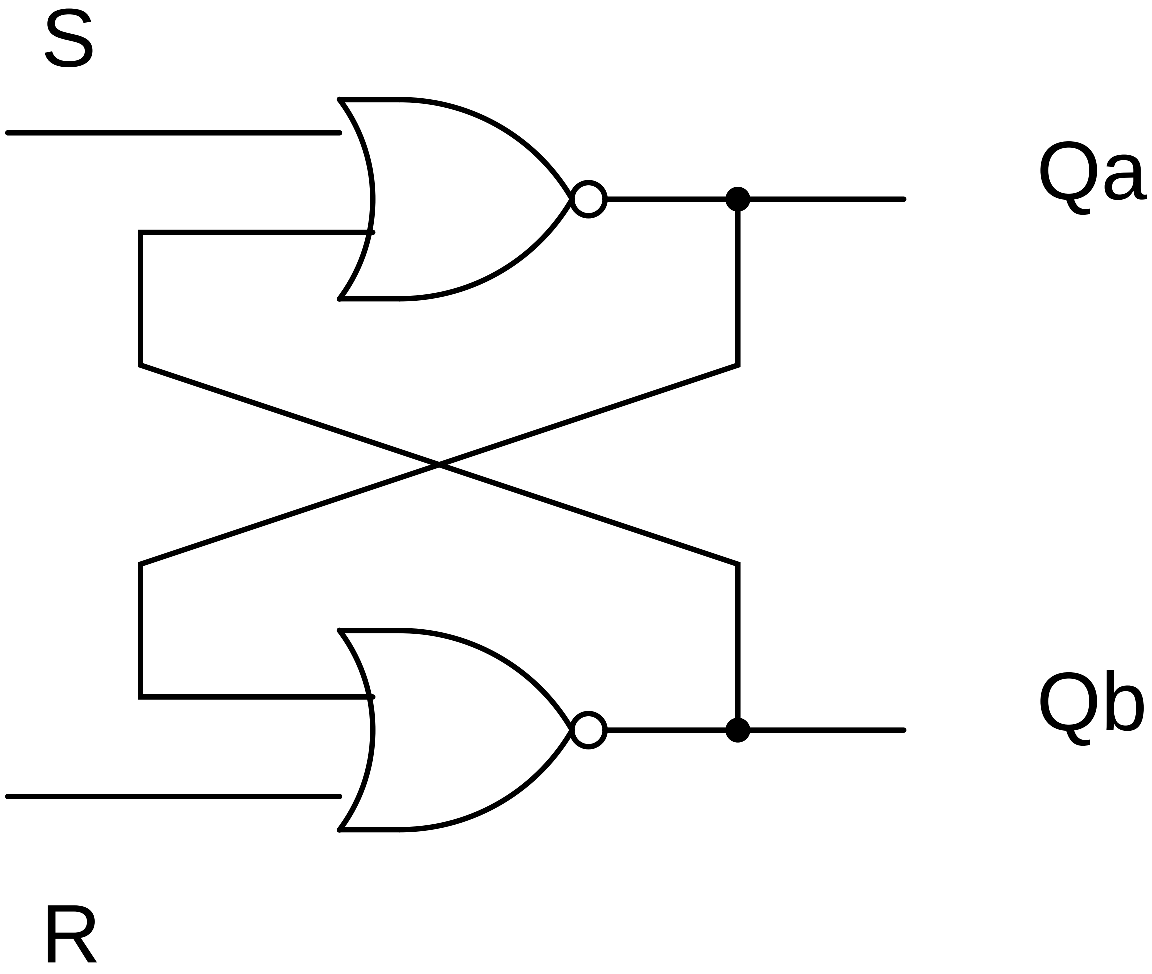
Overview: Sequential Circuits

- 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 outputs for a given input. They are not functions, but *relations*.
- For such 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

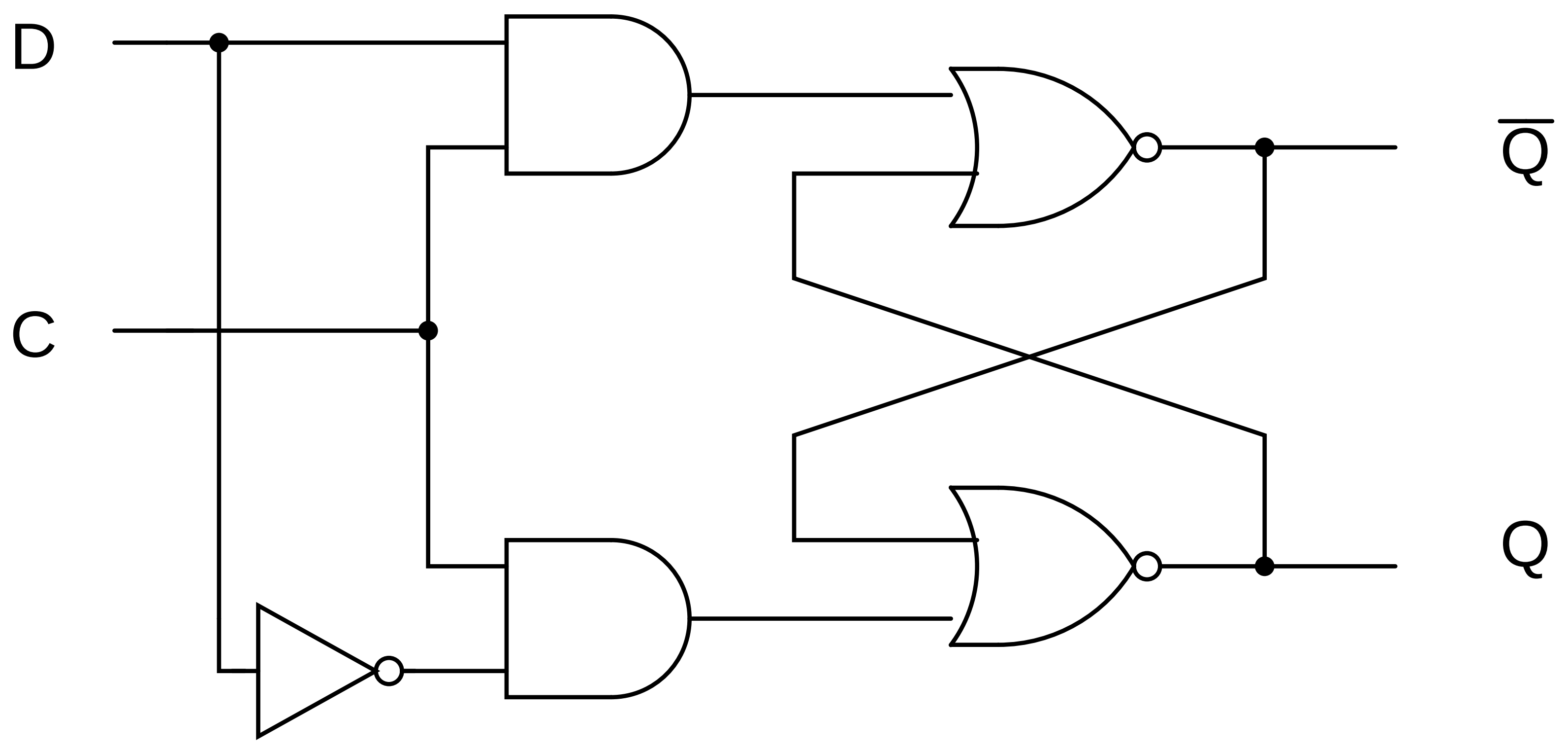
- A model of 1-**bit computer memory** is a pad-locked box storing 0s and 1s. The box has a glass window.

