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’ ∈ \{‘a’, ‘g’, ‘r’, ‘t’\} and 4 ∉ \{‘a’, ‘g’, ‘r’, ‘t’\}.
- \(\mathbb{N} \stackrel{\text{def}}{=} \{0, 1, 2, \ldots\}\) set of natural numbers.
- \(\mathbb{Z} \stackrel{\text{def}}{=} \{\ldots, −2, −1, 0, 1, 2, \ldots\}\) 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

```
n: int  or  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 $uvw x$ denotes a single typical element of $A^4$, ie $uvw x \in A^4$.

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: \{1, 2, 3, 5\} \rightarrow \{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

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

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 \( \{ 3, 8, 9 \} \rightarrow \{ 4, 7, 8, 10, 300 \} \). 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 \( f:A \rightarrow B \) 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?

Computer = Hardware + Software
Software (Programs)

\[
\begin{align*}
    a &:= 0; \\
    c &:= 4; \\
    L &\quad a := a + c; \\
    c &:= c - 1; \\
    \text{if } c &\geq 1 \text{ then } L; \\
    \text{exit};
\end{align*}
\]
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

Hardware = \underbrace{\text{CPU}}_{\text{Control + Datapath}} + \text{Memory} + \text{I/O}
How does a computer work?

Central Processing Unit

- Control
- Datapath

Memory

Input / Output

Bus
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 *organizes* 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 $\text{add } R, A, B$.

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 $I = \text{add } R, A, B$

IR 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

0 0
1 1
2 2
3
4
5
6
7
8
9
10
11

8 bits

32 bits
- 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.
NB contents at T O M would be corresponding binary
Integers are stored in word locations—they can be big! Store $2^{31}$ as $\overline{10}^{bin} = 1000 \ldots 000^{bin}$—a 32-bit word.

7 will also be stored in a word location, as $\overline{0111}^{bin}$.

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**.
block 0

block 1

block B-1

K cells

K cells

2^{n-1}

C-1

1

0

tag

0

1

K-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 $b \times K$. 
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 \( [-] : S \rightarrow \mathbb{N} \)
- 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 $\mathbb{B}^k$, e.g. $10010^\text{bin} \in \mathbb{B}^5$.
- We often use $d$ or $d_i$ to stand for a binary digit.
- Write a $k$-digit binary number as

$$d_{k-1} \ldots d_0^\text{bin} \in \mathbb{B}^k$$
- $d_{k-1} \ldots d_0^{bin}$ is a sequence of digits.
- It denotes an integer $[d_{k-1} \ldots d_0^{bin}]$.

\[
[10^{bin}] = 1 \times 2^1 + 0 \times 2^0 = 2^1 + 0 = 2 + 0 = 2
\]
\[
[10100^{bin}] = 2^4 + 0 + 2^2 + 0 + 0 = 16 + 0 + 2 + 0 + 0 = 20
\]

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

In general

\[
[d_{k-1}d_{k-2} \ldots d_0^{bin}] = d_{k-1} \times 2^{k-1} + d_{k-2} \times 2^{k-2} + \ldots + d_0 \times 2^0
\]
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

$$
[d_{k-1} \ldots d_0 r] \overset{\text{def}}{=} [d_{k-1}] \cdot r^{k-1} + [d_{k-2}] \cdot r^{k-2} + \ldots + [d_0] \cdot r^0
$$
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 \[0\] def \(= 0\), \[1\] def \(= 1\), \ldots \[E\] def \(= 14\), and \[F\] def \(= 15\).
- Thus \(B1A_{\text{hex}}\) represents the integer 2842: here we have \(k = 3\) digits and \(B\) in position 2

\[
\]
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 $d_{k-1} \ldots d_0^{\text{bin}}$ 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 $000011^{\text{bin}}$. 
- We call 0 and 1 **complements** of each other, and write \( \bar{0} \overset{\text{def}}{=} 1 \) and \( \bar{1} \overset{\text{def}}{=} 0 \).

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

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

  $s_3 = (0 + 0 + 1) \ mod \ 10 = 1$
  $s_2 = (9 + 4 + 0) \ mod \ 10 = 3$
  $s_1 = (2 + 3 + 1) \ mod \ 10 = 6$
  $s_0 = (7 + 6 + 0) \ mod \ 10 = 3$
We describe formally the **digitwize algorithm** to compute \( \vec{s}^{\text{dec}} \overset{\text{def}}{=} \vec{d}^{\text{dec}} + \vec{d}'^{\text{dec}} \).

