CS_M33 Computer Systems: 
Processors, Memory and Data
http://www-compsci.swan.ac.uk/~csetzer/lectures/02/compsys/index.html

Dr. Anton Setzer
http://www-compsci.swan.ac.uk/~csetzer/index.html
Michaelmas Term 2002

Welcome

Topics of this Module

- **Basic building blocks** of the computer: logic gates, memory cells, wires.
- **Units** built from these (like register units, control unit).
- Other Hardware components (monitors, hard disks, ...)
- **Representation and manipulation of data.**
- **Machine instructions** and assembly languages.
- Translation of **high level languages** into machine instructions.
- **High performance architectures** like super scalar/pipelined architectures and instruction level parallelism.

Why this Module?

- Understanding of how hardware works, eg.
  - when administrating networks,
  - when working as system administrator.
  - What is a clock cycle?
  - When moving from 1 GHz to 2 GHz, what’s the increase in speed?
  - What is new about the Itanium, and what are the problems?
- Prerequisite for modules like
  - Compilers
  - Operating systems.
  - Networking.
Why this Module? (Cont.)

- High level languages often shield us from details of the machine.
  But not always:
  - Especially when working with C, C++.
  - Pointers, reference parameters.
  - Strange phenomena when using numbers:
    * overflow
    ⇒ numbers become suddenly negative.
    * 0.2 becomes 0.19999999.
    * signed vs. unsigned integers.
    * coding of bit sequences as numbers.

- Understanding of object oriented programming.
  - How does initialization of objects work.

- Understanding of byte code
  - Java virtual machine
  - Microsoft intermediate language (.NET).

- How to speed up programs.

Who am I?

- Lecturer.
- Nationality: German.
- Worked in
  - Swansea and Leeds.
  - Munich.
  - Sweden (Gothenburg, Stockholm, Uppsala).
  - Switzerland (Berne).
  - Hiroshima (Japan).
- Academic Education: Mathematical Logic.
- Worked in both Mathematics and Computer Science Departments.
- Main Research topics:
  - Proof theory (more mathematics).
  - Martin-Löf’s Type Theory (more computer science).
  - Prototype of future programming languages.
  - Recent interest: Foundations of object-oriented languages

Address:
Dr. Anton Setzer
Dept. of Computer Science
University of Wales Swansea
Singleton Park
SA2 8PP
UK

Room  Room 211, Faraday Building
Tel.    (01792) 513368
Fax.    (01792) 295651
Email  a.g.setzer@swansea.ac.uk

Administrative Information

- Assessment:
  - 80% written examination in January.
  - 20% coursework:
    * 2 assignments. Each counts 10%.
    * One probably due at beginning of November.
    * One probably due at beginning of December.
- Two lectures per week.
  - Tuesday, 9:00, Vivian Tower, Room 114.
  - Thursday, 9:00, Glyndwr Building, Room E.
- Contact me if you have questions.
  - Preferably in the afternoon.
  - My usual time table is available from my homepage.
- Ask questions in lectures.
Exam

- You should aim to understand everything which is taught in this module.

- However, not all details will be relevant for the exam.

- In the revision lecture, the amount of material to be learned will be cut down.
  - No need to learn large lists of details.

- Some material to be understood will as well not be relevant for the exam.
  - This will be indicated in the lecture.

Course Material

- Web page contains overhead slides from the lectures. Course material will be continually updated.

- Course material always available from student office in the department of computer science, room 206.
  - From trays on the left side of the entrance.
  - If there is none left, ask the secretary to make additional copies.
  - If the secretary does not have master copies, ask me, and I will make new ones.

- Course material always available from student office in the department of computer science, room 206.

- Please don’t print out lecture notes on departmental printers – ask secretaries for copies. Help to save the tax payers money!!
  - Cost for printing is 10 times the cost of photo copying.

Web Page

- The web page of the module is located at
  http://www-compsci.swan.ac.uk/~csetzer/langhardware/02/index.html

- For copyright reasons, some of the material is password protected. The password is ____________________

- The same password applies to related course material.

- The password protection mechanism is very simple, therefore:
  Please put a link to the password protected page on any world-wide accessible page.

Copyright Notice

Most pictures in these slides are copyrighted. They may not be used elsewhere, see the respective books.

All figures from Computer Organization and Design: The Hardware/Software Approach, Second Edition, by David Patterson and John Hennessy, are copyrighted material (COPYRIGHT 1998 MORGAN KAUFMANN PUBLISHERS, INC. ALL RIGHTS RESERVED). Figures may be reproduced only for classroom or personal educational use in conjunction with the book and only when the above copyright line is included. They may not be otherwise reproduced, distributed, or incorporated into other works without the prior written consent of the publisher.
**Recommended Literature**

- Module is self-contained: No need to buy/borrow any book – but here are some recommendations:


**Other Good Text Books**


**Books I haven't studied yet**


  - Both books seem to be very good.

**Related Topics**

- **R. H. Katz:** *Contemporary logic design.* Benjamin/Cummings Publishing Company, 1994. Book on how circuits are actually designed on gate level.

Overview

2. Boolean Logic, Combinatorial Circuits.
3. Sequential Circuits.
6. Internal Memory.
7. External Memory.
8. CPU-Instructions Sets, Addressing Modes.
10. CPU Structure.
11. High-Performance Computer Architectures.

Levels of Abstraction

1. Application programs: word processors, databases etc.
2. High-level languages: Java, Pascal, Delphi etc.
4. Abstract hardware (digital logic).
5. Low level hardware (electronics).

In this module:
- Machine languages and abstract hardware.
- A first glimpse at compilation from high level languages.

1. Historic Development of Computers and Basic Structure

(a) Mechanical Era.
(b) The First Generation.
(c) Later Developments.
(d) Increase in Performance.
(e) Basic Structure.

(a) The Mechanical Era

- Old tools: Abacus.

Picture copied from book not included
The Mechanical Era (Cont.)

- Mechanical Calculators: Schickhardt (1623), Pascal (1642), Leibnitz (1673).

Punch Cards

- Invented 1801 by Joseph-Marie Jacquard (France)
- For control of patterns of an automatic loom.

Charles Babbage (1791 - 1871)

  - Memory.
  - Arithmetic and logic unit.
  - Punch cards, printer.
  - Instructions contain the following:
    - Arithmetic and logic operations.
    - Load/Store.
    - Unconditional and conditional jumps.
Charles Babbage (Cont.)

- Addition 1 second, multiplication 1 minute.
- Construction not carried out before 1991.

George Boole: Development of Boolean Logic. (1854).

Herman Hollerith
- Construction of a machine based on punch cards for evaluation of the 1890 census in US.
- Founder of the predecessor of IBM

Electromechanical computers
- J. A. Atanasoff (1936-1939) Special purpose computer for solving sets of linear equations.
- Mark I (1944). First general-purpose electromechanical computer.
- Turing (1936): Theoretical model of a computer.

(b) The First Generation: Vacuum Tubes

ENIAC (completed 1946)
- 18 000 vacuum tubes.
- 5000 additions per second.
- Programmed through rewiring of connections between components.
- Decimal digits represented by 10 vacuum tubes.
- Originally developed for calculating ballistic equations for artilleries during world war II.
The von Neumann Machine (1946)

Main Memory

I/O Equipment

Arithmetic Logic Unit (ALU)

Control Unit

- Theoretical model of a computer.
- Basis for most computers developed up to now.
- However, in modern computers, the control unit can communicate directly with I/O Equipment.

Main Principles of the von Neumann Machine

- Stored program-concept. Program in the memory.
- Main components: Memory, ALU, Control unit, I/O
- Both programs and data stored in the same memory.
  - Referred to as the Princeton/von Neumann Architecture
  - Opposed to Harvard Architecture: Memory and connections both separated for instructions and data.
- Sequential execution.

Problems with the von Neumann Machine

- Von Neumann bottleneck: Time for data transfer between main memory and CPU is main hindrance for improving speed.
- Semantic gap between high level programming languages and implementation on a von Neumann machine.

(c) Later Developments

- Generation 1 (1945 - 1957)
  - Vacuum tubes, Magnetic drums
  - Machine code, stored programs.
  - First Computer families (compatibility) UNIVAC I, UNIVAC II.
    - Performance: 2 KB memory, 10 KIPS (Kilo instructions per second).
- Generation 2 (1958 - 1964)
  - Transistors.
  - High level languages.
  - Floating-point arithmetic.
  - Performance: 32 KB memory, 200 KIPS.
- Generation 3 (1964 - 1971)
  - Integrated circuits.
  - Semiconductor main memory.
  - Micro-programming
  - Multi-programming
  - Structured programming
  - 1 MB memory, MIPS (Mega instructions per second).
• **Generation 4 (1972 - 1977)**
  - Large scale integration
  - Networks
  - CD
  - Object-oriented languages.
  - Expert systems.
  - 8 MB memory, 10 MIPS.

• **Generation 5 (1978 - present)**
  - Very large scale integration. (VLSI)
  - 128 - 512 MB memory, 100 MIPS - 500 MIPS.
  - World wide web.

(d) **Increase in Performance**

Picture copied from book not included
(Source: Stallings).
Note that the scale is exponential.

**Gap between Processor Speed and Memory**

![Gap between Processor Speed and Memory](Source: Stallings).
Note that the scale is exponential.

**Performance Increase of Workstations**

Scale: number of times faster than the VAX-11/780
Rate of performance: doubling every 1.6 years.
(Source: Patterson and Hennessy)
Picture copied from book not included
Moore's Law

Moore's law (1965):
- Number of transistors per Chip doubles every 18 months.
- Similarly we can observe an exponential growth of
  - density of memory,
  - speed of processors,
  - memory size.

Components of a Desktop Computer

- Box, keyboard, mouse, monitor, peripherals
- Box contains
  - **Motherboard**
    - Memory and processor – directly or via slots.
    - More slots for
      - Extra memory (various types)
      - Expansion cards:
        - Graphics cards
        - Peripheral interfaces.
        - Networking.
      - PCI - Peripheral Component Interconnect.
      - AGP Accelerated Graphics Port.
      - ISA Industrial Standard Architecture bus.
- Box contains also
  - Power supply, fan, drives (hard disk, CD, floppy)
- Main focus in this module:
  - Memory and processor.
**A Motherboard**

- 1 = Ports / 2 = ISA slots
- 3 = PCI slots
- 4 = AGP slots
- 5 = CPU slots
- 6 = Chipset (Northbridge)
- 7 = Power connector
- 8 = Memory sockets
- 9 = I/O connectors
- 10 = Battery
- 11 = Chipset (Southbridge)
- 12 = BIOS chip

*Picture copied from book not included*

---

**Embedded Systems**

- Most processors not in desktops.
  - They are in *embedded systems*, eg. cars, mobiles, planes, domestic appliances, medical equipment, industrial plants.
  - Even *smart cards*.
  - Relatively low-powered.
- As well *main frames*.
- In this module main focus on desktop computers.

---

**2. Boolean Logic and Combinatorial Circuits**

(a) Combinatorial Circuits.

(b) Boolean Logic.
Main principle of a computer:
Information represented by voltage on/off or positive/negative.
- **voltage on** or **voltage positive** means 1 or true,
- **voltage off** or **voltage negative** means 0 or false.
(The physical representation can be interchanged).

The following parts are used for internal manipulation of information:
- Voltage + ground.
- Clock.
- Capacitor. Temporary storage. (Needs regular refresh and refresh after information taken).
- Logic Gates.
- Wires.

**Logic Gates**
Built using transistors and diodes.
Transistor = electronic version of a relay.
A relay is a electromagnetic switch:

- **Negating Relay**
  p-channel transistors operate like this.

- **Positive Relay**
  n-channel transistors operate like this.

- Often written for \( \neg x \).
- Don’t forget the circle in the symbol for NOT!!
  Without a circle, one obtains a buffer (output = input).
**Relationship to Natural Language “And”**

- “And” corresponds to natural language “and”: Assume the sentence
  - “we are in Swansea and it is raining”.
  This sentence is true, if both
  - “we are in Swansea” and
  - “it is raining”
  get truth values “true”. If
  - the truth value of “we are in Swansea” is x, and
  - the truth value of “it is raining” is y,
  and one looks at the truth table of ·, one sees that the truth value of
  - “we are in Swansea and it is raining”,
  is \(x \cdot y\). Therefore \(\cdot\) formalizes the way truth values are assigned to sentences linked by “and”.

**Relationship to Natural Language “Or”**

- “Or” corresponds to natural language “or”: Assume the sentence
  - “we are in Swansea or it is raining”.
  This sentence is true, if at least one of
  - “we are in Swansea” and
  - “it is raining”
  get truth value “true” (one often mixes this up with the either . . . or. If both are true, then the truth value of the result is true, which would be false if we applied either or).
  If
  - the truth value of “we are in Swansea” is x, and
  - the truth value of “it is raining” is y,
  and one looks at the truth table of +, one sees that the truth value of
  - “we are in Swansea and it is raining”,
  is \(x + y\). Therefore \(+\) formalizes the way truth values are assigned to sentences linked by “or”.

Often \(xy\) or \(x \cdot y\) written for \(x \land y\).

Often \(x + y\) written for \(x \lor y\).
NOR Gate

\[ \text{NOR} = \neg \text{OR} \]

\[
\begin{array}{c|c|c}
  x & y & x + y \\
  \hline
  0 & 0 & 1 \\
  0 & 1 & 0 \\
  1 & 0 & 0 \\
  1 & 1 & 0 \\
\end{array}
\]

**Exercise:** Verify that this is the same as \( \neg (x + y) \):

\[
\begin{array}{c|c}
  x & y \\
  \hline
  0 & 0 \\
  0 & 1 \\
  1 & 0 \\
  1 & 1 \\
\end{array}
\]

---

NAND Gate

\[ \text{NAND} = \neg \text{AND} \]

\[
\begin{array}{c|c|c}
  x & y & \neg (x \cdot y) \\
  \hline
  0 & 0 & 1 \\
  0 & 1 & 0 \\
  1 & 0 & 1 \\
  1 & 1 & 0 \\
\end{array}
\]

**Exercise:** Verify that this is the same as \( \neg (x \cdot \neg y) \):

---

Abbreviations

- \( \bigcirc \) means always negation.

- Gates with more than two inputs:
  \[
  x \quad y \quad z \\
  \hline
  \bigcirc\bigcirc\bigcirc \\
  \bigcirc\bigcirc\bigcirc
  \]
  
  \( x(yz) = (xy)z =: xyz \)

  - Similarly for \( + \).

---

Combination of Gates

With a combination of gates, more complicated functions can be represented:

\[
\begin{array}{c}
  \quad x \\
  \quad y \\
  \quad \bigcirc \\
  \quad z \\
  \quad = \\
  \quad \bigcirc \\
  \quad u \\
  \quad v \\
  \quad w \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
  x & y & z & u & v & w \\
  \hline
  0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 1 & 0 & 1 & 0 & 0 \\
  0 & 1 & 1 & 0 & 1 & 1 \\
  1 & 0 & 0 & 0 & 0 & 0 \\
  1 & 0 & 1 & 0 & 1 & 0 \\
  1 & 1 & 0 & 1 & 1 & 1 \\
  1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]
Exercise

- Simplify the last circuit (and further circuits presented in this section).

(b) Boolean Logic

- **Boolean value** = value true or false.

- Often (and in this module) written as 1 (= true) and 0 (= false).

- **Boolean connectives**
  - not (\(\neg\)), and (\(\wedge\)), or (\(\vee\)).

- **Boolean terms** = expressions formed from
  - variables,
  - 0, 1,
  - Boolean connectives \(\neg\), \(\wedge\), \(\vee\).

- We write
  - \(\neg x\) for \(\neg x\) (except in pictures),
  - \(\cdot\) for \(\wedge\),
  - \(+\) for \(\vee\),
  - even \(xy\) for \(x \cdot y\).

- **Examples of Boolean terms:**
  \((xy) + 1, x + y, \neg (xy)\).

Boolean Terms

- **Order of precedence**
  - As in arithmetic: \(\cdot (\wedge)\) binds more than \(+ (\vee)\).
    
  E.g. \(xy + z = (xy) + z\).
  - \(\neg\) binds more than other connectives.
    
  E.g. \(\neg x + y \cdot z = (\neg x) + (y \cdot z)\).
  - Since \((x + y) + z\) and \(x + (y + z)\) have the same result for choices of \(x, y, z\), similarly for \((xy)z\) and \(x(yz)\),
  we can omit parenthesis in products and sums.

- **Defined connectives**
  - \(x\) NAND \(y := \neg(xy)\) \(= \neg(x \wedge y)\).
  - \(x\) NOR \(y := \neg(x \vee y)\).
  - \(x\) XOR \(y := \neg(xy) + xy\).
  - XOR = exclusive or: either \(x\) or \(y\)
    
  \((x \lor y)\) but not both.

Representation of Circuits by Boolean Terms

- A Boolean term corresponds to a circuit with inputs
  for each variable in it.

- Example: \((xy + \neg(xy)) + z\) represents the following circuit:
Optimization of Circuits

- If the same term occurs twice in a Boolean term, we can construct a circuit which uses this subterm directly twice.

**Example:**
In $xy(xy + z)$, $xy$ occurs twice. $xy(xy + z)$ can be represented by the following circuit:

![Circuit Diagram]

The circuit on the last slide was combinatorial.

A circuit which has a loop is non-combinatorial.

The following circuit is not combinatorial (try to represent it by a Boolean term):

![Circuit Diagram]

$y = x + y$, a recursive equation.

Boolean-Valued Functions

A Boolean valued function is given by a truth table:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>$f(x,y,z)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</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>0</td>
<td>0</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>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Boolean-Valued Functions (Cont.)

A Boolean term in variables $x$, $y$, $z$ represents a function depending on $x$, $y$, $z$. E.g. $xy + z$ represents the function with truth table (an additional column for the intermediate term $xy$ has been inserted):

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>xy</th>
<th>$f(x,y,z)=xy+z$</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>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Computer Systems, CS_M33, Michaelmas term 2002, Sect. 2
Representation of Boolean-Valued Functions

Any Boolean valued function can be represented by a Boolean term (and therefore by a combinatorial circuit).

Consider the following example:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>f(x,y,z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</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>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Representation of Boolean-Valued Functions (Cont.)

Now take all rows, where \( f(x, y, z) \) has value 1.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>f(x,y,z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</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>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Now take the conjunction (= the “and”) of all these terms:

\[
\begin{align*}
\text{introduced term} &= \overline{x} \cdot \overline{y} \cdot \overline{z} + x \cdot \overline{y} \cdot \overline{z} + x \cdot y \cdot \overline{z} \\
\end{align*}
\]

The resulting term is \( \overline{x} \cdot \overline{y} \cdot \overline{z} + \overline{x} \cdot y \cdot \overline{z} + x \cdot y \cdot z \)
Why is this Correct?

The constructed intermediate terms are 1 if and only if the truth values of the variables are exactly those in the row, from which the term was derived:

For instance for the row with entries

\[ x = 0, \ y = 1, \ z = 0, \]

the corresponding intermediate term is

\[ \overline{x}y\overline{z} \]

This term is 1 if and only if

\[ \overline{x} = 1 \text{ and } y = 1 \text{ and } \overline{z} = 1 \]

which is if and only if

\[ x = 0 \text{ and } y = 1 \text{ and } z = 0 \]

ie. if \( x, y, z \) have the values for this row.

Disjunctive Normal Form

In logic, the resulting term is called the disjunctive normal form for this Boolean function. This is because the result is a

- disjunction (disjunction means “an or”)  
- of a conjunction (conjunction means “an and”)  
- of terms \( x \) or \( \overline{x} \); \( y \) or \( \overline{y} \); \( z \) or \( \overline{z} \).

Example

The above mentioned function \( f \) is represented by

\[ \overline{x} \cdot \overline{y} \cdot \overline{z} + \overline{x}y\overline{z} + xyz \]

which corresponds to the circuit: 

- \( x \), \( y \), \( z \) as inputs
- \( \overline{x}y(\overline{z}) \)
- \( (\overline{x})(\overline{y})(\overline{z}) \)
- \( f(x,y,z) \)
- \( xyz \) as output
Trick for Constructing Circuits for Boolean Functions

- Trick with constructing circuits or Boolean formulas for Boolean functions:
  - Take inputs x,y,z, ....
    - For every "1" in the result, take one AND gate and connect it with each of x,y,z, ....
    - If x is 0 in the corresponding row, put a circle (a "0" at the gate), otherwise leave the line as it is (a horizontal "1").
    - If x is 0 in the corresponding row, put a circle (a "0" at the gate), otherwise leave the line as it is (a horizontal "1").
    - etc.
  - OR everything.
  - A Boolean formula can now be read off.

Example

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</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>1</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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[(x \cdot \overline{y} \cdot \overline{z}) + (\overline{x} \cdot \overline{y} \cdot z)\].

Exercise

- Take any other Boolean valued functions with 3 arguments by choosing arbitrary values for \(f(x, y, z)\).
  Compute its disjunctive normal form (ie. a Boolean term representing it as defined above).
  - (Typical exam question).

Logic Array

- **Logic array** with inputs \(x, y, z\) is a circuit
  - which represents all possible intermediate terms for arbitrary functions
  - i.e. all products of \(x\) vs. \(\overline{x}\), \(y\) vs. \(\overline{y}\), \(z\) vs. \(\overline{z}\).
  - all connected with an orgate to the output
  - but with possibility to interrupt connections from the and-gates to the orgate.
  - By interrupting appropriate connections all possible functions can be represented.
### Equality between Boolean Terms

- **We say** two Boolean terms are equal, if they for all choices of truth values for variables occurring in them both have the same value.

- **In order to verify** that two terms are equal, build a truth table (depending on all variables occurring in one of the terms) for both terms, and show that their results are the same.

#### Example 1: Verification of \( x + y = y + x \):  

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( x + y )</th>
<th>( y + x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Example 2: Verification of \( x \cdot \bar{x} = 0 \):

<table>
<thead>
<tr>
<th>( x )</th>
<th>( x\bar{x} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Laws of a Boolean Algebra

- **Simplification of Boolean terms** (and their corresponding circuits) by using equations between Boolean terms.

- \( = \) means: for all choices of \( x, y, z \ldots \) the value of the term is the same.

- **The following laws are due to Boole:**
  - **Commutativity:**  
    \( x + y = y + x \),  
    \( xy = yx \).
  - **Associativity:**  
    \( x + (y + z) = (x + y) + z \),  
    \( x(yz) = (xy)z \).
  - **Distributivity:**  
    \( x(y + z) = (xy) + (xz) \),  
    \( x + (yz) = (x + y)(x + z) \).
  - **1 is neutral for +, 0 neutral for •:**  
    \( x1 = x \),  
    \( x + 0 = x \).
  - **\( \neg \) is inverse for • and +:**  
    \( x + \neg x = 1 \),  
    \( x\neg x = 0 \).
Laws of a Boolean Algebra (Cont.)

- **Remark:** The following laws can be derived from the above ones:
  \[ x + x = x \quad xx = x \quad (\text{idempodence}) \]
  \[ x + 1 = 1 \quad x \cdot 0 = 0 \]
  \[ \overline{\overline{x}} = x \]
  and the de Morgan laws:
  \[ \overline{x + y} = \overline{x} \cdot \overline{y} \quad \overline{x \cdot y} = \overline{x} + \overline{y} \]

- Based on the laws of a Boolean algebra there are good techniques for simplifying circuits (see e.g. Stallings).

Exercise (Cont.)

- For deriving \( x + y = \overline{x} \cdot \overline{y} \),
  derive first
  \[ \overline{x} \cdot \overline{y}(x + y) = 0 \]
  and
  \[ \overline{x} \overline{y} + (x + y) = 1 \]
  Now start with
  \[ \overline{x} + y = \overline{x} + y \cdot (x + y + \overline{x} \cdot \overline{y}) . \]

3. Sequential Circuits

(a) Sequential Circuits.

(b) Latches and Flip-Flops.
Example:

\[
\begin{align*}
x & \rightarrow u \\
y & \leftarrow u \\
z & \\
\end{align*}
\]

Represented program:

\[
\begin{align*}
u & := xy \\
v & := u + z \\
w & := uv \\
\end{align*}
\]

Combinatorial circuits allow to define:
- arbitrary functions,
- therefore complex programs,
- but **no loops**.

### Behaviour for x=1:

- \( y \) is constant 1.
- \( z \) is constant 0.

⇒ **Not intended behaviour.**

### Representation of Loops

Consider the following (useless) toy program:

\[
\text{loop : } x := \neg x \\
\text{jmp loop}
\]

**First Unsuccessful Attempt**

\[
\begin{align*}
x & \rightarrow y \\
z & \rightarrow z \\
\end{align*}
\]

Unclear, what is meant by the connection of \( x, y, z \).
Therefore not a correct way of drawing circuits.
Correct diagram: Use an orgate.

**Orgate = Union of Flows**

- If a bit arrives in one of the two connecting rivers, one gets a bit at the output.
- However, if one bit arrives in both rivers, we only get one bit at the output.
1. Insertion of switch $S$.

2. Separation into program steps: cycles. However: output of the not-gate $z$ should be fed into $S$ at the next cycle.

$S$ outputs $x$ at cycle 1, $u$ at later cycles.

3. Insertion of memory element $M$.

$S$ outputs $x$ at cycle 1, $u$ at later cycles.

$M$ stores input and outputs it in the next cycle.
**Behaviour for \( x = 1 \):**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>u</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>→</td>
<td>1</td>
<td>→</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- \( M \) needs to store one bit and make use of it in the next cycle.
- \( S \) needs to store as well one bit, containing the information, whether it is in the first or a later clock cycle.
- We need a circuit which can store bits.
- Done using latches and flip flops.

**An S-R Latch** is the following circuit:

\[ Q = R + Q' \]
\[ Q' = S + Q \]

- Non-combinatorial because of feed-back loop.
- Input: \( R, S \) (kept fixed).
  Output: \( Q, Q' \).
- Full analysis hard (mathematical technique used: differential equations).
Calculation of Movements of Two Fighters

- In reality, two fighters react continuously over time depending on their own movement and that of the other.

- We can approximate this as follows:
  - Divide time into discrete steps.
  - Steps of time are now numbered 0,1,2,3, . . .
  - Now assume, each fighter makes at time $k+1$ one movement depending on
    * his posture (state)
    * and that of the other fighter
    * both at time $k$.

Calculation of the Behaviour of an S-R-Latch

- Similarly as for the fighters, when analyzing an S-R-latch, we divide time into discrete steps.

- Steps correspond to the gate delay.
  - Time it takes for the gate to change its output depending on a new input.
  - We can assume that this delay is the same for all gates (almost true).

- So we assume that at time $k+1$, the output of a gate is calculated depending on its inputs at time $k$.

\[ R \quad \rightarrow \quad \overline{Q} \]
\[ S \quad \rightarrow \quad Q' \]

**Notation:**
- Let $Q(k)$ be the value of $Q$ in the $k$th step.
- Let $Q'(k)$ be the value of $Q'$ in the $k$th step.

- Assume $R$, $S$ constantly for some time.

- Then
  - $Q(k+1) = R \text{ NOR } Q'(k) = \overline{R + Q'(k)}$.
  - $Q'(k+1) = S \text{ NOR } Q(k) = \overline{S + Q(k)}$.

The above equations don’t define $Q(0)$ and $Q'(0)$.

- When the circuit is switched on, $Q(0)$ and $Q'(0)$ are unknown.
  - By applying reset to it ($R = 0$, $S = 1$ as discussed below) one obtains a defined state.

- At later stages, $Q$ and $Q'$ will initially have the value obtained by the previous input.
Case $R = 0, S = 1$ (Set)

- Assume $R = 0, S = 1$ constant for some time.

- Initially $Q(0)$, $Q'(0)$ are assumed to be unknown (indicated by ?).

- Then
  \[
  Q(k+1) = R + Q'(k) = 0 + Q'(k) = Q'(k).
  \]

  \[
  Q'(k+1) = S + Q(k) = 1 + Q(k) = I = 0.
  \]

So:

- $Q'(1) = 0$.

- $Q'(2) = 0, Q(2) = \overline{Q(1)} = \overline{0} = 1$.

- $Q'(3) = 0, Q(3) = \overline{Q(2)} = \overline{1} = 1$.

The following table shows the values of $Q, Q', R, S$ after 0, 1, 2, 3 gate delays:

<table>
<thead>
<tr>
<th>$k$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$S$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$Q(k)$</td>
<td>?</td>
<td>?</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$Q'(k)$</td>
<td>?</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

From step 3 onwards, $Q(k), Q'(k)$ remain constant.

- So whatever $Q, Q'$ initially are, $Q$ becomes 0, then $Q'$ becomes 1.

Case $R = 0, S = 1$ (Cont.)

Case $R = 1, S = 0$ (Reset)

- $Q(k+1) = R + Q'(k) = 1 + Q'(k) = I = 0$.

- $Q'(k+1) = S + Q(k) = 0 + Q(k) = Q(k)$.

We get the table:

<table>
<thead>
<tr>
<th>$k$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$S$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$Q(k)$</td>
<td>?</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$Q'(k)$</td>
<td>?</td>
<td>1</td>
<td>1</td>
<td>constant</td>
</tr>
</tbody>
</table>

Whatever $Q$, $Q'$ initially are, $Q$ becomes 0, then $Q'$ becomes 1.

Case $R = 0, S = 0$, Stable Case

- Assume initially
  - $Q(0) = 0, Q'(0) = 1$
  - or $Q(0) = 1, Q'(0) = 0$.

- Then
  \[
  Q'(0) = \overline{Q(0)},
  Q(k+1) = R + Q'(k) = 0 + Q'(k) = Q'(k).
  \]

  \[
  Q'(k+1) = S + Q(k) = 0 + Q(k) = Q(k).
  \]

So:

- $Q(1) = \overline{Q(0)} = \overline{Q(0)} = Q(0)$,
  $Q'(1) = Q(0)$.

- $Q(2) = \overline{Q(1)} = \overline{Q(0)} = Q(0)$,
  $Q'(2) = \overline{Q(1)} = \overline{Q(0)}$.

We get the table:

<table>
<thead>
<tr>
<th>$k$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$S$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$Q(k)$</td>
<td>$Q(0)$</td>
<td>$Q(0)$</td>
</tr>
<tr>
<td>$Q'(k)$</td>
<td>$Q(0)$</td>
<td>$Q(0)$</td>
</tr>
</tbody>
</table>
Case $R = 0$, $S = 0$, Stable Case (Cont.)

- $Q$, $Q'$ keep (store) their initial value.

$R = 0$, $S = 0$, Unstable Case (Cont.)

- Assume initially $Q(0) = 0$, $Q'(0) = 0$.

- Again
  
  \[
  Q(k + 1) = Q'(k), \quad Q'(k + 1) = Q(k). 
  \]

- $Q(1) = Q'(0) = 0 = 1$,
- $Q'(1) = Q(0) = 0 = 1$.

- $Q(2) = Q'(1) = 1 = 0$,
- $Q'(2) = Q(1) = 1 = 0$.

- We get the table:

<table>
<thead>
<tr>
<th>$k$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$S$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$Q(k)$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$Q'(k)$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- The latch oscillates (toggles) between 1 and 0.
- Same behaviour if initially $Q(0) = 1$, $Q'(0) = 1$.
- In reality, the delay of the two gates is not exactly the same.
  - Therefore the gates are not exactly synchronized.
  - After some toggling steps, it will be the case that one is 0 while the other is 1.
  - Then the latch stabilizes.
  - But in which state, is unpredictable.

<table>
<thead>
<tr>
<th>Time</th>
<th>$Q$</th>
<th>$Q'$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- If we then release $R, S$, ie. set $R = S = 0$, we obtain the **unstable case**.
S-R-Latch as a Memory Element

- If the output is $Q = 0$, $Q' = 1$ or $Q = 1$, $Q' = 0$, the latch is in a **stable** state.

- A stable state can be achieved by initially setting $R = 1$, $S = 0$, long enough.

- $Q = 1$, $Q' = 0$ means **memory value 1**.
- $Q = 0$, $Q' = 1$ means **memory value 0**.

- If $R = S = 0$ in a stable state, the memory value is **preserved**.

- If we set $R = 0$, $S = 1$ in any state, we obtain **memory value 1**.
  ($S = \text{Set}$).

- If we set $R = 1$, $S = 0$ in any state, we obtain **memory value 0**.
  ($R = \text{Reset}$).

Unstable State of an S-R-Latch

- If the output is
  - $Q = 0$, $Q' = 0$ or
  - $Q = 1$, $Q' = 1$,
  the latch is in a potentially **unstable** state.

- If $R = S = 0$ in this state, the latch becomes unstable and oscillates (toggles).
  This should be avoided.

- If we set $R = S = 1$ in any state, we obtain such a potentially unstable state.

**Exercise**

Analyze the following circuit in a similar way:

- **D-Flip-Flop** consists of a combinatorial circuit and an S-R-latch.

- If the clock is 1 then:
  - If $D = 1$, then $R = 0$, $S = 1$, and therefore in the S-R latch memory value 1 is stored.
  - If $D = 0$, then $R = 1$, $S = 0$, and therefore in the S-R latch memory value 0 is stored.

- If clock is 0, then independently of $D$ we have $R = 0$, $S = 0$, and the S-R-latch keeps therefore its previous value.
Clock Cycles

- The clock signal is a signal, which periodically alternates between 0 and 1. Generated by an external clock.

- The clock signal will be slower than the time it takes for S-R-latches to stabilize (i.e. for Q, Q’ to become constant), if we are not in an unstable situation.
  - For practical purposes one tries to make the clock as fast as possible.
  - As an approximation, we assume that we can neglect the time it takes for the S-R-latches to stabilize.

By one clock cycle we mean the period
- starting when the clock signal becomes 1
- ending when the clock signal is 0 and is just about to switch back to 1 again.

By half a clock cycle we mean the period when the clock cycle is constant either 1 or 0.

Clock Cycles (Cont.)

- By “at clock cycle 1 signal R is 0” we mean:
  - when the clock is the first time 1, R should be 0.
  - This remains constant as long as the clock is 1.
  - What happens when the clock is 0 doesn’t matter.

- By “at second half clock cycle S is 0” we mean that as long as this half clock cycle persists, S is 0.

Clock

The following signal is 1 at clock cycle 1:

The following signal is as well 1 at clock cycle 1:

The following signal is 1 at the 2nd half clock cycle:

Behaviour of a D-Flip-Flop

- Assume clock alternates between 1 and 0.
- Assume further that clock signal lasts much longer than the time needed for the S-R-latch to stabilize.
- The following table illustrates the value of Q, Q’ in response to a (random) sequence of input values for D:

<table>
<thead>
<tr>
<th>Clock</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Q</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Q’</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- So Q adapts to the value of D when the clock is 1, and stores this value, independent of D, when the clock is 0.
- Q’ is the negation of Q.
- Note that Q, Q’ react on D with a short (here neglected) delay.
The following symbol denotes a D-Flip-Flop, with input Ck (for clock) and D and output Q and Q'.

Symbol:

```
<table>
<thead>
<tr>
<th>D</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ck</td>
<td>Q'</td>
</tr>
</tbody>
</table>
```

- If Enable = 1 and Clock = 1, then Q is updated to the value of D.
- If Enable = 0 the value of Q is kept.
- If Clock = 0, the value of Q is kept as well.

**A Storage Cell**

- **Behaviour**, for alternating clock signal:
  - If Clock = 1, then D' is updated to D.
  - If Clock then switches to 0, then
    * Value of D' remains the same.
    * Further Q is updated to D', which is the value of D when clock was 1.
  - If Clock then switches back to 1 again, then
    * Value of Q remains the same.
  - Therefore the output at Q, when the clock is 1, is the input of D when clock was previously 1.
    * which means at the previous clock cycle.

<table>
<thead>
<tr>
<th>Clock</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D'</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Q</td>
<td>?</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- The falling edge D-flip-flop is a buffer, which delays its output by one clock cycle.
  - The output arrives already after half a clock cycle, but is not stable from the beginning.
  - When the clock is 1 again, the output is stable and repeats the input from the previous clock cycle.

The following table illustrates the behaviour of a falling edge D-flip-flop in response to a (random) sequence of inputs on D:
Moving Bits Around

- If Signal is 1, the value saved in the first flip-flop is stored in the second flip-flop and can be read off at Q in the next clock cycle.
- If Signal is 0, the second flip-flop keeps its value.
- Output at $D'$ could be fed into several flip-flops, each with an individual control signal.

Output of Last Circuit

- $D'$ follows D when clock = 1 and keeps this value if clock = 0.
- Q updates to $D'$ when clock = 0, provided Signal = 1.
  If clock = 1 or Signal = 0, it keeps the previous value.

$$
\begin{align*}
\text{Clock} & : 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \\
\text{D} & : 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \\
\text{Signal} & : 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \\
\text{D'} & : 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \\
\text{Q} & : 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 
\end{align*}
$$

Multiplexer

- A multiplexer is a combinatorial circuit corresponding to the Boolean term
  $$C = AS + BS$$

  - If $S = 1$, then
    $$C = AT + B \cdot 1 = A \cdot 0 + B \cdot 1 = B.$$
  - If $S = 0$ then
    $$C = A\overline{S} + B \cdot 0 = A \cdot 1 + B \cdot 0 = A.$$

  So:
  - if $S = 0$, the output is $A$,
  - if $S = 1$, the output is $B$.

Multiplexer (Cont.)

- $S$ controls, which of the two inputs becomes the output.
- Can be extended:
  - with 2 Signals, decisions between 4 incoming signals can be made,
  - with 3 between 8 signals, etc.
- Exercise:
  - Write down the Boolean formula for a 4-to-1 multiplexer (i.e. one with 4 inputs + two control signals, which decide, which of the inputs is the output).
Making Decisions

Clock

- Depending on the control signal, the third flip-flop is updated to the value stored in the first or second one.

- Now we are able to
  - move around bits,
  - store them in registers, and read them again.

- Required: Generation of control signals.

Controlling the Flow of Data in a Circuit

- No need to understand the previous slide (shows a CPU).

- Need to understand:
  - A CPU consists of lots of units.
  - Each unit has an output depending on its input at the previous clock cycle.
  - The Control Unit (CU) controls the flow of data (bits) in the circuit.
  - It gets as input signals from various points in the CPU.
  - It outputs signals, which via multiplexers control the flow between these units.
  - In the diagram only a few example control paths were shown.
  - How does the Control Unit operate?
Operation of the Control Unit

- The control unit has an internal state, encoded by several bits.
  - Essentially the lines in a little program.
- Initially the state will be reset to 0.
- Then in each clock cycle, depending on
  - inputs from various points in the circuit,
  - and its own state from the previous clock cycle,
- the CU computes for the next clock cycle
  - control signals which control the data flow
  - and its own next state.
- This computation is a Boolean function, which can be done by a (very complicated) combinatorial circuit.

4. Computer Arithmetic and Representation of Data

(a) Unsigned Integers.
(b) Arithmetic Operations on Unsigned Integers.
(c) Signed Integers.
(d) Fixed Point Numbers.
(e) Floating-Point Numbers.
(f) Other Scalar Data Types.
(g) Compound Data Types.