- *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.

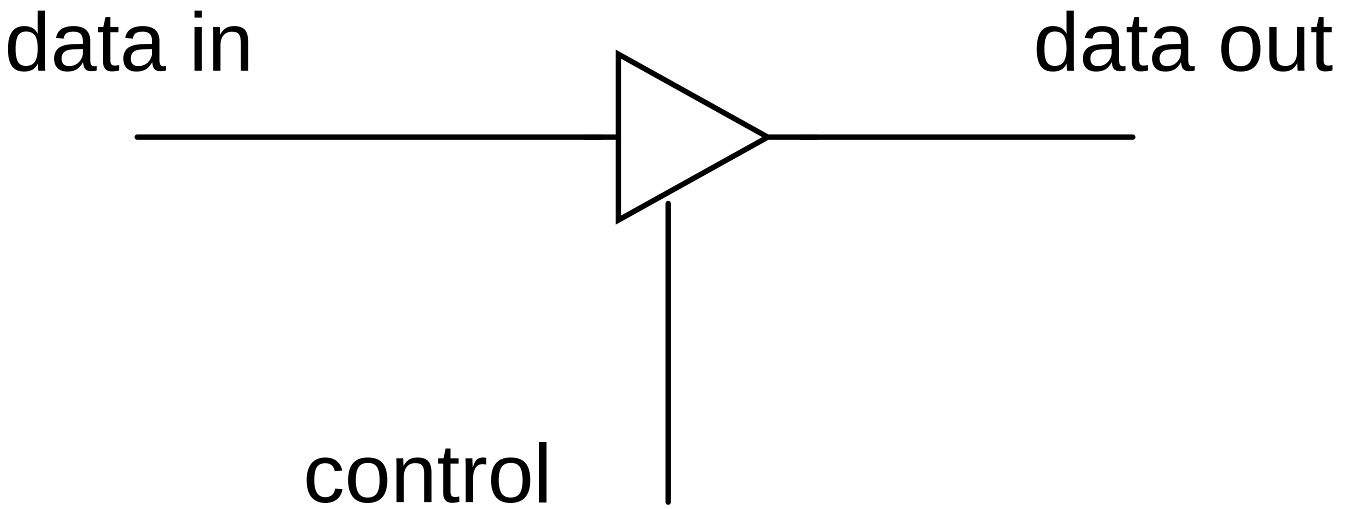
- **clock line** indicates when data can and cannot be stored—a **write enable** line.

- Q holds the stored data.


D
C
$\overline{Q}$
Q

Tri State Buffers

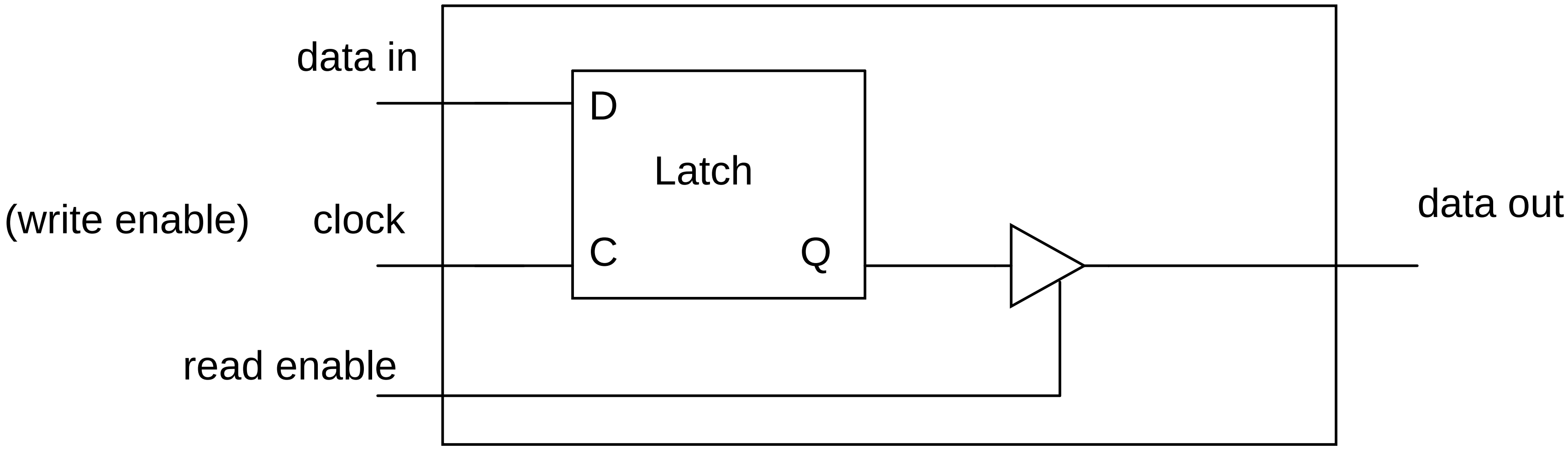
- A Tri State Buffer is an electronic switch.
- We use it to make a **read enable** line for a 1-bit memory.



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:

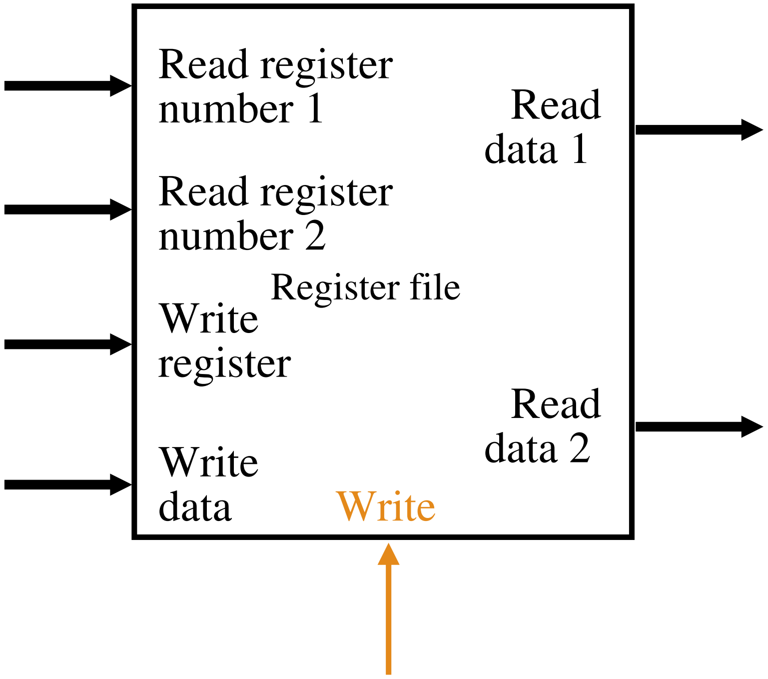


Register Files

- 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:

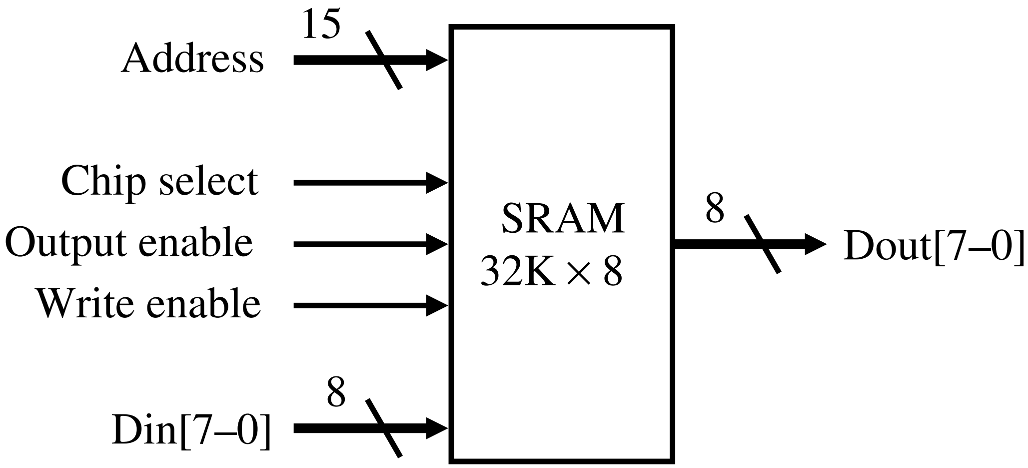


Building Static Random Access Memory

Computers count in binary.

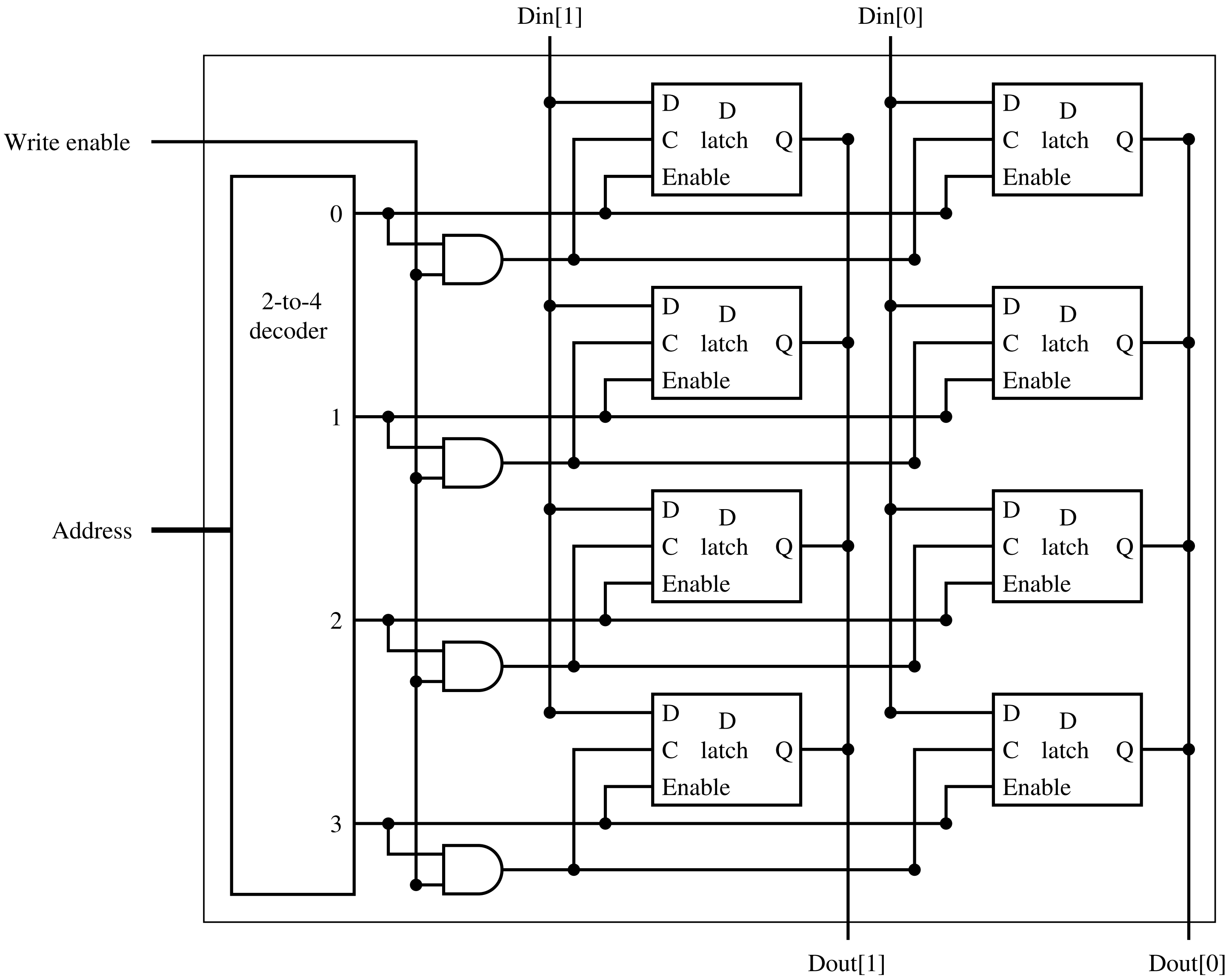
Kilo (K)	2^{10}
Mega (M)	2^{20}
Giga (G)	2^{30}
Tera (T)	2^{40}

- A finite number of 1-bit memories (a cell), written to simultaneously, provides a basic building block for computer memory.
- For example, the MIPS computer has 8-bit cells (byte locations) as the smallest addressable memory unit.
- A computer's *fast* main memory is built from **SRAM**s, that is **Static Random Access Memory**s. An SRAM is an integrated cicuit.