- \( \text{carry}_0 \overset{\text{def}}{=} 0. \)

- If \( i \geq 0 \) then \( s_i \overset{\text{def}}{=} (d_i + d'_i + \text{carry}_i) \ MOD \ 10 \) and \( \text{carry}_{i+1} \overset{\text{def}}{=} (d_i + d'_i + \text{carry}_i) \ \text{DIV} \ 10 \)

- 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:

<table>
<thead>
<tr>
<th>(\vec{d}^{bin})</th>
<th>(\vec{d}'^{bin})</th>
<th>(carry^{bin})</th>
<th>(\vec{s}^{bin})</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Correctness

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

$$\lfloor \vec{s}_{bin} \rfloor = \lfloor \vec{d}_{bin} \rfloor + \lfloor \vec{d'}_{bin} \rfloor$$

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 . . .

\[
\begin{array}{cccc}
\vec{d}^{bin} & 1 & 0 & 1 \\
\vec{d}'^{bin} & + & 1 & 1 & 0 \\
\vec{s}^{bin} & 1 & 0 & 1 & 1 \\
\end{array}
\]
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 bit in position \( k \) will be 0 for positive integers and zero, and 1 otherwise.

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

\[
\langle d_k \ldots d_0^{\text{bin}} \rangle \overset{\text{def}}{=} -d_k \times 2^k + \left[ d_{k-1} \ldots d_0^{\text{bin}} \right] = -d_k \times 2^k + \left( d_{k-1} \times 2^{k-1} + d_{k-2} \times 2^{k-2} + \ldots + d_0 \times 2^0 \right)
\]
If $k = 5$, then

$\langle 000111^{bin} \rangle = -0 \times 2^5 + 7 = 7$

but

$\langle 100111^{bin} \rangle = -1 \times 2^5 + 7 = -32 + 7 = -25$

Note also that

$25 = \langle 011001^{bin} \rangle$. 
- If \( m \) is any integer, its **negation** is defined to be \(-1 \times m\).

- Subtraction performed by adding the negation of an integer.

- The negation of the integer represented by \( \overrightarrow{b}^{\text{bin}} \), is represented by the \( k+1 \) 2s-complement number number \( \overrightarrow{b}^{\text{bin}} + 1^{\text{bin}} \).

- 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:

\[
\begin{bmatrix} \vec{s}^{\text{bin}} \end{bmatrix} = \begin{bmatrix} \vec{b}^{\text{bin}} \end{bmatrix} + \begin{bmatrix} \vec{b}'^{\text{bin}} \end{bmatrix} \quad (EQ?)
\]

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

- If true, algorithm (or computer) is correct.
On paper: \([0111 \text{ } \text{bin}] = 7 = 3 + 4 = [011 \text{ } \text{bin}] + [100 \text{ } \text{bin}]\).

3-bits only: \([111 \text{ } \text{bin}] = 7 = 3 + 4 = [011 \text{ } \text{bin}] + [100 \text{ } \text{bin}]\).
<table>
<thead>
<tr>
<th>$\vec{b}^{bin}$</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vec{b}'^{bin}$</td>
<td>+</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\vec{s}^{bin}$</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- 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

- $\begin{bmatrix} s_{k-1} \ldots s_0 \text{ bin} \end{bmatrix} = \begin{bmatrix} b_{k-1} \ldots b_0 \text{ bin} \end{bmatrix} + \begin{bmatrix} b'_{k-1} \ldots b'_0 \text{ bin} \end{bmatrix}$
- $0 \leq \begin{bmatrix} b_{k-1} \ldots b_0 \text{ bin} \end{bmatrix} + \begin{bmatrix} b'_{k-1} \ldots b'_0 \text{ bin} \end{bmatrix} \leq 2^k - 1$

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!!!

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

\[
\begin{array}{c}
\hline
1 & 1 & 1 \\
+ & 1 & 1 & 0 \\
\hline
(1) & 1 & 0 & 1 \\
\end{array}
\]

\[
\langle 101 \rangle = -3 = \langle 111 \rangle + \langle 110 \rangle
\]
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 $P$ 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.

- \( (s_k \ldots s_0^{\text{bin}}) = (b_k \ldots b_0^{\text{bin}}) + (b'_k \ldots b'_0^{\text{bin}}) \)
- \(-2^k \leq (b_k \ldots b_0^{\text{bin}}) + (b'_k \ldots b'_0^{\text{bin}}) \leq 2^k - 1\)
- \(P\) 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 \( \cdot : \mathbb{B}^2 \to \mathbb{B} \) and \( +_{or} : \mathbb{B}^2 \to \mathbb{B} \). Refer to these as AND and OR.