Three Types of Data Types

- Scalar data types.
  - Hold a single data item.
  - Each data item has a unique representation.
  - Main examples:
    * Number types (integer, floating-point numbers).
    * Characters.
    * Booleans.
    * Finite data types (enumerations types, void).
- Compound data types.
  - Hold several data items.
  - Examples:
    * Strings (some books treat them as scalar).
    * Records, lists, arrays, maps.
- Reference and function types.
  - Have no unique representation.
  - Examples:
    * Pointers.
    * Classes in object-oriented programming.

Number of Values Representable

- With one bit we can represent two different values, “1” and “0”: e.g.: 1 represents true, 0 represents false; or: 1 represents blue, 0 represents red.
- With two bits we can represent four different values: 00, 01, 10, 11.
  - So we can represent for instance red as 00, blue as 01, green as 10 and yellow as 11.
- With three bits we can represent eight different values:
  - 4 values can be represented, using first bit 0: 000, 001, 010, 011.
  - 4 values can be represented, using first bit 1: 100, 101, 110, 111.
  - In total we can represent $2 \cdot 4 = 8$ values.
Number of Values Representable (Cont.)

- With every additional bit, the number of values we can represent doubles.
- With \( l \) bits we can represent \( 2 \cdot \ldots \cdot 2 = 2^l \) different values.

Unsigned Integers (Cont.)

- **Representation in ENIAC:** One decimal digit represented by 10 vacuum tubes, i.e. by 10 bits. (Bit = binary digit = digit 0 or 1)
- **Better approach:** With 4 bits we can represent \( 2^4 = 2 \cdot 2 \cdot 2 \cdot 2 = 16 \) objects. Represent each digit by 4 bits.
  - Appropriate for **financial data** (commercial rounding): Binary coded decimal (BCD): 135 coded as \( \begin{array}{c} 0001 0011 0101 \end{array} \).
  - Otherwise:
    - **Waste of space** (with 4 bits 16 symbols can be stored of which only 10 are used).
    - **Difficult to compute** by a circuit.
    - Better: Use **binary representation** of numbers.

(a) Unsigned Integers

- **Signed** integers = integers, which can be positive or negative.
- **Unsigned** integers = integers, which are always positive.
- Representation of unsigned integers easier.

Exponential Function

- Remember from school:
  \( 10^0 = 1 \).
  \( 10^1 = 10 \).
  \( 10^2 = 10 \cdot 10 = 100 \).
  \( 10^3 = 10 \cdot 10 \cdot 10 = 1000 \).
  \( 10^4 = 10 \cdot 10 \cdot 10 \cdot 10 = 10000 \).
  etc.
- Similarly:
  \( 7^0 = 1 \).
  \( 7^1 = 7 \).
  \( 7^2 = 7 \cdot 7 = 49 \).
  \( 7^3 = 7 \cdot 7 \cdot 7 = 343 \).
  \( 7^4 = 7 \cdot 7 \cdot 7 \cdot 7 = 2401 \).
  etc.
- Important to know the powers of 2:
  \( 2^0 = 1 \)    \( 2^7 = 128 \)
  \( 2^1 = 2 \)    \( 2^8 = 256 \)
  \( 2^2 = 4 \)    \( 2^9 = 512 \)
  \( 2^3 = 8 \)    \( 2^{10} = 1024 \)
  \( 2^4 = 16 \)   \( 2^{11} = 2048 \)
  \( 2^5 = 32 \)   \( 2^{12} = 4096 \)
  \( 2^6 = 64 \)   \( 2^{13} = 8192 \)
**Different Number Systems**

- **Decimal representation** of numbers:

  What's the meaning of 4053?

  We can assign a weight to each digit as follows:

  \[
  4053 = 4 \cdot 10^3 + 0 \cdot 10^2 + 5 \cdot 10^1 + 3 \cdot 10^0.
  \]

  Now the represented number is obtained by taking the sum of the product of each digit with its weight:

  4053 represents \(4 \cdot 10^3 + 0 \cdot 10^2 + 5 \cdot 10^1 + 3 \cdot 10^0\).

- **Binary representation** means: basis is 2:

  \((101101)_2 = 2^5 + 2^3 + 2^2 + 2^0 = 32 + 8 + 4 + 1 = 45\):

  \[
  \begin{array}{cccccccc}
  & 1 & 0 & 1 & 1 & 0 & 1 \\
  \hline
  2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
  \hline
  32 & 16 & 8 & 4 & 2 & 1
  \end{array}
  \]

  Long binary numbers are difficult to read, e.g. 100110001100001101101001.

  Therefore use of hexadecimal and octal representation.

- **Other bases**: Replace 10 by other numbers, e.g. 7.

  Notation \((4053)_7\) stands for 4053 in the number system with basis 7:

  \[
  \begin{array}{cccccccc}
  & 4 & 0 & 5 & 3 \\
  \hline
  7^3 & 7^2 & 7^1 & 7^0 \\
  \hline
  343 & 49 & 7 & 1
  \end{array}
  \]

  \((4053)_7 = 4 \cdot 343 + 5 \cdot 49 + 3 \cdot 7 + 1 \cdot 1 = 1410\).

- **Hexadecimal representation**: Basis 16.

  Lack of digits:

  New digits \(A, B, C, D, E, F\):

  \[
  \begin{array}{c|c}
  \text{digit} & \text{value} \\
  \hline
  A & 10 \\
  B & 11 \\
  C & 12 \\
  D & 13 \\
  E & 14 \\
  F & 15 \\
  \end{array}
  \]

  - Example:

    \[
    \begin{array}{cccccccc}
    & A & F & 0 & 3 \\
    \hline
    16^3 & 16^2 & 16^1 & 16^0 \\
    \hline
    4096 & 256 & 16 & 1
    \end{array}
    \]
Notations

- Hexadecimal: \(0xA03\) (sometimes \(A003\) for \(AF03\)) for \((AF03)_{16}\).
- Binary: \(0b1101110\) (sometimes \(0b1101110\)) for \((1101110)_{2}\).
  The notation \(1b\ldots\) does not occur in this lecture.

MSB. In binary representation with fixed length, the bit corresponding to the highest power of 2 is called the most significant bit, MSB.
- In standard Western writing it is the bit most to the left.
- On most machines it is the bit with the lowest number.
- On some machines this bit will be stored as the one with the highest number.

The bit corresponding to the lowest power of two is called the least significant bit, LSB.

Example:

\[
\begin{array}{cc}
\text{bit No. 0} & \text{bit No. 7} \\
\text{on most machines} & \text{on most machines}
\end{array}
\]

Example of use of Hexadecimal

(From a GameBoy Advance program, written in C)

\[
\text{// ***** REG_DISPCNT defines */}
\]

```
#define MODE_0 0x00
#define MODE_1 0x01
#define MODE_2 0x02
#define MODE_3 0x03
#define MODE_4 0x04
#define MODE_5 0x05
#define BACKBUFFER 0x10
#define H_BLANK_OAM 0x20
#define OBJ_MAP_2D 0x0
#define OBJ_MAP_1D 0x40
#define FORCE_BLANK 0x80
#define BG0_ENABLE 0x100
#define BG1_ENABLE 0x200
#define BG2_ENABLE 0x400
```

Conversion from Binary to Hexadecimal

Trivial transformation:
A binary number is transformed into hexadecimal, by packing from the right four bits into one hexadecimal digit:

Example:

\[
\begin{array}{c}
0b1001110111111111 \\
= 0b1001110111111111 \\
= 0x13BF
\end{array}
\]

Conversion from Hexadecimal to Binary

Write every digit as a binary number with four bits.

Example:

\[
\begin{array}{c}
0x13BF \\
= 0b1001110111111111 \\
= 0b1001110111111111 \\
= 0x13BF
\end{array}
\]

Leading 0's can be omitted.
Conversion from Binary to Octal

Similar to conversion binary to hexadecimal: A binary number is transformed into octal, by packing from the right three bits into one digit:

**Example:**

\[ \begin{array}{c}
0b100111011111 \\
= 0b100 111 011 111 \\
= (11677)_{8}
\end{array} \]

Conversion from Octal to Binary

Write every digit as a binary number with three bits.

**Example:**

\[ (11677)_{8} = (001 001 110 111 111)_{2} \]

Leading 0’s can be omitted.

Conversion from Binary/Hexadecimal/Octal to Decimal

Done using the above calculations. E.g.

- \( 0xF39 = 15 \cdot 16^2 + 3 \cdot 16^1 + 9 \cdot 16^0 \).
- \( 0b1011 = 2^3 + 2^1 + 2^0 \).

Conversion from Decimal to Binary

For small numbers practical method:

- Find highest power of 2 you can subtract.
- Continue with the rest, till you arrive at 0.
- Form a binary number with 1 at the positions corresponding to the powers of 2 used:

**Example**

15321 into binary:

<table>
<thead>
<tr>
<th>Calculation</th>
<th>Posit.</th>
</tr>
</thead>
<tbody>
<tr>
<td>15321 - 2^{14}</td>
<td>13</td>
</tr>
<tr>
<td>7129 - 2^{12}</td>
<td>12</td>
</tr>
<tr>
<td>3033 - 2^{11}</td>
<td>11</td>
</tr>
<tr>
<td>985 - 2^{9}</td>
<td>9</td>
</tr>
<tr>
<td>473 - 2^{8}</td>
<td>8</td>
</tr>
<tr>
<td>217 - 2^{7}</td>
<td>7</td>
</tr>
<tr>
<td>89 - 2^{6}</td>
<td>6</td>
</tr>
<tr>
<td>25 - 2^{4}</td>
<td>4</td>
</tr>
<tr>
<td>9 - 2^{3}</td>
<td>3</td>
</tr>
<tr>
<td>1 - 2^{0}</td>
<td>0</td>
</tr>
</tbody>
</table>

If we describe positions by subscripts we get

\[ 15321 = 2^{13} + 2^{12} + 2^{11} + 2^{9} + 2^{8} + 2^{7} + 2^{6} + 2^{4} + 2^{3} + 2^{0} \]

\[ = 0b111110111101111001 \]

\[ = 0x3BD9 \]

Better Algorithm

If we divide a decimal number by 10 with remainder, we obtain the number shifted to the right by 1, and as remainder the least significant digit:

<table>
<thead>
<tr>
<th>Result</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>153 ÷ 10</td>
<td>15</td>
</tr>
<tr>
<td>15 ÷ 10</td>
<td>1</td>
</tr>
<tr>
<td>1 ÷ 10</td>
<td>0</td>
</tr>
</tbody>
</table>

- The remainder are the digits of the decimal representation.
- Do the same to convert to binary representation.
### Conversion from Decimal to Hexadecimal

Probably easiest by hand: convert to binary, then to hexadecimal.

Otherwise: similar to the above method (divide by 16).

---

### Some Facts about Shifting Binary Numbers

#### Correctness

- If we shift a binary number once to the left, each “1” gets just twice the weight as in the original number.

- The new number represents the sum of the weights of the “1” in it, which is twice the sum of the weights of the “1” in the original number, which is twice the value of the original number.

- So $0b y 0 = 0b y \cdot 2$.

---

### Some Facts about Shifting Binary Numbers (Cont)

#### Fact

Let $y$ be a binary sequence, $\bar{y} 0$ the same sequence shifted once to the left with one new 0 inserted at the right.

Then

$$0b \bar{y} 0 = 0b y \cdot 2.$$  

- Example:

  $$0b 1 0 1 = 2^2 + 2^0 = 4 + 1 = 5$$

  $$\downarrow \downarrow\downarrow 2^2 2^1 2^0$$

  $$= = = =$$

  $$4 2 1$$

Shifted once to the left:

$$0b 1 0 1 0 = 2^3 + 2^1 = 8 + 2 = 10$$

$$\downarrow \downarrow\downarrow 2^3 2^2 2^1 2^0$$

$$= = = =$$

$$8 4 2 1$$
Some Facts about Shifting Binary Numbers

Fact

Shifting a binary number \( l \) times to the right and omitting the old LSB is the same as dividing it by \( 2^l \) with remainder. The \( l \) LSBs form the remainder with this division.

Examples:

- \( 0b110 = 2^2 + 2^1 = 6 \).
  Shifted once to the right: \( 0b11 = 2^1 + 2^0 = (2^2 + 2^1) \div 2^1 = 3 \).
  Remainder is \( 0 \).

- \( 0b111 = 2^2 + 2^1 + 2^0 = 7 \).
  Shifted once to the right: \( 0b11 = 2^1 + 2^0 = (2^2 + 2^1 + 2^0) \div 2^1 = 3 \).
  Remainder of this division is \( 1 \).

- \( 0b1010 = 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 26 \).
  Shifted twice to the right: \( 0b10 = 2^1 = 2 \).

Correctness

Write the original sequence of bits as \( 0byz \), where \( z \) are the \( l \) LSBs and \( y \) are the remaining bits.

Example:

For \( 0b111 \), shifted twice to the right, \( y = 111, z = 01 \).

Now \( 0byz = 0by \cdot 2^l + 0bz \).

In the example above:

\[
\begin{align*}
0by &= 0b11100, \\
0bz &= 0b01, \\
0byz &= 0b11101
\end{align*}
\]

(b) Arithmetic Operations on Unsigned Integers

(i) Addition

Addition of Decimals:

\[
\begin{align*}
1298 & \quad + \quad 751 \\
\text{Carry 11} & \quad \text{2049}
\end{align*}
\]

Note: Carry is at most \( 1 \):
If we add two decimal digits, i.e. two numbers in the interval \( 0 \ldots 9 \) we get at most 18.
It might be that we have to add carry one, then we obtain at most 19.

Addition of Binary Numbers:
Do just the same:

\[
\begin{align*}
&\begin{array}{c}
101101 \\
+ \quad 111110
\end{array} \\
\text{Carry 1111} & \quad 1101011
\end{align*}
\]
Addition of Decimals in Detail

Reconsider example from previous slide:

\[
\begin{array}{c}
1298 \\
+ \phantom{0}751 \\
\hline
\text{Carry} \phantom{0}11 \\
\hline
2049
\end{array}
\]

- **Step 1:** \[8 + 1 = 9, \text{ result } 9, \text{ carry } 0\]
  (carry 0 is not written down):

\[
\begin{array}{c}
1298 \\
+ \phantom{0}751 \\
\hline
\text{Carry} \phantom{0}9
\end{array}
\]

- **Step 2:** \[9 + 5 = 14, \text{ result } 4, \text{ carry } 1\]

Addition of Binaries in Detail

Reconsider example from above:

\[
\begin{array}{c}
101101 \\
+ \phantom{0}111110 \\
\hline
\text{Carry} \phantom{0}1111011
\end{array}
\]

- **Step 1:** \[1 + 0 = 1 = 0b01, \text{ result } 1, \text{ carry } 0\]
  Again carry 0 not written down:

\[
\begin{array}{c}
101101 \\
+ \phantom{0}111110 \\
\hline
\text{Carry} \phantom{0}1
\end{array}
\]

- **Step 2:** \[0 + 1 = 1 = 0b01, \text{ result } 1, \text{ carry } 0\]
Addition of Binaries in Detail (Cont.)

- **Step 6:** \(1 + 1 + 1 = 3 = 0b111\). Result 1, carry 1

\[
\begin{array}{c}
101101 \\
+ 111110 \\
\text{Carry } 111110 \\
\hline
101011
\end{array}
\]

- **Step 7:** \(0 + 0 + 1 = 1 = 0b01\). Result 1, carry 0

\[
\begin{array}{c}
101101 \\
+ 111110 \\
\text{Carry } 111110 \\
\hline
101011
\end{array}
\]

Half-Bit-Adder

Addition of two bits:
We have the following table for addition

<table>
<thead>
<tr>
<th>(A)</th>
<th>(B)</th>
<th>Carry</th>
<th>Sum</th>
<th>Corresponds to</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0b0 + 0b0 = 0b00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0b0 + 0b1 = 0b01</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0b1 + 0b0 = 0b01</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0b1 + 0b1 = 0b10</td>
</tr>
</tbody>
</table>

So we get

- Carry is 1, if both \(A\) and \(B\) are 1:
  \[\text{Carry} = A \land B\]
- Sum is 1, if at least one of \(A\) and \(B\) is 1, but not both:
  \[\text{Sum} = (A \lor B) \land \neg(A \land B)\]

Full One-Bit-Adder

A full one-bit adder allows to add two bits, plus a carry from the previous addition (which can be 1 or 0).
We have the following table:

<table>
<thead>
<tr>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
<th>Carry</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0b0 + 0b0 + 0b0 = 0b00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1b0 + 0b1 + 0b0 = 0b01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0b1 + 1b0 + 0b0 = 0b01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0b1 + 1b1 + 0b1 = 0b10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1b0 + 0b1 + 0b0 = 0b01</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0b1 + 1b0 + 1b1 = 0b10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1b1 + 1b1 + 0b0 = 0b10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1b1 + 1b1 + 1b1 = 0b11</td>
</tr>
</tbody>
</table>
Full One-Bit-Adder (Cont.)

We obtain:
Sum is 1 if exactly one of \(A\), \(B\) and \(C\), or all three are 1:
\[
Sum = (A \cdot \overline{B} \cdot C) + (A \cdot B \cdot \overline{C}) + (A \cdot B \cdot C)
\]

Carry is 1 of at least two of \(A\), \(B\), \(C\) are 1:
\[
Carry = (A \cdot B) + (A \cdot C) + (B \cdot C)
\]

Implementation using Two Half-Bit-Adders:

Correctness:
- Easiest: write a truth table, with inputs \(A\),\(B\),\(C\).
  Very good exercise !!!
- Or see next slide

Correctness:
- If we add \(A\), \(B\) we get result \(\text{Carry}_1\), \(\text{Sum}_1\).
- Now we add \(\text{Sum}_1\) and \(C\), with result \(\text{Carry}_2\), \(\text{Sum}_2\).
- \(\text{Sum}_2\) is now the final \(\text{Sum}\).
- If the first addition has \(\text{Carry}_1=0\), the carry of the complete sum will be \(\text{Carry}_2\).
- If the first addition has \(\text{Carry}_1=1\), then \(\text{Sum}_1=0\). Then the second addition can have only \(\text{Carry}_2 =0\). So the carry of the complete sum is 1 or \(\text{Carry}_1\).
- In both the cases we get:
  \[\text{Carry} = \text{Carry}_1 \lor \text{Carry}_2\]
**Correctness (Revised Version)**

- Adding $A$, $B$, $C$ can be achieved by first adding $A$, $B$ and then adding $C$ to the result:

$$
\begin{array}{c}
A \\
+ \\
\text{Carry1} \\
B \\
+ \\
\text{Sum1} \\
\hline \\
\text{Carry2} \\
C \\
+ \\
\text{Sum2}
\end{array}
$$

- Addition of $A$, $B$ is done by Half-Bit-Adder1.
- Sum2, Carry2 is computed by Half-Bit-Adder2.

- Carry is the result of adding Carry1 and Carry2.
  
  - We could use a half-bit-adder for this.
  
  - However observe:
    
    * If Carry1 = 1, then Sum1 = 0.
      
      (If a half-bit-adder has result Carry = 1, then Sum = 0.)
    
    * If Sum1 = 0, then Carry2 = 0.
      
      (If we add 0 to another bit, the carry is 0).
    
    * So we never have that both Carry1, Carry2 are 1.
    
    * But then the sum of Carry1, Carry2 is simply Carry1 $\vee$ Carry2.
      
      ($A \lor B$ is the sum of $A$ and $B$, if not both $A$, $B$ are 1).
    
    - Therefore Sum = Sum2, Carry = Carry1 $\vee$ Carry2.
      
      That’s the implementation.

**Comparison of the Two Implementations**

- **Advantage of second Implementation:**
  
  Fewer gates needed (7 instead of 9).

- **Disadvantage of second Implementation:**
  
  Delay, because the signal has to pass through more gates:
  
  - In a half bit adder,
    
    carry passes through 1 gate,
    
    sum through 2 gates.
  
  - So sum in the second implementaiton passes through
    
    $\underbrace{2}_{\text{First Adder}} + \underbrace{2}_{\text{Second Adder}}$ gates,
    
    carry passes through
    
    $\underbrace{2}_{\text{First Adder}} + \underbrace{1}_{\text{Second Adder}} + \underbrace{1}_{\text{gate}}$ gates.
    
    - In first implementation each signal has to pass through 2 gates only.
  
  - Addition is the most frequent operation!!

**Addition of l bit numbers**

Remember:

$$
\begin{array}{c}
101101 \\
+ \\
111110 \\
\hline \\
1101011
\end{array}
$$

- Add the two LSBs with carry.
  
  - Sum is LSB of result.

- Add next two LSBs plus carry of previous addition.
  
  - Sum is next LSB of result.

- Etc.
Carry

ABAB ABAB

Possible Carry from previous Addition

Sum

Full adder Full adder Full adder Full adder

Bit 2

MSB

Bit 0 Bit 1

LSB

Bit 3

Output

Input

A Four Bit Adder

Overflow

• If there is carry of the addition of MSBs plus carry, we have an overflow:
  Sum requires one more bits than the summands.

• Example:

\[
\begin{array}{c}
1 \\
1 \\
1 \\
\end{array}
\quad +
\begin{array}{c}
1 \\
1 \\
1 \\
\end{array}
\quad \text{Carry} \begin{array}{c}
1 \\
1 \\
1 \\
\end{array}
\hline
1 \\
1 \\
1 \\
0
\end{array}
\]

Result cannot be represented with 3 bits.

• This extra bit is called carry flag.
  (In general flag are one-bit-registers, which are usually set automatically in response to operations).

Carry Flag

• When adding long numbers, one might carry out addition in two steps:

\[
\begin{array}{c}
0 \\
1 \\
1 \\
\end{array}
\quad 01 \\
\quad 01 \\
\quad 01 \\
\quad \text{Carry} \\
\quad 1 \\
\quad 11 \\
\end{array}
\]

  – First add \(0b10 + 0b11 = 0b101\), by using one two-bit-adder.
  Result is \(0b101\) plus \(\text{Carry} 1\).

  – Now add \(0b01 + 0b01 + \text{previous carry}\)
  using the same two-bit-adder
  Result is \(0b01 + 0b01 + 0b1 = 0b11\).

  – Result is result of second and first addition in sequence.

• In practice, one has for instance a 32-bit adder, and can use it in order to add in two steps two 64-bit-numbers.

Carry Flag

• In machine language, addition of two \(l\)-bit numbers has as result usually a \(l\)-bit number plus a carry, corresponding to the carry of the addition of the MSBs (plus previous carry).

  – Carry might be taken over to the next addition,
  – or might indicate an overflow.

  – Depends on the context (has to be decided by the assembler programmer).
Carry Lookahead

- Problem with $n$-bit adders: The carry-input for the $k$th bit has to pass in worst case through $k - 1$ full bit-adders, which causes delay.

- Solution: Carry Lookahead:
  - Compute carries directly from the bits of the inputs (passes through less gates).
  - Note that the $k$th carry is a function from the least significant $k$ bits of the addends).
  - Becomes complicated with increasing $k$, several techniques for optimization.

Naive Multiplication of Binary Numbers

- Do the same with binary numbers.
  \[ 0b1101 \times 0b1011 = \]
  \[ \begin{array}{c}
  1101 \\
  + 1101 \\
  + 1101 \\
  \hline
  0b10001111
  \end{array} \]

- Steps in this example:
  - Multiply $0b1101$ by 1. Result is $0b1101$.
  - Multiply $0b1101$ by 1. Result is $0b1101$.
  - Shift result once to the left.
  - Multiply $0b1101$ by 0. Result is 0.
  - Shift result twice to the left (or ignore it because it is zero).
  - Multiply $0b1101$ by 1. Result is $0b1101$.
  - Shift result three times to the left.
  - Add the results above.

(ii) Multiplication

Multiplication of Decimal Numbers

\[
345 \times 123 = \\
\begin{array}{c}
1035 \\
+ \ 690 \\
+ \ 345 \\
\hline
42435
\end{array}
\]

Steps in this example:

- Multiply 345 by 3. Result is $1035$.
- Multiply 345 by 2. Result is $690$.
  - Shift result once to the left.
- Multiply 345 by 1. Result is $345$.
  - Shift result twice to the left.
- Add the three numbers.

Naive Multiplication of Binary Numbers (Cont.)

- We will in the following interchange the role of multiplier and multiplicand.
  \[
 0b1101 \times 0b1011 = \]
  \[
  \begin{array}{c}
  1011 \\
  + 1011 \\
  \hline
  0b10001111
  \end{array}
  \]

- Since multiplication is commutative, order of factors doesn't matter.
- This order is better for the presentation.

- In the algorithm below we will make use of a variable sum.
  - sum is initially zero.
  - Whenever we have to add the shifted multiplier, we add it to sum.
  - Then at the end, sum contains the result.

- It's easier to shift the multiplier successively to the left, and add it to sum
  - (rather than shifting it when needed several times to the left).
Naive Multiplication of Binary Numbers (Cont.)

- When calculating by hand, we write blanks in front and add the end of the number to be added. Those blanks stand for the value 0 and have, when working on the computer, to be replaced by zeros.

So the previous multiplication reads as follows:

\[
\begin{align*}
\text{0b1101} \cdot \text{0b1011} &= \text{0b00001011} \\
&\quad + \text{0b00101100} \\
&\quad + \text{0b01011000} \\
&= \text{0b10001111}
\end{align*}
\]

- Example (Naive Algorithm)

<table>
<thead>
<tr>
<th>Step</th>
<th>m</th>
<th>m'</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>011</td>
<td>000 101</td>
<td>000 000</td>
</tr>
<tr>
<td>Step 1a</td>
<td>011</td>
<td>000 101</td>
<td>000 101</td>
</tr>
<tr>
<td>Step 1b</td>
<td>011</td>
<td>001 010</td>
<td>001 101</td>
</tr>
<tr>
<td>Step 2a</td>
<td>011</td>
<td>001 010</td>
<td>001 101</td>
</tr>
<tr>
<td>Step 2b</td>
<td>011</td>
<td>010 100</td>
<td>001 111</td>
</tr>
<tr>
<td>Step 3a</td>
<td>011</td>
<td>010 100</td>
<td>001 111</td>
</tr>
<tr>
<td>Step 3b</td>
<td>011</td>
<td>101 000</td>
<td>001 111</td>
</tr>
</tbody>
</table>

\[
\text{0b1111} \cdot \text{0b101} = 3 \cdot 5 = 15 = \text{0b1111}
\]

Slightly Better Algorithm

- Start with sum 0.

- Shift successively
  - multiplier (m) to the right (n times if we have n bits).
  - as before multiplicand (m') to the left,

- Whenever the LSB of multiplier (m) is 1, add multiplicand (m') shifted appropriately to the sum.

- Result is sum.
Second Algorithm for the Multiplication of $m$ and $m'$

$(m$ is multiplicand, $m'$ is multiplier).
Assume $m$, $m'$ are $k$-bit numbers.

```plaintext
sum := 0
for $i = 1$ to $k$ do
  begin
    if LSB($m$) = 1 then sum := sum + $m'$
    Shift $m$ right one bit
    Shift $m'$ left one bit
  end
sum is result
```

Example (Second Algorithm)

<table>
<thead>
<tr>
<th>Step</th>
<th>$m$</th>
<th>$m'$</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>011</td>
<td>000 101</td>
<td>000 000</td>
</tr>
<tr>
<td>Step 1a</td>
<td>011</td>
<td>000 101</td>
<td>+ 000 101</td>
</tr>
<tr>
<td>Add $m'$ to sum</td>
<td></td>
<td></td>
<td>000 101</td>
</tr>
<tr>
<td>Step 1b</td>
<td>001</td>
<td>001 010</td>
<td>000 101</td>
</tr>
<tr>
<td>Shifts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 2a</td>
<td>001</td>
<td>001 010</td>
<td>+ 001 010</td>
</tr>
<tr>
<td>Add $m'$ to sum</td>
<td></td>
<td></td>
<td>001 111</td>
</tr>
<tr>
<td>Step 2b</td>
<td>000</td>
<td>010 100</td>
<td>001 111</td>
</tr>
<tr>
<td>Shifts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 3a</td>
<td>000</td>
<td>010 100</td>
<td>001 111</td>
</tr>
<tr>
<td>LSb($m$)</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No action</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 3b</td>
<td>000</td>
<td>101 000</td>
<td>001 111</td>
</tr>
<tr>
<td>Shifts</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$0b011 \cdot 0b101 = 3 \cdot 5 = 15 = 0b1111.$

Problem

- We need a $2k$-bit adder when multiplying two $k$-bit numbers.
- However the number we add has always $k$ zeros (in a block left and right of the original multiplier).
- Further before the addition, sum has to the left of the original multiplier only zeros.
  - So the addition will affect only the bits which possibly are occupied by the multiplier ($m'$).

Steps towards the Third Algorithm

- Let the sum in the new algorithm be
  - the sum of the original algorithm,
  - but shifted to the left,
  - so that the multiplier to be added has to be added to the $k$ MSBs of the new sum.
- In our example the new sum is the old sum,
  - in step 1a three bits shifted to the right;
  - in step 2a two bits shifted to the right;
  - in step 3a one bit shifted to the right.
- Then addition takes place only in the $k$ MSBs bits of sum.
  - Requires only an $k$-bit adder.
- However, now the new sum has to be adjusted when moving from one step to the next.
  - After the step, the new sum is one bit less shifted to the left than before.
  - So the new sum has to be shifted one bit to the right after step 1a, 2a, 3a etc.
After adding the multiplier, we might obtain one overflow bit.

- This corresponds to the original sum having one bit to the left of the multiplier.

**Solution:**
- Let sum have $2k + 1$ instead of $2k$ bits.
- We add the multiplier to the $k+1$ MSBs of sum.

---

**Third Algorithm for the Multiplication of $m$ and $m'$**

Assume $m$, $m'$ are $k$-bit numbers.

- sum is a $(2k + 1)$-bit number.
- sum := 0
- for $i = 1$ to $k$ do
  - begin
    - if LSB$(m) = 1$ then add $m'$ to the $k+1$ most significant bits of sum
  - Shift $m$ right one bit
  - Shift sum right one bit
  - end
- sum is result

---

**Example (Third Algorithm)**

<table>
<thead>
<tr>
<th>Initialization</th>
<th>$m$</th>
<th>$m'$</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>011</td>
<td>101</td>
<td>0 000 000</td>
</tr>
<tr>
<td>Step 1a</td>
<td>011</td>
<td>101</td>
<td>+ 101</td>
</tr>
<tr>
<td>LSB$(m) = 1$</td>
<td></td>
<td></td>
<td>0 101 000</td>
</tr>
<tr>
<td>Add $m'$</td>
<td></td>
<td></td>
<td>0 101 000</td>
</tr>
<tr>
<td>Step 1b</td>
<td>001</td>
<td>101</td>
<td>0 010 100</td>
</tr>
<tr>
<td>Shifts</td>
<td></td>
<td></td>
<td>0 010 100</td>
</tr>
<tr>
<td>Step 2a</td>
<td>001</td>
<td>101</td>
<td>+ 101</td>
</tr>
<tr>
<td>LSB$(m) = 1$</td>
<td></td>
<td></td>
<td>0 111 100</td>
</tr>
<tr>
<td>Add $m'$</td>
<td></td>
<td></td>
<td>0 111 100</td>
</tr>
<tr>
<td>Step 2b</td>
<td>000</td>
<td>101</td>
<td>0 011 110</td>
</tr>
<tr>
<td>Shifts</td>
<td></td>
<td></td>
<td>0 011 110</td>
</tr>
<tr>
<td>Step 3a</td>
<td>000</td>
<td>101</td>
<td>0 011 110</td>
</tr>
<tr>
<td>LSB$(m) = 0$</td>
<td></td>
<td></td>
<td>0 011 110</td>
</tr>
<tr>
<td>No action</td>
<td></td>
<td></td>
<td>0 011 110</td>
</tr>
<tr>
<td>Step 3b</td>
<td>000</td>
<td>101</td>
<td>0 001 111</td>
</tr>
<tr>
<td>Shifts</td>
<td></td>
<td></td>
<td>0 001 111</td>
</tr>
</tbody>
</table>

$0b111 \cdot 0b101 = 3 \cdot 5 = 15 = 0b1111$.

- Observe that in the last example at the end of Step 1,
  - the $4 + l$ MSBs of sum (written in italic, in colour blue) are the same as
  - the $4 + l$ LSBs of sum in the previous algorithm.
**Example, in which One Extra Bit is Needed:**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialization</strong></td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td><strong>Step 1a</strong></td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td><strong>Add m’</strong></td>
<td>011</td>
<td>111</td>
</tr>
<tr>
<td><strong>Shifts</strong></td>
<td>001</td>
<td>111</td>
</tr>
<tr>
<td><strong>Step 2a</strong></td>
<td>001</td>
<td>111</td>
</tr>
<tr>
<td><strong>Add m’</strong></td>
<td>000</td>
<td>111</td>
</tr>
</tbody>
</table>

0b111 · 0b111 = 7 · 7 = 49 = 0b110001.

---

**Fourth Algorithm for the Multiplication of m and m’**

We can store m in the least significant bits of the sum. Then we have only to shift sum to the right. (Saves shift logic).

**Algorithm:**

sum is a (2k + 1)-bit number.

```plaintext
sum := 0
```

Set the k LSBs of sum to m for i = 1 to k do

begin

if LSB(sum) = 1 then add m’ to the k+1 MSBs of sum

Shift sum right one bit

end

sum is result

---

**Example (Fourth Algorithm)**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialization</strong></td>
<td>011</td>
<td>101</td>
</tr>
<tr>
<td><strong>Step 1a</strong></td>
<td>011</td>
<td>101</td>
</tr>
<tr>
<td><strong>Add m’</strong></td>
<td>011</td>
<td>101</td>
</tr>
<tr>
<td><strong>Shift</strong></td>
<td>011</td>
<td>101</td>
</tr>
<tr>
<td><strong>Step 2a</strong></td>
<td>011</td>
<td>101</td>
</tr>
<tr>
<td><strong>Add m’</strong></td>
<td>011</td>
<td>101</td>
</tr>
<tr>
<td><strong>Shift</strong></td>
<td>011</td>
<td>101</td>
</tr>
</tbody>
</table>

0b011 · 0b101 = 3 · 5 = 15 = 0b1111.

---

- In the last example, the bar | separates
  - the bits identical with the **sum** in the previous algorithm (written in italic, in colour blue)
  - from m’ (written in **boldface**, in colour green)
  - which are both shifted to the right simultaneously.
Implemented Multiplication

- Implementations will use a fifth algorithm, Booth’s algorithm.
- Booth’s algorithm best explained by referring to signed integers.
- Therefore we break off here, and continue with multiplication algorithms in the subsection about signed integers.

Naive Subtraction of Decimal Numbers

- Example:
  \[
  \begin{array}{c}
  1743 \\
  - \quad 345 \\
  \hline
  \end{array}
  \]
  \[
  1398
  \]

- **Step 1:**
  \[
  3 - 5 \text{ is } 8 \text{ with borrow } 1. \\
  \text{(since } 13 - 5 = 8). \\
  \begin{array}{c}
  1743 \\
  - \quad 345 \\
  \hline
  \end{array}
  \]
  \[
  1398
  \]

- **Step 2:**
  \[
  4 - (4 + 1) \text{ (+1 is the borrow) is } 9 \text{ with borrow } 1. \\
  \text{(since } 14 - (4 + 1) = 9). \\
  \begin{array}{c}
  1743 \\
  - \quad 345 \\
  \hline
  \end{array}
  \]
  \[
  1398
  \]

Naive Subtraction of Binary Numbers

- Naive subtraction of binary numbers similarly:
  \[
  \begin{array}{c}
  0b10101 \\
  - \quad 0b1111 \\
  \hline
  \end{array}
  \]
  \[
  0b00110
  \]

- Steps of the computation:
  - **Step 1:**
    \[
    1 - 1 = 0. \\
    \begin{array}{c}
    0b10101 \\
    - \quad 0b1111 \\
    \hline
    \end{array}
    \]
    \[
    0b10
    \]
  - **Step 2:** 0 - 1 = 1 borrow 1,
    \[
    \text{(since } 0b10 - 1 = 1) \\
    \begin{array}{c}
    0b10101 \\
    - \quad 0b01111 \\
    \hline
    \end{array}
    \]
    \[
    0b10
    \]

Computer Systems, CS_M33, Michaelmas term 2002, Sect. 4
- **Step 3:** \((1 + 1) = 1 - 0b10 = 1\) borrow 1. 
  (since \(0b11 - 0b10 = 1\))

\[
\begin{array}{c}
0b10101 \\
- 0b01111 \\
\hline
0b 110
\end{array}
\]

- **Step 4:** \(0 - (1 + 1) = 0 - 0b10 = 0\) borrow 1. 
  (since \(0b10 - 0b10 = 0\))

\[
\begin{array}{c}
0b10101 \\
- 0b01111 \\
\hline
0b 0110
\end{array}
\]

- **Step 5:** \(1 - 1 = 0\).

\[
\begin{array}{c}
0b10101 \\
- 0b01111 \\
\hline
0b00110
\end{array}
\]

**Observations about Binary Numbers**

- We have
  - \(0b1 = 1 = 2^0\),
  - \(0b10 = 2 = 2^1\),
  - \(0b100 = 4 = 2^2\), etc.
  - In general \(0b1 \overbrace{0\cdots0}^{k} = 2^k\).

- Further

\[
\begin{array}{c}
0b \overbrace{1\cdots1}^{k} = 0b1 \overbrace{0\cdots0}^{k} - 0b1 \\
= 2^k - 1:
\end{array}
\]

\[
\begin{array}{c}
0b10 \cdots 000 \\
- 0b 1 \\
\hline
0b 1 \cdots 111
\end{array}
\]

- \(2^{k-1} - 1 = 0b \overbrace{1\cdots1}^{k}\) is as well the **largest number** representable as an unsigned number. Therefore the range of numbers representable as unsigned numbers with \(k\) bits is \(0, 1, 2, \ldots, 2^{k-1} - 1\).

**Notations**

- If \(y\) is a finite sequence of bits with fixed number of bits,
  - \(0y\) is \(y\), extended by one new MSB 0,
  - \(1y\) is \(y\), extended by one new MSB 1

- E.g. if \(y = 1010\), then
  - \(0y = 01010\),
  - \(1y = 11010\).
(c) Signed Integers

- **Goal**: Representation of positive and negative numbers.

- **Oldest Method**: Sign-magnitude representation.
  - One more additional MSB for representation of the sign.
  - MSB = 0 means +, MSB = 1 means -.
  - The remaining bits represent the absolute value.
  - So, if $y$ unsigned represents $l$, then
    - $0y$ represents $+l$,
    - $1y$ represents $-l$.

- **Example**:
  0011 represents +3,
  1011 represents -3.

### Disadvantages of Sign-Magnitude Representation

- Two representations of 0 (+0, -0).
- Arithmetic complicated. Especially in case of addition one needs 4 different algorithms depending on the sign of the first and second argument.
- So in the following we will use a different representation, called **two's complement representation**.