- 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:
```

- A MIPS *program* is a sequence of either instructions or **labelled instructions** `L : I`.
- Usually instructions are executed in order, first to last.
- But sometimes the next instruction is given by a **label** `L`.

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 $\texttt{\$t1} := \vec{0}$)
- 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 $R := 3$. The convention is that this will be shorthand for $R := \vec{0}11^{\,bin}$.

- 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

$$\texttt{addi}\ R_1,\ R_2,\ 5 \qquad\qquad \texttt{add}\ R_1,\ R_2,\ R_3$$

- 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.
- The source argument looks like $k(R)$.
- k is called a **base** address.
- The contents of R are the **offset**, and indicate how far data is located from the base address.
- Data given by the contents of $B[!R+k]$ or $W[!R+k]$.

- k has denoted any integer.
- So $!R + k$ *means add the 32 digits of* $!R$ *to a 32 digit 2s-complement representation of* k.
- $k \in \mathbb{Z}$ must be stored in a computer *within 16 bits.*
- Thus

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

- A 16-bit representation of k may be copied into a 32-bit register. This will involve a sign extension.

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 …

Logical Category Instructions

These instructions are similar to the `add` instruction. The logical functions are the *bitwise AND* and *OR*.

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

Example

Let $k = -32760$. Compute the contents of R_1 when

$$\texttt{ori}\ R_1, R_2, k$$

is executed, given that R_2 holds

$$b = 11111111.11111111.01111111.11110111$$

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

$$zx(b') = 00000000.00000000.10000000.00001000$$

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

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

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

$$\begin{aligned}\texttt{repeat:}\quad & \texttt{add \$t1, \$t1, \$t0}\\ & \texttt{addi \$t0, \$t0, } 1\\ & \texttt{bne \$t0, \$t2, repeat}\\ & \texttt{next} - \texttt{instruction}\end{aligned}$$

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 digit sequences.
- Show that each digit sequence is made up of special parts.
- Look at the use of labels to represent actual machine addressses.

MIPS programs

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 specifies how the 32 bits are divided up into sections known as **fields**.

