2. Computer hardware

1. Binary representation of data
2. Logic gates and circuits
3. A model computer
4. Assembler-language programs

Inquiry

- How is data represented in digital technology?
- How is data processed?
2. Computer hardware

**Topic objective**

2. Explain the storage and processing of data at the machine level

**Why study hardware?**

- All computing occurs in a physical medium
- Processors aren’t like the brain
- Digital computers have a common *stored-program* architecture

**Background outcomes**

2.0a Use the basic terminology of computer hardware**

2.0b Convert a binary numeral to decimal*

2.0d Use the terminology of computer networking and the Internet*
Subtopic outcomes

2.1 Manipulate binary numerals*
2.2 Convert a circuit or formula to a truth table*
2.3 Describe the fetch-execute cycle*
2.4a Identify instructions of an assembler-language program
2.4b Test an assembler program and describe results†
2.4c Write and trace an assembler program†
2.4d Write and trace an assembler program with loop†

Architecture of a computer

- Data flows as shown by arrows
- Data is communicated and stored as *bits* (on/off pulses, or switches)
- *Logic gates* perform manipulation and storage operations
- The *processor* executes programs in *machine language*, loaded from memory (RAM)
- Input, output, and secondary storage are supported by *peripherals*
1. Binary representation of data

- What is digital about computers?
- How is data represented in a computer?
- What is a bit; a byte?
- What is a system of numerals, e.g., base 10?
- Why do we use base-10 and computers use base-2?

Brain vs. computer

<table>
<thead>
<tr>
<th></th>
<th>Brain</th>
<th>Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Representation</td>
<td>analog</td>
<td>digital</td>
</tr>
<tr>
<td>Storage</td>
<td>associative</td>
<td>addressable</td>
</tr>
<tr>
<td>Computation</td>
<td>stimulus-response</td>
<td>stored-program</td>
</tr>
<tr>
<td>Speed</td>
<td>200 mph</td>
<td>electronic (light)</td>
</tr>
<tr>
<td>Parallelism</td>
<td>high</td>
<td>minimal</td>
</tr>
<tr>
<td>Steps</td>
<td>large</td>
<td>small</td>
</tr>
</tbody>
</table>
Silicon vs. neurons

- The brain stores \textit{analog} information
- Neurons respond to the strength of electrical impulses by \textit{firing} (emitting an impulse)
- Computers process \textit{bits} (binary digits)
- Processors work in \textit{small} steps
- The brain is massively \textit{parallel}; digital computing is \textit{serial} today
- \textit{Emotions} currently play little role in computing systems; play a large role in the brain

The binary system

- Appropriate for two-state devices
- The form in which all information is represented is \textit{bits}
- Uses two digits \{0, 1\} rather than ten
- Like decimal, uses place values
- Examples $1_2 = 1$, $10_2 = 2$, $11_2 = 3$
- Operations
  - Conversion of binary to decimal
  - Decimal-to-binary conversion
  - Binary addition and subtraction
**Hardware representation of numbers**

- Values are stored as sequences of *bits*
- One bit can store either of two values: 0, 1
- Two bits can store 4 different values
- Four bits can store $2^4 = 16$ different values
- One *byte* (eight bits) can store $2^8 = 256$ different values, e.g., 0 .. 255
- On a 32-bit computer, a *register* can store up to $2^{32} \approx 4$ billion different values
- $k$ bits of storage can store $2^k$ different values

**Hexadecimal notation**

- 16 digits: ‘0’ to ‘9’ and ‘a’..’f’
- Four times as compact as binary
- Easy conversion with binary
- Each digit is 4 bits
- Memory addresses are normally expressed in hex
- Examples:
  
  $0a_{16} = 10_{10}$
  
  $12_{16} = 18_{10}$

*expand*
Binary-to-decimal conversion

- Scan from right to left
- Where a 1 is found in binary numeral, add its place value (a power of 2)
- Sum of these place values is the value of the binary numeral

\[ 11010_2 \]
\[ \begin{align*}
0 \times 2^0 &= 0 \\
1 \times 2^1 &= 2 \\
0 \times 2^2 &= 0 \\
1 \times 2^3 &= 8 \\
1 \times 2^4 &= 16 \\
\hline
&= 26_{10}
\end{align*} \]

Decimal-to-binary conversion

Two methods:
(a) Repeatedly divide decimal numeral by 2, jotting down remainder, right to left
(b) Find the set of powers of 2 that add up to the decimal value; record each as a binary 1, left to right

\[ 25_{10} \]
\[ \begin{align*}
25 \div 2 &= 12 \text{ R } 1 \\
12 \div 2 &= 6 \text{ R } 0 \\
6 \div 2 &= 3 \text{ R } 0 \\
3 \div 2 &= 1 \text{ R } 1 \\
1 \div 2 &= 0 \text{ R } 1 \\
\hline
&= 11001_2
\end{align*} \]
## Binary addition

- Do as in decimal, but use binary addition table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0₂</td>
</tr>
</tbody>
</table>

- Carry one bit:

```
  1100₂ + 10110₂ = 100110₂
  + 1100₂       + 10111₂ = 100110₂
  10110₂        + 1101₂
```

## Integer overflow

- If a computation produces a value that exceeds the capacity of its storage, the effect is *overflow*, producing an incorrect computed value.