---

**Two's Complement Representation**

- Idea: If we naively subtract 1 from 0, we obtain a number consisting of infinitely many ones:

  \[
  \begin{align*}
  0 & \quad - \quad 1 \\
  \cdots & \quad \cdots
  \end{align*}
  \]

- Similarly $0 - 10 = \cdots 1110$:

  \[
  \begin{align*}
  0 & \quad - \quad 10 \\
  \cdots & \quad \cdots
  \end{align*}
  \]

- $0 - 11 = \cdots 11101$:

  \[
  \begin{align*}
  0 & \quad - \quad 11 \\
  \cdots & \quad \cdots
  \end{align*}
  \]

---

**Two's Complement Representation (Cont.)**

- Similarly, we can consider positive numbers as sequences of numbers (growing to the left) with infinitely many zeros to the left:

- $0b10$ can be written as $\cdots 00010$,
- $0b101$ is written as $\cdots 000101$. 
Two's Complement Representation (Cont.)

- Since we have only finitely many bits available, we have to cut the number of bits off at some position.

- We fix the number \( k \) of bits in our representation.

- In this representation
  - an MSB 1 means that there are infinitely many 1s to the left,
  - an MSB 0 means that there are infinitely many 0s to the left.

- So we have for \( k = 3 \)
  - 011 represents \( \cdots 0011 \), or 3.
  - 010 represents \( \cdots 0010 \), or 2.
  - 001 represents \( \cdots 0001 \), or 1.
  - 000 represents \( \cdots 0000 \), or 0.
  - 111 represents \( \cdots 1111 \), or \(-1\).
  - 110 represents \( \cdots 1110 \), or \(-2\).
  - 101 represents \( \cdots 1101 \), or \(-3\).
  - 100 represents \( \cdots 1100 \), or \(-4\).

Positive Numbers

Assume for simplicity we cut off after 3 bits. We can represent the following numbers:

\[
\begin{align*}
+3 & = 011 \\
+2 & = 010 \\
+1 & = 001 \\
0 & = 000 \\
-1 & = 111 \\
-2 & = 110 \\
-3 & = 101 \\
-4 & = 100
\end{align*}
\]

- Positive numbers are 2-bit unsigned integers, extended by a new MSB 0.

- With \( k \) bits, positive numbers are \( k - 1 \)-bit unsigned integers, extended by new MSB 0.

- The largest number representable with \( k \) bits is the largest unsigned number representable with \( k - 1 \) bits namely \( 2^{k-1} - 1 \).

Negative Numbers

\[
\begin{align*}
+3 & = 011 \\
+2 & = 010 \\
+1 & = 001 \\
0 & = 000 \\
-1 & = 111 \\
-2 & = 110 \\
-3 & = 101 \\
-4 & = 100
\end{align*}
\]

- Negative numbers are of the form \( \cdots 1y \).

- When we increase \( x \), the number increases.
  - Different from sign-magnitude increases.
  - There \( 1y \) denotes \(-0by\). If one increases \( y \), the number decreases.
Two's Complement Representation
Negative Numbers (Cont.)

+3 = 011
+2 = 010
+1 = 001
0 = 000
-1 = 111 = -4 + 3 = -4 + 0b11
-2 = 110 = -4 + 2 = -4 + 0b10
-3 = 101 = -4 + 1 = -4 + 0b01
-4 = 100 = -4 + 0 = -4 + 0b00

-1x represents -4 + 0bx.

Least number we can represent is -4.

With k bits, 1y represents -2^{k-1} + 0by.
The least number we can represent is -2^{k-1}.

So in total, with k bits, we can represent numbers in the range -2^{k-1}, -2^{k-1} + 1, ... , 2^{k-1} - 1 (with 3 bits the range is -4, -3, -2, -1, 0, 1, 2, 3 with 4 bits it is -8, -7, ... , 6, 7).

Translation of Two's Complement into Decimal Notation

The number represented by finite bit sequence two’s complement form is the sum of the weights of the 1s in it:

* (101101)_{\text{signed}} = -2^5 + 2^3 + 2^2 + 2^0
  = -32 + 8 + 4 + 1 = -19:
  1 0 1 1 0 1
  \text{ weights: } 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
  \text{ sum: } -32 16 8 4 2 1

* (001101)_{\text{signed}} = 2^3 + 2^2 + 2^0
  = 8 + 4 + 1 = 13:
  0 0 1 1 0 1
  \text{ weights: } 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
  \text{ sum: } -32 16 8 4 2 1

Another example:

* (10001110)_{\text{unsigned}} = 128 + 8 + 4 + 2 = 114:
  1 0 0 0 1 1 1 0
  \text{ weights: } 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
  \text{ sum: } -128 64 32 16 8 4 2 1

Notation:
- \langle y \rangle_{\text{signed}} is the value of y in two’s complement form
- As before 0by stands for the value of the sequence y as a binary unsigned number.

- (0y)_{\text{unsigned}} = 0by,
  (1y)_{\text{unsigned}} = -2^{k-1} + 0by.

So the contribution of the MSB in two’s complement form with k bits to the value is -2^{k-1}:
The MSB has weight -2^{k-1}.
- This is the negative value of the weight it would have as a signed number (i.e. 2^{k-1}).

All other bits have the same weight as for unsigned numbers.
Observe

- In two’s complement form, MSB = 1 means the number is **negative**.
- MSB = 0 means the number is **positive**.
- So the MSB indicates whether the number is positive or negative.
- But two’s complement form is not **sign-magnitude representation**:
  For instance 101 denotes not 0 but \(4 + 1 = 3\).

#### Algorithm for Negating Two’s Complement Numbers

- Assume \(m = (z)_{\text{signed}}\), where \(z\) is a \(k\)-bit binary number.
- Assume \(z \neq 1 \underbrace{0 \cdots 0}_{k-1}\).
  - (The negation of \(1 \underbrace{0 \cdots 0}_{k-1}\) = \(-2^{k-1}\) is \(2^{k-1}\), which cannot be represented with \(k\) bits).
- In order to compute \(-m\), do the following:
  - Form the bitwise complement of \(z\) as a \(k\)-bit number.
    i.e.: replace every 0 by a 1, every 1 by a 0.
  - Add one (as if it were an unsigned number), and ignore any carry.
  - The result represents \(-m\).
Algorithm for Translating Decimal Integers into Two’s Complement

- With \( k \) bits we can represent the range \(-2^{k-1}, \ldots, 2^{k-1} - 1\).
  (eg. with 3 bits, this range is \(-4, -3, -2, -1, 0, 1, 2, 3\).)
- If \( l \) is not in this range, we cannot represent \( l \), i.e. we have overflow or underflow.
- Assume \( l \) is in the range.

Example

- Calculation of the representation of 22 with 8 bits
  \[ 22 = 0b0010110 = (00010110)_{\text{signed}}.\]
- Calculation of the representation of \(-22\) with 8 bits:
  \[ 22 = 0b0010110 = (00010110)_{\text{signed}} \]
- Formation of the bitwise complement:
  \[
  \begin{align*}
  00010110 \\
  11101001 \\
  \text{Result is } 11101001.
  \end{align*}
  \]
- Add one. Result is 11101010.
- \(-22 = (11101010)_{\text{signed}}.\)

Expansion of Two’s Complement Numbers

- Assume \( m = (z)_{\text{signed}}, z \) has \( k \) bits.
- The representation of \( m \) with \( k + 1 \) bits is obtained by adding in front of \( z \) the MSB of \( z \).

Examples:

- \( z = 0101 \) is expanded by adding a 0 in front of it:
  \( (0101)_{\text{signed}} = (00101)_{\text{signed}} = (00010110)_{\text{signed}}.\)
- \( z = 1101 \) is expanded by adding a 1 in front of it:
  \( (1101)_{\text{signed}} = (11101)_{\text{signed}} = (111101)_{\text{signed}}.\)
Observe

- If we add a 0 in front of a negative number, we obtain a positive number, not the same number.

- Example:
  - $(11101010)_{\text{signed}} = (-128) + 64 + 32 + 8 + 2 = -22$
    (as a signed 8 bit number):
    \[
    \begin{array}{cccccc}
    1 & 1 & 1 & 0 & 1 & 0 \\
    \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
    -128 & 64 & 32 & 16 & 8 & 4 & 2 & 1
    \end{array}
    \]
  - $(011101010)_{\text{signed}} = 128 + 64 + 32 + 8 + 2 = 234$
    as a 9 bit number.
  - The correct expansion of the signed number $(11101010)_{\text{signed}}$ to 9 bits is $(111101010)_{\text{signed}}$.

Warning

- Don’t mix up sign-magnitude representation with two’s complement.

- 1011 in two’s complement representation is not to be interpreted as $-0011$, but as $-00101$:
  - $(1011)_{\text{signed}} = -8 + 2 + 1 = -5$.

Addition of Two’s Complement Numbers

- Addition of two $k$-bit numbers $m$, $m'$ in two’s complement form is carried out as follows:
  - Add $m$, $m'$ using the same algorithm as for unsigned numbers, with result as a $k$-bit number. Ignore a possible carry.
  - Interpret the result as a signed number $n$.
  - Case 1: $m, m'$ are positive, $n$ is negative as a two’s complement number.
    * Then there is an overflow.
    Result too big to be represented.
  - Case 2: $m, m'$ are negative, $n$ is positive as a two’s complement number.
    * Then there is an underflow.
    Result is negative and too small to be represented.
  - Otherwise there is no under/overflow, $n$ as a $k$-bit number in two’s complement representation is the sum.

Examples

- $(0110)_{\text{signed}} + (0100)_{\text{signed}}$ yields an overflow, since $0110 + 0100 = 1010$, which appears to be negative.

- $(0100)_{\text{signed}} + (0010)_{\text{signed}} = (0110)_{\text{signed}}$, since $0100 + 0010 = 0110$ and the result is positive.

- $(1001)_{\text{signed}} + (1101)_{\text{signed}}$ yields an underflow, since $1001 + 1101 = 0110$ (ignoring a carry), which is positive.

- $(1101)_{\text{signed}} + (1011)_{\text{signed}} = (1000)_{\text{signed}}$, since $1101 + 1011 = 1000$ (ignoring a carry!) which is negative.

- $(1000)_{\text{signed}} + (0101)_{\text{signed}} = (1101)_{\text{signed}}$, since $1000 + 0101 = 1101$.

- $(1001)_{\text{signed}} + (0111)_{\text{signed}} = (0000)_{\text{signed}}$, since $1001 + 0111 = 0000$ (ignoring a carry).
Subtraction of Two’s Complement Numbers

- \( m - m' = m + (-m') \).
- Therefore in order to subtract \( m' \) from \( m \), add the negation of \( m' \) to \( m \).
- Alternatively use naive subtraction, which can be lifted directly to two’s complement.

(i) Multiplication of Two’s Complement Numbers

Result Signed

- A block of ones, \( 0b1 \ldots 1 \), unsigned, represents \( 2^k - 1 \).
- So multiplication of \( 0b1 \ldots 1 \) with \( x \) has result \((2^k - 1)x = 2^k x - x\).
- \( 2^k x \) is \( x \) shifted \( k \) bits to the left.
- Therefore \( 0b1 \ldots 1 \cdot x \) is the result of subtracting \( x \) shifted \( k \) bits to the left \( x \) not shifted.
- For instance \( 0b111 \cdot 0b101 = (0b1000 \cdot 0b101) - (0b1 \cdot 0b101) = 0b101000 - 0b101 = 0b100011 \) is calculated as follows:
  \[
  \begin{array}{c|c|c}
  \hline
  \text{+0b101} & \text{+0b101 \cdot 0b1000} & \text{0b100011} \\
  \text{-0b101} & \text{-0b101} & \text{0b100011} \\
  \hline
  \end{array}
  \]

Result Signed (Cont.)

- Reconsider the example above:
  - Before adding \( 0b101 \) we shifted it, so that the LSB of it is in the first bit to the left of \( 0b111 \).
  - Since \( 0b111 = 0b0111 \), this is at the first bit after the end of the block of ones.
  - So we add first \( 0b101 \), shifted so that its LSB is at the the end of the block of ones, and subtract from it \( 0b101 \) so that its LSB is at the beginning of the block of ones.
- \( 0b1110 \cdot 0b101 = (0b10000 - 0b10) \cdot 0b101 \) can be computed as follows:
  \[
  \begin{array}{c|c|c}
  \hline
  \text{+0b101} & \text{+0b101 \cdot 0b1000} & \text{0b1000110} \\
  \text{-0b101} & \text{-0b101 \cdot 0b10} & \text{0b10000110} \\
  \hline
  \end{array}
  \]
- Again: we add \( 0b101 \) so that it is shifted to the end of the block of ones, and subtract from it \( 0b101 \) shifted to the beginning of the block of ones.
Mult. of Unsigned Numbers
Result Signed (Cont.)

If we start from the right when detecting blocks of ones in the multiplicand, we first detect the beginning of the block of ones, then the end of it. Therefore we have to subtract first \( \text{0b}101 \), and then add it (shifted accordingly).

If we subtract \( \text{0b}101 \) first, we get a negative number. In order to deal with that, we treat the result as signed numbers in two's complement. Then we have to convert \( \text{0b}101 \) into a signed number before subtracting or adding it – can be done by adding at least one 0 on the left side. Further, in order to represent the result, a 6-bit signed number, we need 7 bits in order to represent it as an unsigned number.

The calculation is as follows (all calculations are carried out as signed numbers):

\[
\begin{align*}
\text{0b}111 \cdot \text{0b}101 &= \quad 0 \\
- \quad \text{0101} & \quad \downarrow \\
\text{111011} & \quad \uparrow \\
+ \quad \text{0101} & \\
\text{010011} &
\end{align*}
\]

So when multiplying the above number with \( x \) we

- start with \( 0, y \)
- subtract \( x \) shifted 0 bits to the left, *
  (beginning of the first block of ones)
- add \( x \) shifted 3 bits to the left, *
  (after the end of the first block of ones)
- subtract \( x \) shifted 6 bits to the left, *
  (beginning of the 2nd block of ones)
- add \( x \) shifted 7 bits to the left, *
  (after the end of the 2nd block of ones)
- subtract \( x \) shifted 10 bits to the left, *
  (beginning of the 3rd block of ones)
- add \( x \) shifted 13 bits to the left *
  (after the end of the 3rd block of ones).

Any binary number can be split into several blocks of ones.
- Example: \( 0 \text{111000} \), \( 000 \text{111} \).

A number is the sum of numbers, which consist of selecting one of these blocks only.
- \( \text{0111000100} \text{111} = \text{01110000000000} + \text{00000001000000} + \text{0000000000111} \)

Multiplying \( x \) with the previous number is the result of multiplying it with each of \( \text{0b}11110000000000 \), \( \text{0b}00000000100000 \), \( \text{0b}0000000000111 \), and adding the results.

Example:

\[
\begin{align*}
\text{0b}111011 \cdot \text{0b}110 &= \quad 0 \\
- \quad \text{0110} & \quad \downarrow \\
\text{1111110110} & \quad \uparrow \\
+ \quad \text{0110} & \\
\text{0000100110} & \quad \downarrow \\
- \quad \text{0110} & \quad \uparrow \\
\text{1111000110} & \\
+ \quad \text{0110} & \\
\text{0101000110} &
\end{align*}
\]
Booth’s Algorithm for the Multiplication of $m$ and $m'$ (as Unsigned Numbers, Non-Optimized Version)

- Let $m$, $m'$ be $k$ bit-unsigned numbers. 
- sum is in the following a $2k + 1$ bit-signed number.

```plaintext
sum := 0
for $i = 1$ to $(k + 1)$ do
begin
if $i$th LSB of $m$ is 1 and
($i = 1$ or $(i - 1)$th LSB of $m$ is 0)
then
subtract $m'$ shifted $(i - 1)$ bits left
as a $(2k + 1)$-bit two’s compl. numb.
if $i$th LSB of $m$ is 0 and
$i > 1$ and $(i - 1)$th LSB of $m$ is 1
then
add $m'$ shifted $(i - 1)$ bits left
as a $(2k + 1)$-bit two’s compl. num.
end
sum is result
```

Example for Mult. of Unsigned Numb. with Non-Optim. Booth’s algorithm

<table>
<thead>
<tr>
<th></th>
<th>$m$ (unsigned)</th>
<th>$m'$ (unsigned)</th>
<th>sum (signed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>101</td>
<td>110</td>
<td>0000 000</td>
</tr>
<tr>
<td>Step 1</td>
<td>1st LSB($m$) = 1</td>
<td>$m'$</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>Subtract $m'$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 2</td>
<td>2nd LSB($m$) = 0</td>
<td>1st LSB($m'$) = 1</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>Add $m'$ to sum shifted once</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 3</td>
<td>3rd LSB($m$) = 1</td>
<td>2nd LSB($m'$) = 0</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>Subtract $m'$ shifted twice</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 4</td>
<td>4th LSB($m$) = 0</td>
<td>3rd LSB($m'$) = 1</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>Add $m'$ shifted 3 times</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0b101 \cdot 0b110 = 5 \cdot 6 = 30 = 0b11110$.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(ii) Optimization

- Similar steps as for the naive algorithm:
  - Shift sum to the left, so that subtraction/addition happens only in the $(k + 1)$ MSBs of sum.
  - Therefore in intermediate steps sum has to be shifted to the right.
  - Store $m'$ in the LSBs of sum which are not used.

Optimization (Cont.)

- The following differs from what was done for naive multiplication:
  - We need to consider the LSB of $m'$ and the previous bit.
    - Therefore additional bit $\text{sum}_{-1}$ required, which keeps the previous LSB.
  - We have to carry out the complete procedure $k + 1$ times.
    - Therefore we need to shift $m$ $(k + 1)$ times to the right, and treat an implicit 0 to the left of $m$.
    - This requires one extra bit in sum.
    $\Rightarrow$ Sum needs to be treated as a $(2k + 2)$-bit number.
  - Whenever we subtract, we obtain a sum with a block of ones to the left.
    - A block of ones is represented by a MSB 1.
    - A block of zeros is represented by a MSB 0.
    - Therefore when shifting sum to the right, those hidden zeros and ones turn up.
  - Arithmetic shift required:
    - When shifting sum to the right, the new MSB is the old MSB.
Arithmetic Shift

- **Arithmetic shift** of $x$ to the right means:
  - We shift $x$ to the right one bit.
  - But new MSB is the same as the old MSB.
- Examples:
  
<table>
<thead>
<tr>
<th>Example</th>
<th>$x$</th>
<th>$\text{new MSB}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1 0 1</td>
<td>$\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow$</td>
<td>1 1 0 0 0 1 0</td>
</tr>
<tr>
<td>0 1 0 0 1 0 0</td>
<td>$\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow$</td>
<td>0 0 1 0 0 1</td>
</tr>
</tbody>
</table>

Booth’s Algorithm for the Multiplication of $m$ and $m'$ (as Unsigned Numbers)

Let $m$, $m'$ have $k$ bits.
Sum has in the following $2k + 2$ (not $2k$!!) bits.

- Set the $k$ LSBs of sum to $m$
- $\text{sum}_{-1} := 0$
- For $i = 1$ to $k + 1$
  - begin
    - if LSB($\text{sum}$) = 1 and $\text{sum}_{-1} = 0$
      - subtract $m'$ as a $(k + 1)$-bit number from the $k + 1$ MSBs of $\text{sum}$ and ignore a borrow
    - if LSB($\text{sum}$) = 0 and $\text{sum}_{-1} = 1$
      - add $m'$ as a $(k + 1)$-bit number to the $k + 1$ MSBs of $\text{sum}$ and ignore a carry
  - $\text{sum}_{-1} := \text{LSB}(\text{sum})$
  - Arithmetic shift right of sum by one bit
  - end
- $\text{sum}$ is result

Two examples follow.

<table>
<thead>
<tr>
<th>$m$ (un-)</th>
<th>$m'$ (un-)</th>
<th>$\text{sum}$ (signed)</th>
<th>$\text{sum}_{-1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>011 101</td>
<td>0000 0111</td>
<td>0011 101</td>
<td>0</td>
</tr>
<tr>
<td>Initialization</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 1a</td>
<td>LSB($\text{sum}$) = 1</td>
<td>sum$_{-1}$ = 0</td>
<td>Subtract $m'$</td>
</tr>
<tr>
<td>Step 1b</td>
<td>Arithmetic shift</td>
<td>011 101</td>
<td>1101 1100 1</td>
</tr>
<tr>
<td>Step 2a</td>
<td>LSB($\text{sum}$) = sum$_{-1}$</td>
<td>No action</td>
<td>011 101</td>
</tr>
<tr>
<td>Step 2b</td>
<td>Arithmetic shift</td>
<td>011 101</td>
<td>1110 1100 1</td>
</tr>
<tr>
<td>Step 3a</td>
<td>LSB($\text{sum}$) = 0</td>
<td>sum$_{-1}$ = 1</td>
<td>add $m'$</td>
</tr>
<tr>
<td>Step 3b</td>
<td>Arithmetic shift</td>
<td>011 101</td>
<td>0001 1110 0</td>
</tr>
<tr>
<td>Step 4a</td>
<td>LSB($\text{sum}$) = sum$_{-1}$</td>
<td>No action</td>
<td>011 101</td>
</tr>
<tr>
<td>Step 4b</td>
<td>Arithmetic shift</td>
<td>011 101</td>
<td>0000 1111 0</td>
</tr>
</tbody>
</table>

Remark on Previous Slide

- Subtraction of 0b101 on previous slide can be done using the naive algorithm (**but ignore the borrow!!!**).
- Or instead of subtracting 0b101, add -0b101 as a 4-bit number, which is (1011)_{signed}.
  - Bitwise complement of 0101 is 1010.
  - Adding one yields 1011.
(ii) Multiplication of Signed Numbers

If $m'$ is now a signed number (but $m$ unsigned), the algorithm works as before.
- Except that sum can now be a $(2k + 1)$-bit number.
- Carry can be ignored.

Booth’s Algorithm for the Multiplication of Signed Numbers $m$ and $m'$

Let $m$, $m'$ be $k$-bit signed numbers. Let sum be a $2k$-bit signed number.

\[
\text{sum} := 0
\]

Set the $k$ LSBs of sum to $m$

\[
\text{sum}_{-1} := 0
\]

for $i = 1$ to $k$ do

begin

if LSB(sum) = 1 and sum$_{-1} = 0$ then

subtract $m'$ as a $k$-bit signed number from the $k$ MSBs of sum and ignore any borrow

if LSB(sum) = 0 and sum$_{-1} = 1$ then

add $m'$ as a $k$-bit signed number to the $k$ MSBs of sum and ignore any carry

sum$_{-1} := \text{LSB}(\text{sum})$

Arithmetic shift right of sum by one bit

end

sum is result
Example for Multiplication of Signed Numbers with Booth’s algorithm

<table>
<thead>
<tr>
<th>Initialization</th>
<th>$m$</th>
<th>$m'$</th>
<th>sum</th>
<th>sum $_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000</td>
<td>101</td>
<td>-110</td>
<td>001 010</td>
</tr>
<tr>
<td>Step 1a</td>
<td>101</td>
<td>110</td>
<td>0</td>
<td>101</td>
</tr>
<tr>
<td>Step 1b</td>
<td>101</td>
<td>110</td>
<td>001 010</td>
<td>1</td>
</tr>
<tr>
<td>Step 2a</td>
<td>101</td>
<td>110</td>
<td>+110</td>
<td>111 101</td>
</tr>
<tr>
<td>Step 2b</td>
<td>101</td>
<td>110</td>
<td>000 110</td>
<td>1</td>
</tr>
<tr>
<td>Step 3a</td>
<td>101</td>
<td>110</td>
<td>-110</td>
<td>001 010</td>
</tr>
<tr>
<td>Step 3b</td>
<td>101</td>
<td>110</td>
<td>000 110</td>
<td>1</td>
</tr>
</tbody>
</table>

Veriﬁcation that result on last slide is correct:
- 101 represents $-4 + 1 = -3$.
- 110 represents $-4 + 2 = -2$.
- $(-3) \cdot (-2) = 6$ is represented by 000110 (with 6 bits).
  Note that subtraction of 110 means addition of 010.

(d) Fixed Point Numbers

- In decimal representation, real numbers can be written as numbers with potentially inﬁnite digits after the point:
  - $\pi = 3.1415926 \ldots$

Analysis:
- All digits get a weight in terms of powers of 10.
- Left to the point, this is a positive power of 10,
- Right of it a negative one:

$$\sqrt{\Pi} = 1 0 . 5 3 5 5 \ldots$$

$$\begin{array}{cccccc}
10^1 & 10^0 & 10^{-1} & 10^{-2} & 10^{-3} & 10^{-4} \\
= & = & = & = & = & = \\
10 & \frac{1}{10} & \frac{1}{100} & \frac{1}{1000} & \frac{1}{10000} & \\
\end{array}$$

$$= 1 \cdot 10^1 + 0 \cdot 10^0 + 5 \cdot 10^{-1} + 3 \cdot 10^{-2} + 5 \cdot 10^{-3} + 5 \cdot 10^{-4} + \ldots$$

- Fixed point numbers:
  - Round or cut off after a ﬁxed number of digits after the point.

Binary Fixed Point Numbers

- With basis 2, similar representation:

$$\begin{array}{cccccc}
0b & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 2^1 & 2^0 & 2^{-1} & 2^{-2} & 2^{-3} & 2^{-4} \\
= & = & = & = & = & = & = \\
2 & 1 & \frac{1}{2} & \frac{1}{4} & \frac{1}{8} & \frac{1}{16} & \\
\end{array}$$

$$= 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} + \ldots$$

$$= 1 \cdot 2 + 0 \cdot 1 + 1 \cdot 0.5 + 0 \cdot 0.25 + 1 \cdot 0.125 + 0 \cdot 0.0625 + \ldots$$

$$= 2.6875$$

- With the above technique we can convert from binary to decimal.
Conversion from Decimal to Binary

- Convert the whole-numbered part (the part before the point) into a binary number.
  - E.g. whole-numbered part of 13.25 is 13 = 0b1101

- Convert the fractional part (the part after the point)
  - E.g. fractional part of 13.25 is 0.25 = 0b0.01.

into the fractional part of a binary number with \( k \) bits after the point as follows:

- Multiply the number by 2.
- If the result is \( \geq 1 \), the next bit is 1, subtract 1 from the result.
- If the result is \( < 1 \), the next bit is 0
- Iterate this \( k \) times.

\[ 13 = 0b1101 \]

Example
We write 10.5355 as a fixed point binary number. (This is not \( \sqrt{111} \), since we have rounded this number already).

\[ 10 = 0b1010. \]

<table>
<thead>
<tr>
<th>Fraction</th>
<th>Digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5355</td>
<td>1</td>
</tr>
<tr>
<td>0.0710</td>
<td></td>
</tr>
<tr>
<td>0.1420</td>
<td></td>
</tr>
<tr>
<td>0.2840</td>
<td></td>
</tr>
<tr>
<td>0.5680</td>
<td></td>
</tr>
<tr>
<td>1.1360</td>
<td>1</td>
</tr>
<tr>
<td>0.1360</td>
<td></td>
</tr>
<tr>
<td>0.2320</td>
<td></td>
</tr>
<tr>
<td>0.0001</td>
<td></td>
</tr>
</tbody>
</table>

with 6 bits after the point.

Remark: Note that the first bit obtained is the first bit after the point, the next one, the second etc.

Conversion from Decimal to Binary (Cont.)

- More formally, the following algorithm converts the fractional part

\[ (k = \text{number of bits after point required}). \]

\[ x := \text{fractional part of the number} \]

for \( i = 1 \) to \( k \) do

\[ \text{begin} \]

\[ x := x \cdot 2 \]

\[ \text{if } x < 1 \text{ then } k\text{th bit after point of the result is 0} \]

\[ \text{else } k\text{th bit after point of the result is 1} \]

\[ x := x - 1 \]

\[ \text{end} \]

Remark
- The full (not-fixed-point) binary representation of the decimal number 0.2 is 0.00110011001100110011...

- So numbers might have exact representation as a decimal fixed point number, but not as a binary fixed point number.

- Consequence: Rounding errors occur.

    A calculation with decimal fixed-point representation and with binary fixed-point representation might have different results.

- Important when working with financial data.

- However: There is no optimal number system. Fractions \( \frac{a}{b} \) can be represented exactly:

  - In binary if \( b \) is of the form \( 2^n \).
  - In decimal if \( b \) is of the form \( 2^n5^m \).
  - In a number system with basis 30, if \( b \) is of the form \( 2^n3^m5^k \).
  - etc., there we can always get a better one.
Facts about Shifting Fixed Point Numbers

Fact

Shifting the point in a fixed point number once to the right is the same as multiplying the denoted number by $2$.
Shifting to the right is the same as dividing the denoted number by $2$.

- **Examples:**
  - $\text{0b10.01} = 2^2 + 2^{-2}$.
  - $\text{0b100.1} = 2^3 + 2^{-1} = (2^2 + 2^{-2}) \cdot 2$.
  - $\text{0b10.01} = 2^2 + 2^{-2}$.
  - $\text{0b1.001} = 2^1 + 2^{-3} = (2^2 + 2^{-2})/2$.

Proof:

- Let $y$ be the original fixed point number, $z$ the number shifted once to the right.
- If a bit in $y$ has weight $2^l$, where $l$ is:
  - positive, if occurring left of the point,
  - negative, if occurring right of the point
then the weight of the corresponding bit in $z$ is $2^{l+1}$.
- $0by$ is the sum of the weight of ones in $y$.
- $0bz$ the sum of the weight of ones in $z$.
which is the sum of twice the weight of the ones in $z$, which is 2 times $0bz$.
- Shifting a number to the left is just the inverse of the previous operation, resulting in dividing it by 2.

Operations on Fixed Point Numbers

**General Method:**

- Write the operands as fixed point numbers with the same number of bits to the right of the point.
- Perform the operation as for integers.
- Now reintroduce the point in the result:
  - **Case addition and subtraction:**
    - If it initially was after the $k$th bit from the right, it should be in the result after the $k$th bit from the right.
  - **Case multiplication:**
    - If it initially was after the $k$th bit from the right, it should be in the result after the $2k$-th bit from the right.
At the end one might omit

- leading zeros
  - e.g. convert 010.01 into 10.01;
- zeros as least significant bits after the point
  - e.g. convert 0.010 into 0.01.

### Examples (all in binary)

#### Fixed Point Numbers

<table>
<thead>
<tr>
<th>Fixed Point Numbers</th>
<th>Corresponding Integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.01 + 0.01 = 1.10:</td>
<td>101 + 001 = 110:</td>
</tr>
<tr>
<td>1.01</td>
<td>101</td>
</tr>
<tr>
<td>0.01</td>
<td>001</td>
</tr>
<tr>
<td>1.10</td>
<td>110</td>
</tr>
</tbody>
</table>

#### Example with overflow in the addition

10.01 + 1.11 = 100:
1001 + 0111 = 10000:

| 10.01 | 1001 |
| 1.11  | 111  |
| 100.00| 10000|

10 - 1.11 = 0.01:
1000 - 111 = 1:

| 10.00 | 1000 |
| 1.11  | 111  |
| 00.01 | 0001 |

1.01 · 1.11 = 10.0011:
101 · 111 = 100011:

| 1.01 | 101 |
| .101 | 101 |
| 101  | 101 |
| 10.0011| 100011|

### Correctness of the Algorithms

- Case addition:
  - Sum of \( n, n' \) is \( 2^k \cdot (m + m') \).
  - Reintroducing the point after \( k \)-bits means division by \( 2^k \).
  - Result is \( m + m' \), the correct result.

- Case subtraction: Similarly.

- Case multiplication:
  - Product of \( n, n' \) is \( 2^k \cdot 2^k \cdot m \cdot m' = 2^{2k} \cdot m \cdot m' \).
  - Reintroducing the point after \( 2k \) bits means division by \( 2^{2k} \).
  - Result is \( m \cdot m' \), the correct result.
(e) Floating-Point Numbers

- **In decimal representation** numbers represented like
  \[ 0.23578 \cdot 10^5, -0.99998 \cdot 10^{-5}, 0.0 \cdot 10^0. \]

- In general we obtain the form
  \[ s \cdot a \cdot 10^k, \]
  where
  - \( s = +1 \) or \( s = -1 \); \( s \) is called the **sign**.
  - \( a \) is a fixed point number (in decimal representation) s.t. \( 0.1 \leq a < 1 \) or \( a = 0 \);
    \( a \) is called **significand** or **mantissa**.
  - \( k \) is a signed integer called the **exponent**.

Note that
\[ 0.349 \cdot 10^8 = 34900000 \]

- So \( 10^{\text{exponent}} \) (in the example \( 10^8 \)) is **10 times the weight** of the digit in decimal representation most to the left that is not equal to 0 (in the example \( 10^7 \)).

- The significand is formed by the sequence of **digits** starting with this digit (in the example 349).

---

Binary Floating-Point Numbers

- Similarly, only basis 2:
  We write numbers as
  \[ s \cdot a \cdot 2^e \]
  - where \( s \) is the sign as before,
  - \( 0 \leq a < 1 \) or \( a = 0 \)

- If \( a \) is chosen as above, we call this a **normalized floating-point representation**.

- If we omit the condition on \( a \), \( s \cdot a \cdot 2^3 \) is called a **non-normalized floating-point representation** of this number.

- There are several **non-normalized representations** of the same number, e.g.
  - \( 0b0.1 \cdot 2^5 \)
  - \( 0b0.01 \cdot 2^6 \)
  - \( 0b1 \cdot 2^4 \)
  represent the same number, of which the first one is the normalized representation.

---

Omission of Leading Ones

- If \( a \neq 0 \) then \( a \) is always of the form \( 0b0.1xyz \cdots \), with \( x, y, z \in \{0,1\} \).
  - So the first 1 has not to be stored, except for \( a = 0 \).
Biased Exponents

Instead of storing positive and negative exponents as signed integers, one does the following:

- We add to the exponent a fixed number \( b > 0 \).

- The result is called the **biased exponent**.
  * If the biased exponent is positive, the exponent is represented by the biased exponent as an **unsigned** number.
    The resulting bit sequence **encodes** the original unbiased exponent.
  * Otherwise the exponent cannot be represented.

- So a biased exponent of \( a \) represents (encodes) the exponent \( a - b \).

- With \( k \) bits we can represent unsigned numbers and therefore biased exponents in the range \( 0, \ldots, 2^k - 1 \).

  * Therefore we can represent unbiased exponents in the range \( -b, \ldots, -b + (2^k - 1) \).
    - If \( k = 8 \), \( 2^k - 1 = 255 \), and a good choice of \( b \) is 127 or 128.
      (For historic reasons, 126 is used).

The IEEE 754 Floating-Point Standard

We consider only the single format with 32 bits, there exists as well a double format with 64 bits.

Floating-points are written in the form

<table>
<thead>
<tr>
<th>Bit positions</th>
<th>Function</th>
<th>0</th>
<th>1 – 8</th>
<th>9 – 31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sign</td>
<td>biased exponent</td>
<td>significand</td>
<td></td>
</tr>
</tbody>
</table>

- Sign 0 means a positive, 1 a negative number (so \( s \in \{ 0, 1 \} \) represents sign \( (-1)^s \)).

- The exponent is **biased with bias 126**, and represented by **8 bits**.
  - So exponents in the range \( -126, \ldots, 129 \) could be represented.
  - However exponent 129 used for **infinity (\( \infty \)) and NaN**, see below, giving an effective range of \( -126, \ldots, 128 \) for the exponent.
  - If the exponent is \( > -126 \), the significand is interpreted as having an additional leading 1, so the effective significand has 24 bits, not 23 bits.

- Let \( k = 8 \), \( 2^k = 256 \).

- If we fix the bias 126, then we have the following representations:

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Biased Exponent</th>
<th>Binary Representation (with 8 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-100</td>
<td>26</td>
<td>00011010</td>
</tr>
<tr>
<td>0</td>
<td>126</td>
<td>01111110</td>
</tr>
<tr>
<td>10</td>
<td>136</td>
<td>10001000</td>
</tr>
<tr>
<td>-126</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>129</td>
<td>255</td>
<td>11111111</td>
</tr>
</tbody>
</table>

So the bit sequence 00011010 is a code for the exponent -100.
With 8 bits we can represent exponents in the range \( -126, \ldots, 129 \).

Exponent -126

- If the exponent is \( -126 \) (so the biased exponent is 0), a leading 1 is **not implied**.
  - Especially significand 0 and biased exponent 0 means value 0.
    - Because of the sign bit, **two representations of zero** \( (+0, -0) \) exist.
  - This allows to represent numbers which in normalized form have exponent less than \( -126 \) as **non-normalized numbers** with exponent \( -126 \).
    - E.g. \( 0 \times 10^{-128} = 0 \times 0.001 \times 2^{-126} \).
  - Therefore the area of numbers close to 0 is represented in uniformly
Therefore there is no gap between 0 and $0b0.1 \cdot 2^{-126}$, the smallest normalized number greater > 0 with exponent $-126$:

\[
\begin{array}{c|c|c}
\text{Using this approach} & 0b0.1 \cdot 2^{-126} \\
\text{non-normalized numbers} & 0b1.2^{-126} \\
\text{Without it} & 0b1.2^{-126}
\end{array}
\]

Without this approach, the smallest number > 0 we can represent is $0b0.1 \cdot 2^{-126} = 2^{-127}$. In IEEE 754, it is $0b0.000000000000000000000001 \cdot 2^{-126} = 2^{-149}$.

**Infinity and NaN**

- Exponent of all ones (0b11111111) with significand 0 expresses positive or negative infinity, depending on the sign (denoted by $+\infty$, $-\infty$). Used when overflow occurs:
  - If $n > 0$ then $n/0 = +\infty$.
  - If $n < 0$ then $n/0 = -\infty$.
  - If $n > 0$ then $n \cdot (+\infty) = +\infty$.
  - If $n < 0$ then $n \cdot (\infty) = +\infty$.
- If $n, m > 0$, $n \cdot m$ cannot be represented, the result is $+\infty$.

- Exponent of all ones and non-zero significand represents NaN, not a number, for undefined (error, which cannot be associated to positive or negative infinity). Examples:
  - $0/0$ yields NaN.
  - $(+\infty) + (\infty)$ yields NaN.
  - But $(+\infty) + (+\infty)$ yields $+\infty$.

**Remarks on the Bias in the IEEE Floating-Point Standard**

- In the literature, one often sees the bias 127 instead of 126 as on the last slides.
- The reason for this difference is that they expand significand $f$ to $1.f$ instead of $0.1f$ as we did it.
Examples

\[ -0.75 = (-1) \cdot 0.6011 \cdot 2^0 \]
\[ = (-1) \cdot 0.6011 \cdot 2^{-126+126} \]
\[ = (-1) \cdot 0.6011 \cdot 2^{-3+0} \]

is represented by

(first 1 of the significand omitted!)

\[ \underbrace{0111111100000000000000000000000}_{\text{sign exponent}} \overbrace{10000000000000000000000}_{\text{significand}} \]

0.4375 = 0.6011  \cdot 2^1
\[ = 0.6011 \cdot 2^{-126+126} \]
\[ = 0.6011 \cdot 2^{-3+0} \]

is represented by

(first 1 of the significand omitted!)

\[ \underbrace{0111110111000000000000000000000}_{\text{sign exponent}} \overbrace{10000000000000000000000}_{\text{significand}} \]

Arithmetic Operations on Floating-Point Numbers

- **Simplification here:**
  - We use here the unbiased (original) exponents.
  - We don’t omit in the representation of the significand a leading 1.