Here is an example of `add` R_1, R_2, R_3

`add $t0, $s1, $s2`

Format	Fields					
R	op	rs	rt	rd	shamt	funct
	$\vec{0}$	R_2	R_3	R_1	$\vec{0}$	`add`
	$\vec{0}$	`$s1`	`$s2`	`$t0`	$\vec{0}$	`add`
	000000	10001	10010	01000	00000	1000000
# Bits	6	5	5	5	5	6

- Opcode (op) 0^{dec} indicates an R-format instruction.
- Function field (funct) indicates addition.
- The second and third fields specify the *numbers* of the source registers (17^{dec} and 18^{dec})
- The fourth field, 8^{dec}, specifies the destination register.
- Finally, the fifth field is set to 0^{dec} ALWAYS in CO1016.

I-Format

I-format with syntax *name* $R_1,\ k(R_2)$

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

Translating Assembly to Machine Language

- An **assembler** is a program that will translate assembly language into machine language.
- We use a **field translation table** (opcode map); and
- the field layouts of the last lecture(s); and
- a **register number table**.

Field 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)
`$t0`	8	temporary (not preserved across call)
`$t1`	9	temporary (not preserved across call)
`$t2`	10	temporary (not preserved across call)
`$s0`	16	saved temporary (preserved across call)
`$s1`	17	saved temporary (preserved across call)
`$fp`	30	frame pointer (preserved across call)

Example

What is the machine code for

$$\texttt{lw \$s1},\ 13(\texttt{\$t1})?$$

Answer: I-format instructions look like *name* R_1, $k(R_2)$. The opcode for `lw` is 35. The number for `$s1` is 17 and `$t1` is 9. The 16 digit representation of $k = 13$ is 0000000000001101.

6 bits	*5 bits*	*5 bits*	*16 bits*
`lw`	R_2	R_1	k
100011	01001	10001	0000000000001101

Branch Addressing

- Recall `beq` R_1, R_2, L. The label `L` points to the next instruction.
- Such instructions have a machine code:

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

- Address field "indicates"an **offset**, o, of how far in bytes the **next instruction** `I`$'$ **to be executed** is from the *next instruction* `I` *in memory.*

a	`beq ......`
$a+4$	`I ......`
	$\vdots$
$a' = (a+4)+o$	`I' ......`

- But the address field actually measures words. Therefore $o = 4 * \text{address}$.

- Suppose that the next instruction is six word locations after the branch. Then

$$a' = (a+4)+(4*\text{address}) \qquad = \qquad a+4*6 = a+24$$

and so

$$4+4*\text{address} = 24 \qquad \text{and so} \qquad \text{address} = 5$$

- At the end of execution, the new value of the PC will be given by