- We will sometimes write sequences \( d_k \ldots d_0 \) 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 \(d \cdot d'\).

■ \(d \cdot d'\) is 1 precisely when both \(d\) and \(d'\) are 1, and otherwise \(d \cdot d'\) 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 \( \cdot \) 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 \( \cdot \).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>( A \cdot B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>( A +_{or} B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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

\[ \mathbb{B}^k \rightarrow \mathbb{B}^{k'+k} \quad \overrightarrow{b} \mapsto \overrightarrow{0b} \]

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

- To move a $k+1$ bit 2s-complement number $\vec{δb}$ 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

$$B^{k+1} \rightarrow B^{k'+k+1} \quad \vec{δb} \mapsto \vec{δδb}$$

where $\vec{δ}$ means a sequence of $k'$ $\vec{δ}$s.

- We will sometimes write the output as $sx(\vec{δb})$. 
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: 
\((\mathbb{B}, \cdot, +_or, \overline{\cdot}, 0, 1)\).

■ The functions satisfy certain properties.

<table>
<thead>
<tr>
<th>Property</th>
<th>(+_or)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idempotent</td>
<td>(A +_or A = A)</td>
</tr>
<tr>
<td>Complement</td>
<td>(A +_or \overline{A} = 1)</td>
</tr>
<tr>
<td>Associative</td>
<td>(A +_or (B +_or C) = (A +_or B) +_or C)</td>
</tr>
</tbody>
</table>

■ \(\overline{A} \cdot B +_or C\) stands for \(((\overline{A}) \cdot B) +_or C\).
Implementing Functions by Circuits

- We implement $n \mid m$-ary functions over $\mathbb{B}$ as digital circuits. For example, $f: \mathbb{B}^3 \to \mathbb{B}^4$ 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, e.g. 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 \( \cdot \); there are other gates . . .
There are also \( m \) \( 1 \)-ary versions of \( . \) and \( +_\text{or} \). If \( m = 3 \) then \( \mathbb{B}^3 \rightarrow \mathbb{B} \) and we write \( A \cdot B \cdot C \) for the output to input \((A, B, C)\). The output \( A \cdot B \cdot C = 1 \) only when \( A = 1 \) and \( B = 1 \) and \( C = 1 \).

\[
\begin{array}{ccc|c}
A & B & C & A \cdot B \cdot C \\
\hline
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
\vdots & & & \\
1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 \\
\end{array}
\]
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:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$g(A, B)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

We construct a **Sum-of-Products** expression …
<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$f_0(A, B, C)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Step 2: Implement S-o-P Example

- We have

\[ g(A, B) = \overline{A} \cdot \overline{B} + \text{or} \ A \cdot B \]

\[ f_0(A, B, C) = \overline{A} \cdot \overline{B} \cdot \overline{C} + \text{or} \ \overline{A} \cdot B \cdot C + \text{or} \ A \cdot \overline{B} \cdot C + \text{or} \ A \cdot B \cdot C \]

- 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:

\[
\overline{A} \cdot \overline{B} \cdot C +_{or} \overline{A} \cdot \overline{B} \cdot C +_{or} \overline{A} \cdot \overline{B} \cdot C +_{or} A \cdot B \cdot C
\]

\[
= \overline{A} \cdot \overline{B} \cdot (\overline{C} +_{or} C) +_{or} (\overline{A} +_{or} A) \cdot B \cdot C
\]

\[
= \overline{A} \cdot \overline{B} \cdot 1 +_{or} 1 \cdot B \cdot C
\]

\[
= \overline{A} \cdot \overline{B} +_{or} B \cdot C
\]
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 ($d_i \in \mathbb{B}$)

<table>
<thead>
<tr>
<th>$C_2$</th>
<th>$C_1$</th>
<th>$C_0$</th>
<th>$D_7$</th>
<th>$D_6$</th>
<th>$D_5$</th>
<th>$D_4$</th>
<th>$D_3$</th>
<th>$D_2$</th>
<th>$D_1$</th>
<th>$D_0$</th>
<th>$M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$d_7$</td>
<td>$d_6$</td>
<td>$d_5$</td>
<td>$d_4$</td>
<td>$d_3$</td>
<td>$d_2$</td>
<td>$d_1$</td>
<td>$d_0$</td>
<td>$d_5$</td>
</tr>
</tbody>
</table>

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

\[ D(2^n - 1) \]

\[ D2, D1, D0 \]

\[ C \]

\[ D3, D2, D1, D0 \]

\[ C1, C0 \]
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 $carry_{i+1}$. 