- **Addition:**
  - Represent both numbers in the form \( s \cdot a \cdot 2^k \),
    where \( k \) is the maximum of the exponents of the normalized form of both numbers.
  - Let the **signed significand** be \( s \cdot a \).
  - Add the two signed significands.
  - Write the result as \( s \cdot a \) where \( s \in \{-1, 1\} \), \( a \) is positive.
  - A non-normalized form of the result is \( s \cdot a \cdot 2^k \).
  - Normalize the result, i.e. write it as \( s \cdot a' \cdot 2^{k'} \)
    s.t. this representation is in normalized form.

\[ 0.5 \cdot 2^{-126} = 0.601 \cdot 2^{-126+0} \]

is represented by

(first 1 of the significand not omitted!)

\[ \underbrace{00000000}_{\text{sign}} \overbrace{10000000000000000000000}_{\text{significand}} \]

\[ \begin{align*}
0.25 \cdot 2^{-126} &= 0.6001 \cdot 2^{-126+0} \\
&= \underbrace{00000000}_{\text{sign}} \overbrace{01000000000000000000000}_{\text{significand}}
\end{align*} \]

\[ 0 \text{ is represented by} \]

\[ \underbrace{00000000}_{\text{sign}} \overbrace{00000000000000000000000}_{\text{significand}} \]

\[ \begin{align*}
\text{and} \quad &\underbrace{00000000}_{\text{sign}} \overbrace{00000000000000000000000}_{\text{significand}} \\
\text{Infinity is represented by} \quad &\underbrace{00000000}_{\text{sign}} \overbrace{11111111000000000000000000000000}_{\text{significand}} \quad \text{is one (of many) representation of NaN.}
\end{align*} \]
Examples (Cont.)

\[ (-1) \cdot 0b0.1011 \cdot 2^{06110} + (-1) \cdot 0b0.101 \cdot 2^{06101} = (-1) \cdot 0b0.1011 \cdot 2^{06110} + (-1) \cdot 0b0.0101 \cdot 2^{06110} = (-0b0.1011 - 0b0.0101) \cdot 2^{06110} = -0b1.0 \cdot 2^{06110} = -0b0.1 \cdot 2^{06111} \]

Subtraction of Floating-Point Numbers

- **Subtraction** is carried out similarly, carry out subtraction instead of addition for the signed significands.

Multiplication:

\[
(s_0 \cdot a_0 \cdot 2^{k_0}) \cdot (s_1 \cdot a_1 \cdot 2^{k_1}) = (s_0 \cdot s_1) \cdot (a_0 \cdot a_1) \cdot 2^{k_0+k_1}
\]

- Therefore the **sign** of the result is \( s_0 \) XOR \( s_1 \).

- The **significand** of the result is the product of the significands.

- The **exponent** is the sum of the exponents.
  
  - (when working with biased exponents, one would add both biased exponents and subtract from the sum the bias).

- If one does not obtain a number in normal form, adapt the result.

Examples:

\[
(0b0.1 \cdot 2^{0610}) \cdot (0b0.1 \cdot 2^{0610}) = 0b0.01 \cdot 2^{0610} = 0b0.1 \cdot 2^{0610}
\]

\[
((-1) \cdot 0b0.11 \cdot 2^{0610}) \cdot ((-1) \cdot 0b0.11 \cdot 2^{0610}) = 0b0.1001 \cdot 2^{06100}.
\]

(f) Other Scalar Data Types

Characters, Texts

**Standard Representation of Characters:** ASCII, uses 7 bits which represent numbers between 0 and 127 to represent on character. Table:

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Char</th>
<th>ASCII</th>
<th>Char</th>
<th>ASCII</th>
<th>Char</th>
<th>ASCII</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>16</td>
<td>DLE</td>
<td>32</td>
<td>SP</td>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
<td>17</td>
<td>DC1</td>
<td>33</td>
<td>!</td>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
<td>18</td>
<td>DC2</td>
<td>34</td>
<td>&quot;</td>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>ETX</td>
<td>19</td>
<td>DC3</td>
<td>35</td>
<td>#</td>
<td>51</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
<td>20</td>
<td>DC4</td>
<td>36</td>
<td>$</td>
<td>52</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
<td>21</td>
<td>NAK</td>
<td>37</td>
<td>%</td>
<td>53</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
<td>22</td>
<td>SYN</td>
<td>38</td>
<td>&amp;</td>
<td>54</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
<td>23</td>
<td>ETB</td>
<td>39</td>
<td>'</td>
<td>55</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
<td>24</td>
<td>CAN</td>
<td>40</td>
<td>(</td>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>HT</td>
<td>25</td>
<td>EM</td>
<td>41</td>
<td>)</td>
<td>57</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>LF</td>
<td>26</td>
<td>SUB</td>
<td>42</td>
<td>*</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>VT</td>
<td>27</td>
<td>ESC</td>
<td>43</td>
<td>+</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>FF</td>
<td>28</td>
<td>FS</td>
<td>44</td>
<td>.</td>
<td>60</td>
<td>&lt;</td>
</tr>
<tr>
<td>13</td>
<td>CR</td>
<td>29</td>
<td>GS</td>
<td>45</td>
<td>-</td>
<td>61</td>
<td>=</td>
</tr>
<tr>
<td>14</td>
<td>SO</td>
<td>30</td>
<td>RS</td>
<td>46</td>
<td>.</td>
<td>62</td>
<td>&gt;</td>
</tr>
<tr>
<td>15</td>
<td>SI</td>
<td>31</td>
<td>US</td>
<td>47</td>
<td>/</td>
<td>63</td>
<td>?</td>
</tr>
</tbody>
</table>
IBM has extended ASCII (extended ASCII) by adding 128 characters, especially symbols occurring in other European languages, as well some mathematical symbols.

- In total therefore 8 bits (= 1 byte) are needed to encode them.
- Characters 0 - 127 are identically with ASCII and 128 - 255 are new.
- Unfortunately variants of this set of characters exist, partially due to the fact that IBM’s set is not allowed to be used on non-IBM-licensed PCs.
- Characters 128 - 255 are not standardized.
- Some characters are control symbol, which cannot be displayed or printed (eg. ASCII 7 is an acoustic signal, “bell”; ASCII 10 is a line feed).

- **Unicode** (allows to represent international characters including Chinese, Japanese and Korean, with 16 bits.)
  - Standard in Java, XML.
  - Problem: requires twice as much space.
  - Variants exist, which encode the standard Western alphabet using 1 bytes, non-Western alphabets using 2 bytes, and control characters to switch between 1-byte and 2-byte character encodings.
  - Unicode still doesn’t accomodate all existing alphabets (especially not the full range of Chinese/Korean/Japanese-Chinese characters).
- **ISO/IEC 10646-1** (Universal Multiple-Octet Coded Character Set): system, which uses 32 bit encodings (although only 31 bits are effectively used).
  - Allows to accomodate for all alphabets.
- Lots of other encodings exist.

---

Other Character Encodings

- **Unicode** (allows to represent international characters including Chinese, Japanese and Korean, with 16 bits.)
  - Standard in Java, XML.
  - Problem: requires twice as much space.
  - Variants exist, which encode the standard Western alphabet using 1 bytes, non-Western alphabets using 2 bytes, and control characters to switch between 1-byte and 2-byte character encodings.
  - Unicode still doesn’t accomodate all existing alphabets (especially not the full range of Chinese/Korean/Japanese-Chinese characters).
- **ISO/IEC 10646-1** (Universal Multiple-Octet Coded Character Set): system, which uses 32 bit encodings (although only 31 bits are effectively used).
  - Allows to accomodate for all alphabets.
- Lots of other encodings exist.

---

Representation of Texts

- Text represented as sequences of characters.
- **Advanced formats**: Programming languages for storing text with formatting information (postscript, pdf, TeX, RTF, Word, Openoffice XML-format).
- Various compressing algorithms in order to save space.

Representation of Graphics and Multimedia

- Graphics stored as bit maps for representing pixel with colours.
  - Graphics interchange format, **gif**, with some compression.
- Music, video data in various formats.
**Compression of Multimedia**

- **Problem when storing multi-media:** data require a lot of space. Good compression algorithms needed.
- **Use of the fact that small changes between consecutive images, sounds.**
- **Some compression algorithm lead to loss of data (lossy algorithms).**
- **For instance omission of invisible colour changes in areas where texture is particularly vivid.**
- **Examples:** JPEG (images), MPEG (video, audio; computationally complex to code and decode), Quicktime (video).
- **MPEG-2 achieves compression rates of more than 100:1.**

**Compound Data Types**

- **Compound data types are data types, where each unit consists of several units.**
- **Main examples:**
  - Arrays.
  - Strings.
  - Records.
  - Classes (not treated in this lecture).
- **Main ideas:** Store data items in sequence.
- **Strings stored as arrays of characters.**

**Storage of Records**

- **Records are stored by saving the elements in a sequence:**
  - **Example:**
    ```
    studentMark= record
    studentID: integer
    mark: double
    end
    ```
  - Stored as one integer followed by one double precision floating-point number.
    E.g., if an integer is stored as 4 bytes, a double-precision floating-point number is stored as 8 bytes, and addresses referring to byte level:
    * Addresses 0x0000 to 0x0003 contain studentID,
    * 0x0004 to 0x000B contain mark,
    * Each entry requires 12 bytes.

**Storage of Arrays**

- **Assume a:array[0...n] of int and int requires 4 bytes.**
- **a** can be stored by storing **a[0], a[1], a[2]** etc. in sequence.
- **Assume that memory addresses refer to bytes, and that a[0] is to be stored at location 0x1000.**
- **Then we store**
  - **a[0] at addresses 0x1000 – 0x1003.**
  - **a[1] at addresses 0x1004 – 0x1007.**
  - **a[2] at addresses 0x1008 – 0x100B.**
  - **a[3] at addresses 0x100C – 0x100F.**
  - **a[i] at addresses 0x1000 + 4 · i to 0x1003 + 4 · i.**
Two-Dimensional Arrays

- Assume
  - $a$: array[0..199,0..99] of int,
  - int stored as 4 bytes,
  - addresses up to byte level,
  - a stored beginning with memory location $A$.
- Then the address of the first byte
  - of $a[0,0]$ is $A$,
  - of $a[0,1]$ is $A + 4$, etc.
  - of $a[0,k]$ is $A + 4 \cdot k$,
  - of $a[0,99]$ is $A + 4 \cdot 99$,
  - of $a[1,0]$ is $A + 4 \cdot 100$,
  - of $a[1,1]$ is $A + 4 \cdot 100 + 4$,
  - of $a[2,0]$ is $A + 2 \cdot 4 \cdot 100$,
  - of $a[2,1]$ is $A + 2 \cdot 4 \cdot 100 + 4$,
  - of $a[1,0]$ is $A + i \cdot 4 \cdot 100$ and
  - of $a[1,j]$ is $A + i \cdot 4 \cdot 100 + j \cdot 4$.

So the picture is like this
(elem = element of the array, addr = address):

<table>
<thead>
<tr>
<th>elem.</th>
<th>addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a[0,0]$</td>
<td>$A$</td>
</tr>
<tr>
<td>$a[0,1]$</td>
<td>$A + 4$</td>
</tr>
<tr>
<td>$a[i]$</td>
<td>$4i$</td>
</tr>
<tr>
<td>$a[0,99]$</td>
<td>$396$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>elem.</th>
<th>addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a[1,0]$</td>
<td>$400$</td>
</tr>
<tr>
<td>$a[1,1]$</td>
<td>$400 + 4i$</td>
</tr>
<tr>
<td>$a[1,i]$</td>
<td>$396$</td>
</tr>
<tr>
<td>$a[1,99]$</td>
<td>$396$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>elem.</th>
<th>addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a[j,0]$</td>
<td>$400j$</td>
</tr>
<tr>
<td>$a[j,1]$</td>
<td>$400j + 4i$</td>
</tr>
<tr>
<td>$a[j,i]$</td>
<td>$396$</td>
</tr>
<tr>
<td>$a[j,99]$</td>
<td>$396$</td>
</tr>
</tbody>
</table>

Exercise: What is the address of $a[i,j]$, if
- $a$: array[k..l, m..n] of B,
- each element of B requires $f$ bytes
- $a[k,m]$ is stored at address $A$?
Supplementary Material for Sect. 4.

(a) Correctness proofs of algorithms on signed numbers
(b) Signed fixed point numbers.

Correctness of the Algorithm for Negating Numbers in Two’s Complement Form

- First observe: forming the bitwise complement of a $k$-bit unsigned number is the same as subtracting it from $0b1\cdots1 = 2^k - 1$:
  - Example:
    
    $\begin{array}{c}
    \text{0b1111 - 0b1010} = \\
    \text{0b 1 1 1 1} \\
    \text{− 0b 1 0 1 0} \\
    \text{0b 0 1 0 1}
    \end{array}$

Correctness of the Algorithm (Cont.)

- **Case 1**: $m > 0$.
  - Then $z = 0x$ where $x$ is a $k - 1$-bit number, $m = (z)_{\text{signed}} = 0bx = 0bz$.
  - Bitwise complement of $z$ represents the unsigned number $(2^k - 1) - 0bz = 2^k - 1 - m$.
  - Adding one yields an $y$ s.t. $0by = 2^k - 1 - m + 1 = 2^k - m$.
  - That number has MSB 1, so it is negative as a signed number.
  - By the lemma above $(y)_{\text{signed}} = 0by - 2^k = (2^k - m) - 2^k = -m$.

Correctness of the Algorithm (Cont.)

- **Case 2**: $m = 0$.
  - Then $x = 0\cdots0$. Bitwise complement of $x$ is $1\cdots1$. Adding one and ignoring carry yields $0\cdots0$, which is as signed number 0 or $-m$. 


Correctness of the Algorithm (Cont.) (Cont.)

- **Case 3:** $m < 0$.
  - Then $z = 1x$ where $x$ is a $(k - 1)$-bit number, and $m = -2^{k-1} + 0bx$.
  - Bitwise complement of $1x$ is $0y$ where $y$ is the bitwise complement of $x$.
    * $(0y)_{\text{signed}} = 0by$
    * If $x = \overline{0\cdots 0}$, then $y = \overline{1\cdots 1}$ and adding one to $0by$ yields $1\overline{0\cdots 0}$ which is not $-m$.
    $(-m = 2^{k-1}$ is out of the range of representable numbers and can therefore not be represented.)
    * Otherwise, adding one to $0y$ yields a number $0u$ and we have
      
      \[
      (0u)_{\text{signed}} = \begin{cases} 
      0bu & \text{if } y = 0\overline{0\cdots 0} \\
      (2^{k-1} - 1 - 0bx + 1 = 2^{k-1} - 0bx = 2^{k-1} - (m + 2^{k-1}) \\
      = -m & \text{if } y = 1\overline{0\cdots 0} 
      \end{cases}
      \]

Some Facts about Shifting Numbers in Two’s Complement Form

- **Fact**
  (The following facts are only intended for those interested in proofs).

- **Examples:**
  - $(\overline{011})_{\text{signed}} = 2^1 + 2^0$.
  - $(\overline{0110})_{\text{signed}} = 2^3 + 2^0 = 2^2(2^1 + 2^0)$.
  - $(\overline{101})_{\text{signed}} = -2^2 + 2^0$.
  - $(\overline{1010})_{\text{signed}} = -2^3 + 2^1 = 2(-2^2 + 2^0)$.

Some Facts about Shifting Numbers in Two’s Complement Form

- **Proof:**
  A bit in $y \overline{0\cdots 0}$ has $2^l$ times the weight in $y$ as signed numbers:
  - If it is the MSB, it has weight $-2^{k-1}$ in $y$, and weight $-2^{k-1+1} = -2^{k-1} \cdot 2^l$ in $y \overline{0\cdots 0}$.
  - Otherwise, it has weight $2^l$ in $y$, and weight $2^{l+1} = 2^l \overline{2\cdots 2}$ in $y \overline{0\cdots 0}$.
  - The $l$ LSBs don’t contribute to the value of $0by \overline{0\cdots 0}$.
  - So $(y \overline{0\cdots 0})_{\text{signed}}$ is the sum of the weight of the ones in $y$, each multiplied by $2^l$, which is $(y)_{\text{signed}} \cdot 2^l$.
Some Facts about Shifting Numbers in Two’s Complement Form

- **Proof**:
  - \((yz)_{\text{signed}} = (y0\cdots0)_{\text{signed}} + 0bz\).
  - \((y0\cdots0)_{\text{signed}} = (y)_\text{signed} \cdot 2^t\).
  - \(0bz < 2^t\) (follows as in the proofs for unsigned numbers above).
  - So \((yz)_{\text{signed}} = (y)_{\text{signed}} \cdot 2^t + 0bz\) with \(0 \leq 0bz < 2^t\).
  - This means that \((y)_{\text{signed}} = (yz)_{\text{signed}} \div 2^t, 0bz\) is the remainder.

Correctness of the Addition Algorithm (Cont.)

- **Case** sum of \(x, y\) as unsigned \((k-1)\)-bit numbers gives no overflow.
  - Let sum be \(z\).
  - Then sum of \(0x\) and \(0y\) as signed numbers is \(0z\):
    - \(0x\)
    - \(+0y\)
    - \(0z\)
  - This is the sum of \(0x\) and \(0y\).
  - The algorithm computes the correct result.
Correctness of the Addition Algorithm (Cont.)

- **Case** sum of \(x, y\) as unsigned \((k-1)\)-bit numbers has an overflow.
  - Then sum is of the form \(z\) (\(z\) is a \((k-1)\)-bit number) plus carry.
  - Since
    - \(z\) ignores this carry,
    - carry would have weight \(2^{k-1}\),
  - it follows that
    \[
    0bz = 2^{k-1} + m + m' - 2^{k-1}.
    \]
  - Sum of \(1x, 1y\) as unsigned number is \(1z\) plus carry:
    \[
    \begin{array}{c}
    1x \\
    + 1y \\
    \hline
    1z
    \end{array}
    \]
  - \((1z)_{\text{signed}} = -2^{k-1} + (2^{k-1} + m + m')\).
  - Algorithm gives result \(1z\) (carry ignored), the correct result.

Correctness of the Addition Algorithm (Cont.)

- **Case** sum of \(x, y\) as unsigned \((k-1)\)-bit numbers has no overflow.
  - Then sum is of the form \(z\) (\(z\) is a \((k-1)\)-bit number) plus carry.
  - Since
    - \(z\) ignores this carry,
    - carry would have weight \(2^{k-1}\),
  - it follows that
    \[
    0bz = 2^{k-1} + m + m' + m' - 2^{k-1}.
    \]
  - Sum of \(1x, 0y\) as unsigned number is \(0z\) plus carry:
    \[
    \begin{array}{c}
    1x \\
    + 0y \\
    \hline
    0z
    \end{array}
    \]
  - \((0z)_{\text{signed}} = m + m'\).
  - Algorithm gives result \(0z\) (carry ignored), the correct result.

Correctness of the Addition Algorithm (Cont.)

- **Case** sum of \(x, y\) as unsigned \((k-1)\)-bit numbers has an overflow.
  - Then sum is of the form \(z\) (\(z\) is a \((k-1)\)-bit number) plus carry.
  - Since
    - \(z\) ignores this carry,
    - carry would have weight \(2^{k-1}\),
  - it follows that
    \[
    0bz = 2^{k-1} + m + m' - 2^{k-1}.
    \]
  - Sum of \(1x, 0y\) as unsigned number is \(0z\) plus carry:
    \[
    \begin{array}{c}
    1x \\
    + 0y \\
    \hline
    0z
    \end{array}
    \]
  - \((0z)_{\text{signed}} = m + m'\).
  - Algorithm gives result \(0z\) (carry ignored), the correct result.
Correctness of the Addition Algorithm (Cont.)

- **Case** sum of $x$, $y$ as unsigned $(k-1)$-bit numbers has no overflow.
  - $0z = 2^{k-1} + m + m'$.
  - Sum of $1x$, $0y$ as unsigned number is $1z$:
    \[
    \frac{1x}{+0y} \quad \frac{1z}{1z}
    \]
  - This represents
    \[
    -2^{k-1} + 2^{k-1} + m + m' = m + m',
    \]
    the correct result.
  - Algorithm has as result $1z$, the correct result.

Operations on Signed Fixed Point Numbers

- **Addition/subtraction/multiplication** of signed fixed point numbers $m$, $m'$ carried out as for unsigned fixed point numbers:
  - Assume we have $k$ bits to the left of point and $l$ bits to the right of the point.
  - Write $m$, $m'$ with this number of bits $l$. (Important, that both have the same format).
  - Omit the point.
  - Add/subtract/multiply $m$, $m'$.
  - Reintroduce the point with $k$ bits after point (addition/subtraction) and $2k$ bits after point (multiplication).

- **Correctness of the above**:
  - Multiplication of a signed number $l$ with $k$ bits after the point by $2^n$ is the same as shifting the point $n$ bits to the right.
    - MSB which had weight $-2^{k'}$ gets new weight $-2^{k'+n}$.
    - Any of the other bits, with weight say $2^{k''}$, gets new weight $2^{k''+n}$.
  - Now correctness follows exactly as for unsigned numbers.

- **Signed fixed point numbers.**
  - Similarly to fixed point numbers, the MSB gets weight $-2^{k-1}$, if $k$ bits to the left of the point.

- **Conversion**:
  - Convert whole-numbered part as a signed binary number and the fractional part as before.

- **Examples ($k=2$)**:
  - $(1001.0)_{\text{signed}}$:
    - $(1001)_{\text{signed}} = -2^3 + 2^0 = -8 + 1 = -7$.
    - $(0.01)_{\text{unsigned}} = 0.25$.
    - Therefore
      - $(1001.01)_{\text{signed}} = -7 + 0.25 = -6.75$.
  - $(0010.11)_{\text{signed}}$:
    - $(0010)_{\text{signed}} = 2$.
    - $(0.11)_{\text{unsigned}} = 0.5 + 0.25 = 0.75$.
    - Therefore
      - $(0010.11)_{\text{signed}} = 2 + 0.75 = 2.75$.
Examples (Addition, Signed Numbers)

Fixed Point Numbers Corresponding Integers

Two positive integers

\[(01.01)_{\text{signed}} + (00.10)_{\text{signed}} = (01.11)_{\text{signed}} \quad \text{(decimal: } 1.25 + 0.5 = 1.75)\]

Two negative integers (Carry ignored)

\[(11.01)_{\text{signed}} + (11.11)_{\text{signed}} = (11.00)_{\text{signed}} \quad \text{(decimal: } -0.75 + (-0.25) = -1)\]

Opposite signs (Carry ignored)

\[(01.01)_{\text{signed}} + (10.11)_{\text{signed}} = (00.00)_{\text{signed}} \quad \text{(decimal: } 1.25 + (-1.25) = 0)\]

Example (Multiplication, Signed Numbers)

- Calculation of \((10.1)_{\text{signed}} \cdot (11.1)_{\text{signed}}\) as fixed point numbers.
  - Booth’s algorithm yields:
    \[(101)_{\text{signed}} \cdot (111)_{\text{signed}} = (000011)_{\text{signed}}.\]
  - So result of the above is \((0000.11)_{\text{signed}}.\)
  - This is correct, since
    \[\ast (10.1)_{\text{signed}} = -2 + 0.5 = -1.5.\]
    \[\ast (11.1)_{\text{signed}} = -2 + 1 + 0.5 = -0.5.\]
    \[\ast (0000.11)_{\text{signed}} = 0.75 = (-1.5) \cdot (-0.5).\]

Examples (Addition, Signed Numbers, Cont.)

Overflow

\[(01.01)_{\text{signed}} + (01.01)_{\text{signed}} = (10.10)_{\text{signed}} \quad \text{(decimal: } 1.25 + 1.25 = 2.5 \geq 2)\]

Underflow

\[(10.00)_{\text{signed}} + (10.00)_{\text{signed}} = (00.00)_{\text{signed}} \quad \text{(decimal: } -2 + (-2) = -4 < -2)\]

5. CPU and Interconnecting Structure

(a) Overview.
(b) Execution of Commands.
(c) The Fetch-Decode-Execute Cycle.
(d) Registers.
(e) The ALU.
(f) Buses
(a) Overview

Remember the von Neumann Architecture (with additional connections CU ← I/O):

- Main memory stores information:
  - Data,
  - programs.

(b) Execution of Commands

Execution of High Level Commands

- We will consider the interaction between
  - Arithmetic-Logic Unit, (ALU).
  - Main memory.
  - Basic registers (temporary storage).
  during the execution of programs.

- Consider the high level language command
  \[ A := (B + C) \times (C+D) \]
  - Too complicated in order to be executed directly.

- Decompose it into 3 commands:
  \[
  \begin{align*}
  \text{Aux1} & := B + C \\
  \text{Aux2} & := C + D \\
  A & := \text{Aux1} \times \text{Aux2}
  \end{align*}
  \]
  - Variables are usually stored in main memory.
  - During executions, most commands deal with registers.
  - Registers = small storage cells in the CPU.
  - Allow fast storage and retrieval.

Execution of High Level Commands (Cont.)

- Initially, we will use in the following only one user visible register, AC (Accumulator).

- Use of very simple instructions with simple instruction codes (little man computer):
  - LOAD \( A \)
    “Load AC with content at memory address \( A \)”.
  - ADD \( A \)
    “Add to AC content at memory address \( A \), store result in AC”.
  - MULT \( A \)
    “Multiply AC by content at memory address \( A \), store result in AC”.
  - STORE \( A \)
    “Store content of AC at memory address \( A \)”.
Execution of High Level Commands (Cont.)

- The above command translates then into the following
  (A, B, C, D, AUX1, AUX2 are the addresses where the corresponding variables are stored, ";" indicates comments):

  ```
  LOAD B ; AC = B
  ADD C ; AC = B + C
  STORE AUX1 ; AUX1 = B + C
  LOAD C ; AC = C
  ADD D ; AC = C + D
  STORE AUX2 ; AUX2 = C + D
  LOAD AUX1 ; AC = (B+C)
  MULT AUX2 ; AC = (B+C) * (C+D)
  STORE A ; A = (B+C) * (C+D)
  ```

Execution of Machine Instructions

Consider the following piece of program code:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x300</td>
<td>LOAD 0x940</td>
<td>0x1940</td>
</tr>
<tr>
<td>0x301</td>
<td>ADD 0x941</td>
<td>0x5941</td>
</tr>
<tr>
<td>0x302</td>
<td>STORE 0x941</td>
<td>0x2941</td>
</tr>
</tbody>
</table>

- Codes for the instructions in this toy example consist of
  - a 4 bit operation code (or opcode), representing the instruction (the 1 in 1940).
  - a 12 bit operand, which is here the address of an element in main memory.

- Execution will use the following internal registers:
  - PC = Program Counter.
  - IR = Instruction Register.
  - Contains next instruction to be executed.

Execution of Machine Instructions (Cont.)

0. Initially, the Program Counter (PC) points to address 300.

1a. Fetch instruction from address 300.
    Result stored in IR.
    Simultaneously, increment PC by 1.

1b. Load AC with content of memory location 940.

2a. Fetch instruction from address 301.
    Result stored in IR.
    Increment PC by 1.

2b. Add contents of AC and contents of location 941 and store result in AC.

3a. Fetch instruction from address 302.
    Result stored in IR.
    Increment PC by 1.

3b. Store contents of AC at location 941.

Memory

<table>
<thead>
<tr>
<th>300</th>
<th>1940</th>
</tr>
</thead>
<tbody>
<tr>
<td>301</td>
<td>5941</td>
</tr>
<tr>
<td>302</td>
<td>2941</td>
</tr>
</tbody>
</table>

CPU Registers

<table>
<thead>
<tr>
<th>300</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td></td>
</tr>
<tr>
<td>IR</td>
<td></td>
</tr>
</tbody>
</table>

Step 0: Initial State

(All numbers are hexadecimal)
**The Fetch-Decode-Execute Cycle**

- **Three phases carried out for each instruction:**
  - **Fetch Cycle:**
    Get next machine instruction from main memory.
  - **Decoding Cycle:**
    Decode the fetched instruction.
  - **Execution Cycle**
    Carry instruction out.
    Done by sending control signals to the units.
    Once this is done, repeat this cycle:
    fetch next instruction etc.

---

### Step 1a: Fetch Instruction 300

<table>
<thead>
<tr>
<th>Memory</th>
<th>CPU Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>300 1940</td>
<td></td>
</tr>
<tr>
<td>301 5941</td>
<td></td>
</tr>
<tr>
<td>302 2941</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>940 0003</td>
<td></td>
</tr>
<tr>
<td>941 0002</td>
<td></td>
</tr>
</tbody>
</table>

### Step 1b: Execute LOAD 940

<table>
<thead>
<tr>
<th>Memory</th>
<th>CPU Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>300 1940</td>
<td></td>
</tr>
<tr>
<td>301 5941</td>
<td></td>
</tr>
<tr>
<td>302 2941</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>940 0003</td>
<td></td>
</tr>
<tr>
<td>941 0002</td>
<td></td>
</tr>
</tbody>
</table>

### Step 2a: Fetch Instruction 301

<table>
<thead>
<tr>
<th>Memory</th>
<th>CPU Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>300 1940</td>
<td></td>
</tr>
<tr>
<td>301 5941</td>
<td></td>
</tr>
<tr>
<td>302 2941</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>940 0003</td>
<td></td>
</tr>
<tr>
<td>941 0002</td>
<td></td>
</tr>
</tbody>
</table>

### Step 2b: Execute ADD 941

<table>
<thead>
<tr>
<th>Memory</th>
<th>CPU Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>300 1940</td>
<td></td>
</tr>
<tr>
<td>301 5941</td>
<td></td>
</tr>
<tr>
<td>302 2941</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>940 0003</td>
<td></td>
</tr>
<tr>
<td>941 0002</td>
<td></td>
</tr>
</tbody>
</table>

### Step 3a: Fetch Instruction 302

<table>
<thead>
<tr>
<th>Memory</th>
<th>CPU Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>300 1940</td>
<td></td>
</tr>
<tr>
<td>301 5941</td>
<td></td>
</tr>
<tr>
<td>302 2941</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>940 0003</td>
<td></td>
</tr>
<tr>
<td>941 0002</td>
<td></td>
</tr>
</tbody>
</table>

### Step 3b: Execute STORE 941

<table>
<thead>
<tr>
<th>Memory</th>
<th>CPU Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>300 1940</td>
<td></td>
</tr>
<tr>
<td>301 5941</td>
<td></td>
</tr>
<tr>
<td>302 2941</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>940 0003</td>
<td></td>
</tr>
<tr>
<td>941 0005</td>
<td></td>
</tr>
</tbody>
</table>
The Fetch Cycle

- Need to know address in main memory, to fetch instruction from.
- Done via Program Counter (PC).
- Instruction stored in instruction register (IR).

Realization of Fetch Cycle on a Digital Level

- The following diagrams shows the layout for a fictitious architecture with
  - Only 4 memory cells.
  - Therefore a 2 bit PC used.
  - Only the connections for the first bit of the IR are shown.
    (For the other bits, there are similar connections, which operate in parallel).
  - Connection to PC, assertion of read/write controlled by the CU.
Here you see how the instruction stored in memory location 01 is loaded:

![Diagram of memory and instruction register](image)

1. **Enable Mem 0**
2. **Enable Mem 1**
3. **Enable Mem 2**
4. **Enable Mem 3**

**IR**

1. 0
2. 0
3. 1
4. 1

**PC**

1. 0

The Decoding Cycle

- Identify the bits representing:
  - the **operation code (op-code)**
  - what kind of instruction: addition, multiplication, storage etc.
  - and the **operands**
    * addresses for memory to be loaded, stored
    * register numbers
    * next instructions
    * constants

- Might not require any memory cycles, if decoding is easy (as in the above example).

RISC vs. CISC

- **RISC (reduced instruction set computers)** have simple instruction formats.
  - More modern concept.
  - Decoding very simple and does usually not take any memory cycle.
  - Examples: PowerPC (Macintosh), SUN SPARC and descendants

- **CISC (complex instruction set computers)** have very complex instruction formats
  - Older concept.
  - Length of op code can vary very much.
  - Decoding might take several cycles.
  - Example: Intel Pentium.

- In the 80s, 90s expectation RISC will eventually outperform CISC.

- But still CISC very successful.
  - Trend: mixture between CISC and RISC.

- More later.

The Execution Cycle

- Execute instruction.
  - **Example 1**: LOAD R\textsubscript{m}, M\textsubscript{n}.
    * Meaning: load memory location n into register m.
    * Execution:
      - Connect bits in the instruction containing address n as before PC with memory.
      - Connect lines from memory at the CPU with register m.
      - Assert read to main memory
      - Assert write to register m.
The Execution Cycle (cont.)

- **Example 2**: ADD $R_k, R_l, R_m$
  
  - Meaning: Add register $R_l$, $R_m$; store result in register $R_k$.
  
  - Execution:
    - Cycle 1:
      * Connect registers $R_l$, $R_m$ with the ALU, assert Addition.
      * Store result temporarily in some temporary register.
    - Cycle 2:
      * Connect that temporary register with register $R_k$.
  
  - This takes 2 cycles.
  
  More complex instructions can take much more cycles.

Refined Instruction Cycle

- More refined instruction cycle see next slide.
  - Address of instruction might need to be calculated.
  - Execution cycle split into
    * operand address calculation (might be more complicated, see section on addressing modes).
    * operand fetch (operands might be fetched separately)
    * data operation (e.g. ADD)
    * operand store address calculation,
    * operand store.
  
  - For most instructions some of these sub-steps of the execution cycle might be trivial or omitted altogether.
(d) Registers

- Two kinds of registers
  - User Visible Registers.
  - Control and status Registers.

User Visible Registers

- General-purpose registers.
  - Use of more expensive but faster technology than for main memory.
  - General-purpose registers used for frequently used data (e.g. the $i$ in a loop “for $i=1$ to $10000 \ldots$”). Can be read and written by the machine language.
  - Sometimes specific ones for
    * data vs. programs
    * floating-point vs. integer data.
  - This distinction allows to save address space.

- Condition codes, often referred to as flags.
  - Are one-bit registers.
  - Examples are overflow or result zero flags from last ALU operation.
  - Condition codes are usually set by the ALU and (sometimes) by the CU and only read by the user.
  - However some might be changed (in exceptional cases) by the machine language.

Control and Status Registers

- Registers, used while carrying out the instructions in the CPU.

- Typical ones are:
  - Program Counter (PC)
  - Instruction Register (IR)
  - Memory Data Register (MDR) or Memory Buffer Register (MBR): Stores data received from main memory, which can not directly be passed on to a register.
  - Memory Address Register (MAR): * Contains addresses of memory locations, from which data is to be fetched or into which data is to be stored.
(e) ALU

- Carries out operations like
  - addition, subtraction
  - bitwise and, or, negation.

(f) Buses

- A **bus** consists of 50 - 100 separate lines,
  - each of which assigned with a particular meaning or function,
  - each line carries one bit.

- Most buses have
  - **Data lines** for transfer of data (including instructions).
    * Collection of data lines is called **data bus**.
    * Data buses have typically 8, 16, 32, 64 separate lines.
    * This number is called **width** of the data bus.
  - **Address lines**.
    Carry
    * memory address,
    * possibly as well addresses to I/O ports.
    · Collection of address lines is called **address bus**.
    · Usually **higher order bits** select particular module on the bus.
    · Lower order bits select **memory location** or **I/O port** within that module.

- **Control lines**. For instance:
  - Collection of control lines is called **control bus**.
  - Memory write/read.
  - I/O write/read.
  - Transfer acknowledgment.
  - Request of bus and grant of bus access.
  - Interrupt request and acknowledgment.
  - Clock.
  - Reset
**Dedicated and Multiplexed Lines**

- **Dedicated lines.**
  - Lines of the bus which serve only one particular purpose.
  - Example: transfer of data bit 4.

- **Multiplexed lines.**
  - Carry many signals multiplexed through time.
  - Suitable for I/O (memory has much higher traffic which needs to be treated much faster).

**Arbitration**

- **Arbitration** = process of determining which component may write to the bus.

- **Two methods:**
  - **Centralized,** by one piece of hardware
    - (bus controller or arbiter).
  - **Distributed:**
    - Each module has access control logic,
    - modules act together to share the bus.

- **With both methods,**
  - a **master** is determined,
  - which initiates a data transfer with some other device,
  - called **slave.**

**Timing**

- **Synchronous:**
  - All bus operations synchronized by a clock.
  - **Advantage:** Simpler to implement and test.
  - **Disadvantage:**
    - Tied to a fixed clock rate,
    - no advantage if one device is replaced by a faster one.

- **Asynchronous:**
  - No clock for the bus.
  - Use special control signals to synchronize them.
  - Very difficult to verify.

- **Example:**
  - With synchronous timing it is clear that after one clock signal, one bit put on the data bus has been received by the recipient.
  - With asynchronous timing, an acknowledgement signal is required by the recipient to indicate that the bit has arrived.
  - So the protocol is more complicated and requires extra control lines.

**Multiple-Bus Hierarchies**

- Bus forms the **von-Neumann-bottleneck:**
  - Relatively slow.
  - Blocks increase in performance.

- Methods taken for solving this problem:
  - A **cache** is placed between processor and memory.
    - Frequently used memory contents do not have to be moved via the system bus.
    - Sometimes cache integrated into the processor.
  - Formation of **hierarchies of buses**
    - s.t. some memory traffic does not go via the system bus.
  - Design of I/O modules with **controllers, buffers,**
    - so that memory access is needed only, when bus is free.
    - Memory transfer carried out in **blocks.**
Supplementary Material for Sect. 5

(a) The PCI Protocol.

(b) Synchronous vs. Asynchronous Timing.

(c) Data Transfer Types

Explanation of the modules on last slide:

- **SCSI** = small computer system interface = type of bus which supports local disk drives and other peripherals.

- **LAN** = local area network, for instance Ethernet.

- **Fire Wire** = high speed bus arrangement for support of high-capacity I/O devices.
Description of the PCI-Protocol (Last Slide)

- If a signal is at the top line means deasserted, at the bottom line asserted.

- Protocol is **synchronized** by a clock.

- In this protocol, no treatment of bus arbitration
  - (i.e. decision who is **master**, who is **slave**).
    - In case of PCI, **master** is called **initiator**, **slave** is called **target**.
  - Separate protocol (via a centralized PCI arbiter).
  - We assume, initiator has gained control over the bus.

Bus Lines Controlled by the Initiator

- **FRAME** asserted indicates
  - initiator has started a request and not received all data yet.

- **C/BE##**
  - When address given, indicates whether read or write requested.
  - When data given, tells which bytes of the word on AD are valid.
  - (Any of the four bytes of a word can be selected individually.
  - Master/slave might not be able to receive/send a full word of 4 bytes in one step).