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

Chapter 6

- Build a datapath in which ISA instructions are executed in one clock cycle.
- Design control for this.

In notes; non-examinable:

- Build a datapath in which ISA instructions are executed in many clock cycles.
- Design control for this.

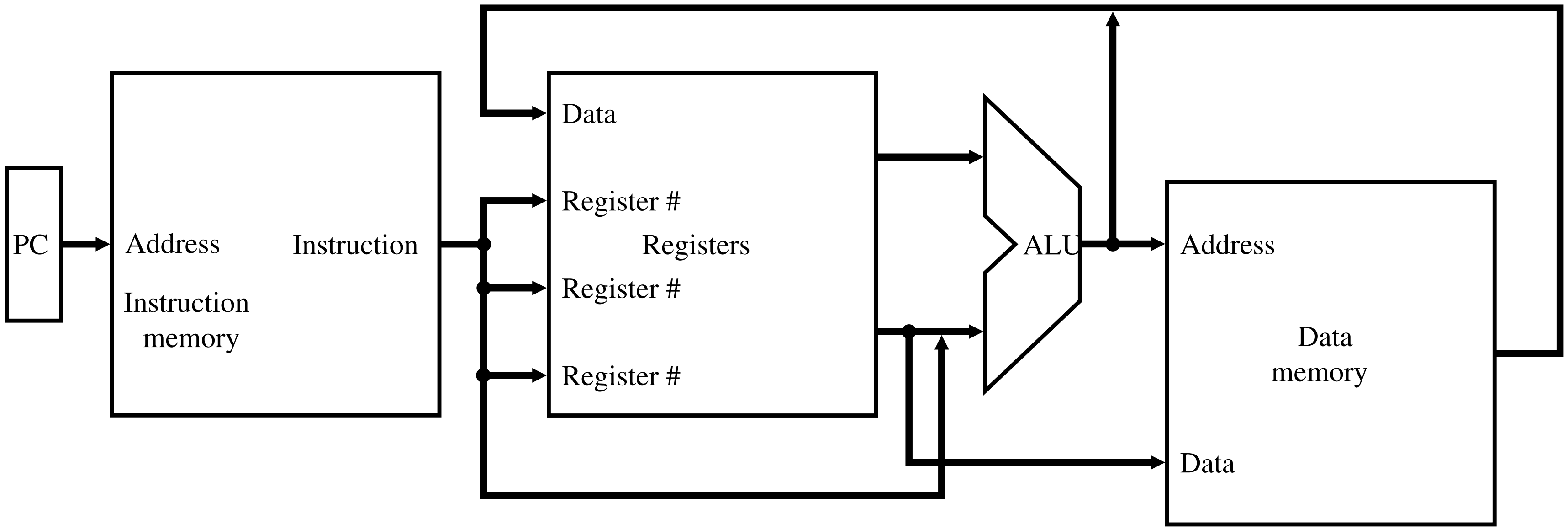
Overview: Introduction, Timing and Performance

- Introduce the idea of a micro-architecture: the details of a processor.
- Explain the timing and performance of a processor.
- Give equations for processor performance.

Introduction

- Show how to build a small CPU.
- Design of the CPU is called its **architecture**.
- **Micro** suggests 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`.

This chapter will develop datapaths based on the following abstract model:



- Each thick black arrow denotes a bus; widths vary.
- PC points to next instruction.
- Each instruction field is **passed** along a bus …
- Separate circuits are called **elements** or **units**.
- Control will *perform* the FDE cycle.
- Control provides **settings** for the datapath elements.

Timing and Performance

- Recall that a *clock* generates pulses of a fixed length—used to give a measure of time.
- Clock period called a **cycle**.
- Control unit will set datapath elements (eg ALU to $+$) once per cycle.
- If a whole instruction can be executed in one clock cycle, we call it a **single cycle instruction**.
- If a whole instruction can only be executed in many clock cycles, we call it a **multi 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**.
- *In this chapter we shall show how to build a CPU with only a single cycle datapath.*
- But first we look at timing in general, not just for a single cycle 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.

- Write $Cl_{CPI}(P)$ for the set of classes of instructions with identical CPI in a program P.

- Write $|P|_c$ for the number of instructions in P of class c.

- Also write C_c and C_P.

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

A 6 instruction 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

$$C_P = \sum_{c \in Cl_{CPI}(P)} |P|_c C_c \qquad Cl_{CPI}(P) = \{\alpha, \beta, \gamma\}$$

Class c	$\lvert P\rvert_c$
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

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

We shall write π for a clock period. Of course

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

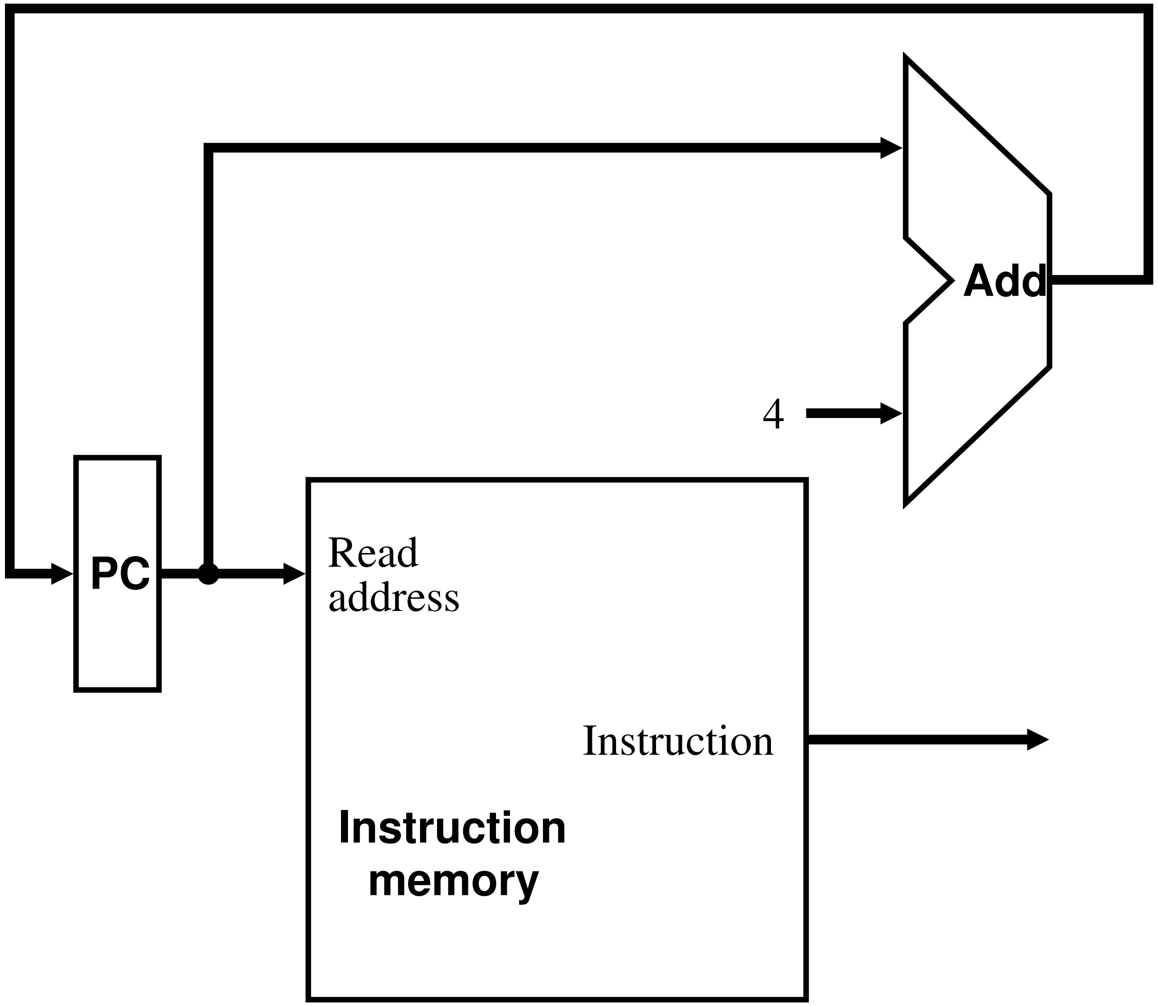
It also follows that the time taken to execute P is