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>CarryIn</th>
<th>CarryOut</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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” \texttt{slt} function which returns

\[ r_{31} = 0, \ldots, r_1 = 0, r_0 = 1 \quad \text{if} \quad (|\vec{a}_{bin}| < |\vec{b}_{bin}|) \]

\[ r_{31} = 0, \ldots, r_1 = 0, r_0 = 0 \quad \text{if} \quad \text{otherwise} \]

\begin{itemize}
  \item A single bit output line \texttt{Zero} which returns 1 when \textit{any} result is \(\vec{0}\), and 0 otherwise.
  \item A single bit output line which will hold 1 when a function causes overflow, and will hold 0 otherwise.
\end{itemize}
Result

Operation

CarryIn

CarryOut

a

b

Binvert

0

1

2

+
<table>
<thead>
<tr>
<th>Result</th>
<th>Operation</th>
<th>CarryIn</th>
<th>CarryOut</th>
<th>Overflow detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Binvert</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Less</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### a.

- Less
- CarryIn
- Operation
- Result
- CarryOut
- Overflow detection

### b.

- Less
- CarryIn
- Operation
- Result
- CarryOut
- Overflow detection
Set a31 0
Result0 a0
Result1 a1
Result2 a2
Result31 a31

Operation b31
Zero

ALU0
Less
CarryIn
CarryOut

ALU1
Less
CarryIn
CarryOut

ALU2
Less
CarryIn
CarryOut

ALU31
Less
CarryIn
CarryOut

Overflow

Bnegate

...
Overview: Sequential Circuits

- Sequential circuits may have different “outputs” 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

Qa

Qb
### Stable States

<table>
<thead>
<tr>
<th>$S$</th>
<th>$R$</th>
<th>$Q_a$</th>
<th>$Q_b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Unstable States

<table>
<thead>
<tr>
<th>$S$</th>
<th>$R$</th>
<th>$Q_a$</th>
<th>$Q_b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$S$ 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.
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:

![Diagram of a latch with inputs and outputs labeled: data in, clock, write enable, read enable, data out. The latch symbol is D, C, Q.]
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:

- Read register number 1
- Read register number 2
- Write register
- Write data

(COPYRIGHT 1998 MORGAN KAUFMANN PUBLISHERS, INC. ALL RIGHTS RESERVED.)
Building Static Random Access Memory

Computers count in binary.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kilo (K)</td>
<td>2^{10}</td>
<td></td>
</tr>
<tr>
<td>Mega (M)</td>
<td>2^{20}</td>
<td></td>
</tr>
<tr>
<td>Giga (G)</td>
<td>2^{30}</td>
<td></td>
</tr>
<tr>
<td>Tera (T)</td>
<td>2^{40}</td>
<td></td>
</tr>
</tbody>
</table>
- The number of bits in a cell is sometimes called the chip’s **width**, and the number of cells its **height**.

- We talk about an $h \times w$-SRAM.
The 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.
Recall

\[
a := 0; \quad c := 4;
\]

\[
\text{L} \quad a := a + c;
\]

\[
c := c - 1;
\]

\[
\text{if } c \geq 1 \text{ then } \text{L};
\]

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 (e.g., $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

<table>
<thead>
<tr>
<th>–</th>
<th>$t1$</th>
<th>$t2$</th>
<th>$t0$</th>
<th>–</th>
<th><em>add</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>01001</td>
<td>01010</td>
<td>01000</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

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 $\omega$ 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 $R := \overrightarrow{0111}_{bin}$.

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

\[
\text{addi } R_1, R_2, 5 \quad \text{add } R_1, R_2, R_3
\]

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, $t1, 24`.
- The semantics of the instruction is given by `$t0 := !t1 + 24$`.
- We sometimes refer to data (operands) given by immediate addressing as **constants**.
- `+` means digitwise 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

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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $R_1$, $R_2$, $R_3$</td>
<td>$R_1 := !R_2 + !R_3$</td>
</tr>
<tr>
<td>sub $R_1$, $R_2$, $R_3$</td>
<td>$R_1 := !R_2 - !R_3$</td>
</tr>
<tr>
<td>addi $R_1$, $R_2$, $k$</td>
<td>$R_1 := !R_2 + sx(k)$</td>
</tr>
</tbody>
</table>

See example ...