- **IRDY** = indicates whether initiator is ready to receive data.

Functionality of the Bus Lines

- **CLK** = clock.

- **AD**
  - Multiplexed address/data lines.
    - 32 bits.
  - An extension to 64 bits exists
  - In case of a write operation, AD controlled by the initiator.
  - In case of a read operation, address controlled by the initiator, data controlled by the target.
**Bus Lines Controlled by the Slave**

- **TRDY**
  - During read operation: Valid data is on AD.
  - During write operation: Slave ready to receive data.

- **DEVSEL**
  - Slave has received address.

---

**Description of the Steps in the PCI Protocol**

- **a.**
  - Initiator asserts FRAME.
  - Initiator puts requested address on address bus.
  - Initiator asserts read on C/BE#.

- **b.**
  - Target has received the address.

- **c.**
  - Initiator assumes that address has been received,
  - removes therefore address from the bus.
  - **Turnaround cycle** required, since a different device (target) controls now those bus lines.
  - Initiator indicates on C/BE#, which bytes of AD should contain valid data.
  - Initiator asserts on IRDY# that it is ready to receive data.

---

**Description of the Steps in the PCI Protocol Example (Cont.)**

- **d.**
  - Target asserts DEVSEL#, since it has recognized its address and will respond.
  - It replaces requested data on AD.
  - Target asserts IRDY, since valid data is on AD.

- **e.**
  - Initiator reads data.
  - Changes C/BE# and indicates which bytes it wants to read next.

- **f.**
  - As an example, here we assume that target needs time before second data item is ready.
  - Therefore TRDY# is deasserted.
  - When target is ready, it asserts TRDY# again, puts data on AD.

- **g.**
  - As an example, here we assume that initiator needs time before it can received 3rd data item.
  - Therefore it deasserts IRDY#.
  - Data item will remain on AD until IRDY# is asserted again.

- **h.**
  - Initiator will now receive its last data item (which it knows is already there because of IRDY# on AD).
  - It therefore deasserts FRAME#.
  - It asserts IRDY#, since it is now ready to receive data.
Description of the Steps in the PCI Protocol Example (Cont.)

i.
- Initiator deasserts IRDY#, since it has received its data.
- Target deasserts TRDY# and DEVSEL#, and removes data from AD.
- FRAME#, AD, C/BE# have now turnaround time, so that another initiator can use them one clock cycle later.
- IRDY#, TRDY#, DEVSEL# contain still valid information ("deasserted");
- they have their turnaround time afterwards.
- Note the turnaround time for these signals during a., b.

(b) Synchronous vs. Asynchronous Timing

Consider the following timing diagram.

The Synchronous Timing Protocol

- Figure 3.19 (a).
  - Description of a simple CPU/memory read request.
- Master puts a read signal, the requested address on the bus.
- It further asserts a start signal.
- In the next cycle, slave puts data on the data lines.
- Slave asserts an acknowledgment signal, to indicate that it has received the address and put data on the data lines.
The Asynchronous Timing Protocol

- Figure 3.19 (b).
- Master places address + read signal on the bus.
- It takes some time, before address has stabilized. Known by the master. (With synchronous timing, this time is 1 clock cycle).
- When this is the case, master asserts MSYN (master synchronization) line.

(c) Data Transfer Types

Several data transfer types allowed (used in order to decrease the amount of time for switching between different modes).

- **Read.**
  - Master puts address on the bus.
  - Then slave puts data on it.
  - In case of multiplexed lines delay for stabilization of the bus in between.

- **Write.**
  - Master puts address and data on the bus
  - (if multiplexed, consecutively).

- **Read-Modify-Write.**
  - Read followed by a write.
  - Master puts address on the bus.
  - Slave puts data on it.
  - Master puts then data on it (no second address transfer needed).

The Asynchronous Timing Protocol (Cont.)

- When address is stable and slave has received it, it puts data on data lines.
- As soon as data is stable, slave asserts SSYN (slave synchronization) line.
- When master has read data, it deasserts MSYN.
- Slave responds by deasserting SSYN.
- It then removes data from data lines.
- Master removes address from address bus.

Data Transfer Type (Cont.)

- **Read-After-Write.**
  - Write followed by a read transfer.
  - Master puts address and data on the bus.
  - Slave puts afterwards the received data on the bus.
  - Used for checking purposes.

- **Block transfer.**
  - Only once the address is put on the bus.
  - Then follow several data transfers from adjacent memory locations,
  - controlled by control lines.
6. Internal Memory

(a) Categories of Internal and External Memory.

(b) Cache Memory.

(c) Stacks.

(a) Categories of Internal and External Memory

Location

- Internal Memory
  - On-chip (processor; registers, some cache).
  - Off chip. (Main memory, some cache).
- External (eg. harddisk, CD-ROM, tape).

Capacity

External and main memory measured in terms of Bytes:
1 byte = 8 bits.

- 1 Kilobyte (KByte) = $2^{10}$ byte = 1024 byte,
- 1 Megabyte (MByte) = $2^{20}$ byte = 1024 Kilobyte.
- 1 Gigabyte (GByte) = $2^{30}$ byte = 1024 Megabyte.
- 1 Terabyte (TByte) = $2^{40}$ byte = 1024 Gigabyte.
- 1 Petabyte (PByte) = $2^{50}$ byte = 1024 Terabyte.
- 1 Exabyte (EByte) = $2^{60}$ byte = 1024 Petabyte.

- Sometimes one needs to work with Kilobits, Megabits etc.:
  - Kilobit = 1024 bits.
  - Megabit = 1024 Kilobits.
  - etc.
Remark on Units

When dealing with metric units we have:

- **Kilo** means \(10^3 = 1000\).
- **Mega** means \(10^6 = 1000000 = 1000\) Kilo.
- **Giga** means \(10^9 = 1000000000 = 1000\) Mega.
- **Tera** means \(10^{12} = 1000\) Giga.
- **Peta** means \(10^{15} = 1000\) Tera.

This applies to meter, seconds, Hertz (see next slide).

For bytes, bits one uses kilo for 1024, since 1000 bytes is not a natural unit since addresses are in binary.

### Hertz

- **Hertz (Hz)** = 1 signal per second.
  - 1 Hz = 1 (signal) per second.
  - 2 Hz = 2 (signals) per second.
  - 1 KHz = 1000 Hz. (**not 1024 Hz!!**).
  - 1 MHz = 1000 kHz = 1 million (signals) per second.
  - 1 GHz = 1000 MHz.

Note that 1kHz is **not** 1024 Hz!!!

So the period between two impulses is in case of

<table>
<thead>
<tr>
<th>Hertz</th>
<th>duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1Hz</td>
<td>1s (second)</td>
</tr>
<tr>
<td>2Hz</td>
<td>0.5s (seconds)</td>
</tr>
<tr>
<td>1KHz</td>
<td>(\frac{1}{1000}) s = 1ms (millisecond)</td>
</tr>
<tr>
<td>1MHz</td>
<td>(\frac{1}{1000000}) s = 1µs (microsecond)</td>
</tr>
<tr>
<td>1GHz</td>
<td>(\frac{1}{1000000000}) s = 1ns (nanosecond)</td>
</tr>
</tbody>
</table>

### Addressable Units

- **Word** = natural unit of organization of memory.
  - Typically: Number of bits used to represent a number or smallest length of an instruction
  - Typically 2, 4 or 8 byte.

**Addressable units.**

- Usually word.
- Sometimes byte level addressable.
- Examples:
  - Assume
    - Addressable unit = 2 byte = 1 word
    - Address length = 16 bit
  - Then the first words have addresses
    - 0x0000, 0x0001, 0x0002.
  - If the addressable unit is 1 byte, the same words have addresses
    - 0x0000, 0x0002, 0x0004.
  - With
    - 2 bits we can address \(2^2 = 4\) addressable units,
    - with 3 bits \(2^3 = 8\) addressable units
    - with 4 bits \(2^4 = 16\) addressable units
    - with \(k\) bits \(2^k\) addressable units.

### An Analogy for Addressable Units

The difference between addressable units “byte” and “word with word = 2 byte” is similar to the two number systems for a semi-detached houses:

- Assume a row of semidetached houses, ie. each house has two house numbers.
- We could give them two kind of house numbers:
  - House 1 has numbers 1a, 1b. House 2 has numbers 2a, 2b. Etc.
  - House 1 has numbers 1, 2. House 2 has numbers 3, 4. Etc.
- See picture next slide.
- Addresses are like house numbers, but they start with 0 instead of 1.
- “Addressable unit byte” is like the system with numbers 1, 2, 3, 4 etc.
- “Addressable unit word” with 1 word = 2 byte is like the system with numbers 1a, 1b, 2a, 2b etc. (a and b are left and right byte of the word addressed).
**Method of Access.**

- **Sequential Access:**
  - Access of memory sequentially.
  - Long seek time, before information is accessed.
  - But then following block accessed relatively fast.
  - Main example: tape units.
  - Nowadays mainly used for backups.

- **Direct Access:**
  - Blocks have a direct address.
  - Location within such a block is accessed sequentially.
  - Example: disks, CDs, DVDs.

- **Random Access:**
  - Each addressable location of the memory can be addressed and accessed directly.
  - Example: Main memory, some cache memory, registers.

**Method of Access (Cont.)**

- **Associative:**
  - Random-access type memory.
  - Used for cache.
    - Cache stores data which is expected to be used in the near future.
    - Faster access to cache than to main memory.
    - In order to look up data, one has to check whether its address is in cache.
    - One part of the address selects directly a sequence of addresses in the associative memory.
    - The associative memory verifies in one step, if the rest of the address (tag) coincides with one of the tags of the addresses stored in it.
    - If success, the value stored is retrieved.
    - If unsuccessful, data has to be retrieved from main memory.
Performance Parameters

- **Access time:**
  - For **random-access memory**, time to perform read or write operation:
    Time from the instant that address is presented to the instant that data are stored or made available for use.
  - For **non-random-access memory**, time till read-write mechanism reaches the desired location.

- **Memory cycle time:** Applies to random-access memory.
  Access time plus additional time, till next access is possible.

Physical Types

- **Semi-conductor memory.**
- **Magnetic surface memory** (disk, tape).
- **Optical memory.**
- **Magneto-optical memory** (currently not very important).
Types of Memory by Permanence

- **Volatile Memory**: information lost after power switched off.
- **Non-volatile**: information remains.
  - (e.g. magnetic surface memory, ROM, PROM, EPROM, EEPROM, flash memory, old core memory; see below).
- **Erasable memory**: memory can be overwritten.
- **Nonerasable memory**.
  - Nonerasable memory is necessarily nonvolatile.

Types of Semiconductor Memory

- **RAM**.
  - Abbreviates Random-Access Memory.
  - Abbreviation is mainly historic; RAM should be called
    * erasable (volatile) random access memory.
  - Two Sorts:
    * **Static RAM (SRAM)**:
      - Use of flip-flops
      - Holds data as long as power is supplied (no refresh necessary).
      - More expensive per bit but faster than dynamic RAM.
      - Used for small portions of memory, which require fast access:
        (registers, cache).
    * **Dynamic RAM (DRAM)**.
      - Cheaper.
      - Can be packed more densely

Realization of DRAM

- Current main technique: small capacitors.
- When storing memory, voltage is set on the capacitor, and it is charged.
- Capacitor keeps charge for a few milliseconds.
- When reading it, the capacitor is discharged. **Refreshing is necessary.**
- As well, charge in capacitor fades away.
  - Regularly (after some milliseconds), memory has to be refreshed.
    * The part to be refreshed for a moment unavailable.
    * But refreshing can be done very fast.
    * Only small percentage of time used for refreshing.

Core Memory

- Old technique for what is now DRAM:
  - Ferrite core.
  - Rings of magnetic cores
    (⇒ terminology core memory).
  - Advantages:
    * Robust.
    * Non-volatile.
  - Slow and expensive.
  - Remains (or remained?) in use for military applications and in space.
Other Types of Semiconductor Memory

- **Read-Only Memory (ROM).**
  - Non-erasable.
  - Created like a circuit chip with data stored using gates.
  - Large cost in development, no room for error.
  - Used for
    * Microprogramming of the CPU.
    * Function tables, frequently wanted functions, system programs.

- **Programmable ROM (PROM).**
  - Nonvolatile, can be written only once.
  - Writing process performed electrically, can be performed after the chip is produced.
  - Special equipment for programming needed.
  - Still attractive for high-volume productions.

- **Read-mostly Memory:**
  - Write operation difficult, but can be performed several times.
  - **EPROM.**
    * Erasable Programmable Read-Only Memory.
    * Read and written electrically.
    * One transistor per chip, therefore high density.
    * Erasure possible using UV radiation.
    * Erasure takes 20 minutes.
  - **EEPROM**
    * Electrically Erasable Programmable Read-Only Memory.
    * Can be written to without erasing prior contents.
    * Write operation takes approx several hundred microseconds per byte.
    * More expensive and less dense than EPROM.

- **Flash memory.**
  - Electrical erasing technology like EPROM, EEPROM.
  - Entire memory can be erased in one or a few seconds.
  - Blocks of memory can be erased.
  - No byte-level erasure.
  - Density as for EPROM.
  - Slower and more expensive than the harddisk – therefore no substitute for harddisk.
  - Used in digital cameras, voice recorders, PDAs.
  - Sometimes used in order to store the BIOS (as a substitute for EPROM).

Memory Hierarchy
Memory Hierarchy (Cont.)

- Different kinds of storage devices:
- Expensive (per bit), but fast ones.
  - Very fast: Registers.
  - Fast: Used for cache.
  - Based on SRAM.
- Medium price/speed: Main memory
  - DRAM
- Cheap, slow:
  - Outboard storage
    * Not in main memory.
    * But fast available
  - Offline storage
    * Can be carried around.

(c) Cache Memory

(i) Overview.

(ii) Mapping Functions.

(iii) Replacement Algorithms.

(iv) Write Policies.

(i) Overview

- Problem: Access to main memory slow.
- Therefore use of cache.
- **Cache** = buffer between CPU and main memory.
  - Stores data from memory, which is expected to be accessed soon.
  - Usually data which has been accessed recently.
- Normally constructed from SRAM (static RAM).
- Stores **blocks** of memory used before, so that the data can be retrieved fast.
  - If one memory location is used, usually adjacent memory locations are required as well.
- Typical length of a block: 2/4/8/16 byte.
Main memory much bigger than cache.

⇒ In each line of cache several different blocks from main memory may be stored.

Therefore one needs to attach a tag to each line in cache.

- Identifies, which of the memory blocks, which can be stored in this line, this block is referring to.

### General Read Policy
Cache operates as follows:

- **If master**
  - (CPU or any other device with access to memory) requires memory access, it is first checked whether the data is in cache.

- **If data is in cache**
  - a **hit** is declared,
  - cache returns the data.

- **Otherwise**
  - a **miss** is declared,
  - cache reads block of memory containing memory location into a cache line,
  - passes data to master.

- **(Next slide: RA = real address (= physical address when using paging, addresses in the programs are virtual addresses which are translated into physical addresses. See module on operating systems).**

### Cache Size

- **Tradeoff** between different cache sizes:
  - The bigger the cache the slower it is.
    * If cache is bigger, distances increase.
  - Principle of **locality of reference**:
    * Memory references tend to cluster.
  - Studies suggest that cache sizes between 1K and 512 K words are most effective.
Two level Cache

- **On-chip cache** is now technical standard (integrated in the processor).
  - Typical size: 8-16 KByte.
- Additionally **off-chip cache** often used.
  - Can store more data (order of 256 KByte)
- Originally
  - L1-cache (level 1-cache) for on-chip cache,
  - L2-cache (level 2-cache).
- Nowadays (e.g. Pentium, PowerPC) often both
  - L1-cache small (eg. 8KByte) and fast, contains mostly recently used data,
  - L2-cache is big (256 KByte) and slightly slower.

Split vs. Unified Cache

- **Unified cache** contains both data and instruction.
- **Split cache** provides separate caches for data and instructions.
- **Advantages of split cache** in pipelining:
  - Instruction cache can be accessed for fetching instructions, while simultaneously data cache can be accessed for fetching data.
  - Allows to choose different strategies for replacing data and instructions in cache (replacement algorithms), which allow to have higher hit rates in cache.

(ii) Mapping Functions

- **Question**: Which memory blocks can be stored in which cache lines?
- We assume in the section about mapping functions the following architecture:
  - Word size: 4 byte.
  - Addresses refer to words.
  - Cache size: 128 KByte = $2^7$ KByte = $2^7 \cdot 2^{10}$ byte = $2^{17}$ byte = $2^{15}$ words.
  - Block size: 16 byte = 4 words.
  - Therefore Cache has $2^{13}$ lines of 16 byte each.
  - Main memory: 16 MByte = $16 \cdot 2^{20}$ byte = $2^{24}$ byte = $2^{22}$ words.
  - Therefore address has 22 bits.
  - Further we apply write through policy.
    - (i.e. if data in cache is changed, main memory is updated immediately; more about this later).

Block Number

- We have a block size of 4 words, and addresses referring to words.
- Therefore with every 4th word a new block starts.
- **Block number** =
  - the number of the block an address belongs to.
  - Block number computed by dividing the address by 4 (with remainder).
  - Division of a binary number by 4 means shifting it 2 bits to the right.
    - Division by $2^n$ means to shift it $n$ bits to the right.
  - So in our example the 20 MSBs of an address form the block number:
**Word Number**

- **Word number** = number of the word relative to its block.
  
  - Calculated as the remainder of the division of the address by 4.
  - The remainder of division of a binary number by 4 are its 2 LSBs.
  - Remainder of division by $2^n$ would be the $n$ LSBs.
  - We briefly call the word number word.
  
  - If word = byte, then we refer to the word number as byte number or byte.

- So in our example an address is divided as follows:

  \[
  \begin{array}{c|c}
  \text{block number} & \text{word} \\
  \hline
  20 \text{ bits} & 2 \text{ bits}
  \end{array}
  \]

**General Situation**

- In general, in this section word is the addressable unit.
  
  - So if the addressable unit is a byte, “word” and “byte” are interchangeable.

- Assume in general, that we have
  
  - a block size of $2^k$ words and
  
  - an address space of $2^n$ words.
  
  - Therefore the address has $n$ bits.

**General Situation (Cont.)**

- Then we have the following:
  
  - The block number is the result of dividing the address by $2^k$.
  
  - The word number is the remainder, if we carry out this division.
  
  - Division by $2^k$ is the same as shifting the address $k$ bits to the right.
  
  - The remainder of this divisions are the bits shifted beyond the boundary of the address with this shift.
  
  - Therefore the word number are the $k$ LSBs of the address.
  
  - the word number are the remaining $n-k$ MSBs of it.
  
  - The complete address is divided as follows:

  \[
  \begin{array}{c|c}
  \text{block number} & \text{word} \\
  \hline
  n-k \text{ bits} & k \text{ bits}
  \end{array}
  \]

**First Mapping Function: Direct Mapping**

- Each block of memory is associated with a unique cache line.

- **Locality of reference.**
  
  - Neighbouring blocks should be associated with different cache lines.
  
  - Otherwise, when we are close to a boundary of a block, we would often have to exchange content of cache.
In our example cache has $2^{13}$ lines, containing one block each.

So
- the first $2^{13}$ blocks will be mapped each to one cache line in sequence;
- the next $2^{13}$ blocks are mapped each to one cache line in sequence;
- etc.
- So the 7th block will be mapped to the 7th cache line.
- The $2^{13} + 7$th block will be mapped to the same line. Etc.
- So the line number, a block is associated with, is the remainder of division of the block number by $2^{13}$.
- This remainder are the 13 LSBs of the block number.
- A block number is divided as follows:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 bits</td>
<td>13 bits</td>
</tr>
</tbody>
</table>

Example

A hexadecimal address 0x2B 7AE6 is binary 10 1011 0111 1010 1110 0110.

It is divided into tag, line word as follows:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Line</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 bits</td>
<td>13 bits</td>
<td>2 bits</td>
</tr>
</tbody>
</table>

So the block number is divided as follows:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 bits</td>
<td>13 bits</td>
</tr>
</tbody>
</table>

This has to be stored in cache together with the content of a block
- in order to identify the block number, a block corresponds to.

The remaining 7 bits
- i.e. the result of division of the block number by $2^{13}$,
- identify the block among all the blocks mapped to the same cache line.

A complete address is divided as follows:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Line</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 bits</td>
<td>13 bits</td>
<td>2 bits</td>
</tr>
</tbody>
</table>

Now
- the line number of a cache (the actual address of this line),
- the position of a word within this line and
- the tag
determine uniquely the main memory address, this word corresponds to.

Computers Systems, CS M33, Michaelmas term 2002, Sect. 6
General Situation

In general, assume we have
- an address of \( n \) bits
- a block size of \( 2^k \) words,
- a cache of \( 2^l \) lines.
(This means that it has a size of \( 2^l \cdot 2^k = 2^{l+k} \) words).

Then
- The block number is formed by the \( n - k \) MSB bits of the address.
- The tag is obtained by dividing the block address by \( 2^l \).
- The line number is the remainder by this division.
- Therefore the line number consists of the \( l \) LSBs of the block number.
- The tag number consists of the remaining \( n - k - l \) MSBs of the block number.
- The address is therefore divided as follows:

<table>
<thead>
<tr>
<th>tag</th>
<th>block number</th>
<th>word</th>
</tr>
</thead>
<tbody>
<tr>
<td>n-k-l bits</td>
<td>l bits</td>
<td>k bits</td>
</tr>
</tbody>
</table>

Reading from Memory (Direct Mapping)

Assume master requests a word from main memory.

Divide address into tag \( t \), line \( l \), word \( w \).
- Compare \( t \) with the tag \( t' \) of cache line no. \( l \).
  - If \( t = t' \), we have a **hit**.
    * Send word \( w \) stored in cache line \( l \) to master.
  - Otherwise we have a **miss**.
    * Load block with tag \( t \), line \( l \) into cache.
    * Replace tag of cache line \( l \) by \( t \).
    * Send word \( w \) to master.

Example

- Cache with 2 lines only.
  For simplicity, we divide memory addresses into (tag,line,word).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cache Line after Instruction Carried out</th>
<th>Tag stored in Cache Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially</td>
<td>0 invalid</td>
<td>1 invalid</td>
</tr>
<tr>
<td>Load (0,0,2)Miss</td>
<td>0 invalid</td>
<td>1 invalid</td>
</tr>
<tr>
<td>Load (3,1,3)Miss</td>
<td>0</td>
<td>1 3</td>
</tr>
<tr>
<td>Load (0,0,1)Hit</td>
<td>0</td>
<td>1 3</td>
</tr>
<tr>
<td>Load (1,0,1)Miss</td>
<td>0</td>
<td>1 3</td>
</tr>
<tr>
<td>Load (0,0,1)Miss</td>
<td>0</td>
<td>1 3</td>
</tr>
</tbody>
</table>

Writing to Memory (Direct Mapping)

Assume master requests to store data in memory.
Divide address into tag \( t \), line \( l \), word \( w \).
- Write word to main memory (write through policy).
  - Compare \( t \) with the tag \( t' \) of cache line no. \( l \).
  - If \( t = t' \), store data at word-position \( w \) in cache line \( l \).
  - If \( t \neq t' \),
    * load corresponding (already updated) block in main memory into cache,
    * replace tag by \( t \).
- Alternatively, if \( t \neq t' \),
  * only store word \( w \) in main memory.
Second Mapping Function: Fully Associative Mapping

- Each block of memory can be associated with any cache line.
- Use of associative memory in order to find whether block is in cache.
- So the tag has to be the full block number.
  - In the example the 20 MSBs of the address
- Memory address divided as follows:
  
<table>
<thead>
<tr>
<th>Tag</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 bits</td>
<td>2 bits</td>
</tr>
</tbody>
</table>

  - Calculation of tag and word from an address as before.
  - In the general situation we have as well tag = block number.

Reading from Memory (Associative Mapping)

- Cache memory stores in each line a block of memory together with its tag.
- Assume master requests a word from memory. Divide address into tag \( t \) and word \( w \).
  - Verify whether any cache line has same tag.
    - If yes, hit.
      - Send word \( w \) stored in that cache line to master.
    - Otherwise miss.
      - Load block with tag \( t \) into one cache line which gets tag \( t \).
        - (Which one see below).
      - Send word \( w \) to master.
Example

- Cache with 2 lines only. Memory addresses written as (tag,word).
- FIFO Replacement (see later).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>cache line</th>
<th>tags stored in cache line after Instruction Carried out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially</td>
<td>0</td>
<td>invalid</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>invalid</td>
</tr>
<tr>
<td>Load (0,2)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Miss</td>
<td>1</td>
<td>invalid</td>
</tr>
<tr>
<td>Load (3,1)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Miss</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Load (0,3)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hit</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Load (1,0)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Miss</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Load (0,1)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Miss</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Writing to Memory (Associative Mapping)

- Assume master requests to store a word in memory. Divide address into tag $t$ and word $w$.
  - Write word to main memory (write through policy).
  - Compare $t$ with the tags of all cache lines.
  - If $t$ is equal to one of these, store word $w$ in that cache line $l$.
  - Otherwise
    * load corresponding (already updated) block in main memory into one cache line,
    * set tag to $t$.
    * (Modification of cache might be omitted).

Third Mapping Function: Set-Associative Mapping

- Compromise between direct and associative mapping.
- Cache is divided into $v$ sets of $k$ lines each.
  - In our example, let $k = 4$, therefore $v = 2^{13}/4 = 2^{11}$.
  - With each block we associate a unique set.
    * The block can be stored in one of the $k$ lines of this set.
  - The set number is calculated as the line number in direct mapping.
    * In our example determined by the 11 LSBs of the block address.
  - The remaining bits form again the tag.
    * In our example these are the 9 MSBs of the block address.
Set-Associative Mapping (Cont.)

- So in our example the memory address is divided as follows:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Set</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 bits</td>
<td>11 bits</td>
<td>2 bits</td>
</tr>
</tbody>
</table>

- A block of memory with tag $t$ and set $s$ can be mapped into any cache line of set $s$.

- Memory type used is associative memory.

General Situation

- In general, assume we have
  - an address of $n$ bits
  - a block size of $2^k$ words,
  - a cache of $2^l$ lines
    * (so the cache size is $2^{l+k}$ words),
  - and each set consists of $2^m$ lines.
  - Therefore cache has $2^{l-m}$ sets.

Then

- The block number is formed by the $n-k$ MSB bits of the address.
- The tag is obtained by dividing the block address by the number of sets, ie. by $2^{l-m}$.
- The set number is the remainder by this division.
- Therefore the set number consists of the $l-m$ LSBs of the block number.
- The tag number consists of the remaining $n - k - (l-m)$ LSBs of the block number.
- The address is therefore divided as follows:

<table>
<thead>
<tr>
<th>tag</th>
<th>set</th>
<th>word</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n-k-(l-m)$ bits</td>
<td>$l-m$ bits</td>
<td>$k$ bits</td>
</tr>
</tbody>
</table>

Reading from Memory (Set-Associative Mapping)

- Cache memory stores in each line a block of memory together with its tag.

- Assume master requests a word from memory. Divide address into tag $t$, set $s$ and word $w$.
  - Verify whether any cache line of set $s$ has tag $t$.
    - If yes, hit.
      * Send word $w$ stored in that cache line to master.
    - Otherwise miss.
      * Load block with tag $t$, set $s$ into one cache line of that set.
      * That cache line gets now tag $t$.
      * Send word $w$ of that block to master.
**Example**

- Cache with 2 sets of 2 lines each.
  Memory addresses written as (tag,set,word).
- LFU Replacement (see later).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Set</th>
<th>Line</th>
<th>Tags</th>
<th># of Stored accesses after Instruction Carried out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td>Load (0,0,2)</td>
<td>Miss</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td>Load (1,0,2)</td>
<td>Miss</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>invalid</td>
<td></td>
</tr>
</tbody>
</table>

**Writing to Memory**

- Assume master requests to store a word in memory.
  Divide address into tag $t$, set $s$ and word $w$.
  - Write word to main memory.
  - Compare $t$ with the tags of cache lines of set $s$.
  - If $t$ is equal to one of these, store word $w$ in that cache line.
  - Otherwise
    * load corresponding (already updated) block in main memory into one cache line of set $s$,
    * set tag to $t$.
    * (Modification of cache might be omitted).
Comparison of Mapping Functions

- **Direct mapping:**
  - Cheapest and fastest.
  - But if one is working in two blocks with same line number interleaved, one often has to exchange the two blocks.

- **Fully-associative mapping:**
  - More expensive lookup logic needed.
  - Looking up takes longer.
  - But more often hits.

- **Set-associative mapping** good compromise.
  - Typically 2 or 4 lines per set.

---

(iii) Replacement Algorithms

If we want to load a new block into a cache which is full, we have to remove one block.

- In case of direct mapping there is only one block possible, replace that one.

- For associative and set-associative cache choose from the lines corresponding to that address one according to one of the following algorithms:
  - **Least recently used (LRU).** Replace block which has been in cache longest without a read or write access.
  - **First in first out (FIFO).** Replace block which has been in cache longest.
  - **Least frequently used (LFU).** Replace block which had the least number of read or write accesses.
  - **Random.** Choose a block at random.

Performance with random replacement algorithm is not much worse than with the other algorithms.

---

Example

- Assume a **fully associative cache** with 3 lines.
  (artificial example, since number of lines usually a power of 2).

- Assume cache is **initially empty**, followed by load requests for addresses with block numbers (= tags) \(0,0,1,2,2,2,0,1\)

Then blocks with tags 0,1,2 will be stored in cache.

- Assume now a load request for address with block number 3.
  One of the blocks in cache has to be replaced.
  - W.r.t. **LRU policy**, cache line containing block 2 will be replaced.
  - W.r.t. **LFU policy**, cache line containing block 1 will be replaced (2 accesses, the others had 3 accesses).
  - W.r.t. **FIFO policy**
    cache line containing block 0 will be replaced.

(iv) Write Policies

- When we store data in memory via cache, when do we replace data in main memory?

- Two main policies:

  (a) **Write Through Policy**

  - When changing data which is in cache, change both cache and main memory.
  - When a cache line has to be overwritten, no modification of main memory necessary.
Write Back Policy

- Associate with line \( l \) in cache one bit \( \text{modified}(l) \).
  - \( \text{modified}(l) = 0 \) means:
    - Cache line coincides with main memory.
  - \( \text{modified}(l) = 1 \) means:
    - Cache line was modified without changing main memory.

- Whenever loading main memory into cache line \( l \):
  - \( \text{modified}(l) := 0 \).

- When modifying data which is in cache line \( l \):
  - \( \text{modified}(l) := 1 \).
  - Main memory remains unchanged.

- When replacing cache line \( l \):
  - Case \( \text{modified}(l) = 1 \):
    - Content of old cache line must be written back to main memory.
    - If \( \text{modified}(l) = 0 \),
      - Writing back to main memory not required.

---

(c) Stacks

- If we look at a stack of paper:

  we see the following:
  - Only the top sheet is accessible.
  - We can add a sheet on top of it (PUSH \((p)\)).
    Then the new sheet is the top one.
  - we can take a sheet from its top (POP) and use it.
    Then the sheet below it is new top one.

- A stack is now a data structure which is derived from this example.

---

Stack Addressing (Cont.)

- A stack is a finite linearly ordered sequence of elements, of which only the last element can be accessed at a time.

- For the ordering one uses the analogy of above/below/top element:
  - The last element is called the top of the stack.
  - The element immediately before one element is called the element below it.
  - The first element is called the bottom element of the stack.

- Usually we have only the following operations:
  - PUSH, which moves a new element on top of a stack (at the end of it).
  - POP, which removes an element from the top of a stack. Afterwards the element below it is new top element.

---

Stack Addressing (Cont.)

Stacks which are used for arithmetic operations have additionally operations:

- Unary operations, which pop the top element, apply a unary operation to it (like negation, shift operations, bitwise NOT) and push the result on the stack.

- Binary operations, which pop two elements from the stack, apply a binary operation (like addition, multiplication), and push the result on the stack.
  - ADD denotes this operation for addition,
  - MULT for multiplication,
  - SUB for subtraction,
  - DIV for division.
Stack Machines

- Machines built using stacks as main principle.
  - Used in some simple machine (not very common).
- Almost all architectures have a stack for dealing with function and procedure calls (see later).
- Java virtual machine (JVM) is based on a stack machine.
  - JVM is a real machine language, but it is mainly interpreted on other machines.

Stacks and Evaluation of Expressions

- Stacks can be used for evaluating expressions.
- Assume we want to calculate \((2 + 3) \cdot 6\) with a stack.
  Can be done as follows:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Resulting Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH 2</td>
<td>2</td>
</tr>
<tr>
<td>PUSH 3</td>
<td>3 2</td>
</tr>
<tr>
<td>ADD</td>
<td>5</td>
</tr>
<tr>
<td>PUSH 6</td>
<td>6 5</td>
</tr>
<tr>
<td>MULT</td>
<td>30</td>
</tr>
</tbody>
</table>

Stack Implementation

- Usually the items in the stack are stored in some special area of main memory.
- In registers one stores:
  - **Stack Base.** Points to the bottom element of the stack.
  - **Stack Pointer.** Points to the top element. (Problem if the stack is empty; stack should point to an address **below** the stack base!).
    To avoid this problem (eg. if Stack Base = 0), sometimes one has stack pointers pointing to the first address **above the stack**.
  - **Stack Limit.** Points to the topmost element of the stack (alternatively: to the first address not part of the stack).
Variations:
- Stacks might grow downwards instead of upwards — so the element below an element gets a higher rather than a lower memory address.
  * This is the case in the Pentium.
- Stack base and/or stack limit might be hardwired.
- For efficiency reasons one might have registers containing the content of the top element and of the second element of the stack.

Error messages are issued if the CPU wants to
- Pop an element from an empty stack.
- Push an element on a stack which is full.
  (This often indicates that one has a procedure which is infinitely often calling itself).

The next two slides show two implementations of a stack

Example of Stackimplementation
- In the following we follow the implementation on slide 6-75.
- We assume as base the address 0x1000, as limit the address 0x2000.
- We use an implementation where the pointer points to the top element.
  - If the stack is empty, the pointer will point to the first address below the beginning of the stack i.e. to address 0x0FFF.
After Execution of **PUSH 0x0002**

Registers:

- 0x2000
- 0x0002
- 0x1000
- 0x0100
- 0x2000
- 0x0003
- 0x0002

Memory:

- undefined
- undefined
- undefined
- undefined
- undefined
- undefined
- undefined

Result returned: 0x0003.

After Execution of **POP**

Result returned: 0x0003.

Registers:

- 0x2000
- 0x0002
- 0x1000
- 0x0100
- 0x2000
- 0x0003
- 0x0002

Memory:

- undefined
- undefined
- undefined
- undefined
- undefined
- undefined
- undefined
Supplementary Material for Sec. 6.

(i) Organization of Main Memory.
(ii) Cache Coherence

Example: 16-Mbit DRAM (next slide but one).
- Organized as four arrays of
  \[2048 \cdot 2048 = 2^{11} \cdot 2^{11}\] bits each.
- Requires two 11-bit addresses to access a single bit.
- Organized in rows and columns.
- Address is sent multiplexed, ie. sent in two cycles:
  - First an 11 bit address is sent to identify a row.
  - Then an 11 bit address is sent to identify the column.
- Whether address identifies row or column done via signals:
  - Row address select (RAS) and
  - Column address select (CAS).

Two extremes:
- Addressable up to to word level. Cheaper, more dense, higher access time.
- Addressable up to bit level.

Refresh circuity:
- When refresh is done,
  * Address buffers are disabled,
  * Refresh counter steps through all rows.
  * For each row, the complete column is refreshed in one step.
Next but one slide: EPROM and DRAM chips.

- A0 - A19: Address lines.
- D0- D7: data lines.
- Vcc: Power supply.
- Vss: Ground pin.
- CE (chip enable): indicates, whether address is valid for the chip or not. (since there are more memory chips in use).
- Vpp: program voltage used during programming (write operation, for EPROM).

- WE: Write enable (for DRAM chip).
- OE: Output enable (for DRAM chip).
- RAS: row access select (for DRAM chip).
- CAS: column access select (for DRAM chip).
- NC: not connected (so that pins of the chip match pins of the socket).
Module Organization:
- Next slide: Organization of 256 KByte memory, using eight 256 kilobit chips.
- Each chip stores 1 bit of the word.
- (Word length = 8 bit = 1 byte).

(ii) Cache Coherence

- **Problem** when several processors (or I/O with access to memory) share portions of main memory.
  - **Problem 1**: Other cache might contain this block. Content no longer correct.
  - **Problem 2**: Other cache might load later this block from main memory. But content in memory no longer correct.

**Problems**, One Master modifies block in its cache.

- **Problem 1**: Other cache might contain this block. Content no longer correct.
- **Problem 2**: Other cache might load later this block from main memory. But content in memory no longer correct.

Problems of **Cache Coherence**.

**Approaches for achieving Cache Coherency**

- **Bus watch with write through.**
  - Write through policy for all caches. Solves Probl. 2.
  - Cache of each processor watches address bus for writes to shared memory. If cache contains address written to it, invalidate cache line. Solves Probl. 1.

- **Non-cacheable memory.**
  - Forbid cache of a processor to store memory shared by other processors.

- **Hardware transparency.**
  - Extra hardware ensures that writes to a cache are mirrored in main memory (solves Probl. 2) and other caches holding that line. (Solves Probl. 1).

- Use of special protocols like the **MESI-Protocol** (see supplementary material).
The MESI Protocol

- **MESI** = **Modified-Exclusive-Shared-Invalid**

- One of 4 states associated with every cache line:
  - **Modified**: Line in cache has been modified (different from main memory), available only in this cache.
  - **Exclusive**: Line in cache is the same as that in main memory, and is not present in any other cache.
  - **Shared**: The line in cache is the same as that in main memory, and may be present in another cache.
  - **Invalid**: The line in cache does not contain valid data.

- Let in the following master be any unit which can request access to memory via the cache.

Coherence Conditions, MESI Protocol

We will consider invalid lines as non existent.
MEIS ensures that at any block in main memory exactly one of the following 4 cases holds:

- **Case 1**: Block corresponds to no cache line:

  - Main memory: X
  - Cache A, B, C: 

- **Case 2**: Block corresponds to exactly one cache line,
  - which has state exclusive,
  - and content there is identical with main memory:

  - Main memory: X
  - Cache A: X
  - Cache B: 
  - Cache C: 

  exclusive

- **Case 3**: It corresponds to exactly one cache line,
  - which has state modified:

  - Main memory: X
  - Cache A: Y
  - Cache B: 
  - Cache C: 

  modified

- **Case 4**: It corresponds to lines in several caches,
  - all of which have state shared,
  - all contents of those lines coincide with main memory.

  - Main memory: X
  - Cache A: X
  - Cache B: X
  - Cache C: X

  shared shared

Initial Case (MESI):

- All cache lines obtain state invalid.
Assume master requests data from main memory.

- Case address corresponds to no cache line:

  - Block corresponding to that address is loaded into cache and
  - passed through to master.
  - Cache line obtains state exclusive.