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

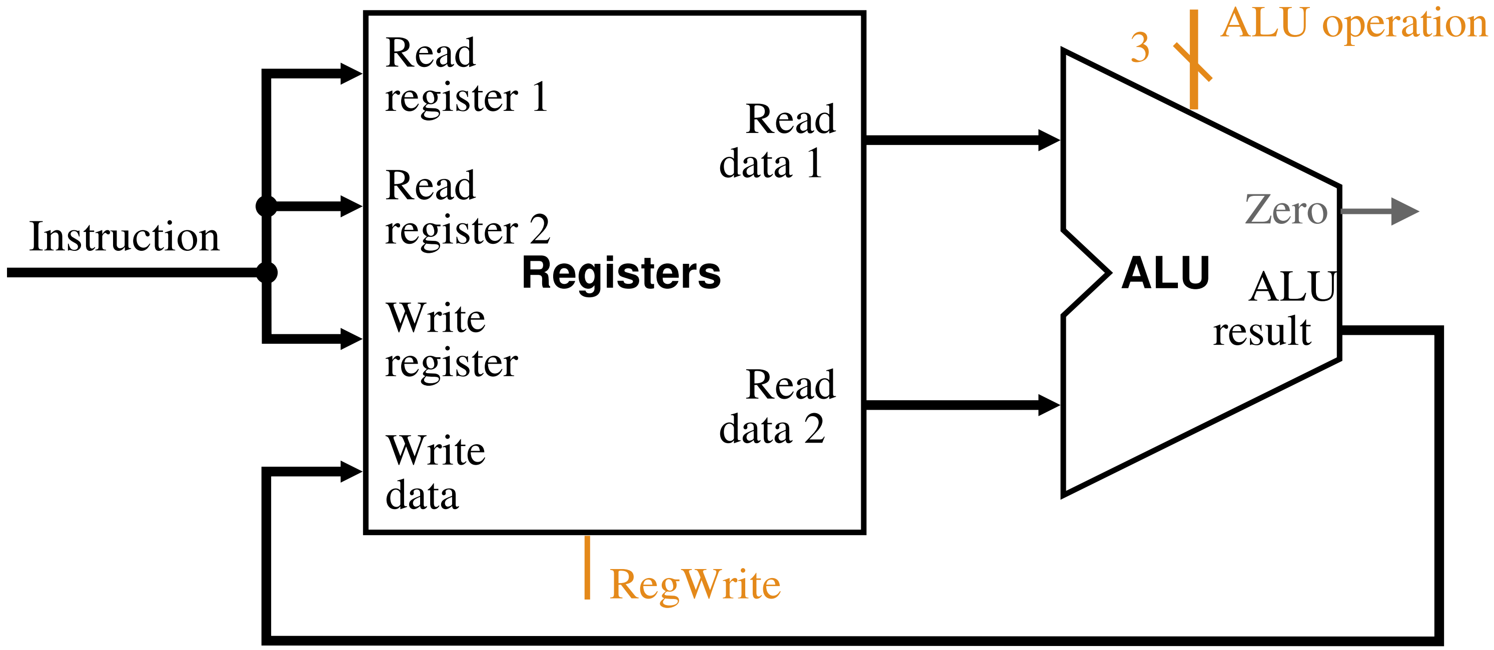
Overview: Datapath Components

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

Basic Instruction Fetch and PC Increment

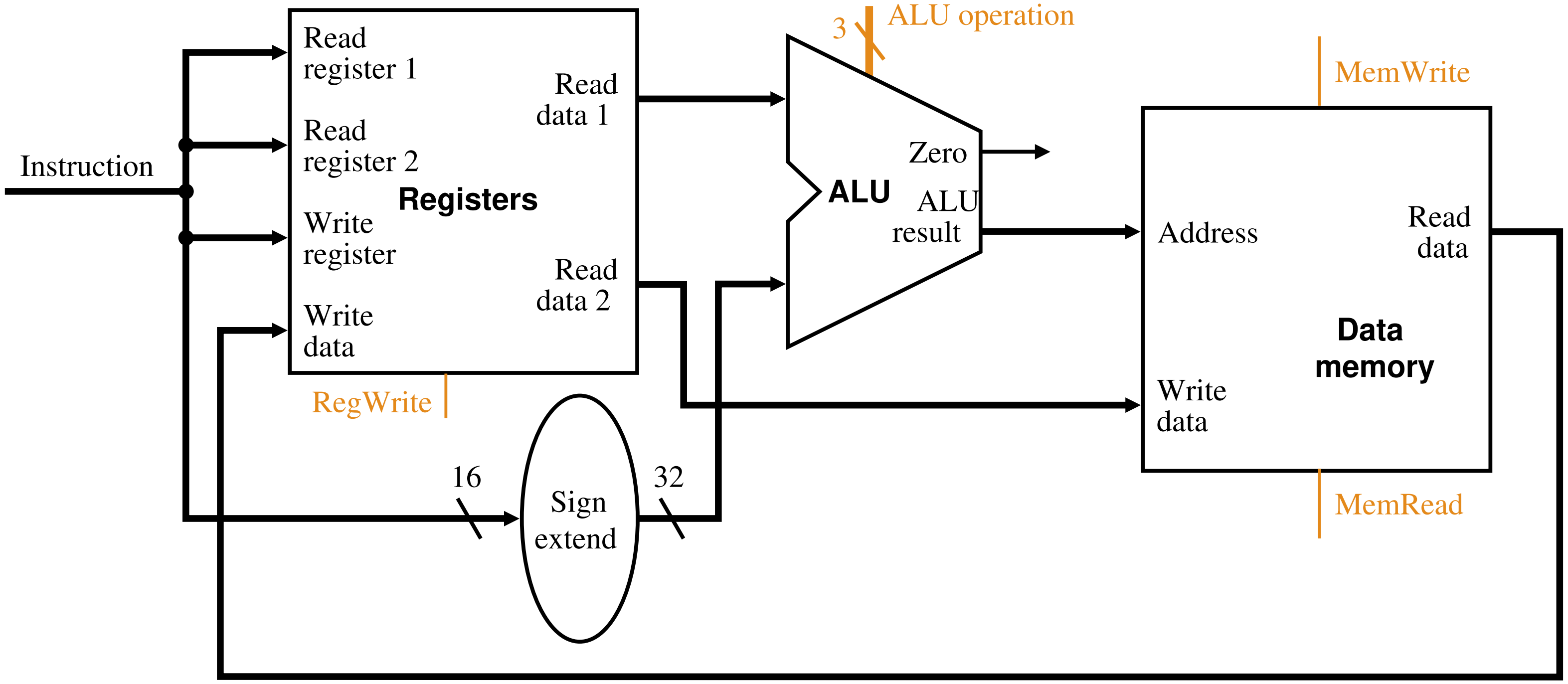


R-Format Instructions



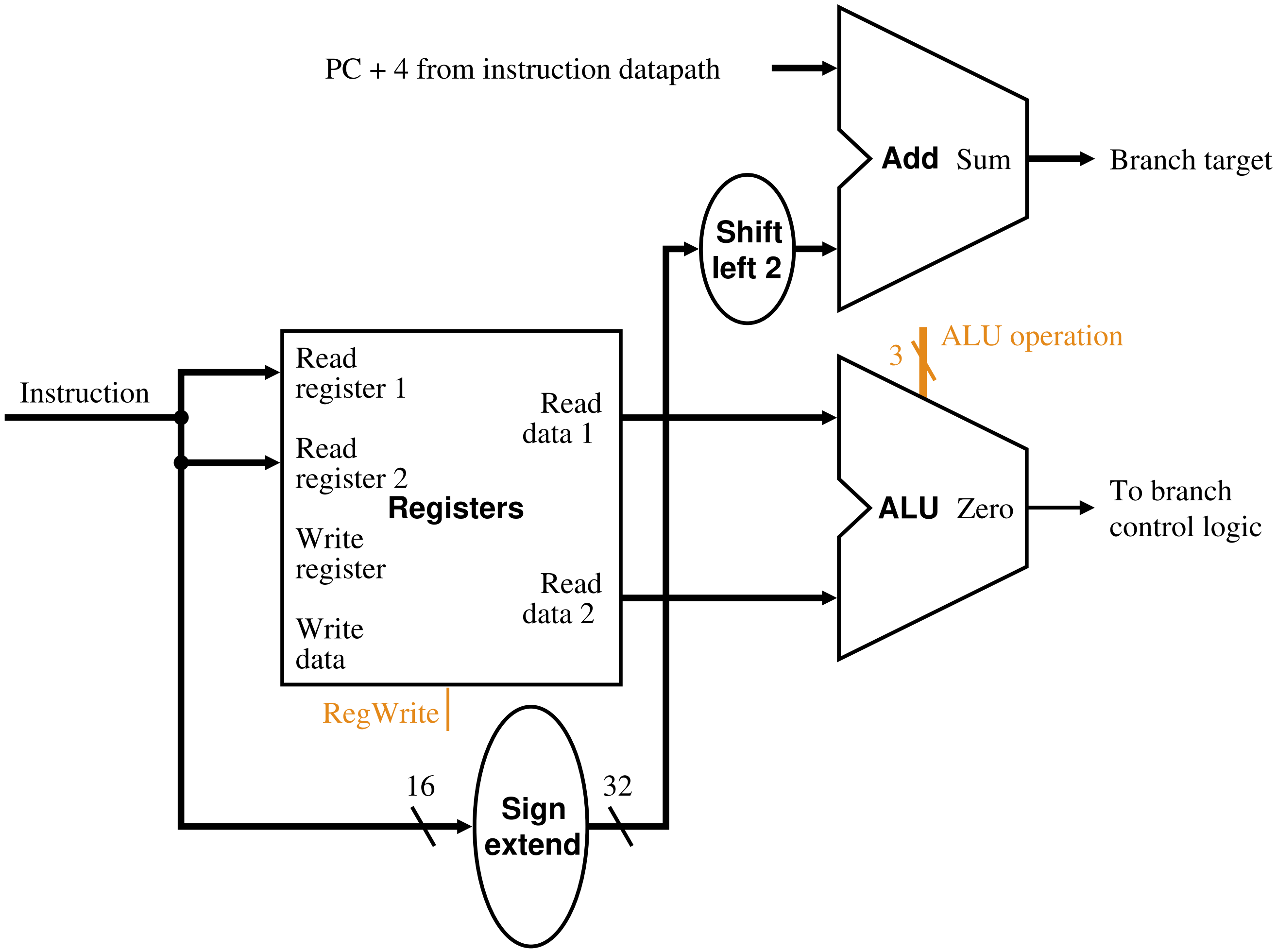
- These have the form *name* R_1, R_2, R_3.
- Machine instruction ($\mathtt{I}[31-0]$) passed along 32-bit wide `Instruction` bus; the fields are divided up …
- The numbers of R_2 and R_3 ie $\mathtt{I}[25-21]$ and $\mathtt{I}[20-16]$ are passed along `Read register 1` and `Read register 2`
- Destination register number $\mathtt{I}[15-11]$ passed along 5 bit `Write register` bus.
- Source operands appear on the `Read data 1` and `Read data 2`
- ALU is set by control and result is written …

Load/Store Instructions



- We examine `lw` $R_1,\ k(R_2)$ where $R_1 := !W[!R_2 + sx(k)]$.
- Source register R_2 number `I`$[25-21]$ goes to `Read register 1`
- 16-bit k in `I`$[15-0]$ 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, the 32-bit word $!R_2 + sx(k)$.
- `Data memory` sends contents at a into `Write data`.
- The destination register R_1 number `I`$[20-16]$ is in `Write register`, so the contents at a 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 ...

- use the ALU line `Zero` with ALU calculating $!R_1 - !R_2$

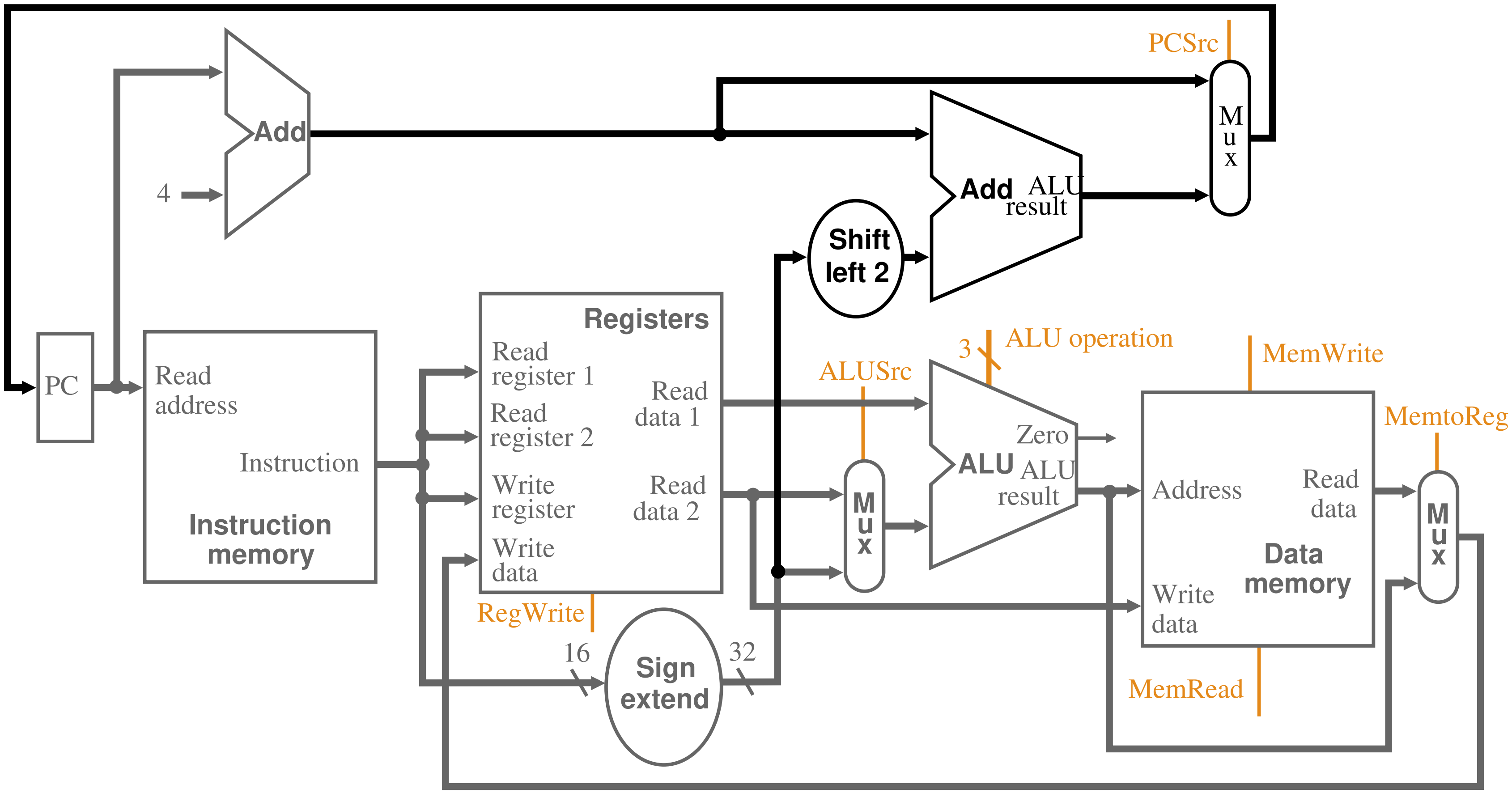
$!R_1 = !R_2$	$\vec{r} \stackrel{\text{def}}{=} !R_1 - !R_2$	`Zero`
True	$= \vec{0}$	1
False	$\neq \vec{0}$	0

- Update the PC:

$$\begin{array}{rll} (!\mathtt{PC}+4)+(4*\text{address}) & \text{if} & \mathtt{Zero}=1 \\ (!\mathtt{PC}+4) & \text{if} & \mathtt{Zero}=0 \end{array}$$

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.

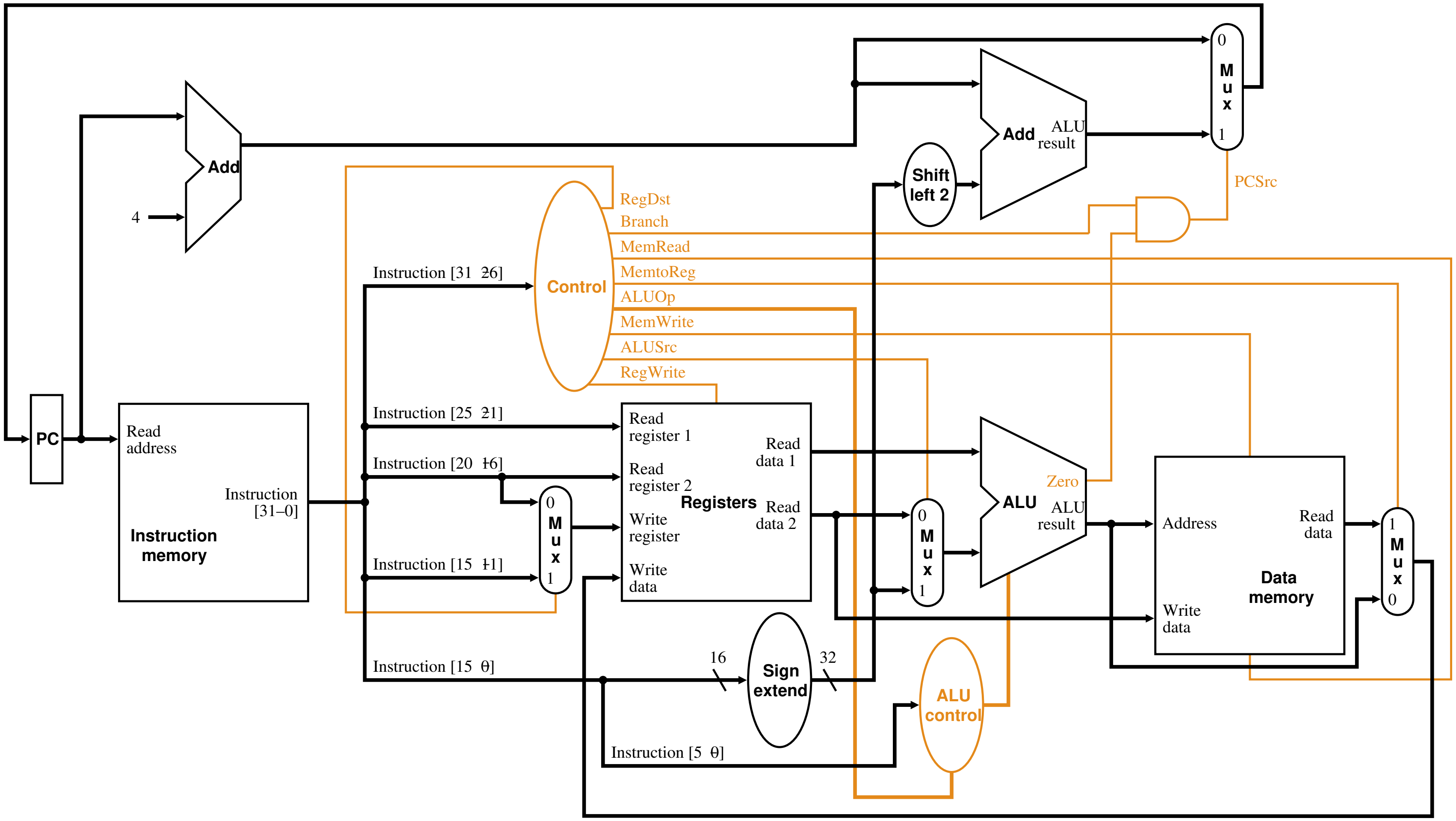

PCSrc
Add
4
M
u
x
Add ALU result
Shift left 2
Registers
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
PC
Read address
Instruction
Instruction memory
ALUSrc
3
ALU operation
MemWrite
MemtoReg
Zero
ALU
ALU result
Address
Read data
Data memory
Write data
RegWrite
16
Sign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

Multiplexor Settings

- `PCSrc`:
  - R-format, data transfer: next instruction at address $!\texttt{PC}+4$. So `PCSrc`=0.
  - Branch instruction executed, conditional true, next instruction at address $(!\texttt{PC}+4)+(sx(\texttt{I}[15-0])\ll 2)$. So `PCSrc`=1.
- `ALUSrc`:
  - R-format and `beq`: second ALU operand from `Read data 2`.
  - `lw` and `sw`: second ALU operand from sign extend.