# Data Transfer Instructions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw R_1, k(R_2)</code></td>
<td><code>R_1 := !W[sx(k) + R_2]</code></td>
</tr>
<tr>
<td><code>sw R_1, k(R_2)</code></td>
<td><code>W[sx(k) + R_2] := !R_1</code></td>
</tr>
<tr>
<td><code>lb R_1, k(R_2)</code></td>
<td><code>R_1 := sx(!B[sx(k) + R_2])</code></td>
</tr>
</tbody>
</table>
## Conditional Instructions

**Syntax**

**Semantics**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>$R_1, R_2, L$</td>
<td>if $!R_1 = !R_2$ then goto $L$</td>
</tr>
<tr>
<td>bne</td>
<td>$R_1, R_2, L$</td>
<td>if $!R_1 \neq !R_2$ then goto $L$</td>
</tr>
<tr>
<td>slt</td>
<td>$R_1, R_2, R_3$</td>
<td>if $!R_2 &lt; !R_3$ then $R_1 := 1$ else $R_1 := 0$</td>
</tr>
</tbody>
</table>

```python
repeat:  add $t1, $t1, $t0
        addi $t0, $t0, 1
        bne $t0, $t2, repeat
next - instruction
```
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 addresses.
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 known 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* $R_1, R_2, R_3$

<table>
<thead>
<tr>
<th>Fields</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
<tr>
<td>$\bar{0}$</td>
<td>$R_2$</td>
<td>$R_3$</td>
<td>$R_1$</td>
<td>$\bar{0}$</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>$\bar{0}$</td>
<td>$s1$</td>
<td>$s2$</td>
<td>$t0$</td>
<td>$\bar{0}$</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100000</td>
<td></td>
</tr>
</tbody>
</table>


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

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>addi, andi, ori, lw, sw, lb, lbu, sb</td>
<td>$R_2$</td>
<td>$R_1$</td>
<td>$k$</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0 contents always zero</td>
</tr>
<tr>
<td>$v0</td>
<td>2 expression evaluation and function results</td>
</tr>
<tr>
<td>$v1</td>
<td>3 expression evaluation and function results</td>
</tr>
<tr>
<td>$a0</td>
<td>4 first argument component (preserved across call)</td>
</tr>
<tr>
<td>$t0</td>
<td>8 temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t1</td>
<td>9 temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t2</td>
<td>10 temporary (not preserved across call)</td>
</tr>
<tr>
<td>$fp</td>
<td>30 frame pointer (preserved across call)</td>
</tr>
</tbody>
</table>
Example

What is the machine code for \texttt{lw} 
$s1, 8($t1) \? 

- \textit{name} \ R_1, \ k(R_2) \text{ is I-format.}
- \text{Use look-up table for } \texttt{lw} \text{ opcode: 35.}
- \text{Use register number table for } \texttt{s1}: 17 \text{ and for } \texttt{t1}: 9.

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>16 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{lw}</td>
<td>\texttt{R}_2</td>
<td>\texttt{R}_1</td>
<td>\texttt{k}</td>
</tr>
<tr>
<td>100011</td>
<td>01001</td>
<td>10001</td>
<td>000000000000001000</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>beq</td>
<td>$R_1$</td>
<td>$R_2$</td>
<td>$L$</td>
</tr>
</tbody>
</table>
Looking up the opcode and register numbers is easy!

The address field specifies the distance (offset) \(ow\) in \textit{words}\ from the address of the \textit{next instruction} \(I\) \textit{in memory} to the address of the labelled \textit{instruction} \(I'\)

\[
\begin{align*}
& a \quad \text{beq } R_1, R_2, L \\
& a + 4 \quad \boxed{I \ldots \ldots} \quad \text{1 word} \\
& \vdots \quad 2,3,4, \ldots \quad \text{words} \\
& \quad \boxed{\ldots \ldots} \quad ow \quad \text{words} \\
& a' \quad L : \quad I' \ldots \ldots
\end{align*}
\]
To be precise $ow = (|\text{address}|) \in \mathbb{Z}$ 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?