- Case address corresponds to one line in current cache.

  - Data is read from cache.
  - State of cache line remains unchanged.

- Case address corresponds to no line in current cache, but to exclusive or shared line in some other cache.

  - Data in main memory is valid.
  - Cache reads block from memory.
  - All cache lines (including the current one) corresponding to that address obtain status shared.

  (Picture refers to shared/shared example above.)
Write Command (MESI)

Assume master writes data to memory.
- Case address corresponds to no cache line.

\[
\begin{array}{cccc}
\text{Main memory} & \text{Cache A} & \text{Cache B} & \text{Cache C} \\
X & & & \\
\end{array}
\]

- Data from memory is written into cache,
- data to be changed is modified,
- line obtains state modified.

\[
\begin{array}{cccc}
\text{Main memory} & \text{Cache A} & \text{Cache B} & \text{Cache C} \\
X & Y & & \\
\end{array}
\]

modified

(Alternatively write data only to main memory and not to any cache line).

Write Command (MESI; Cont.)

- Case address corresponds to line in current cache.
- Data is written to current cache,
- line obtains state modified.
- Other corresponding cache lines (if line was shared) obtain state invalid.

\[
\begin{array}{cccc}
\text{Main memory} & \text{Cache A} & \text{Cache B} & \text{Cache C} \\
X & Y & & \\
\end{array}
\]

modified

Write Command (MESI; Cont.)

- Case address corresponds to no line in current cache,
- but to a modified line in some other cache.

\[
\begin{array}{cccc}
\text{Main memory} & \text{Cache A} & \text{Cache B} & \text{Cache C} \\
X & & & \\
\end{array}
\]

modified

- The other cache writes its block back to main memory,
- sets state of its cache line to invalid.
- Current cache loads data from main memory,
- modifies data in its cache,
- state of line is now modified.

\[
\begin{array}{cccc}
\text{Main memory} & \text{Cache A} & \text{Cache B} & \text{Cache C} \\
Step 1,2: Y & Y & Y & \\
\end{array}
\]

Step 3:
\[
\begin{array}{cccc}
& Z & & \\
\end{array}
\]

modified invalid
7. External Memory

(a) Magnetic Disk.
(b) RAID.
(c) Magnetic Tape.
(d) Optical Drives.

(a) Magnetic Disk

- Data recorded on the surface of a disk as magnetic impulses.
- Head of disk drive = solenoid
  - which serves as an electromagnet.
- By changing the polarity of voltage, electromagnet changes polarity of magnetic field.
  - Magnetic field with opposite polarity of head imprinted on disk.
  - Different polarities used for storing 0 or 1.
  - Information from disk loaded by passing solenoid over disk.
    * Magnetization induces electrical impulses in the head.
  - Data organized in circular tracks on the disk.
- Nowadays usually sealed, since small articles could cause catastrophic damage (disk crash).

Fixed Head Disks

(One head per track)

Movable Head

- One head only, moves over tracks.
- Access time higher, since head has to move first to find the track.
Data Organization

- Circular tracks over disk,
  - each divided into sectors.

- Inter-track gaps between tracks, inter-sector gaps between sectors.

- Disk rotated with constant angular velocity (CAV)
  (eg. the rotational speed of the disk, measured in rotations per second, is constant).

- Outer tracks of disk store same quantity of information as inner tracks – outer tracks less dense than inner tracks.

- Additional information stored in a sector to allow the disk drive to
  - identify track and sector under head;
  - align data reads and writes;
  - carry out error detections.

Different Kinds of Disks

- Single vs. double sided.

- Removable (eg. floppy disk) vs. non-removable (eg. hard disk).

- Single platters (one disk) vs. multiple platters (see next slide)

- Different head mechanisms:
  - Contact between disk and head (eg. floppy disk).
  - Fixed Gap (Hard Drives)
  - Aerodynamic Gap (Winchester drives):
    Head rests on disk when disk is motionless. Head is lifted by air pressure generated by the spinning disk.
    Advantage: head closer to disk than with fixed gap, therefore higher density of data possible.

Multi-Platter Disk
(b) RAID

- **RAID** = Redundant Array of Independent Disks.
  - Standardized scheme for multiple-disk design.

- 6 different levels.

- Principle characteristics:
  - RAID is a set of physical drives observed as one logical drive.
  - Data is distributed across the physical drives.
  - Except for RAID 0, parity information or backups used in order recover information on disk in case of failure.

- Performance will be assessed by
  - I/O request rate (both read and write).
    * Many requests, small amount of data per request.
    * Response time should be as small as possible.
  - Data transfer rate (both read and write).
    * Few requests with large amounts of data each.

Picture copied from book not included

Computer Systems, CS_M33, Michaelmas term 2002, Sect. 7  
7-9

### RAID 0

- No parity information stored.

- Data striped across the disks.

- Transfer of contiguous blocks of data can be done in parallel by accessing strips on different disks simultaneously.

- Performance:
  - Excellent I/O request rate for small strips.
  - Excellent data transfer rate for large strips.

- Used for applications
  - requiring high performance.
  - with non-critical data.

Computer Systems, CS_M33, Michaelmas term 2002, Sect. 7  
7-11

### RAID 1

- As RAID 0, but a second copy held of each disk.

- Reading requests use copy with shortest access time.

- Writing requests executed as fast as the slowest mirror.

- Performance:
  - Good/fair I/O request rate. Excellent if mainly read requests.
  - Fair data transfer rate. Excellent if mainly read requests.

- Suitable if immediate recovering of data in case of disk failure necessary.

- Typical application:
  - System drives
  - Critical data

Computer Systems, CS_M33, Michaelmas term 2002, Sect. 7  
7-12
**RAID 2**

- Data strips are small (often a single bit or a word).
- Usually all disks synchronized. Parallel access possible.
- Extra disks provided to store error detecting codes (Hamming codes). These codes allow to detect errors and recover data, if the number of wrong bits is not too big. (In the figure, $f_0(b)$, $f_1(b)$, $f_2(b)$ are parity bits for $b_0$, $b_1$, $b_2$, $b_3$ together.)
- Performance:
  - Poor I/O request rate (No possibility to deal with independent requests in parallel).
  - Excellent data transfer rate. (Single requests can be dealt with in parallel).
- Good if disks are highly unreliable. Because of good quality of disks, RAID 2 currently not used.

**RAID 3**

- As RAID 2, but only one disk with parity bits.
- Error of one strip, and therefore failure of one disk can be recovered.
- Performance:
  - One single I/O request can be treated with parallel access to all disks, therefore high performance. However, only one I/O request can be treated at a time.
    - Poor I/O request rate.
    - Excellent data transfer rate.
  - Typical application:
    - Applications with large amount of data per request (e.g., imaging, CAD).

**RAID 4**

- Large strips.
- Disks operated independently (good for dealing in parallel with high I/O requests)
- Parity bits stored on parity disk, recovering of failure of one disk possible.
- Parity disk delays write requests.
- Performance:
  - Excellent/fair I/O request rate.
  - Fair/poor data transfer rate.
  - Because writing is a bottleneck, RAID 4 currently not used.
RAID 5

- As RAID 4, but parity strips distributed across the disks:
  - For $n$-disk array, parity strips on a different disk for the first $n$ strips,
  - then again for the $n + 1$th to $2n + n$th strip etc.
- Therefore problem from RAID 4 reduced.
- Performance:
  - Excellent/fair I/O request rate.
  - Fair/poor data transfer rate.
- Applications:
  - High request rates, read intensive, data lookup

RAID 6

- As RAID 4, but two different parity calculations per strip, which are:
  - stored on different disks (distributed similarly as before.
  - (In the illustration written as P and Q).
- Therefore even failure of two disks can be recovered.
- However, for having logically $N$ disks available, $N + 2$ disks needed.
- Performance:
  - Excellent/fair I/O request rate.
  - Fair/poor data transfer rate.
- Applications:
  - Applications which require extremely high availability.
  - (Where failure of a second disk while repairing the failure of one disk is not completed would be fatal).

(c) Magnetic Tape

- Tape surface divided into tracks.
- Traditionally 9 tracks:
  - First 8 tracks store a byte.
  - Ninth track stores parity.
- Modern tape drives store 2 or 4 bytes on 18 or 36 tracks.
- Data is recorded in blocks, separated by inter-record-gaps.
- Sequential access.
  - Typically for backups and temporary storage of large amounts of data.

Nine-Track Magnetic Disk Format

Picture copied from book not included
(c) Optical Drives

- **CD ROM:**
  - Data stored as pits on the surface of a disk. Imprinted at manufacture time.
  - To read data, laser directed to surface. Reflection of beam alters by presence/absence of pits.
  - **Constant linear velocity used (CLV)**
    (eg. the speed of the track passing by the head is constant, therefore for inner tracks the rotational speed is faster than for outer tracks).
  - Data density per angle on inner and outer areas of the disk different (Different from magnetic disks).
  - Data stored along one spiral, that originates from the centre of the disk.
  - To locate sector:
    * Disk head moves to approximate vicinity.
    * Minor adjustments to find and read desired sector.

---

**Advantages/Disadvantages of CD-ROM**

- **Main advantages of CD ROM:**
  - Huge storage capacity. (774.57 Mbyte = 550 3.25 inch diskettes)
  - Mass production of CDs inexpensive.
  - CD removable.

- **Main disadvantages of CD ROM:**
  - It is read only.
  - Access time much longer than magnetic disks: up to 1/2 seconds.

---

**Digital Video Disk (DVD)**

- As CD-ROM but
  - Bits packed more closely than in CD
  - Two layers of pits one on top of the other.
  - Two-sided.

- Can store 8.5 GB (single sided), 17 GB (double-sided).
**Erasable Optical Disk**

- CD which can be rewritten. (Like EPROM vs. ROM).
- Known as CD-RW.
- High capacity.
- Removable.
- Reliable.
- However, only 500,000 - 1,000,000 erase cycles possible.
- Exists as well for DVD: DVD-RW.

---

**Magneto-optical Disk (MO)**

- Magnetic recording.
- Optical laser used to focus the magnetic recording head, therefore higher capacity than magnetic disks.
- Optical reading. Polarized laser light changes its rotation according to the magnetic field.
- Cheaper per Mbyte than magnetic storage.
- Can be rewritten more often than erasable optical Disk.

---

**8. CPU-Instructions Sets, Addressing Modes**

(a) Basic Types of Instructions.
(b) Basic Form of Instructions.
(c) Translation of Higher Level Instructions into Assembly Language.
(d) Addressing Modes.
(e) Endianess.
(f) Instruction Formats.
(g) Machine Language Instructions for Procedure Calls
(h) Parameters.

---

**Assembly Languages**

- Commands executed by the machine are words in binary.
- Too difficult to program by human beings. Instead use of assembly languages:
  - Like a programming language.
  - But each assembly language instruction corresponds exactly to one machine instruction.
  - Assembly languages are translated by a program, called assembler into machine code.
  - Only small and simple calculations (eg. of addresses) are carried out by the assembler, especially no optimizations.
    - Assembly language code is essentially identical to machine code.
    - Compilation of higher level languages into machine code is more complicated and the programmer has no direct control over, which machine instructions are actually used.
    - When writing time-critical code, assembly language code is usually much faster (typically twice as fast).
Typical Assembly Language Instructions

- Typical instructions in assembly languages have the form:
  \[ \text{INSTR} \ A_1, \ A_2, \ A_3, \ldots \ A_n \]
  where
  - \text{INSTR} is an instruction, typically a word in upper case of 3 or more characters like MOVE, SUB, ADD;
  - \( A_1, \ldots, A_n \) are operands or destinations of the operation.

- For instance \( \text{ADD} \ R_1, R_2, R_3 \) means add contents of registers \( R_2, R_3 \) and store result in \( R_1 \).

(a) Basic Types of Instructions

There are essentially four groups of instructions:

(i) Instructions for data processing.

(ii) Instructions for data transfer.

(iii) Instructions for Input/Output (I/O).

(iv) Control Instructions.

(i) Instructions for Data Processing

- Perform common arithmetic and logic operations, and shift operations (see below).

- On RISC architecture all arithmetic/logic operations have a simple format:
  \[ \text{Name destination, source1, source2} \]
  where destination, source1 and source2 are registers (eg. \( R_3 \)) or constants.

- On CISC machine, much more sophisticated formats available.

- Advantage of RISC instruction set:
  - More uniform execution (same number of cycles per instruction).
  - More uniform and shorter instruction format.
  - Expected to be much faster.

Examples

- \( \text{ADD} \ R_{10}, R_1, \#5 \): \( \Rightarrow R_{10} := R_1 + 5 \).
  Add 5 to content of \( R_1 \), store result in \( R_{10} \).

- \( \text{SUB} \ R_1, R_0, R_3 \): \( \Rightarrow R_1 := R_0 - R_3 \).
  Subtract \( R_3 \) from \( R_0 \), store result in \( R_1 \).

- Similarly \( \text{MULT}, \text{DIV} \) (multiplication, division).

- \( \text{XOR} \ R_0, R_1, \#0xC8 \):
  Bitwise XOR of \( R_1 \) and \( 0xC8 \), store result in \( R_0 \). Eg. bitwise XOR of \( 0b0110 0110 \) and \( 0b1100 1000 \) is:
  \[
  \begin{array}{cccc}
  0b & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
  0b & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
  0b & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
  \end{array}
  \]

- Similarly \( \text{AND}, \text{OR}, \text{NOT} \) (bitwise).
Examples (Cont.)

- Shift left, shift right:
  - 0b0100 0000 shifted right becomes 0b0010 0000.
  - 0b0100 0010 shifted left twice becomes 0b0001 0000
- Arithmetic shift.
- Rotate operations
  (rotate left: new LSB is old MSB;
  rotate right: new MSB is old LSB):
  ROTATELEFT #0b10101 has result 0b01011
  ROTATERIGHT #0b10101 has result 0b11010

(ii) Instructions for Data Transfer

- Movement of data.
- On RISC Machines only these instructions can move data between registers and main memory.
- Two main instructions on RISC
  - Load instructions (from main memory into some register):
    LOAD R20, [15]: Load memory location 15 into register R20.
  - Store instructions (from some register into main memory):
    STORE [12], R1: Store content of register R1 into memory location 12.
- On CISC machines instructions for arbitrary movements, for instance from main memory to main memory; often called MOVE.

Comparison with High-Level Languages

- Above mentioned instructions correspond to high level statements:
  \[
  \begin{align*}
  i &:= 15; \quad \text{STORE} \ R_i, #15 \\
  i &:= j+10; \quad \text{ADD} \ R_i, R_j, #10 \\
  k &:= l-i; \quad \text{SUB} \ R_k, R_l, R_i
  \end{align*}
  \]
- Here we assume that i,j,k,l are stored in register Ri, Rj, Rk, Rl.
- If they are in main memory, they have first to be loaded (or other addressing modes be used).
- More complex statements like
  \[
  i := (j+k) \times l \\
  \]
  have to be decomposed:
  \[
  \begin{align*}
  \text{aux} &:= j+k; \\
  i &:= \text{aux} \times l.
  \end{align*}
  \]

(iii) I/O Instructions

- Transfer data to and from I/O.
- Instructions for control of I/O
  (ie. start/stop execution of I/O).
(iv) **Control Instructions**

- **Unconditional jumps** (goto in high level languages).
- **Branch instructions** (conditional jumps).
- **Conditional skip** of next instruction.
- **Jump to subroutine or procedure** (see later).
- **Return from subroutine** (see later).
- **Stop** program execution.
- **Wait** until condition (clock or I/O) is satisfied.
- **No operation**.

---

**Unconditional Jump**

- Unconditional Jump written as JMP A.
  - Jump to address A. (Corresponds to goto in higher level languages).
  - Address usually indicated by a label in assembly languages, eg.:
    
    ```
    JMP LabelA
    ...
    Label A: Next Instruction to be executed.
    ```

---

**Branch Instructions**

- An instruction like
  ```
  if <condition> then <instruction>
  ```
  requires **three** steps to be carried out:
  - Calculation of `<condition>`.
  - Depending on `<condition>`, either jump at the end of `<instruction>` or not.
  - Execution of `<instruction>` (in case `<condition>` was true).

- For the second step machine languages provide usually **branch instructions**.
  - Typical mnemonic like BEQ (branch if equal), BNEQ (branch if not equal) etc.
  - BEQ A means: branch, if the result of the last calculation was zero, to address A.
    - Address A is given usually in relative addressing (see later).
    - In assembly languages usually given by a label (as for unconditional jumps).
  - The condition, depending on which one branches, is determined by flags.

---

**Flags**

- Flag registers are **one-bit registers**, which are usually set automatically depending on the last arithmetic/logic operation performed.

- **Main flags are**
  - **Z** flag
    - *Zero* – flag.
    - *Set if result (of last arithmetic/logic operation) was zero.*
  - **N** – flag.
    - *Set if result was negative.*
  - **C** – flag.
    - *Set if result had carry/borrow.*
    - (= Over/flow for arithmetic operations on unsigned numbers).
  - **V** – flag.
    - *Set if result gave over/underflow.*
    - for arithmetic operations on signed numbers.
      - (Note that arithmetic operations on signed and unsigned numbers coincide. The difference is how to determine when over/flow occurs).
### Conditional Jumps

- Branch instructions advice to jump, if one flag is set (or not set). Typical branch instructions are
  - **BEQ** labelA
    - Branch, if Z flag is set, to labelA.
    - *(If compared values were equal).*
  - **BNQ** labelA
    - Branch, if Z flag is not set, to labelA.
    - *(If compared values were not equal).*
  - **BRN** labelA
    - Branch if N flag is set to labelA.
    - *(If first compared argument less than the second).*
  - **BRP** labelA
    - Branch if N flag is not set (positive result) to labelA.
    - *(If first compared argument greater or equal to second).*
  - **BCS/BCC** labelA
    - Branch, if C flag is set/not set to labelA.
  - **BVS/BVC** labelA
    - Branch, if V flag is set/not set to labelA.

### Comparison Instructions

- For interpretation of a high-level language instruction
  
  ```
  if r = s then ... else ...
  ```
  one needs to
  - subtract r from s,
  - this sets zero flag, if the result is zero,
  - then depending on this flag make the decision.
  - Result of subtraction can be **thrown away**.

- Therefore **CMP** instruction:
  - **CMP** R1,R2
    - calculates R1 - R2,
    - but throws result away.
    - Only result of flags relevant.
  - So if R1 < R2, then the negative flag is set, otherwise not.
    - If R1 = R2, then the zero flag is set, otherwise not.

- **CMP** is a **data processing instruction**.

### Conditional Jumps and “Spaghetti-Code-If”

- The two instructions:
  
  ```
  CMP R0,R1
  BEQ label1
  ```

  correspond to the “Spaghetti-Code”-Instruction:
  
  ```
  if R0 = R1 then goto label1.
  ```

- The two instructions:
  
  ```
  CMP R0,R1
  BNQ label1
  ```

  correspond to the “Spaghetti-Code” Instruction:
  
  ```
  if R0 ≠ R1 then goto label1
  ```
(b) Basic Form of Instructions.

- An instruction consists of an operation and operands.
- The CPU needs to obtain up to four kinds of informations from the instruction:
  - The operation the CPU has to perform:
    * Which instruction,
    * including size of the data (eg. bit, byte, word, double word),
    * format of data (eg. unsigned integer, floating-point etc).
  - The source of the operation:
    Not needed in case of unconditional and flag-dependent control operations,
  - The destination of the operation:
    Not needed in case of unconditional and flag-dependent control operations,
  - The location of the next instruction to be fetched.
    Usually the instruction following the current one, except for control instructions.

Number of Addresses

- Most arithmetic and logic functions have one or two arguments.
- Together with the destination, this would require two or three addresses.
- Four addresses needed if we have an address of the next instruction (rarely used).
- Formats used where one or more addresses can be omitted.
  (Register numbers considered here as an address as well).
  This is done in order to save space needed for saving operations.
  (Increases speed, since fetching of instructions requires time).

Basic Form of Instructions. (Cont.)

- The destination of the operation.
  Not needed in case of unconditional and flag-dependent control operations,
- The location of the next instruction to be fetched.
  Usually the instruction following the current one, except for control instructions.

Reduction of Number of Addresses

In case of functions with two arguments we could have:

- Three addresses.
  Eg. SUB A,B,C expresses: A := B-C.

- Two addresses.
  Eg. SUB A,B expresses: A := A-B.

- One address.
  Used if one has one specific register in the CPU, called accumulator, AC.
  Eg. SUB A expresses: AC := AC - A.

- Zero address.
  Used if one has one specific stack.
  SUB expresses:
  - Pop two uppermost elements from stack,
  - subtract first popped from the second popped,
  - push result on the stack.
  (Push and Pop however require at least one address).
Example: \( A = \frac{(B+C)}{(D-(E\cdot F))} \)

### With three-address instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Interpretation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD A,B,C</td>
<td>A:= B + C</td>
<td></td>
</tr>
<tr>
<td>MULT Z,E,F</td>
<td>Z:= E \cdot F</td>
<td>Z= E\cdot F</td>
</tr>
<tr>
<td>SUB Z,D,Z</td>
<td>Z:= D-Z</td>
<td>Z= D-(E\cdot F)</td>
</tr>
<tr>
<td>DIV A,A,Z</td>
<td>A:= A/Z</td>
<td></td>
</tr>
</tbody>
</table>

### With two-address instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Interpretation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD A,B</td>
<td>A:= B</td>
<td></td>
</tr>
<tr>
<td>ADD A,C</td>
<td>A:= A + C</td>
<td>A= B + C</td>
</tr>
<tr>
<td>LOAD Z,E</td>
<td>Z:= E</td>
<td></td>
</tr>
<tr>
<td>MULT Z,F</td>
<td>Z:= Z \cdot F</td>
<td>Z= E\cdot F</td>
</tr>
<tr>
<td>SUB U,D</td>
<td>U:= D</td>
<td></td>
</tr>
<tr>
<td>SUB U,Z</td>
<td>U:= U-Z</td>
<td>U = D -</td>
</tr>
<tr>
<td>DIV A,U</td>
<td>A:= A/U</td>
<td>A= (B+C)/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(D-(E\cdot F))</td>
</tr>
</tbody>
</table>

### With one-address instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Interpretation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD B</td>
<td>AC:= B</td>
<td></td>
</tr>
<tr>
<td>ADD C</td>
<td>AC:= AC + C</td>
<td>AC = B+C</td>
</tr>
<tr>
<td>STORE A</td>
<td>A:= AC</td>
<td>A = B+C</td>
</tr>
<tr>
<td>LOAD E</td>
<td>AC:= E</td>
<td></td>
</tr>
<tr>
<td>MULT F</td>
<td>AC:= AC \cdot F</td>
<td>AC = E\cdot F</td>
</tr>
<tr>
<td>STORE Z</td>
<td>Z:=AC</td>
<td>Z = E\cdot F</td>
</tr>
<tr>
<td>LOAD D</td>
<td>AC:= D</td>
<td></td>
</tr>
<tr>
<td>SUB Z</td>
<td>AC:= AC - Z</td>
<td>AC = D - (E\cdot F)</td>
</tr>
<tr>
<td>STORE Z</td>
<td>Z:= AC</td>
<td>Z = D - (E\cdot F)</td>
</tr>
<tr>
<td>LOAD A</td>
<td>AC:= A</td>
<td>AC = B+C</td>
</tr>
<tr>
<td>DIV Z</td>
<td>AC:= AC/Z</td>
<td>AC = (B+C)/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(D-(E\cdot F))</td>
</tr>
<tr>
<td>STORE A</td>
<td>A:= AC</td>
<td>A = (B+C)/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(D-(E\cdot F))</td>
</tr>
</tbody>
</table>

### With zero-address instructions (except for \textit{PUSH} and \textit{POP}):

- **PUSH** A means push content at memory location A on stack.
- **POP** A means pop uppermost element from stack and store it in A.

<table>
<thead>
<tr>
<th>Instruction Stack (rightmost = top element)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH B</td>
</tr>
<tr>
<td>PUSH C</td>
</tr>
<tr>
<td>ADD</td>
</tr>
<tr>
<td>PUSH D</td>
</tr>
<tr>
<td>PUSH E</td>
</tr>
<tr>
<td>PUSH F</td>
</tr>
<tr>
<td>MULT</td>
</tr>
<tr>
<td>SUB</td>
</tr>
<tr>
<td>DIV</td>
</tr>
<tr>
<td>POP A</td>
</tr>
</tbody>
</table>
When translating high level commands into machine language, one should usually not make conversions like
- replacing $A + (B + C)$ by $(A + B) + C$.
- replacing $A \cdot (B + C)$ by $A \cdot B + A \cdot C$.

Since we cannot represent floating-point numbers precisely, floating-point arithmetic does not follow those laws.

(c) Translation of High-Level Instructions into Assembly Language

Interpretation of If-Clauses

- Interpretation of
  
  if $x=y$ then begin $<$ifAction$>$ end;
  $<$actionInAllCases$>$
  
  - Assume $x, y$ in registers $R0, R1$.
  - First we translate the statement into “Spaghetti-code”:
    
    if $x \neq y$ then goto Label1
    $<$action$>$
    
    Label1: $<$actionInAllCases$>$
  
  - Replace the if-statement by the instructions
    
    CMP R0, R1
    BNQ Label1
    
  - We obtain as assembly code:
    
    CMP R0 R1
    BNQ Label1
    $<$action$>$
    
    Label1: $<$actionInAllCases$>$

Translation of If-Then-Else

- Translation of
  
  if $x=y$ then begin $<$ifAction$>$ end;
  else begin $<$elseAction$>$ end;
  $<$actionInAllCases$>$
  
  - Assume $x, y$ in registers $R0, R1$.
  - First we translate the statement into “Spaghetti-code”:
    
    if $x \neq y$ then goto Label1
    $<$ifAction$>$
    goto Label2
    
    Label1: $<$elseAction$>$
    Label2: $<$actionInAllCases$>$
  
  - Note that the goto-statement is necessary.
  - Then we replace the if-statement by the instructions:
    
    CMP R0, R1
    BNQ Label1
    $<$action$>$
    
    Label1: $<$actionInAllCases$>$

  and goto by JMP
Translation of If-Then-Else (Cont.)

- We obtain as assembly code:

  ```assembly
  CMP R0,R1
  BNQ Label1
  <ifAction>
  JMP Label2
  Label1: <elseAction>
  Label2: <actionInAllCases>
  ```

- **Crucial:** The JMP instruction (in boldface). Frequent Mistake to omit it.

- Note as well that **no JMP** at the end of `elseAction` required.

Interpretation of a While Loop

- Interpretation of

  while x ≠ 0 do begin `<whileAction>` end;
  `<actionAfterwards>`

  - Assume `x` in register `R0`.
  - First we translate the statement into “Spaghetti-code”:
    ```assembly
    Label1: if x=0 then goto Label2
    <whileAction>
    goto Label1
    Label2: <actionAfterwards>
    ```
  - Note that the goto-statement is necessary.
  - Then we replace the if-statement by the instructions:
    ```assembly
    CMP R0,R1
    BEQ Label2
    <whileAction>
    ```
    ```assembly
    Label2: <actionAfterwards>
    ```
  - Further replace goto by JMP.

Optimization (While Loop)

- Body of while/for/repeat loops are usually the most time consuming parts of a program.
  - Might be executed for instance a million times whereas the initialization is only executed once.
  \[ ⇒ \text{worth to safe every instruction}. \]

- If we check the condition at the end and then use the branch as a jump we can safe one instruction.

- We have only initially to jump to the test so that, in case initially the condition is not fulfilled, the while-loop is never executed.

- The result is as follows:

  ```assembly
  JMP Label2
  Label1 <whileAction>
  Label2 CMP R0,#0
  BNQ Label1
  <actionAfterwards>
  ```
(d) Addressing Modes

- Question:
  - Where is the source and target of an operation located?
    - E.g. in an instruction
      ADD A, B, C
      What does a hexadecimal value A mean:
      - a constant,
      - a memory address,
      - a register number?
  - Where is the next instruction (in case of jump, branch)?

Address vs. Operand

- When we want to address the target of an operation (where the result of an operation is to be stored), what matters is the effective address (EA), i.e. the address in main memory or the register number, where the data is to be stored.
  - EA might be a physical address. When paging is used (see later), it is a virtual address, which is then mapped by a combination of hardware/software to some physical address.

- When we want to use the addressing mode to determine in case of branch or jump the address of the next instruction to be executed, again, what matters is the effective address, since the PC is to be updated to this address.

- However, when we want to address the source of an operation (arithmetic/logic operation, the data to be stored or loaded), what matters is the operand, which is the content of the data at the EA.

Remark on Example Instructions

- Use of pseudo assembler code (like ADD IMMEDIATE) in the following.

- In real assembler code, abbreviations used which subsume both operation and addressing mode.

- Main example will add one operand to the Accumulator (AC).
  - Not realistic (since in complex architectures, we usually don’t have a specific register AC).
  - Only for simplification.
  - Usually second operand arbitrary register content or constant, but complicated addressing modes might as well be possible.

Syntax for Denoting Addressing Modes

In this module we use the following notation:

- When register numbers are meant, we write expressions R#, R1#, R2# etc. This will be used as well for registers with special names like SR#.

- When writing expressions like R, R1, the content of the register is meant. So \( R := R + 1 \) means that the content of register \( R \) is increased by one.

- The previous two notations are specific for this module only. (Standard notations not used in a systematic way).
Syntax for Denoting Addressing Modes (Cont.)

- The following is standard:
  - When we want to refer to the content of a register *given by its number*, we put brackets around it and write for instance $\langle R1\# \rangle$.
    - So $\langle R1\# \rangle$ and $R1$ mean the same.
  - If $A$ is an address in main memory, $\langle A \rangle$ is the content of memory location $A$.
  - More complex expressions are defined by iterating this syntax:
    - E.g. $\langle \langle R1\# \rangle \rangle$ is the content of memory location at the address which is the content of register number $R\#$.
    - $\langle \langle \langle R1\# \rangle \rangle \rangle$ is the content at memory location given by the previous expression.

Main Addressing Modes

(i) Immediate Addressing.

(ii) Direct Addressing.

(iii) Indirect Addressing.

(iv) Register Addressing.

(v) Register Indirect Addressing.

(vi) Displacement Addressing.

(vii) Stack Addressing.

(viii) More Advanced Addressing Modes.

(i) Immediate Addressing

- Address = Operand
  (So operand is a constant, ie. a degenerated address.
  This mode cannot be used for the target of a store/source of a load operation or for denoting jump/branch addresses).

```
 Opcode  Address Field
        A

Operand
```

(Opcode specifies the instruction to be executed, eg. “ADD”).

- Notation: OPERAND = A.

- ADD IMMEDIATE 17 expresses:
  \[ AC := AC + 17. \]

Notations

- We write ADD #17 for ADD IMMEDIATE 17.
- By default, 17 means (unless specified differently) decimal number 17.
  For denoting hexadecimal 17, we write 0x17.
Advantages/Disadvantages of Immediate Addressing

- **Fast** (not even necessary to look up in memory).
  - However, when loading the instruction code, the address has to be looked up in main memory.

- Good for dealing with **constants** in programming languages (especially small constants like 2 in `i := i+2`).

**Limitations:**
- Since both operation and addresses have to be stored in the instruction code
  - either **limited range of data** to be stored in the instruction,
    - (both number and complete instructions stored as one word)
  - or **two or more words per instruction** needed.
    - Loading of these words requires extra time.

Advantages/Disadvantages of Direct Addressing

- **EA** is the Effective Address.

- **This might be the physical address.**

- or might be a **virtual address**, which is then translated using paging tables or other mechanisms to a physical address. (Invisible to the programmer; see later).

(ii) Direct Addressing

- Address = actual memory address.

**Opcode Address Field**

```
Address Field

Opcode

A

Main Memory

Content

Memory Address

A

C

(Operand is C)
```

- **EA** = A (Effective Address = A).

- **Operand** = (A).

- **ADD DIRECT 0x17** means:
  - \( AC := AC + (0x17) \)
  - Sometimes written as \( AC \leftarrow AC + (0x17) \).

Advantages/Disadvantages of Direct Addressing

- **Fast**, since **no calculation** of memory address required.
  - Suitable for reference to **scalar variables** in high level programming languages.

- **However**, range of address space **limited** or **longer instruction codes required**, similarly as for immediate addressing.
(iii) Indirect Addressing

- If address in instruction code is \( A \), effective address calculated as follows:
  - Look up content \( B \) at memory location with address \( A \).
  - Effective address is \( B \).

\[
\text{Opcode} \quad \text{Address Field} \\
\begin{array}{c}
A \\
\end{array}
\]

\begin{array}{c}
\text{Main Memory} \\
A \\
B \\
C \\
\end{array}

\begin{array}{c}
\text{Memory addresses} \\
\end{array}

\begin{array}{c}
\text{Memory Content} \\
\end{array}

\begin{array}{c}
\text{(Operand is C)} \\
\end{array}

- \( EA = (A) \)  
  (\( EA = \) content of memory at address \( A \)).

- Operand = \((A)\).

ADD INDIRECT 0x17 means:
\[
AC := AC + \text{content at memory location given at memory location 0x17,} \\
AC := AC + ((0x17))
\]

Suitable for looking up pointer references  
(the pointer variable provides an address at which the content the pointer is pointing to can be looked up).

Suitable for looking up variables in a lookup table.

Full range of addresses available for the address \( B \) looked up.

However, looking up of addresses is time consuming.

(iv) Register Addressing

- Address is number of a register:

\[
\text{Opcode} \quad \text{Address Field} \\
\begin{array}{c}
R\# \\
\end{array}
\]

\begin{array}{c}
\text{Registers} \\
A \\
\end{array}

(Operand is \( A \))

- Register number \( R\# \) acts as an address (referring to the register unit however).

- Operand = \((R\#)\).

ADD REGISTER 17 means:
\[
AC := AC + \text{content of register 17,} \\
AC := AC + R17 \\
\text{(or } AC := AC + (R17\#))
\]

Operand can be fetched very fast.

Very small address space needed, so space available for complicated opcodes or multiple addresses.

RISC machines are optimized to use registers as much as possible. Arithmetic operations on RISC machines usually use this addressing mode only.

Suitable for simple variables (i.e. no arrays, string etc.) which are used frequently. Eg. \( i \) in for \( i=1 \) to \( 10 \) do \( a[i]:= a[i+20] \).

Limitation: Number of registers limited.
Register Indirect Addressing

- Effective Address is content of a register:
  
  \[ EA = (R\#) \]
  
  (EA = content of register with number R#).

- Operand = \((R\#)\).

**Opcode Address Field**

- R#

**Main Memory**

- A

**Registers**

- R#

- A

  (Operand is B)

Example

- Assembler program which computes the sum of elements in an array with 100 integer variables (enumerated from 0 to 99).

  In high level language the program is as follows:
  
  \[
  \text{sum} := 0; \\
  \text{for i=0 to 99 do sum:= sum + a[i];}
  \]

  Translated into spaghetti code it reads:
  
  \[
  \text{sum := 0; } \\
  \text{i := 0; } \\
  \text{limit := 100; } \\
  \text{L1: sum:= sum + a[i]; } \\
  \text{i:= i+1; } \\
  \text{if i ≠ limit goto L1;}
  \]

  Assume now
  
  - i not used afterwards,
  - each element of \(a[i]\) requires 4 bytes,
  - addresses refer to bytes.

Example (Cont.)

- Store
  
  - sum in register R1;
  - i in register R2;
  - index pointing to \(a[i]\) in R3;
    * initially \(R3 = A\), where \(A\) is the address of \(a[0]\);
    * in each step \(R3\) increased by 4;
    * \(a[i]\) is now accessed as \((R3)\) (short for \(\text{REGISTER INDIRECT 3}\)).
  - limit stored in register R4.
Example (Cont.)

LOAD R1,#0 ; R1 := 0
; R1 stores sum.
LOAD R2,#0 ; R2 stores array index \( i \).
LOAD R3,#A ; R3 := A, 
; A = address of \( a[0] \)
; available at compile time.
; R3 is pointer to \( a[i] \)
LOAD R4,#100 ; R4 stores \( 99 + 1 \)
L1: ADD R1,(R3) ; R1 := R1 + \( a[i] \)
; \( \text{sum} := \text{sum} + a[i] \)
ADD R2,#1 ; \( i := i + 1 \)
ADD R3,#4 ; Updating of pointer
; to \( a[i] \)
CMP R2,R4
BNE L1 ; Branch if \( R2 \neq R4 \)
; to L1

Optimization

- Index \( R2 \) is not really needed.
- Instead use \( R3 \).
  - When \( R3 \) has reached address of \( a[100] \), the program stops.
- Let \( A' = A + 400 = \text{address of } a[100] \).

LOAD R1,#0 ; R1 = sum
LOAD R3,#A ; R3 is pointer to \( a[i] \)
LOAD R4,#A' ;
L1: ADD R1,(R3) ; R1 := R1 + (R3)
; \( \text{sum} := \text{sum} + a[i] \)
ADD R3,#4 ; Increment pointer to \( a[i] \)
CMP R3,R4
BNE L1 ; Branch if \( R3 \neq R4 \)
; to L1

(vi) Displacement Addressing

- Address calculated by adding to one address content contained at another address.
- Five main variants:
  (vi.1) Relative Addressing
  (vi.2) Base-Register Displacement Addressing.
  (vi.3) Index Displacement Addressing.
  (vi.4) Post-Indexed Indirect Addressing
  (vi.5) Pre-Indexed Indirect Addressing.

(vi.1) Relative Addressing

- Effective Address is result of addition of content of address field to the program counter.
- Content will be considered as a signed number: both backwards and forwards references possible.

Opcode | Address Field
\[ \text{(EA = PC + A)} \]

Main Memory

\[ \text{(EA = program counter + A).} \]
Relative Addressing (Cont.)

- Main addressing mode used for jump addresses (conditional, unconditional and subroutine jumps).
  - Whereas full jump address requires many bits, usually the distance to it from current address is small and requires only few bits.
  - However, one addition necessary.
    * This is still faster than using direct addressing, since having to load longer instructions from main memory takes longer than performing this addition.
    * Further, one can reach addresses, which are unreachable using direct addressing (unless the direct address has the same length as ordinary addresses).
- JMP REL DISPL +0x10 means: jump 16 addresses forward.
- JMP REL DISPL -0x10 means: jump 16 addresses backward.

Since this addressing mode is usually only used for determining jump/branch addresses, the operand (B in the last picture) is not important (it’s a code for the next instruction to be executed), only the EA (PC + A) matters. The new PC value is, if the branch/jump is taken, the EA.

Relative Addressing (Cont.)

- Often, while fetching an instruction, the PC is already incremented, so that it contains the address of the instruction following the current one.
  - Then the next instruction can be already fetched and decoded.
  - However, in this case the displacement in rel. addressing has to be added to the address of the next instruction.
  - Whether this takes place is architecture dependent.

(vi.2) Base-Register Displacement Addressing

- Effective address is result of addition of an offset given in the address to the contents of a register.
- Two addresses needed:
  - immediately given offset,
  - number of the register.

Base-Register Displacement Addressing (Cont.)