- *Example* (4 bits):
  
  \[
  1000₂ + 1000₂ = 0000₂ (1 \text{ is discarded})
  \]

- To avoid overflow, use numeric data types of sufficient capacity.
Signed binary values

- Values that may be positive or negative are stored in *twos-complement* format
- The leftmost bit is reserved for the sign: 
  1 = negative, 0 = non-negative
- To convert to 2’s complement, flip bits, add 1
- Negative values are stored in such a way that binary addition yields a correct result

\[
\begin{align*}
0101_2 &= 5 \\
+1101_2 &= -3 \\
0010_2 &= 2
\end{align*}
\]

Binary fractions

- Note that \( \frac{1}{2} = 0.5 = 5 / 10^1 \)
  \( 1/4 = 0.25 = 25 / 10^2 \)
- Similarly, \( \frac{1}{2} = 0.1_2 = 1/2^1 \)
  \( 1/4 = 0.01_2 = 1/2^2 \)
- A binary fraction is a sum of negative powers of 2:
  \( \frac{3}{4} = 0.11 = 1 \times 2^{-1} + 1 \times 2^{-2} \)
Floating-point representation

- Concept is same as that used in exponential (scientific) notation; e.g., $3200 = 3.2 \times 10^3$
- Values are stored using a binary point
- For precision and flexibility, 3 fields are used: sign bit; normalized binary fraction; exponent
- The NBF results from moving the binary point just to the left of the leftmost 1 bit
- Multiplying NBF by $2^{exponent}$ compensates for shifting the binary point
- Significant bits may be lost

Subtopic outcome

2.1 Manipulate binary numerals*

* Significance and example usage
2. Logic gates and circuits

- How is data manipulated at the lowest level in digital computing?
- What are the building blocks of electronic hardware, and how are they combined?

Gates

- Used to manipulate binary data
- 1 or 2 bit input, 1-bit output
- Specified using truth tables
- NOT (negation)
- AND (conjunction)
- OR (disjunction)
- Used as components of combinational circuits: NAND, NOR, XOR, adders, etc.
Gates and truth tables

<table>
<thead>
<tr>
<th>Gate</th>
<th>Schematic</th>
<th>Truth table</th>
</tr>
</thead>
</table>
| NOT  | ![Diagram](NOT.png) | \[
\begin{array}{c|c|c}
a & \text{not } a \\
\hline
1 & 0 \\
0 & 1 \\
\end{array}
\] |
| OR   | ![Diagram](OR.png) | \[
\begin{array}{c|c|c|c|c}
a & b & a \text{ or } b \\
\hline
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\] |
| AND  | ![Diagram](AND.png) | \[
\begin{array}{c|c|c}
a & b & a \text{ and } b \\
\hline
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\] |

XOR and NOR circuits

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a xor 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>0</td>
</tr>
</tbody>
</table>

- **Problem**: Can you build a NOR gate from NOT, AND, and OR gates?
Circuits and logic formulas

- A circuit corresponds to a propositional-logic formula
- Examples:
  - \( a \text{ xor } b = (a \lor b) \land \neg (a \land b) \)
  - \( a \text{ nor } b = \neg(a \lor b) \)

Combinational circuits

- The XOR and NOR circuits are examples of hardware that combines gates to process data
- An \textit{adder} circuit performs addition on binary numeric data stored in registers of a processor
- A \textit{flip-flop} circuit stores a bit; as in RAM
Bitwise operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Java operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complement</td>
<td>~</td>
<td>~ 10000000₂ = 01111111₂</td>
</tr>
<tr>
<td>OR</td>
<td></td>
<td>1100₂</td>
</tr>
<tr>
<td>AND</td>
<td>&amp;</td>
<td>1100₂ &amp; 1001₂ = 1000₂</td>
</tr>
<tr>
<td>Left shift</td>
<td>&lt;&lt;</td>
<td>1101₂ &lt;&lt; 1 = 11010₂</td>
</tr>
<tr>
<td>Right shift</td>
<td>&gt;&gt;</td>
<td>11000₂ &gt;&gt; 2 = 110₂</td>
</tr>
<tr>
<td>XOR</td>
<td>^</td>
<td>1001₂ ^ 1010₂ = 0011₂</td>
</tr>
</tbody>
</table>

Subtopic outcome

2.2 Convert a circuit or formula to a truth table*
3. A model computer

- What are the main components of a computer?
- How does a program execute, at the hardware level?
- What are the programmed operations of a processor?
- What is a register?