Designing Control

- Given an instruction, control sets ALU etc.
- The elements which have control lines/busses are
  - the multiplexors and memory elements, and
  - the ALU.
- There are two control units, **Control** and **ALU Control**.
- Settings for multiplexors and memory elements determined completely by the instruction opcode $\texttt{I}[31-26]$; setting of ALU also requires the funct field $\texttt{I}[5-0]$.


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

Each control unit is a combinational circuit—to construct each one, all we need is a truth table.

Unit	Inputs	Outputs
`Control`	`I`$[31-26]$	settings for mult'rs & mem's
`ALU Control`	`I`$[31-26]$ and `I`$[5-0]$	settings for the ALU

- The opcode `I`$[31-26]$ must be fed into `Control`.
- For our small selection of instructions there are only *four* opcodes, for `lw`, `sw`, `beq` and R-format.
-
  - `lw` and `sw` require ALU addition.
  - `beq` require ALU subtraction.
  - R-format sets ALU using funct, ie `I`$[5-0]$.

  So there are only $4-1=3$ different situations.
- we use `Control` to feed simpler output signals to `ALU Control` along a 2-bit bus `ALUop`.

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

The situation now is

Unit	Inputs	Outputs
`C'l`	`I`[31 – 26]	settings for mult'rs & mems
`ALU C'l`	`ALUOp0`, `ALUOp1` and `I`[5 – 0]	settings for the ALU

Building Control

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

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

Signal Name	Effect when 0	Effect when 1
`RegDst`	The register destination number for the `Write register` comes from the rt field.	The register destination number for the `Write register` comes from the rd field.
`Branch`	The contents of `PC` can only be updated with !`PC` + 4	The contents of `PC` may be updated with a branch target; this only happens if `Zero` is high.
`RegWrite`	The register file cannot be written to.	The register specified by the value of the `Write register` input is written with the value of the `Write data` input.

Inst	Opcode field						RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp2
`lw`	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0
`sw`	1	0	1	0	1	1	X	1	X	0	0	1	0	0	0
`beq`	0	0	0	1	0	0	X	0	X	0	0	0	1	0	1
R-f't	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0

Instruction (ALU operation)	ALUOp1	ALUOp2	Funct Field						ALU-Control Output		
`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
`or` (*OR*)	1	0	1	0	0	1	0	1	0	0	1
`slt`	1	0	1	0	1	0	1	0	1	1	1