\[
\begin{align*}
    a & \quad \text{beq } R_1, R_2, L \\
    a+4 & \quad \text{I \ldots \ldots 1 word} \\
    \vdots & \quad \text{2,3,4, \ldots words} \\
    \ldots & \quad \text{ow words} \\
    a' = (a+4) + ob & \quad L: \quad \text{I' \ldots \ldots }
\end{align*}
\]

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

\[
oob = 4 \times ow = 4 \times (|\text{address}|) \in \mathbb{Z}
\]
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

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

- 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

\[ a' = (a + 4) + ob \in \mathbb{Z} \quad \text{if test TRUE} \]
\[ a + 4 \in \mathbb{Z} \quad \text{if test FALSE} \]

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

\[
\begin{align*}
\text{PC} & := (\neg \text{PC} + 4) + (4 \times \text{address}) \quad \text{if test TRUE} \\
\text{PC} & := (\neg \text{PC} + 4) \quad \text{if test FALSE}
\end{align*}
\]
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

Instruction memory

PC

Read address

Instruction

Add

4
# R-Format Instructions

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#R₂</td>
<td>#R₃</td>
<td>#R₁</td>
<td>0</td>
<td>name</td>
</tr>
</tbody>
</table>

## Diagram

- **Instruction**
- **Read register 1**
- **Read register 2**
- **Write register**
- **Write data**
- **Read data 1**
- **Read data 2**
- **ALU**
- **ALU operation**
- **Zero**
- **ALU result**
- **RegWrite**
These have the form name $R_1, R_2, R_3$.

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

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

Destination register number $I[15-11] \in \mathbb{B}^{5}$ 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

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>16 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>#R₂</td>
<td>#R₁</td>
<td>k</td>
</tr>
</tbody>
</table>
- We examine \( \text{lw } R_1, k(R_2) \) where \( R_1 \leftarrow W[!R_2 + sx(k)] \).

- The number of register \( R_2 \) ie \( I[25 - 21] \in \mathbb{B}^5 \) goes to Read register 1.

- 16-bit representation of \( k \) ie \( I[15 - 0] \in \mathbb{B}^{16} \) 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 \( I[20 - 16] \in \mathbb{B}^5 \) is in Write register, so \( !W[a] \in \mathbb{B}^{32} \) is written in \( R_1 \).
Branch if Equal Instruction

- Look at \texttt{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 \( \vec{r} \overset{\text{def}}{=} !R_1 - !R_2 \). \texttt{Zero} tests if \( \vec{r} = \vec{0} \).

<table>
<thead>
<tr>
<th>( !R_1 =!R_2 )</th>
<th>( \vec{r} \overset{\text{def}}{=} !R_1 - !R_2 )</th>
<th>\texttt{Zero}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{True}</td>
<td>= \vec{0}</td>
<td>1</td>
</tr>
<tr>
<td>\text{False}</td>
<td>\neq \vec{0}</td>
<td>0</td>
</tr>
</tbody>
</table>
PC + 4 from instruction datapath

Instruction

Read register 1
Read register 2
Write register
Write data

Registers

Read data 1
Read data 2

ALU

Shift left 2

Add Sum

Branch target

RegWrite

16 32

Sign extend

3

ALU operation

To branch control logic

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>16 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>#R₁</td>
<td>#R₂</td>
<td>address</td>
</tr>
</tbody>
</table>
Executing beq updates the PC:

\[ PC := (\!PC + 4) + (4 \times \text{address}) \quad \text{if} \quad \text{Zero} = 1 \]
\[ PC := (\!PC + 4) \quad \text{if} \quad \text{Zero} = 0 \]

\[ a \overset{\text{def}}{=} (\!PC) \in \mathbb{Z} \]

\[ a + 4 \]

\[ a' = (a + 4) + 4 \times (\text{|address|}) \]

Branch target

\[ \text{beq } R_1, R_2, L \]

\[ \text{I . . . . . .} \quad 1 \text{ word} \]

\[ \vdots \quad 2, \ldots, (\text{|address|}) \text{ words} \]

\[ L : \text{I'} . . . . . . \]
There is a control $\text{PCSrc}$ for a multiplexor. In fact (see later on) $\text{PCSrc}$ is set equal to $\text{Zero}$ when a branch instruction is executed.

- So if $\text{PCSrc}=\text{Zero}=0$ then the PC will—see circuit—be updated with $\text{PC} + 4$.

- And if $\text{PCSrc}=\text{Zero}=1$ the PC is updated with $(\text{PC} + 4) + (sx(\text{I}[15-0]) \ll 2) = (\text{PC} + 4) + (4 \times \text{address})$.