Address

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Address Field 1</th>
<th>Address Field 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R#</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

Main Memory

C

B+A

Registers

R#

B

(Operand is C)

Register number

- \( EA = (R#) + A \)
  (EA = content of register with number R# plus offset A).
- \( \text{Operand} = ((R#) + A). \)
Base-Register Displacement Addressing (Cont.)

- Used for segmentation:
  **Segmentation**: Several processes (programs) run on the same machine interleaved.

- Variables used in these processes organized in memory segments.

- Segments are allocated dynamically at run time, when the process is created (or requires more memory).

- Use of base-register displacement addressing:
  - Store in \((R\#)\) the beginning of the address of a segment, known at run time.
  - \(A\) contains the at compile time known offset.

- Usually \((R\#)\) is considered as a **long** address (eg. 32 bit) whereas \(A\) is a **small** displacement (eg. 8 bit) within the segment.

- Register might be **implicit** (a standard register for this mode, eg. segment register SR in the Intel instruction set).

\[\text{Computer Systems, CS M33, Michaelmas term 2002, Sect. 8} \]

**Base-Register Displacement Addressing (Cont.)**

\[\text{Offset known at compile time}\]

\[\text{Segment}\]

\[\text{next Segment}\]

\[\text{Beginning of Segment (Runtime)}\]

\[\text{previous Segment}\]

\[\text{Computer Systems, CS M33, Michaelmas term 2002, Sect. 8} \]

\[\text{8-59} \]

\[\text{8-60} \]

\[\text{8-61} \]

\[\text{8-62} \]

\[\text{(vi.3) Index Displacement Addressing}\]

- Similar formula as before:
  \[\text{EA} = A + (R\#).\]
  But now
  - \(A\) is a possibly **long** starting address (eg. 32 bit number).
  - \((R\#)\) is a **small** offset (eg. 8 bit number),

- Used for **arrays**.
  A contains starting address of the array, \(R\) the displacement.
Auto Incrementing

- Index displacement addressing often combined with auto increment:

**Post-increment** After carrying out the address calculation, as part of the instruction the register is incremented by one:

\[ EA = A + (R\#) \]
\[ (R\#) := (R\#) + 1 \]

**Post-decrement**:

\[ EA = A + (R\#) \]
\[ (R\#) := (R\#) - 1 \]

**Pre-increment** Register incremented before calculating the address:

\[ (R\#) := (R\#) + 1 \]
\[ EA = A + (R\#) \]

Auto Incrementing (Cont.)

- Similarly **Pre-decrement**.

- Useful when iterating over all elements of an array.

- Even higher post/pre/in/decrement is used (for instance by 2, 4, 8).

- All the above used for loops through an array, eg.

\[
\text{for } i=1 \text{ to } 10000 \text{ do } a[i] := a[i] * 32
\]

- But less efficient then using register indirect addressing.

Post-Indexed Indirect Addressing

- Effective Address is result of addition of the content of a register to the content of an address given in the address field.

- Of limited use.

**Post-Indexed Indirect Addressing (Cont.)**

- Used for process control blocks, PCBs (which provide control information for one individual process, like priority or time it was started)

- A contains a pointer to the beginning of the PCB of the current process.

- So when switching to another process, \((A)\) will be modified.

- Process control is not carried out often, therefore one does not use a register for storing \((A)\).

- \((R\#)\) contains the displacement within that block, which allows to access individual fields in the PCB.
Post-Indexed Indirect Addressing (Cont.)

- Used for multiway branch tables:
  - \( A \) contains the address of the beginning of a branch table.
  - Branch table contains addresses, to one of which one jumps.
  - Depending on some calculation, one of these addresses is to be chosen.
  - Result of this calculation will provide an offset stored in \( (R#) \).
  - Next instruction to be executed will be the address contained at \( A + (R#) \).

- Usually not both pre-indexing and post-indexing indirect found in the same architecture.
(vii) Stack Addressing

- Operations refer to stack.
  - Eg. ADD: Pop two topmost elements from the stack, add them store result on the stack.
  - No explicit address needed, implicitly one refers to the stack.
- PUSH, POP have
  - one implicit address referring to the stack and
  - one explicit address using another addressing mode, denoting the element to be pushed or popped.

More advanced addressing modes are used.
Example: Pentium II has the following addr. modes (SR = segment register, I = index register, B = base register):

- **Standard Addressing modes**:
  - Immediate Addressing.
  - Register Addressing.
  - Relative Addressing.

- **Standard Addressing modes**, but with one extra addition of SR# (segment register).
  - Relativized direct:
    * EA = (SR#) + A
    * This is base register displacement with a special register.
  - Relativized register indirect:
    * EA = (SR#) + (B#).
    * This is post-indexed addressing with a special register.
  - Relativized base register displacement:
    EA = (SR#) + (B#) + A.

(e) Endianess

- If an address or number consists of several bytes, what is the ordering of the individual bytes?
- Example: 0x1234 5678 can be stored at address 0x0256 in the following to ways:

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Value</td>
</tr>
<tr>
<td>0x256</td>
<td>0x12</td>
</tr>
<tr>
<td>0x256</td>
<td>0x34</td>
</tr>
<tr>
<td>0x256</td>
<td>0x56</td>
</tr>
<tr>
<td>0x256</td>
<td>0x78</td>
</tr>
</tbody>
</table>

More Advanced Addressing Modes (Cont.)

- Advanced Addressing Modes:
  - **Scaled index with displacement** (Scaling factors: S=1,2,4,8;
    eg. for arrays, where each element requires 1,2,4,8 bytes):
    \[ EA = (SR\#) + (I\#) \cdot S + A. \]
  - **Base with index and displacement** (eg. for two-dimensional arrays, or arrays on a stack):
    \[ EA = (SR\#) + (B\#) + (I\#) + A. \]
  - **Base with scaled index and displacement** (eg. for arrays on a stack):
    \[ EA = (SR\#) + (I\#) \cdot S + (B\#) + A. \]
**Endianess (Cont.)**

- Left way is called **big endian**.
  - The most significand byte has the lowest address.
  - Used by IBM 370/390, Motorola 680x0, Sun SPARC, most other RISC machines.

- Right way is called **little endian**.
  - The least significand byte has the lowest address.
  - Used by Intel 80x86, Pentium, VAX, Alpha.

- PowerPC is **bi-endian**: it supports both modes (selected by a flag).

- Differences in performance.
  - On big endian machines, the most significand byte is accessed first, on little endian machines the least significand byte is accessed first.
  - **Difference in performance minor**.

- However the layout of data stored different.

---

**Terminology goes back to Part I, Chapter 4 of "Gulliver’s Travels" by Jonathan Swift.**

- It contains the story of a religious war between two groups, one that breaks eggs at the **big end** and one that breaks them at the **little end**.

---

**Instruction Formats**

- Instruction formats = layout of bits in the code of an instruction.

- Main fields in it are
  - **Opcode (operation code)**:
    - Code for the operation to be executed.
  - **Operand address mode**:
    - Addressing mode used.
    - Often included in the opcode or implicit since only one addressing mode allowed for an operation.
  - **Operand address fields**:
    - Actual addresses.

- **Opcode** often supplemented by additional fields like
  - length of operand address;
  - format of operands (floating-point, signed integer etc.).

- Allocation of bits is done so that instructions can be **decoded and executed efficiently**.
The allocation of bits is determined by the following parameters:

- **Number of Operands.**
- **Number of addressing modes.**
- **Number of registers.**
- **Range of addresses** (8 bit, 16 bit, 32 bit, etc. addresses). Longer addresses require more bits.
- **Address granularity.** (i.e. whether addressing up to byte, 2-byte, 4-byte etc. level)

There is a tradeoff between short and long instruction codes.

- With long instructions we are able to define a **richer instruction set,** which is closer to higher level language constructions, with optimized codes.
- Shorter instructions can be loaded and evaluated faster with a more uniform architecture.

**Remark**

- SPARC format is a little-endian machine.
- IBM mainframe system is a big-endian machine.

**Two Examples**

- Both examples only serve as an illustration how instruction formats look like.
- **Example 1: CISC** instruction set: the IBM mainframe format.
  - (The most important example for CISC language is the Pentium, but its instruction set is very complicated).
  - **Abbreviations used**
    - R data register
    - B base register
    - X index register
    - D relative displacement
    - L length
    - src source
    - dst destination
  - **Example 2: RISC** instruction set: Sun SPARC RISC format.

---

**IBM Mainframe Formats (partial set)**

- **Register to Register**
- **Register to Indexed Storage**
- **Register to Storage**

---

**Computer Systems, CS M33, Michaelmas term 2002, Sect. 8**

8-79

8-80

8-81

8-82
(g) Machine Language Instructions for Procedure Calls

Procedure/function calls occur quite often in high-level languages. Many machine languages have a corresponding mechanism.

- One instruction for calling a procedure:
  - CALL A
  - CALL Sub1
  - CALL Sub2

- One instruction for returning from a procedure:
  - RETURN
  - RETURN

Procedure Sub1
(at address labelled Sub1)

CALL Sub2
ADD R10, R1, 0x15
STORE R10, [A4]
ADD R2, R2, [A9]

Procedure Sub2
(at address labelled Sub2)

LOAD R2, [A9]
ADD R2, R2, R3

Return to the program calling the current instruction.
### Order of RETURN

- A procedure might have multiple entries:
  - Program1:
    MLT R1,R1,#14
    CALL Label1
  - Program2:
    ADD R1,R1,#34
    CALL Label2
- Subroutines:
  Label1 ADD R1,R1,#0x14
  Label2 MLT R1,R1,#0x3

- or multiple RETURN-statements:
  BNE Label 1
  RETURN
  Label1 ADD R1,R2,#13
  RETURN

### Execution of Procedure Calls

- When calling a procedure, the address which would be executed after returning from the procedure
  - address of the instruction immediately following the CALL statement
    has to be stored somewhere.

- Problem of nested procedure calls:
  Unbounded many procedure calls might occur.
  - Standard method:
    Push return address on a stack.

### Execution of Procedure Calls Using a Stack

- Execution of CALL A:
  - Address of next instruction to be executed is pushed on the stack.
  - Jump to the address A is executed.
    (A usually specified in assembly language by a label).

- Execution of RETURN:
  - Pop address from the stack.
    * If stack is empty, error occurs.
    - Execute interrupt handler for this (see later).
    * Otherwise, jump to the address retrieved from the stack.
Example of Recursive Function Calls

- Procedures/functions might even call themselves recursively.

- Example:
  
  **Factorial** function $n!$ defined by:
  
  - $0! := 1$
  - $(n + 1)! := n! \cdot (n + 1)$.

- So
  
  - $1! = 1 \cdot 1$,
  - $2! = 1 \cdot 1 \cdot 2$.
  - $3! = 1 \cdot 1 \cdot 2 \cdot 3$
  - $n! = 1 \cdot 1 \cdot 2 \ldots \cdot n$

- Program:
  
  ```
  function fac (i: integer): integer;
  begin
    if i = 0 then fac := 1
    else fac := i \* fac(i-1)
  end;
  ```

Example of Recursive Function Calls (Cont.)

- So
  
  - fac(2) will call
  - fac(1) which will call
  - fac(0).
When calling a procedure, parameters have to be passed on to the new procedure.

Could be done via registers or main memory.
- However one needs to keep track of multiple nested function calls
- Parameters for each function call have to be stored.
- Easiest way of organization by storing parameter on the stack.

So
- Before calling a procedure, parameters are pushed on the stack.
- Then CALL is executed, therefore address of next instruction of the calling program is pushed as well on the stack.
- Address of the parameters is result of subtracting from the new stack pointer a certain offset, known when compiling the called procedure.
- Additional information is stored on the stack. See appendix to this section.

Two kinds of parameters.
- Call by value.
  * The value of the parameter is passed to the procedure.
  * Changes to the parameter in the called procedure, doesn’t affect the original parameter. Therefore such changes will be lost.
  * Modelled by pushing the actual content of the variable on the stack.
  * Inefficient, when dealing with arrays.
    - Complete array has to be copied.
    - Arrays can be extremely big.
    - In Java, for this reason arrays are always passed by reference.
- Call by reference.
  * Only the address to the original variable is passed to the procedure.
  * So address of the parameter stored in the list of parameters.
  * Only possible, if parameter is actually a variable
    (not an expression like i – 1 in fac).
  * Changes to this parameter will affect the original variable.

Consider the following program:
```pascal
var i: integer;
...
procedure p([var] i: integer);
begin i:= i+1 end;
...
i:=5;
p(i);
writeln(i);
```
- If $i$ is passed by reference, result is 6 ("var"-parameter).
- If $i$ is passed by value, result is 5.
(a) Parameter passing and local variables.

(b) Call by name.

- A procedure might have local variables.
  - When calling another procedure, local variables have to be saved.
  - When returning, local variables have to be retrieved again.

- Best way to store local variables on the stack.
  - Can be done just before executing CALL.
  - More efficient to use the stack as the place where these variables are kept throughout the procedure.
  - In order to access them we use a register, containing the address on the stack, where the current local variables start.
  - All registers can be accessed by adding to this register a certain offset.

So when a procedure is called, the following is done:

- The location of the stack pointer is saved in a register Frame Pointer.
- Then space is created on the stack for all local variables.
- Can be done by just increasing the stack pointer.
- Then
  * the frame pointer points to the beginning of the local variables for the current procedure,
  * the stack pointer points to the top element of the stack, which is the last of the local variables.
- Later we will see, that we need one extra field on the stack below the local variables, to which the frame pointer points.
Parameter Passing

- Parameters have to be passed on to the called procedure.
  - Can be done via main memory or registers.
  - However one needs to keep track of multiple nested function calls
    Parameters for each function call have to be stored.
  - Therefore better to store the parameter on the stack.
    - Before calling a new procedure, push the parameters for the procedure on the stack.
    - Then call the procedure.
    - The called procedure can access the parameters by subtracting from the frame pointer a certain offset.

Old Frame Pointer

- When returning from a procedure, we need to retrieve the old frame pointer.
- So when calling a procedure, the old frame pointer has to be pushed on the stack as well.
  - Done in the called procedure.
  - Old Frame Pointer will be pushed first on the stack.
  - The new frame pointer will point to this pointer.
  - Local variables will be stored on top of it.
Complete Procedure Call

- So the complete high level language procedure call results in the following operations:
  
  - In the calling procedure:
    - The parameters for the called procedure are pushed on the stack.
    - CALL is executed, which results in pushing the return address on the stack.
  
  - In the called procedure:
    - The register containing the (old) frame pointer is pushed on the stack.
    - The frame pointer is updated to to the current stack pointer.
    - Therefore it points to the location of the old frame pointer.
    - Stack pointer is increased to make space for the local variables.

Stack Frame

- Notion of a stack frame.
  
  - Stack frame of procedure is the part of the stack containing
    * parameters passed on to this procedure,
    * the return address to the calling program,
    * the frame pointer of the calling program,
    * the local variables for the called procedure.

(b) Call by Name

- A third kind of parameter type are call by name parameters.
  
  - It occurs mainly in functional programming.
    - The called expression is passed on to the called procedure.
    - In the example of `fac`, call by need for the parameter `i` would mean that the expression `i - 1` is passed on to the called procedure.
    - Whenever the parameter is accessed, the expression is evaluated.
    - Advantages:
      * Efficient, if the parameter might not be accessed at all, and if it is accessed, it will be accessed at most once.
      * Especially, the expression used might not terminate at all.
        If the parameter is not needed, the call of the procedure might still terminate.
    - Inefficient, if the parameter is accessed more than once.
      - If the expression for computing the parameter has some side effect, the result might differ from when using call by value.
Example of Call By Name

- Consider the following program:

```pascal
var i: integer;

function f: integer;
begin
i:=i+1;
f:=1;
end;

function p(j: integer);
begin
i:=i+j;
f:=j;
end;

i:=1;
p(f(0));
write(i);
```

- With call by value for the parameter of `p`, `f` is executed only once, the program prints 2.
- With call by name, `f` is executed twice, the program prints 3.

Example 2 of Call By Name

- Replace in the above example `p` by the following:

```pascal
function p(j: integer);
begin
i:= i+1;
f:=1;
end;
```

- Now `p` doesn’t make use of `j`.
  - With call by value, `f(0)` has still to be executed, the program prints as before 2.
  - With call by name `f(0)` is never executed, the program prints 1.

9. Input/Output, Interrupts

(a) I/O Modules.

(b) Control Methods for I/O.
  - (i) Programmed I/O.
  - (ii) Interrupt Driven I/O.
  - (iii) Direct Memory Access I/O.

(c) Memory Mapped vs. Isolated I/O

(d) Parallel vs. serial ports.

- External devices are:
  - **Input**: keyboard, mouse, voice, scanner.
  - **Output**: Printer, monitor.
  - **Input and output**: Networks.
  - **Storage** (functions structurally as I/O devices): Magnetic disk/tape, optical disk/tape.

- Computer systems communicate with external devices through an intermediate **I/O module** (also called controller).

- An I/O module serves both as
  - interface between CPU via system bus and as
  - interface with one or more external devices.

- I/O module is often part of I/O devices.
(b) Control Methods for I/O

Three types of control methods used

(i) Programmed I/O.
(ii) Interrupt Driven I/O.
(iii) Direct Memory Access I/O.

(i) Programmed I/O

Oldest method.

CPU controls directly I/O.

When I/O module has received information to be passed on to the CPU, it sets appropriate flags in its status register.

It is the task of the CPU to check status of the I/O module until it finds operation is complete.

Disadvantages

- Processor has quite often check the status of all I/O modules or
- processor has to wait, if request is not dealt with.
- Complicated to program efficiently.

Tasks of the I/O Module:

- **Command decoding.** Decoding of commands from CPU.
- **Data Transfer.** Transfer of data via the system bus.
- **Status Reporting.** Report of status to CPU: Busy or ready.
- **Address recognition.** Identification of addresses of associated peripherals and sending of control signals.
- **Sending of interrupt signals.** (in case of interrupt driven I/O and DMA).

Sending of requests from CPU to I/O modules unproblematic.

Problem, to deal with data supplied by I/O modules, since waiting time is usually unpredictable.
(ii) Interrupt Driven I/O

(a) Overview.

(b) Handling of Interrupts.

(c) Identification of Requesting Module.

- I/O modules have now possibility to send interrupt signals, once they have data available.

- At the end of an instruction cycle of the CPU, an interrupt check is included. (See next slide).

- In case of interrupt a hardware routine is started, which does the following:
  - It stores information about current status of processor in memory.
  - Calls interrupt handler.
    * Interrupt handler = software program for dealing with the interrupt.
    * Address for finding it is stored at some fixed location.
  - When interrupt handler is finished, current status of processor is retrieved.
  - Continuation with program execution.

- Note that CPU still has to control every byte of data passed from I/O to memory.
**Other Reasons for Exceptions**

- Interrupts are one form of an exception. Interrupts are caused by
  - I/O requests.
  - Timed interrupts, when pre-determined time has elapsed.
  - Hardware errors.
- The second group of exceptions are called traps.
  - Caused by software errors, e.g. invalid instructions, word boundary violations, division by zero, etc.
- Traps will be handled in the same way as interrupts.

**Handling of Interrupts**

- It might happen that while handling an interrupt, a further interrupt happens.
  - Nesting can be arbitrarily deep.
- Similar to nested interrupts are recursive function/procedure calls.
  - E.g. the function computing the factorial:
    ```
    function factorial(n:int)
    begin
      if n > 0 then fac := factorial(n-1) * n
      else fac := 1
    end
    ```
- Handling of nested interrupts (and of recursive function calls) can be carried out via a stack:
  - When calling an interrupt handler or a sub program, we push data about current state on a stack.
  - When the interrupt handler has finished, we pop data from the stack.
When interrupt occurs the following is carried out:

- Acknowledgement of the interrupt is signalled.
- Processor pushes PSW on the control stack.
  * PSW = Process status word:
  * Contains information about status of processor.
  * flags indicating carry, result zero from last arithmetic operation,
  * flags for interrupt, supervisor mode.
  * Possibly content of some other registers.
- Device controller or other system hardware issues an interrupt
- Processor finishes execution of current instruction
- Processor issues an interrupt
- Processor pushes PSW and PC onto control stack

Then a jump to a software program, the interrupt handler is called.

- This means that the current PC is pushed on the stack,
- and the program jumps to the location of the interrupt handler.
- The interrupt handler does the following:
  - It stores further process state information (usually on the same or some other stack.)
  - Then it deals with interrupt.
  - At the end it restores its original status,
  - and returns.
  * This means that the original PC is recovered.
  * Further the status according to the PSW is recovered,
  * and execution of the original program continues.

Continue on Next Slide
Example of Nested Interrupts

- Assume twice nested interrupts, as on the next slide. Below we show the content of stack after each step. (Nested function calls are treated similarly).

- **Step 0:** Main Program is running. Stack is empty.

- **Step 1:** Interrupt is received.
  - Information about current status is pushed on the stack
  - (separation between hardware/software see later).
  - Interrupt routine is called.

- **Step 2:** Interrupt routine receives a further interrupt.
  - Information about current status is pushed on the stack
  - Second interrupt routine is called.

Example of Nested Interrupts (Cont.)

- **Step 3:** 2nd Interrupt routine terminates.
  - Information about status of processor in the first interrupt program is popped from stack.
  - Processor’s registers, flags are set to their status in the first interrupt procedure, when interrupt occurred.

- **Step 4:** First interrupt routine is terminated.
  - Information about status of processor in the main program is popped from stack.
  - Processor’s registers, flags are set to their status in the first interrupt procedure, when interrupt occurred.
Identification of Requesting Module

- Problem with multiple interrupts, since usually not one interrupt bus line for every module:
  - Identification of requesting module necessary.
  - Several modules might send interrupt at the same time.

- Methods used:
  - Multiple Interrupt lines.
  - Software polling.
  - Daisy chains.
  - Bus arbitration.

Multiple Interrupt Lines

- Use one interrupt line for each I/O module.
- Only possible if there are few I/O modules.
- Method used for PCs, if there are not too many (typical 16) modules attached to it.

Software Polling

- Processor asks each I/O module (or its status register) whether it issued the interrupt.
- Disadvantage: time consuming.

Daisy Chain

- All modules have direct access to interrupt lines.
- Modules chained together subsequently.
- When processor senses interrupt, it sends interrupt acknowledgment signal.
- Signal propagates through the series of I/O module.
- First module, which has sent an interrupt, sends back a vector, which identifies this module.
  (Priority of modules ordered by their position in the chain).
**Bus Arbitration.**

- I/O module must gain control over bus when raising interrupt request.
  - Carried out via general bus protocols.
- Therefore only one module at a time can raise an interrupt.
- When processor detects request, it responds via interrupt acknowledgment line.
- Module places a vector identifying it on the data or address lines.

**Direct Memory Access (DMA)**

- Additional module on the system bus: The DMA-module.
- DMA module can transfer data from an I/O module to memory without CPU interaction.
- Processor sends I/O request to DMA module, and then continues with other tasks.
- When transfer complete, DMA module sends interrupt signal.

**DMA (Cont.)**

- DMA must use bus only, when processor does not use it.
  - Either use the bus when processor does not need it.
  - Or force processor to suspend operation temporarily (cycle stealing; usual method).
- Three main kinds of configurations (see next slide).
Reading Data from I/O (DMA)

Instruction
Read I/O
in main Program

Issue Read Command to DMA Module

CPU DMA
Do something else

Interrupt
DMA CPU

Read Status of DMA Module

Next Instruction

(d) Memory Mapped vs. Isolated I/O

- Memory mapped I/O: I/O devices and memory share the global address space.
  - No I/O instructions for load and store needed.
- Isolated I/O: I/O device addresses and memory addresses separate.
  - Additional control signal indicate whether an address is valid for memory or I/O.

(e) Parallel vs. Serial Ports

- Parallel ports allow simultaneous transfer of many bits (usually a word) to/from the I/O module.
  - Broad cables, port with many pins.
  - Used for devices to/from which large amounts of data per request are transferred, e.g. connection to monitor, printer.
- Serial ports transfer only one bit at a time in sequence.
  - E.g. connection to mouse, keyboard.
10. CPU Structure

(a) Data Lines.
(b) Control Lines.
(c) Data Flow during an Instruction Cycle.
(d) The Control Unit.
   (i) Hardwired CU.
   (ii) Microprogramming.

More Detailed View of a Typical CPU Structure

- Slide 10-5 shows the registers, basic registers, ALU and the connections with main memory.
  - The **CU** is not shown here. It is external to it and controls this diagram.
  - It has control lines with which it can
    * check the **status of flags and certain bits** of the registers (esp. of the IR).
    * **open and close data paths** between the units shown.
    * **issue control signals** like read and write to main memory or the register unit, which ALU operation is to be performed, which constant is chosen.
  - This is an example architecture. There might be additional registers involved. Further there will be additional internal logic (like shift logic) involved.

More Detailed View of a Typical CPU Structure (Cont.)

- Slide 10-6 shows the **data paths**.
  - Note that internally in the CU both addresses and data are here treated in the same way (as data paths).
  - When looking below at steps in the instruction cycle, it will become clear which data paths are needed.
  - An explanation can be found in the appendix.
  - The data paths from the IR refer to **different bits in the IR**, which contain the different address fields in the instruction code. So later when stating that we open a connection from IR we always state which bits of the IR we mean.
Remarks on the Data Flow Diagram

- The **principal design** is such that all connections are
  - either *from* some *register* to some *register*, or
  - *from* the register unit output or data bus of main memory to some *register*, or
  - *from* some *register* to the data input (Rdatain) of the register unit (Rdatain) or to the system bus.

Other connections would not be finished within one memory cycle.

- In general a CPU will not have all data paths shown.
- Some additional registers might be present.
- Some shift logic required is not shown.
- The **main memory** is of course *not* part of the CPU, only the bus system bus is connected with it. The picture given makes it easier to describe the data flow.
- The **data bus** of main memory is usually **bidirectional**. We have separated it for simplicity into “data bus (in)” and “data bus (out)”.
- Some data paths will be **permanently open**:
  - From the outputs of the register unit into registers A, B.
  - From the ALU to register C.
• There is a register unit which administers the general-purpose registers. It has:
  - Two input lines (R#1in, R#2in) for providing register numbers of two registers simultaneously.
  - Control lines for asserting read or write to the register unit.
  - Two output lines (R1out, R2out), where, if read is asserted, the content of the registers specified by R#1in, R#2in, will be output. This allows to look up two arguments of an arithmetic operation with register addressing mode in one go.
  - Data lines in (Rdatain), for providing data to be stored in a register. (The number of the register is specified in R#1in.)

• Data paths will often be split into two or more, which can operate independently. Especially the lines from the IR are separated into
  - lines corresponding to up to two register numbers
  - lines corresponding to memory addresses or immediate data.

(b) Control Lines

ALU

Flags

1st Argument

2nd Argument

Result

Control Signals

(ADD, SUB, AND, OR, ...)

ALU (Cont.)

• The arithmetic and logic unit is a blind calculating engine.

• It has as input usually two arguments and control signals, indicating the operation to perform, and as output the result plus some flags.

• There are additional shift operations and tests carried out in the CPU, which don’t belong to the ALU itself.
Control Paths to the CU

The control unit controls the flow of data in the CPU.
- There are control paths to the CU from
  - the IR,
  - the flags of the ALU,
  - control lines from the system bus:
    * interrupts raised,
    * successful/unsuccesful loading of data from
      main memory.
  - In this lecture control lines from the system bus ignored.

Control Paths from the CU

The CU has control paths to:
- switches, which open/close data paths.
  - Connections from the register unit to A, B, and
    from ALU to C are permanently open, i.e. no
    switch.
- main memory to assert read or write operation,
- the register unit to assert read or write operation,
- the ALU, to determine which arithmetic/logic
  operation to perform,
- the constants which are possible arguments for the
  ALU (which one),
- shifting units (Ignored in this lecture).
- other control lines of the system bus, for dealing
  with interrupts, delayed memory responses etc.
  (Ignore in this lecture).

States, Clock and CU

- The CU has further some internal memory,
  - stores the current state of the CU.
    * (Every instruction requires several clock cycles,
      * during which different operations are carried
        out.
      * See for instance steps carried out during a
        multiplication.)
    - Each such cycle corresponds to one state of the
      CU.
- Further a clock signals is provided to the CU.
Operation of the CU

Depending on
- the current state and
- the incoming control signals
  in one clock signal the CPU
- determines the outgoing control signals,
- and the next state, which is then updated.

How the CU and its states are organized will be discussed later.

c) Data Flow during an Instruction Cycle

- We will look now at the flow of data during an instruction cycle.

Examples treated:

(i) Instruction Fetch.
(ii) Instruction Decoding.
(iii) Operand Fetch: Immediate Addressing.
(iv) Operand Fetch: Direct Addressing.
(v) Operand Fetch: Indirect Addressing.
(vi) Operand Fetch: Register Addressing.
(vii) Operand Fetch: Register Indirect Addressing.
(viii) Operand Fetch: Base-Register Displacement.
(ix) Data Operation.
(x) Operand Store.
(xi) Instruction Address Computation.

General Outline

For each example we will do the following:

1. We will specialize the data flow diagram of the CPU and include only the units and paths involved.
2. We will add labels to all control paths involved.
3. We will describe the instruction cycle.
4. We give a microprogram for this specialized situation.

We first review briefly the instruction cycle (we omit the interrupt cycle).
• **Two operations** have to be carried out:
  - The instruction has to be loaded into the IR.
  - The PC has to be increased by 1.

• Both can be done simultaneously in **one cycle**:
  - **Loading of Instruction**:
    * Data path (a) from PC to the address bus is opened.
    * Read from main memory (b) is asserted.
    * Data path (c) from data bus (out) to the IR is opened.
    * At the beginning of the next cycle, **instruction with address given in the PC** is **available in the IR**.

• **Increase of PC by 1**:
  * Data path (d) from PC to argument 1 of the ALU is opened.
  * Constant 1 (e) is asserted for the second output and path opened for it to the argument 2 of the ALU.
  * Data path (f) from ALU to PC is opened.
  * Addition (o) is asserted at the ALU.
  * Then the beginning of next cycle **PC is incremented by 1**.
Microprograms

- Microprogram = program inside the CU, which determines the control signals to be issued by the CU.
- Microprogram has lines for each of the steps taken during the instruction cycle.
  - For each line it encodes the control signals to be issued plus
  - depending on information available, the next instruction to be executed.
  - We take here a simplified approach:
    The microprogram has
    * a 1-bit field for each control line of the CU, which determines whether this control line is asserted or not,
    * and a field for the next instruction, where we use pseudocode to denote it.
  - We will only write the control signals, which play a role during the cycle. All others are assumed to be 0.

The microprogram for the instruction fetch cycle is:

<table>
<thead>
<tr>
<th>Line-#</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>o</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

“Line #” is the line number of the current instruction. “Next #” is the number of the next instruction.
(ii) Instruction Decoding

One Cycle required.
- Depending on the instruction fetch choose next microprogram line.
- The microprogram asserts no control signals, only chooses the next micro program line depending on the control signals reaching the CU:

<table>
<thead>
<tr>
<th>Line-#</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Choose next microprogram line depending on incoming control signals (coming from certain bits in the IR).

(iii) Operand Fetch: Immediate Addressing

- The operand is part of the instruction.
- Therefore it is already available for data operations.
- No specific cycle needed.

(iv) Operand Fetch: Direct Addressing

One cycle required:
- Data path (g) from the bits of the IR corresponding to the operand’s address opened to the address bus.
- Read from main memory (b) is asserted.
- Data path (h) from data bus (out) to MDR is opened.

At the beginning of next cycle, data will be in MDR and can be used for the operation.

The microprogram is (again only control lines used written down; note that (g) refers to the bits in the address field of the IR):

<table>
<thead>
<tr>
<th>Line-#</th>
<th>b</th>
<th>g</th>
<th>h</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Line # for second operand fetch.
Two cycles required:

- **Cycle 1:**
  - Data path (g) from the address field in the IR opened to the address bus.
  - Read from main memory (b) is asserted.
  - Data path (h) from data bus (out) to the MDR is opened.
  
  At the beginning of cycle 2, EA will be available in MDR.

- **Cycle 2:**
  - Data path (i) from MDR to the address bus is opened.
  - Read (b) from main memory is asserted.
  - Data path from data bus (out) to the MDR (h) is opened.

At the beginning of cycle 3, operand is in MDR.
(v) Operand Fetch: Indirect Addressing

Cycle 1  Cycle 2

At end of cycle 2
MDR contains operand
(vi) Operand Fetch: Register Addressing

One cycle required:

- Data path (j) from the bits in the address field in the IR to R#1in is opened
- Read from register unit (k) is asserted.

(Note that connection from R1 out to A is always open).

At the beginning of next cycle, operand is in A.

The microprogram is (for first argument):

<table>
<thead>
<tr>
<th>Line-#</th>
<th>j</th>
<th>k</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Line # for second operand fetch

Similarly, a second register can be read. This can be done in parallel with the first one.

(vi) Operand Fetch: Register Addressing

Two cycles required:

- **Cycle 1:**
  - Data path (j) from the address field in the IR (containing the register number) to R#1in is opened.
  - Read (k) from register unit is asserted. At the beginning of cycle 2, EA is available in A.

- **Cycle 2:**
  - Data path (l) from A to the address bus is opened.
  - Read (b) from main memory is asserted.
  - Data path (h) from data bus (out) to the MDR is opened.

At the beginning of cycle 3, operand is in MDR.

The microprogram is:

<table>
<thead>
<tr>
<th>Line-#</th>
<th>b</th>
<th>h</th>
<th>j</th>
<th>k</th>
<th>l</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Line # for second operand fetch
(vii) Operand Fetch: Register Indirect Addressing

Cycle 1

At the end of cycle 2 MDR contains ((R#)) = (R)
(viii) Operand Fetch: Base-Register Displacement Addr.

$EA = (R#) + A$.

Three cycles required:

- **Cycle 1:**
  - Data path (j) from the bits in address field in the IR containing the register number to $R#1_{in}$ is opened.
  - Read (k) from register unit is asserted.
  - At beginning of cycle 2, $R#$ in register “A”.

- **Cycle 2:**
  - Data path (m) from A to the first argument of ALU is opened.
  - Data path (n) from the bits in the address field in the IR containing the base address to the second argument of ALU is opened.
  - Add (o) asserted to ALU.
  - At the beginning of cycle 3, EA is in register “C”.

- **Cycle 3:**
  - Data path (p) from C to the address bus is opened.
  - Read (b) from main memory is asserted.
  - Data path (h) from data bus (out) to the MDR is opened.

- At the beginning of cycle 4, operand is in MDR.

The microprogram is:

<table>
<thead>
<tr>
<th>Line-#</th>
<th>b</th>
<th>h</th>
<th>j</th>
<th>k</th>
<th>m</th>
<th>n</th>
<th>o</th>
<th>p</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

Line # for 2nd operand fetch

```
10-49
10-50
```

```
10-51
10-52
```
(viii) Operand Fetch: Base Register Displacement (operand = ((R#) + A))

**Cycle 1**
At the end of cycle 1
"A" contains (R#)

**Cycle 2**
At end of cycle 2
"C" contains (R#) + A

**Cycle 3**
Read
At end of cycle 3
MDR contains ((R#) + A)
(ix) Data Operation

Assume that after the operand fetch, one operand is in A (output of the register unit) and the other in the MDR. Assume further that the ALU operation requires only one cycle.
One cycle required:

- Data path (p) from register A to first argument of the ALU is opened.
- Data path (q) from MDR to second argument of the ALU is opened.
- Appropriate ALU operation (r) is asserted.

Since path from ALU to C is always open, at the beginning of the next cycle the result of the operation is in C.

The microprogram is:

<table>
<thead>
<tr>
<th>Line-#</th>
<th>p</th>
<th>q</th>
<th>r</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

(xii) Operand Fetch: Base Register Displacement (operand = (R#) + A)

Complete Data Flow Diagram

Assume operand1 in "A", operand 2 in MDR

Complete Data Flow Diagram

Assume operand1 in "A", operand 2 in MDR

Complete Data Flow Diagram
(x) Operand Store

- Done similarly to operand fetch:
  - First calculate the EA, (might require several cycles).
  - Then move the value to be stored (in C or in case of a pure store operation in the MDR, A or B) to data bus (in) of main memory or Rdata in of the Register Unit,
  - and assert a write control signal.

(xi) Instruction Address Computation

- Case 1: No branching/jump instruction: New PC value was already calculated, jump directly to instruction fetch. No cycle required.
- Case 2: Jump with relative addressing mode: Add to PC offset contained in the IR and feed back result to PC. One cycle required.
- Case 3: Branching instruction:
  - Case comparison of two registers: A data operation has to be carried out which has a result not needed but sets some flag. Requires one cycle. Then make decision depending on these flags. Requires another cycle.
  - Case branch depending on some flags: Directly choose next microprogram line depending on these flags. Requires one cycle.

Conclusion

- Arithmetic operations (including multiplication, which might take several cycles) and operand store can be carried out similarly.
- In the current structure, if one has more than one operands, one has to be determined by register addressing mode.
  - Typical for most current machines.
  - For more complicated addressing modes, further registers might have to be added.
- For floating-point arithmetic as well additional registers needed.

(d) The Control Unit

There are two ways of implementing the control unit.

(i) Hardwired CU.
(ii) Microprogramming.
(i) Hardwired Control Unit

Enumerate all states of the CU. Store the number of current state in a state register. Introduce directly a circuit which, depending on

- the incoming control signals and
- the current state number,
determines
- the outgoing control signals for the next clock cycle and
- the number of the next state.

Connect now this circuit with control lines and state register as in the next slide. Now the following simple loop is carried out:
For every clock signal

- Enable the outgoing control signals determined by the circuit,
- Store next state in the state register.

(ii) Microprogramming

Write for every state of a CU a microprogram instruction which contains:
- information of all the outgoing control signals,
- information how to select the next state depending on the incoming control signals.

Now construct a small computer with the following components:
- It has a small memory containing all micro program lines. The content is called microprogram or firmware, since it is half between software and hardware.
- It has a logic, which for every micro program line determines the outgoing control signals for the next clock cycle and the next microprogram line.
Problems:

- Control signals require a lot of storage. Encode them in a more compact way, which can be decoded easily.

- Determination of the next instruction complicated. Three main cases:
  - In most cases next line to be selected.
  - In some cases jumps or depending on incoming control signals next line or another line to be selected.
  - In few cases large case distinction depending on incoming signals into many different cases to be selected.

Solution for problem of selecting next line:
Introduce an address select logic which, depending on information encoded in the microprogram and control signals:

- either selects the current microprogram number increased by one or
- determines using
  - a hardwired circuit,
  - a branch table
  - or an address directly written into the microprogram
the next micro program number.
The condition for a branch (depending on which signals to choose the branch) might be hardwired or encoded in the microprogram.

Comparison

- **Hardwired CU:**
  - Is faster.
  - Implementation is very complex and sensitive to errors.
  - Main technique used for RISC instruction sets.