A simplified model computer

- RAM (random-access memory) contains programs and data
The processor

- Initiates all actions
- Has two units:
  - Control: Determines order of operations
  - Arithmetic logic: Executes operations on data
- Communicates with memory
- Has three registers: program counter, instruction register, accumulator

Machine-language programs

- *Instructions* are represented in binary *operation codes*
- These may have *operands*, which specify *address* of data to be operated on
- All operations are simple
- Instructions and data re stored in consecutive RAM locations
The fetch-execute cycle

Program Counter
register gets address 0

\[
\begin{align*}
\text{PC} & \leftarrow 0 \\
\text{Repeat} & \\
\text{IR} & \leftarrow \text{MEM(PC)} \\
\text{PC} & \leftarrow \text{PC} + 1 \\
\text{Execute instruction in IR} & \\
\text{until instruction is STOP}
\end{align*}
\]

Instruction
Register gets contents of RAM referenced by value stored in PC

The language of the model processor

- Device I/O: input, output
- Copy data between ACC and RAM: load (copy from mem to ACC) store (copy from ACC to mem)
- Calculate from values in ACC and RAM, leaving result in ACC: add, sub
- Control execution: stop, jump, jump0, jump−
- Allocate and label data space: data, sdata
Types of storage

- **Register** (inside processor, high speed)
- **Cache memory** (inside processor, high speed)
- **RAM**: electronic “primary storage”, slower than within processor
- **Memory stick**: interface is like hard disk but speed is electronic
- **Hard disk**: slow, mechanical device
- **Compact disk (CD) and DVD**: slow, optical-mechanical devices

A program

```
// echo.asm; echoes input
input a
output a
stop
a data 0
```

Sample input/output:

[Input:] 26
[Output:] 26
Subtopic outcome

2.3 Describe the fetch-execute cycle*

4. Assembler-language programs

• How does the hardware
  – Give values to variables?
  – Manage memory?
  – Implement branches and loops?
• Have you programmed?
Machine and assembler languages

- Each processor design has its own *machine language* (in binary) and *assembler language* (*mnemonic* text abbreviations)
- Programmer may give names to addresses, using *labels*
- Machine and assembler languages are called *lower level* and are harder to read than HTML or JavaScript

The assembler

- An *assembler* program converts assembly-language code to machine language
Mnemonics, operands, and labels

- In an assembler program, a mnemonic is an easily remembered abbreviation for a machine-language operation.
- An operand specifies the address on which an instruction is to operate.
- A label is a name for an address.
- In the model-processor assembler language, column 1 is for labels, column 2 for mnemonics, column 3 for operand labels.

A program to add

```
// ADD.ASM: Displays sum of 2 inputs
input input1
input input2
load input1
add input2
store sum
output sum
stop

input1 data 0
input2 data 0
sum data 0
```

- `input` copies data from keyboard to RAM.
- `load` and `add` alter contents of ACC.
- `store` copies from ACC to memory.

```
Variables

- A *variable* is a named memory location that stores a value.
- Variables in programs must be *declared*:
  \[
  \text{sum data 0} \quad \text{(assembler)} \\
  \text{int sum;} \quad \text{(Java)}
  \]
- Variables are *assigned* values by *load-store*
  \[
  \text{load 2} \quad \text{(ACC} \leftarrow 2) \\
  \text{store sum} \quad \text{(sum} \leftarrow 2)
  \]

A program to find absolute value

```
// abs.asm
input n
load n
jump- negate
jump end
negate sub n
sub n
end
store abs
output abs
stop
n data 0
abs data 0
```

Mnemonic *jump-* triggers jump to operand address (label *negate*) if ACC holds negative value.
### Program to count from 10 to 1

```
loop    load count
jump-exit
output  count
sub     one
store   count
jump    loop

exit    stop
count   data 10
one     data 1
```

- Mnemonic `jump` triggers unconditional jump to operand address (label `loop`, here)

---

### Counting to add 1 + 2 + … + 5

```
count ← 0
sum ← 0

while count ≥ 0
  display count
decrement count
```

```
Loop count.asm
```

Counting to add 1 + 2 + … + 5
Assembler code for counted loop

- On each iteration, this program adds the counter to a variable, \textit{sum}.

- Exit condition holds when \((5 - \textit{count}) < 0\), i.e., \(\textit{count} > 5\).

- See loop instructions (red), assignments (blue, green).

\begin{verbatim}
repeat load five
sub count
jump- end
load sum
add count
store sum
load sum
add one
store count
jump repeat
\end{verbatim}

\begin{verbatim}
end output sum
stop
\end{verbatim}

\begin{verbatim}
count data 1
one data 1
five data 5
sum data 0
\end{verbatim}

[add1to5.asm]

Sentinel-driven loop

- Adds all inputs until input is 0.
- 0 is the sentinel here.
Subtopic outcomes

2.4a Identify **instructions** of an assembler-language program
2.4b Test an assembler program and describe results†
2.4c Write and trace an assembler program†
2.4d Write and trace an assembler program with loop†

References

D. Keil. The binary system of numerals.