Eg $1011 \ll 2 \overset{\text{def}}{=} 1011.00$ and

$$\langle 1011.00 \rangle = -20 = 4 \times -5 = 4 \times \langle 1011 \rangle$$
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.
The diagram illustrates the control logic for a computer system, focusing on the instruction memory and the data memory operations. The control logic includes the following steps:

1. **Instruction Memory**: Fetches the instruction from memory, with instructions like [31-0], [25-21], [29-16], [15-11], and [15-0].
2. **Zero Check (Zero)**: Checks if the ALU result is zero.
3. **Addressing Mode**: Determines the mode of addressing for memory operations.
4. **Data Memory**: Reads and writes data from memory.
5. **ALU Operations**: Performs arithmetic and logic operations on the data.
6. **ALU Control**: Controls the ALU operations based on the instruction's ALUOp field.
7. **RegWrite**: Writes the result back to a register.
8. **MemRead**: Reads data from memory.
9. **MemWrite**: Writes data to memory.
10. **Branch**: Determines if a branch instruction should be executed.
11. **MemToReg**: Transfers data from memory to a register.
12. **RegDst**: Determines which register to use in the ALU operation.
13. **ALUSrc**: Determines the source of the ALU operation.
14. **MemRead Data 1**: Reads data from memory for the first operand.
15. **MemRead Data 2**: Reads data from memory for the second operand.
16. **Write Register 1**: Writes data to a register.
17. **Write Register 2**: Writes data to a register.
18. **Read Register 1**: Reads data from a register.
19. **Read Register 2**: Reads data from a register.
20. **Add (Add)**: Adds the operands.
21. **Shift Left 2 (Shift left 2)**: Shifts the result left by two bits.
22. **Sign Extend (Sign extend)**: Extends the sign bit of the result.
23. **ALU Write result (ALU Write result)**: Writes the ALU result.
24. **Shift Write result (Shift Write result)**: Shifts the result before writing.
25. **PC Src (PC Src)**: Determines the source of the program counter.
26. **Data Memory Write Data (Data Memory Write Data)**: Writes data to memory.
27. **Data Memory Read Data (Data Memory Read Data)**: Reads data from memory.
28. **Data Memory Address (Data Memory Address)**: Determines the memory address for read or write operations.
29. **Data Memory Write Data (Data Memory Write Data)**: Writes data to memory.
30. **Data Memory Read Data (Data Memory Read Data)**: Reads data from memory.
31. **Data Memory Address (Data Memory Address)**: Determines the memory address for read or write operations.
32. **Data Memory Write Data (Data Memory Write Data)**: Writes data to memory.
33. **Data Memory Read Data (Data Memory Read Data)**: Reads data from memory.
34. **Data Memory Address (Data Memory Address)**: Determines the memory address for read or write operations.
35. **Data Memory Write Data (Data Memory Write Data)**: Writes data to memory.
36. **Data Memory Read Data (Data Memory Read Data)**: Reads data from memory.
37. **Data Memory Address (Data Memory Address)**: Determines the memory address for read or write operations.
38. **Data Memory Write Data (Data Memory Write Data)**: Writes data to memory.
39. **Data Memory Read Data (Data Memory Read Data)**: Reads data from memory.
40. **Data Memory Address (Data Memory Address)**: Determines the memory address for read or write operations.
41. **Data Memory Write Data (Data Memory Write Data)**: Writes data to memory.
42. **Data Memory Read Data (Data Memory Read Data)**: Reads data from memory.
43. **Data Memory Address (Data Memory Address)**: Determines the memory address for read or write operations.
44. **Data Memory Write Data (Data Memory Write Data)**: Writes data to memory.
45. **Data Memory Read Data (Data Memory Read Data)**: Reads data from memory.
46. **Data Memory Address (Data Memory Address)**: Determines the memory address for read or write operations.
47. **Data Memory Write Data (Data Memory Write Data)**: Writes data to memory.
48. **Data Memory Read Data (Data Memory Read Data)**: Reads data from memory.
49. **Data Memory Address (Data Memory Address)**: Determines the memory address for read or write operations.
50. **Data Memory Write Data (Data Memory Write Data)**: Writes data to memory.
51. **Data Memory Read Data (Data Memory Read Data)**: Reads data from memory.
52. **Data Memory Address (Data Memory Address)**: Determines the memory address for read or write operations.
53. **Data Memory Write Data (Data Memory Write Data)**: Writes data to memory.
54. **Data Memory Read Data (Data Memory Read Data)**: Reads data from memory.
55. **Data Memory Address (Data Memory Address)**: Determines the memory address for read or write operations.
56. **Data Memory Write Data (Data Memory Write Data)**: Writes data to memory.
57. **Data Memory Read Data (Data Memory Read Data)**: Reads data from memory.
58. **Data Memory Address (Data Memory Address)**: Determines the memory address for read or write operations.
59. **Data Memory Write Data (Data Memory Write Data)**: Writes data to memory.
60. **Data Memory Read Data (Data Memory Read Data)**: Reads data from memory.
61. **Data Memory Address (Data Memory Address)**: Determines the memory address for read or write operations.
62. **Data Memory Write Data (Data Memory Write Data)**: Writes data to memory.
63. **Data Memory Read Data (Data Memory Read Data)**: Reads data from memory.
64. **Data Memory Address (Data Memory Address)**: Determines the memory address for read or write operations.
65. **Data Memory Write Data (Data Memory Write Data)**: Writes data to 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

<table>
<thead>
<tr>
<th>Inst</th>
<th>Opcode field $I_{[31-26]}$</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1w</td>
<td>1 0 0 0 1 1</td>
<td>0 1 1 1 1 1 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>1 0 1 0 1 1</td>
<td>X 1 X 0 0 1 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>0 0 0 1 0 0</td>
<td>X 0 X 0 0 0 1 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-f’t</td>
<td>0 0 0 0 0 0</td>
<td>1 0 0 1 0 0 0 1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The table lists different instructions with their respective opcode fields and settings for controlling multiplexors and memory read/write enable.
<table>
<thead>
<tr>
<th>Instruction (ALU operation)</th>
<th>ALUOp1</th>
<th>ALUOp2</th>
<th>Funct Field = I[5-0]</th>
<th>ALU-Control Output: sets ALU operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw/sw (+)</td>
<td>0</td>
<td>0</td>
<td>X X X X X X</td>
<td>0 1 0</td>
</tr>
<tr>
<td>beq (-)</td>
<td>0</td>
<td>1</td>
<td>X X X X X X X</td>
<td>1 1 0</td>
</tr>
<tr>
<td>add (+)</td>
<td>1</td>
<td>0</td>
<td>1 0 0 0 0 0</td>
<td>0 1 0</td>
</tr>
<tr>
<td>sub (-)</td>
<td>1</td>
<td>0</td>
<td>1 0 0 0 1 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>or (OR)</td>
<td>1</td>
<td>0</td>
<td>1 0 0 1 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>slt (Less)</td>
<td>1</td>
<td>0</td>
<td>1 0 1 0 1 0</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>
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 (e.g., 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 $P$

<table>
<thead>
<tr>
<th>Instruction $I$</th>
<th>class</th>
<th>cycles $C_I$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw ...</td>
<td>$\alpha$</td>
<td>4</td>
</tr>
<tr>
<td>lw ...</td>
<td>$\alpha$</td>
<td>4</td>
</tr>
<tr>
<td>add ...</td>
<td>$\beta$</td>
<td>5</td>
</tr>
<tr>
<td>sub ...</td>
<td>$\beta$</td>
<td>5</td>
</tr>
<tr>
<td>bne ...</td>
<td>$\gamma$</td>
<td>6</td>
</tr>
<tr>
<td>sw ...</td>
<td>$\alpha$</td>
<td>4</td>
</tr>
</tbody>
</table>
Write $|P|_c$ for the number of instructions in $P$ of class $c$ and $C_c$ for the number of cycles to run any class $c$ instruction.

| Class $c$ | $|P|_c$ | Class $c$ | $C_c$ |
|-----------|--------|-----------|-------|
| $\alpha$  | 3      | $\alpha$  | 5     |
| $\beta$   | 2      | $\beta$   | 4     |
| $\gamma$  | 1      | $\gamma$  | 6     |

Thus the number of clock cycles to execute $P$ is

$$Total \ Clock \ Cycles \ = \ C_P = (3 \times 5) + (2 \times 4) + (1 \times 6) = 29$$
Write \( Cl_{CPI}(P) \) for the set of classes of instructions with identical CPI in a program \( P \).

\[
Total \ Clock \ Cycles \ = \ C_P \ \overset{\text{def}}{=} \sum_{c \in Cl_{CPI}(P)} |P|_c \ast C_c
\]

We shall write \( \pi \) for a clock period. Of course

\[
t_P = C_P \ast \pi \quad t_I = C_I \ast \pi
\]

It also follows that the time taken to execute \( P \) is

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