- **Microprogrammed CU:**
  - Easier to implement.
  - Easy to change if mistake occurs (not much change in the factory).
  - Can even be programmed in some cases (emulation of other architectures possible).
  - However slightly slower.
  - Main approach taken for CISC instruction sets. (Implementation of CISC processors require extremely many states).
Summary Microprogramming vs. Hardwired CU

- **Hardwired CU:**
  Selection of next state and control signals done by a circuit.

- **Microprogramming:**
  Microprogram is stored in some memory in the CU. Only a small selection logic for selecting the next line in the microprogram is implemented directly as a circuit. CU is essentially constructed like a small computer.

Merry Christmas
Good Luck in the Exam

Supplementary Material for Sect. 10

- Supplementary Material on (a) Basic Structure.
  (i) More on User-visible Registers
  (ii) More on Control and Status Registers
  (iii) Remark on the Memory Unit

- Supplementary Material on (b) Data Lines.
  (iv) Explanation of the Data Lines

- Supplementary Material on Control Lines.
  (v) Remark on the Register Unit.
  (vi) More on the ALU.

Supplementary Material for Sec. 10(Cont.)

- Supplementary Material on (d) Data Flow during an Instruction Cycle.
  (vi) Remarks on the Instruction Fetch Cycle.
  (vii) Remarks on the Instruction Decoding.
(i) More on User-visible Registers

- Sometimes the **general-purpose registers** are divided into
  - **Data registers**, which can store only data.
  - **Instruction registers**, which can store only instructions.
- Useful in order to **reduce address space** for register references:
  - For most operations it is clear whether they refer to data or addresses, one bit less needed for their address.
- Sometimes **finer distinctions** (for instance floating-point number registers) occur.
- **Condition Codes** are especially used when an interrupt occurs and a jump to an interrupt handler is carried out.

(ii) More on Control- and Status-Registers

- Other control- and status-registers, occurring in the literature, but not used here, are:
  - **Memory Address Register (MAR)**:
    * Stores addresses in main memory, the CPU wants to load data from or store data into.
    * Not needed in this module, since addresses are always available in some other register.
  - **Memory Buffer Register (MBR)**:
    * Contains both data and instructions read from memory or stored into memory or was read from memory.
    In this module
    - MBR used for storing data loaded from main memory.
    - Instructions always loaded directly into IR.
    - Data to be stored in main memory made available in some other register.

(iii) Remark on the Memory Unit

- By main memory we mean in the following main memory accessed via the cache:
  - Main memory read or write might actually result only in a cache read and write operation.

(iv) Explanation of the Data Lines

- From the **PC** there are data paths to:
  - The **address bus** of the system bus.
    * Needed to fetch the next instruction.
  - **One argument of the ALU**.
    * Needed in order to compute the next instruction:
      Add 1,2,4 or 8 in case of no branch/jump.
      Add offset in case of branch/jump.
- From **data bus (out)**, data paths exist to
  - **IR**.
    Used if an instruction was fetched.
  - **MDR**.
    Used if data was fetched.
(iv) Explanation of the Data Lines (Cont.)

- From the IR, data paths exist to:
  - R#1in and R#2in.
    * Used in case of addressing modes involving register numbers.
  - Rdain.
    * Used, if immediate data is to be stored in a register.
  - The two arguments of the ALU.
    Used for transmitting
    * offsets in case of displacement addressing modes,
    * immediate data on which ALU operation is to be performed.
  - Data bus (in) of main memory.
    * Used, if immediate data is to be stored in main memory.
  - Address bus of main memory.
    * Used, in case of direct or indirect addressing mode, in order to fetch data or addresses from main memory.

Explanation of the Data Lines (Cont.)

- The output of the register unit will always be stored in register A, B. (Connection is permanently open).

- From the registers A, B data paths exist to
  - The two arguments of the ALU.
    * Used for carrying out arithmetic operations on registers.
      This is the main and most efficient use of registers.
    * Used as well for calculating addresses in case of displacement addressing modes involving registers.
  - Data bus (in) of main memory.
    - In order to store data from a register in main memory.
  - Address bus of main memory.
    - Used, in case of register indirect addressing mode.

Explanation of the Data Lines (Cont.)

- From the MDR, data paths exist to:
  - Rdain for the register unit.
    * Used, in order to store data fetched from memory into a register.
  - The two arguments of the ALU.
    * Used if operations are to be performed on fetched data.
    * In case of displacement addressing modes, used for calculating the address.
  - Data bus (in) of main memory.
    * In order to store data back to main memory.
      (In case of bidirectional system bus this means that data is sent back to the data bus).
  - Address bus of main memory.
    * Used, in case of indirect addressing mode.

Explanation of the Data Lines (Cont.)

- The output of the ALU will always be stored in register C. (Connection is permanently open).

- There is as well a direct connection to the PC in order to store the result of a calculation of the address of the next instruction in it.
  - All other connections from the ALU are buffered by C, since they don’t end directly in registers.
From **register C** data paths exist to

- The two arguments of the **ALU**.
  - Used when more arithmetic operations which require multiple cycles are be carried out.

- **Rdatain** of the register unit.
  - In order to store the result in a register.

- **Data bus (in)** of main memory.
  - In order to store the result in main memory.

- **Address bus** of main memory.
  - Used, if an address was calculated in an indirect addressing mode with displacement.

---

**(v) Remark on the Register Unit**

- One might give the register unit separate address ports for read and write.

- Allows to simultaneously read and write and therefore enhance the throughput.

---

**(vi) More on the ALU**

**Implementation of the ALU**

- Integer addition and subtraction can be implemented by adders directly, and require only one clock cycle.

- Integer multiplication can be implemented by a cascade of adders. This requires however several clock cycles.

- Floating-point operations require more clock cycles.

---

**Example: Pentium**

- Integer multiplication requires 4 cycles.
  - But multiplication unit is pipelined,
  ⇒ carried out in 4 consecutive steps,
  - after each step next multiplication operation can already start.
  - Therefore throughput of 1 operation per cycle.

- Floating-point addition requires 5 cycles
  - is pipelined,
  - giving ideally a throughput of 1 operation per 2 cycles.

- Floating-point division is not pipelined, requires many cycles.
(vii) Remarks on the Instruction Fetch Cycle

- The example given requires that the next instruction is (unless a jump/branch is followed) at address given by the PC plus one. If it is at PC + 2, or PC + 4 (which means if addressable unit is byte, that the instruction length is 2 or 4 bytes), the constant has to be changed accordingly.

- If the instruction is longer than what can be fetched in one cycle, fetching has to be repeated.
  - If the length is fixed, one simply repeats the above several times.
  - If we have variable length, one loads the minimum length and then makes a decision.
  - This might require an extra cycle: If the decision depends on the last piece of instruction fetched, one cycle is needed to decide, whether one more instruction has to be fetched.
  - Alternatively, fetch in all cases the next instruction.

---

Microprogram for Fetching of Longer Instructions

- If the instruction length is one or two words, we get the following microprogram
  - (the IR can store now 2 words;
  - $h'$ is the control bit obtained from the first word in the IR, depending on which it is decided whether the next word has to be fetched as well;
  - $c$ connects now data bus from memory with the first word in the IR, $c'$ connects it with the second word).

<table>
<thead>
<tr>
<th>Line-#</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>c'</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>o</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1'</td>
</tr>
<tr>
<td>1'</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2'</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Note that the PC is incremented twice
If the instruction format has 3 or more words, we need to add even more lines to the microprogram.

---

(viii) Remarks on the Instruction Decoding

- If the instruction has length several words, it might be possible to carry out the decoding while fetching the other words by using information contained in the first word. Then no cycle is required for decoding.

- If a decoding cycle is necessary, one might carry out during this cycle operations which are likely to be done.
  - Especially send fields which correspond to register numbers to $R\#1in$, $R\#2in$ and assert read from the register unit.
    (Not harmful if this is not needed afterwards; therefore this is usually done).

- Similar optimizations may be done in other cycles.

---

11. High Performance Microprocessor Architectures

(a) Pipelining.

(b) RISC and CISC Computers.

(c) Superscalar Processors.

(d) Explicit Instruction-Level Parallelism.
**Laundry Analogy:** Assume several students want to do their laundry. They have to do the following 4 steps:

1. Wash the clothing using the washing machine.
2. Dry it in a dryer.
3. Fold clothing.
4. Put them away.

This can now be done in a pipeline:
- When one has washed his/her clothing, the next one can use the washing machine.
- Similarly for the dryer and the table for folding clothing.

The following slide illustrates the **speed up**.

---

**Pipelining of the Instruction Cycle**

The Instruction Cycle can now be divided into similar pieces. Assume a sequence of instructions of the form:

- LOAD R1, (R1 + A)

i.e. the second argument has addressing mode base-register displacement.

We divide it into **5 stages**:

1. Instruction fetch (1 cycle).
2. Register load (R1; 1/2 cycle).
3. ALU operation (calculation of R1 + A; 1 cycle).
4. Data access (load of (R1+A); 1 cycle).
5. Register save (storage of result; 1/2 cycle).

All 5 stages can be **carried out independently**. (For the increase of the PC by one in 1. one needs an extra adder). The next slide demonstrates the speed up. It is ideally by the factor 4.
Remark

- No increase in speed of a single operation.
  - In the previous example, it takes even longer to execute a single operation (4.5 instead of 4 clock cycles).
- But increase of throughput of instructions.
  - In the laundry example, everybody needs the same amount of time as before for his laundry.
- The Pentium IV has a 20 stage pipeline.

Problems

- The stages require different amount of time. The pipeline is only as fast as the slowest device. Can be improved, but not avoided by using as small stages as possible.
- Data dependencies between the stages.
  Eg. the second instruction requires a register, in which a value was stored by the first register:
  - Solutions:
    * Either second instruction has to be delayed until first instruction is ready.
    * Or (nowadays standard) pass on the data from one unit to the other, if possible.

Problems (Cont.)

- Most difficult problem are branches, since we do not know, whether the next instruction is the one at the next location or we have to follow the branch.
  - Example:
    BEQ R1, R2, Label1
    ADD R2, R2, 1
    Label1 LOAD A, REGISTER INDIRECT R2
    Depending on the comparison of R1 and R2, the next instruction is executed.

The next slide illustrates the additional delay (of one cycle), we get, if we have to wait until the condition of a branch is computed.

- We assume extra hardware, such that, while loading R4 and R5, one can
  * directly compare them,
  * compute the result of adding to the PC the offset,
  * and depending on the comparison update the PC to that value.

Then after loading the registers the PC has the next address to be executed.
Solutions for Problem with Branches

- **Delay next instruction** until decision is taken. Takes much time (branch instructions occur frequently).

- **Always continue both with next instruction and follow the branch.**
  If it is determined which was the next instruction, throw away the other one. **Problem:** might require unnecessary data access with removal of data from cache.

- **Branch Prediction:**
  Guess the next instruction. If the guess was wrong, throw away the instruction (and don’t change memory or registers before this decision is taken).

- **Delayed Branches:** Used on RISC machines. See Subsection (b), (iii) on RISC computers.

Strategies for Branch Prediction

- **Branch never taken.**
  Always fetch the next instruction.

- **Branch always taken.**
  Always follow the branch (ie. assume temporarily that the branching condition is true).

- **Predict by opcode**
  Decide by the kind of branch whether to follow the branch or not.

- **Taken/not taken switch.**
  Add to the instruction bits which reflect history taken. Usually stored when instruction is in cache or special buffer.

- **Branch history table.**
  Store information in this table. Information not lost if instruction removed from cache.

Loop Buffers

- **Loop Buffer** = Some sort of cache, but
  - only for instructions,
  - stores always a sequence of instructions.

- Instructions are **prefetched** if possible so that even if a small forward jump is done, often the next instruction is already in the buffer.

- Loop buffers are **well suited for loops in programs**, if loop fits within the buffer
  - Instructions in the loop need to be fetched from main memory only once.

- When a branch is reached, the next instruction can **often be fetched from the loop buffer.**

Conflict of Resources

- In general there will be conflict of resources.

- Most resources can be **doubled** in order to avoid this problem.
  - **Except of access for memory.**
    Required for instruction fetch, data load and data storage.
  - **Solution**, to solve this:
    * Divide cache into instruction cache and data cache.
    * Since usually instructions are not rewritten, unproblematic.
Example: MIPS

Next slide shows the preparation of a CPU for pipelining.

- **Two extra adders** were added.
  (for calculating the next instruction: one for addition of 1 to PC and one for addition of an offset to PC in case of a jump).

- Separate accesses to data and instructions build (corresponds to separate caches).

The slide afterwards shows the pipelined version, where extra registers (IF/ID, ID/EX, EX/MEM, MEM/WB) where added which store the data between the stages (step to and from them requires 1/2 memory cycle each).

Stages of the MIPS Pipeline

- The **stages of the pipeline** are
  - Instruction fetch (in the area up to the first bar).
  - Register fetch and instruction decoding.
  - Arithmetic operation.
  - Save result in main memory.

- This is **optimal only** for instructions which apply an arithmetic operation to two register contents and store the result in main memory.

For **other operations** the order of use of the stages of the architecture is altered and some stages are used more than once, which leads to a delay of the pipeline (next instruction has to wait until current one is completed).
(b) RISC and CISC Computers

(i) CISC Computers.
(ii) Analysis of CISC Computers.
(iii) RISC Computers.
(iv) CISC vs. RISC and Current State.

Definition:
- CISC = complex instruction set computers.
- RISC = reduced instruction set computers.

Semantic Gap: Gap between constructs in higher-level programming languages (HLL) and assembly languages instructions.

Problems of increasing use of higher-level program constructs:
- Decreasing efficiency.
- Increasing size of the compiled program.
- Increasing complexity of compilers.

Attempt to close it by introducing more and more complex instructions. This resulted in:
- Increase of number of machine instructions and addressing modes.
- Instructions with long execution time which require complicated hardware.
- Long instruction formats in order to accommodate a lot of instructions.

(ii) Analysis of CISC Computers

Studies carried out in the late 70’s and early 80’s on usage of machine instructions in computers.

Results:
- Complex instructions are not exploited by compilers.
  Possible reasons:
  - They are too specific.
  - Optimizing generated code easier with simple instructions.
- Assignment statements dominate, therefore simple movement of data is of high importance.
- Conditional and unconditional branches occur as well frequently.
- Procedure calls and returns are most time-consuming operations in HLL.
- Most references to variables are to simple scalar variables (integer, floating-point or strings which store one item only as opposed to arrays, lists etc.).
From the above the following conclusions were drawn:

- Instead of creating complex instructions one should optimize performance of most time-consuming features of HLL.

- The following main principles of RISC Computers were suggested in order to achieve this:
  - Use of a large number of registers. Try to use them as much as possible. Effective since variables are often local and registers are accessed much faster.
  - Use delayed branches and delayed loads to make pipelining easier and avoid the choice of the wrong branch.

Delayed Branches

- **Delayed branch** means that a branch instruction only takes place after the next instruction following it.
  - So in a language with delayed branches the program
    
    ```
    BEQ R1 Label1
    LOAD R2, #13
    ADD R2, #3
    ```
  
  is executed as the program in a language without delayed branches
  ```
  LOAD R2, #13
  BEQ R1 Label1
  ADD R2, #3
  ```

  - Then in pipelining we can start executing the instruction following the branch, even before we know whether the branch takes place.

Delayed Branches (Cont.)

- If we cannot move the branch ahead, one has to insert a NOP instruction (NOP means no operation, an operation which does nothing):

  The program without delayed branches
  ```
  BEQ R1 Label1
  ADD R2, #3
  ```

  can be replaced by the program with delayed branches
  ```
  BEQ R1 Label1
  NOP
  ADD R2, #3
  ```

  - Depending on the architecture, one might delay execution of branches even more by 2, 3 or 4 instructions.
  - Note that delayed branches means that the semantics of the program is changed (program don’t have to be interpreted in the order they occur).

Delayed Loads

- Similarly, **delayed loads** means that the effect of load instruction (from main memory) takes place only after the instruction following the load instruction. Before this the register value cannot be used.
  - So
    ```
    LOAD R1, REGISTER INDIRECT R2
    LOAD R2, R1
    ```

    would be either illegal, or R2 would get the result R1 had before this sequence of instructions.

  - In order to have the desired effect, either move the first load ahead, or insert NOP (no operation):
    ```
    LOAD R1, REGISTER INDIRECT R2
    NOP
    LOAD R2, R1
    ```
Main Principles of RISC Computers, (Cont.)

- Reduce the number of instructions and instruction formats and fix the instruction length. The reason for this is:
  - Simple instruction format and fixed instruction length makes pipelining more easier, more uniform and therefore faster.
  - Simple instruction format allows to carry out opcode decoding and register operand accessing simultaneously.
    (From the format it is clear which bits form the register numbers, even before we have decoded the instruction).
  - Simple instructions are faster executed and pipelined and the hardware can be more optimized.

Register Files and Register Windows

- Use of many registers in a register file, which is organized in a circular way. See Fig 12.2 below.

- When a new procedure called the following happens:
  - Parameters which pass data from the calling procedure to the called one and back again are saved in new free registers of the register window. These parameters are temporary registers for the calling procedure and parameter registers for the called procedures.
  - Further new local registers are assigned for the local variables of the new register.
  - Parameter-, local and temporary registers of a procedure together form the register window of the procedure.
  - The register windows overlap.

Register Windows

- The total number of registers is usually fixed. (Typical size is 16 or 32 registers per window).

- The window allows only to accommodate a few most recent procedure activations (limited nesting of procedure calls). Typical numbers are 8, 16, 32.

- If a procedure call requires more registers than available, the content of old procedures are stored back into main memory.

- When returning from a procedure, the registers of which have been overwritten, the registers have to be loaded back from main memory.

- Two pointers needed for organizing this:
  - the current-window pointer (CWP) points to the end of the currently used register area in the window.
  - the saved window pointer (SWP) points to the beginning of the currently used register area in the window.
Register Windows (Cont.)

- Because of the limited depth of recursions, the writing back of registers into main memory is not often needed.

- For global variables (common to several procedures) some additional global registers might be provided.

On the next slide the circular register file is illustrated.

- A.in, B.in ... are the parameter registers of procedure A, B ..., 

- A.loc, B.loc ... are the local variables of procedure A, B, ..., and

- w0, w1 ... are the register windows of A, B, ...

Comparison Cache vs. Register Files

- Register windows and cache have similar functionality.

- Advantage of cache: It can store non-scalar variables, instructions etc.

- Advantage of registers:
  - Can be addressed with very few bits.
  - Address computation is simple and fast.

Compiler Based Register Optimization

- Some RISC architectures don't have register windows but still large number of registers.

- Then one instead optimizes compilers, so that they try to use registers in such a way that transfers to and from memory are minimized.
**RISC Pipelining**

- RISC computers **execute always the next instruction** following an instruction, even in case of a branch.
- In case of branches therefore instructions have to be **reordered**. For instance the branch can be moved forward, if it doesn’t depend on the previous operation.
- In case there is no suitable next instruction, a **NOP (no operation)** instruction is to be **inserted**.
- Similarly, in case of **dependency of one instruction** on a previous one because of **register use**, reordering of instructions is applied.
- RISC pipelines are therefore very **efficient**, since the pipeline is most of the time in full use.

**Characteristics of RISC Instruction Sets**

- **One instruction per machine cycle.**
  (Machine cycle = length it takes to fetch two operands from registers, perform an ALU operation and to store the result in a register). That results in a uniform pipeline.
- Most operations should be **register-to-register**, only load and store operations access memory. This simplifies the instruction set and therefore the control unit.
- Use of **simple addressing modes**, so that addresses can be computed fast.

**Examples** of full RISC processors are MIPS R4000 (used by NEC, Nintendo, Silicon Graphics, Sony), Sun SPARC, Power PC.

- Not clear whether RISC computers are so much faster. Examples of **directly comparable RISC and CISC computers are missing**.
- Tendency towards **mixture of CISC and RISC features**:
  - Addition of **some CISC instructions to RISC architectures**.
  - For instance the Pentium has an initial interpretation procedure, in which the fetched CISC instructions are interpreted as internal RISC instructions which are then evaluated in a RISC pipeline. Instructions are cached in the form of the interpreted RISC instructions.

- **Difficulty of Intel to move from the CISC instruction set** (would destroy compatibility with previous processors).
- It seems to be easier to **market sophisticated CISC instruction sets**. (Compare with advertisement of cameras, stereo equipment etc. in which one points out the new switches they have, even so they are rarely used).
(c) Superscalar Processors

(i) From Superpipelining to Superscalar.

(ii) Design Issues for Superscalar Architectures.

Superpipelining =
Use the fact that if data is to be passed from one register to the next one, it arrives half a clock cycle later instead of a full clock cycle later. Use pipeline stages so that every half clock cycle the next instruction is fetched. See next slide.

Above = traditional usage, which is changing. Nowadays superpipelining means often only a very long pipeline with very small individual units (e.g. for Pentium IV has a 20 stages pipeline).

Superscalar Architecture =
- Execute several instructions fully in parallel.
- Requires to have multiple functional units, all of which are implemented as a pipeline.
  For instance one has two integer ALUs, two floating-point ALUs, but usually only one access to memory.

Origin of name superscalar because superscalar processors are optimized for processing scalar data, as opposed to vector processors, which are optimized for processing vector data (arrays) in parallel. The latter are important for numerical applications (e.g. weather forecast, other simulations of physical systems, engineering).

Superscalar is now the technical standard for processors in computers (there are as well processors in cars, calculators . . . .
Power PC from RS/6000 onwards, all Pentium processors, MIPS R10000, Sun’s UltraSPARC-II are examples of it.

Superscalar approach is an instance of instruction-level parallelism: multiple instructions in sequence, which are independent, are executed in parallel.
Limitations of Superscalar Architectures

Instructions can not always executed in parallel because of:

- **True data dependency** between multiple instructions (the next instructions depends on the result of a previous one).
  
  Example:
  
  \[
  R_1 := R_3 + 3; \\
  R_2 := R_1 + 4.
  \]
  
  Partial solution: pass result of one computation directly to the next instruction (in the example, as soon as the result \( R_1 + 3 \) of the first instruction is obtained, it can be used in the second one at the same time as storing it in \( R_1 \)).

- **Procedural dependency.**
  
  - Dependencies on branching instructions.
    
    Example:
    
    \[
    \text{BEQ } R_1, R_2, \text{Label2} \\
    \text{ADD } R_3, \#4;
    \]
    
    Second instruction executed only if \( R_1 \neq R_2 \).
  
  - Can be solved as in pipelining by branch prediction.
    
    *(Delayed branch strategy for RISC architectures turns out to be less effective).*

- **Resource conflict.** (Two instructions require the same functional unit of the CPU). Two instruction may use the same resource. Can be resolved by adding more resources. However, access to memory and registers is limited.
  
  - E.g.:
    
    \[
    \text{LOAD } R_1, \text{DIRECT A} \\
    \text{LOAD } R_2, \text{DIRECT B}
    \]
    
    Both loads require access to main memory.

- **Output dependency.** If two instructions store values in the same register, the later one has to be completed after the first one:

  E.g. in case of
  
  \[
  R_1 := R_2 + R_3; \\
  R_1 := 3,
  \]
  
  if the second instruction is completed before the first one, at the end a wrong result might be stored in \( R_1 \).

  Solution: **Register renaming.** Register numbers in the instructions are virtual, which are mapped by some logic to actual registers.

  In the above example, the \( R_1 \) in the third instruction (and in later instructions) is mapped to a different actual register than the \( R_1 \) used in the first instruction.

- **Antidependency.** If one instruction reads a register which is overwritten by a later instruction, the later one shouldn’t be completed before the first one.

  Example:
  
  \[
  R_1 := R_2 + 3; \\
  R_2 := 3.
  \]
  
  If \( R_2 := 3 \) is completed before \( R_2 \) is fetched in the first one, the first instruction has a wrong result.

  Can be solved again by register renaming.

  **Register renaming** turns out to be crucial in order to obtain any substantial speed-up.
Usually instructions are not fully independent. Three orderings are important:

- Order in which instructions are fetched.
- Order in which instructions are executed.
- Order in which result of an instruction is stored in
  registers or memory.

The terminology used is

- Issue for the order in which fetched instructions are passed on to the units executing it.
- Completion for the order in which their results are stored in registers and main memory.

### Three Policies for Superscalar Architectures

- **In-order issue with in-order completion.**
  Both issue and completion of instructions in the order they occur in the program. If there is a conflict, execution of later instructions is delayed. Rarely used.

- **In-order issue with out-of-order completion.**
  Now completion can be out of order. However, decoding of instructions only done up to the point of dependency or conflict. More complex instruction-issue logic required.

### Superscalar Execution

In the next slide an overview over superscalar execution of programs is shown.

- In the instruction fetch process, a dynamic stream of instructions is created. Branch prediction logic is applied simultaneously. Dependencies are examined and if possible removed.

- The instructions are then dispatched into a window of execution. This is done in the order of dependency instead of the order in the program.

- At the end instructions are issued to the retiring unit.

- The retiring unit guarantees that data conflicts when writing data to memory and registers are resolved, so that the result is as in an execution in true order. Results not yet retired are kept in temporary storage. If a wrong branch was taken, wrong results have to be thrown away.
Units Required in Superscalar Architectures

- Good instruction fetch strategies combined with branch prediction logic.
- Logics for determining and if possible solving true dependencies.
- Mechanisms for issuing multiple instructions in parallel.
- Resources for parallel execution of instructions, both for functional units and for access to main memory (in parallel).
- Mechanisms for designing the retiring unit.

(d) Explicit Instruction-Level Parallelism

- New processor of Intel, developed together with HP: IA64, with code name for first processor Itanium.
- Original code name was Merced.
- Step towards 64 bit processors.
- As for superscalar processors
  - large number of registers (256 64-bit-register and 64 1-bit predicate registers).
  - multiple execution units.
- Starts completely new family of processors with completely new instruction sets. (Break with an over 20 years long history, starting in the late 1970s with the x86 family).

Explicit Instruction-Level Parallelism (Cont.)

- Explicit parallelism.
  - Called ILW = Instruction Level Parallelism.
- New instruction format, which holds 3 instructions, stored as 128 bits, which can be executed in parallel.
  - Because of the long instruction format, this is called very large instruction word, VLIW.
- Additional information in the instruction, which indicates that several of these instruction bundles can be executed in parallel.
- Dependencies between instructions are given by additional flags (predicates).
  - Therefore resolving of dependencies and determining of order done at compile time rather than at execution time. Compiler has much more time to resolve such issues.
**Predicated Execution**

- In order to allow the execution of many instructions in parallel, they are **predicated** (has to be done at compile time).
  - If one instruction contains a condition, e.g. "if \( x = 3 \)", and other instructions depend on positive and negative outcome of it, e.g. the "then"- and "else" clause in an if-then-else construct, two predicates (from the predicate registers) are chosen.
  - As soon as the condition is resolved, one becomes true, the other false.
  - The instructions which are to be executed depending on this condition will now be **prefixed with these predicates**.

**Example of Predication**

If \( a=b \) then \( k:= k+1 \) else \( l:= l+1 \) is translated into

\[
\begin{align*}
P2, P3 &:= \text{compare}(a, b) \\
P2 &< k := k+1 \\
P3 &< l := l+1
\end{align*}
\]

**Speculative Load**

- **Bottleneck** is **access to main memory**.
  - Therefore want to **move load instructions ahead**, so that they can be carried out as soon as access to main memory is free.

**Problem** with moving load instructions over branch instructions:

- Wrong load might raise an exception, since it refers to an invalid memory address or we get page fault.
- However this load might not be required and to raise an exception might be unnecessary.

**Solution:**

- Introduction of **new instructions**, which replace an ordinary load instruction:

**Speculative Load (Cont.)**

- **Speculative load** written for instance as

\[
\text{ld.s. } R4, (R3)
\]
  - "load speculative (R3) into R4".

Executed as follows:

- Load (R4) into a register R4, but
  - **don’t deliver the result**, and
  - **don’t raise an exception**.

- An additional **checking instruction** is required
  - e.g. \( \text{chk.s } R4 \)
  - Placed now where one would put the **load instruction without speculative loading**.
  - \( \text{chk.s } R4 \) does the following
    * it **delivers the result** of a previous speculative loading of \( R4 \) to \( R4 \).
    * it **raises an exception**, if an exception occurred.
  - If there is a branch in between, \( \text{chk.s} \) might be **predicated**, which means that the result of the loading is only delivered and an exception is only raised in case the predicate is true.
Current Development

- Explicit parallelism seems to be the right way.

- Problems with backward compatibility.
  - Itanium in compatibility mode slower than Pentium.
  - Full speedup only, when compilers and (partly) operating systems have been specially rewritten.
  - Implies that software has to be adapted as well (new compilers will not always behave in exactly the same way).

- AMD is developing 64 bit architectures, which extend the old x86 family in a traditional way.
  - Market forces might favour this approach, even so this means to continue with not very well designed architectures.

- Degree of parallelism is expected to increase in the near future.

Merry Christmas
Good Luck in the Exam

Revision Lecture

2. Boolean Logic, Combinatorial Circuits.
3. Sequential Circuits.
6. Internal Memory.
7. External Memory.
8. CPU-Instructions Sets, Addressing Modes.
10. CPU Structure.
11. High-Performance Computer Architectures.

General Remarks

- Essentially everything needed should be contained in the notes.
(No absolute guarantee can be given however.)

- Books are of course helpful
  * to widen your perspective,
  * to get a deeper understanding
  * to understand what is meant by short sentences on the slides.
  * But you probably don’t need to know topics which only can be found in books.
  * Other lecturers might have different policies.
Structure of the Exam

- Some questions ask for **bookwork** (eg. 3 I/O control methods, configuration of an S-R-latch).
  - Require an understanding of the areas taught, not only the knowledge of some names.

- Most questions require **working out/calculating** something rather than repeating what you have learned by heart.
  - In that case always write down how you obtained your solution.
  - Most marks are given for the algorithm – small mistakes in calculations will not count that much, if the way of solving it was correct.

- Therefore material involving calculations or programming is more important than other topics.

Anonymous Marking

- There is a **seal** you put over your name.
  - I usually break the seal **after** the exam has been marked completely in order to cross-check that the names are correct.
  - It's a pain if you seal it heavily – one strip glued over one corner suffices.

Seal

- I will visit the exam during the first 1/2 hour and will walk around. Please look for me in case something is unclear in a question.

- **No calculators** will be allowed but dictionary (applies to most exams).
  - No complicated calculations will be demanded from you.
Structure of the Exam (Cont.)

- Exams at the department (and probably other departments at UWS) usually have **usually 3 questions**;
  you can **select 2** out of 3;
  each has 25 marks (in total therefore 50):
  so 1 mark counts 2%.
  - If you answer all 3 questions, **only the best 2 will count**.
  - When trying 3 questions, in most cases the **third question is answered badly**, due to time pressure — it’s usually better for you to spent more time on answering the other questions carefully.
    * Except if you realize when answering one question that it didn’t go that well with it.

Structure of the Exam (Cont.)

- The difficulty of parts in questions varies.
  - Some very easy parts (mainly bookwork).
  - Some parts of **medium difficulty**.
    * Eg. multiply using Booth’s algorithm.
  - One or two **more tricky parts**.

- Expect to be under **time-pressure** during the exam (might be different from what you are used to from your home country).

- This exam might be **slightly more difficult** than previous one.

- An exam is **no beauty contest** (as for instance essay writing, presentations). Don’t waste time with using writing your results down neat and tidily.

Structure of the Exam (Cont.)

- Whenever appropriate write down how you obtained your solution or why your solution is correct.

- Write down **everything which is relevant** in your answers. Often it is more important to show how you obtain your result rather than the result itself
  - If you have a long **detailed calculation and with tiny mistake**, you will get a very small reduction of your marks.
  - If you **just present your result**, you might loose marks because you don’t show how to obtain it, and might **loose all marks**, in case your result is not correct.
  - Sometimes I find something worth extra marks in the **scratch area** — most lecturers wouldn’t bother to look there.

Cheating in the Exam

- Cheating in the exam might have severe consequences.
  - You do not only get 0 marks, but might not get a degree at all.
  - British universities might be more strict than what your are used to in your home country.
Importance

A = very relevant, B = minor questions, C = not directly relevant for the exam.

1 Historic development of computers.  
2 Boolean Logic, Combinatorial Circuits  
2 Sequential Circuits  
4 Computer Arithmetic and Representation of Data.  
5 Computer Components, Interconnecting Structure.  
- Execution of Commands  
- Fetch-Decode Cycle  
- Other Topics  
6 Internal Memory.  
- Cache  
- Other Topics  
7 External Memory.  
8 CPU-Instruction Sets, Addressing Modes.  
9 Input/Output, Interrupts.  
10 CPU Structure.  
11 High-Performance Computer Architectures  
- Pipelining:  
- Other topics:  

No guarantee for completeness.

Four Main Topics

- Boolean Logic, Circuits  
  - Circuits, latches/flip flops, circuits for addition.  
- Computer Arithmetic  
  - Binary representation, multiplication algorithms.  
  - Signed numbers, floating-point numbers.  
- Internal Memory  
  - Cache: direct/associative/set-associative.  
  - Tag, line/set, word.  
  - Replacement algorithms.  
- CPU instruction sets.  
  - Addressing Modes, assembly language programs, instruction cycle, execution of instructions, data flow.

Additionally minor questions from (B- and A-) topics.

Best Preparation

(a) Coursework 1  
(b) Coursework 2  
(c) Old exam.  
(d) Learn some details.  
  (Guidance: exam).

Because: Most questions demand to do something on your own.

1. Historic Development and Basic Structure

- No years, numbers to be learned.  
- Basic periods with basic information.  
  - Mechanical era – from mechanical to electromechanical computers  
    - Schickhardt, Pascal, Leibnitz.  
    - Babbage, analytical machine.  
  - 1st generation (vacuum tubes).  
  - 2nd generation (transistors).  
  - 3rd generation (integrated circuits).  
  - 4th generation (very large scale integration).  
- Moore’s law, increase in performance.  
- Von Neumann machine. Design principles.  
- Harvard vs. von Neumann architecture.  
- Components. Motherboard. (ie. what is a motherboard?)
2. Boolean Logic and Combinatorial Circuits

- **Gates** (symbols – often symbols for + and · mixed up!), truth tables (for a gate; for an expression like \( \overline{x} \cdot y + x \cdot \overline{y} \)).
  - Draw circuits carefully (filled bullet = connection and bullets are required; circles = negation symbol).
  - Note the difference between \( \overline{x} \cdot \overline{y} \) and \( \overline{x} \cdot \overline{y} \). Be careful when writing such expressions.

- Truth tables (of gates, logic operations, Boolean expressions).

- **Conversion** between circuits and Boolean terms.

---

**Circuits for Boolean Functions**

- Trick with constructing circuits for Boolean functions:
  - Take inputs A,B,C, ...
  - For every “1” in the result, take one AND gate and connect it with each of A,B,C, ...
  - If A is 0 in the corresponding row, put a 0 at the gate.
  - If B is 0 in the corresponding row, put a 0 at the gate.
  - etc.
  - OR everything.

- A Boolean formula can now be read off.

---

**Example**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ (A \cdot \overline{B} \cdot \overline{C}) + (\overline{A} \cdot \overline{B} \cdot C). \]
3. Sequential Circuits

- **Combinatorial vs. sequential circuits** (definition). Why do we need sequential circuits?
- **S-R-latches** (called as well R-S-latches; configuration, stable/unstable state, inputs for modifying it, how does modification take place, memory values stored).
- **Circuits built from latches** like D-flip-flop, storage cells.
  - How does output of a D-flip-flop and falling edge D-flip-flop depend on its inputs?
- How does a **clock** operate?
- How are **bits moved around**?
- **Multiplexers** (What do they do? Implementation? How do they work?).
- How does the **CU** operate?

---

4. Computer Arithmetic and Representation of Data

- **Scalar/Compound/Reference and function types**.
- How much can be represented with k bits ($2^k$ elements).
- Binary representation of numbers vs. BCD (binary coded decimals).
- Conversion binary $\leftrightarrow$ hexadecimal $\leftrightarrow$ decimal. (What is 19 in binary, hex?)
  - 0x19 stands for 19 in hexadecimal, 0b0010 for 0010 in binary.
- Other number systems (what is $(14)_7$, what is $(19)_{10}$ in the system with basis 7)?
- **Octal numbers**.
- Shifting binary numbers means multiplying/dividing them by 2.

---

Computer Arithmetic (Cont.)

- Adders (half, full, n-bit; circuits for them. Why two implementations of full-bit adders? Which one is better?).
- Carry lookahead.
- Addition (unsigned, signed). Subtraction (add the negated number).
- Multiplication
  - Mainly Booth’s algorithm, signed and unsigned (Concrete: carry out multiplication for 3 and $-2$ using Booth’s algorithm; note differences between signed and unsigned version).
- Signed numbers (two’s complement; sign-magnitude, and why not used?)
  - Conversion decimal $\leftrightarrow$ binary (complement + add one).
  - Addition, subtraction of signed numbers.
- Carry, overflow (how calculated; examples!)

---

Computer Arithmetic (Cont.)

- Unsigned fixed point numbers (convert binary 0.375 into fixed point with 7 bits after the point).
  - (Solution 0b0.0110000).
- Arithmetic operations on unsigned fixed point numbers.
- Floating-point numbers. (Compute decimal $\leftrightarrow$ binary).
  Biased exponent. Cannot represent numbers precisely.
- IEEE 754 standard.
  - Normalized, non-normalized numbers.
  - NaN, infinity.
  - No need to remember the precise bias, number of bits)
### Representation of Data

- How are texts represented (Ascii, Unicode)?
- Lossy compression.
- Compound data types (how are records/arrays stored?; two-dimensional arrays).
  - At which address is $a_{16,5}$ stored (requires of course additional information like the size of each element of the array, range of the indices)?
  
  **Answer to exercise slide 4-152:**
  
  $$A + ((i - k) \cdot (n - m + 1) + (j - m)) \cdot f$$

  - $A$ is the starting address.
  - $i$ is the $(i-k)$th index in sequence.
  - There are $(n - m + 1)$ possible indices for $j$.
  - $j$ is the $(j-m)$th index in sequence.
  - $f$ is the size of each element of the array.

### CPU and Interconnecting Structure

#### 5. Basic structure of the von Neumann Architecture (ALU, Control Unit (CU), I/O, main memory).

- PC, IR, AC.

#### Execution of high level commands (section (b))

- How does the content of PC, IR, AC and main memory change during execution of concrete instructions? (Consider example instructions).

#### Implementations on digital level.

- Instruction cycle state diagram (Instruction address calculation $\rightarrow$ instruction fetch $\rightarrow$ instruction decoding $\rightarrow$).

- Registers (user visible vs. control/status registers; definition, name typical ones).

- ALU (what is an ALU?)

### CPU and Interconnecting Structure (Cont.)

- Buses
  - 3 groups: data/address/control lines
  - Multiplexed and dedicated lines (what’s that?)
  - Arbitration (centralized, distributed).
  - Synchronous, asynchronous timing.
  - Problems with simple structure, with bus connecting everything (von Neumann bottleneck); solution (cache; hierarchies of busses etc.).

### 6. Internal Memory

#### Categories of memory

- Location (internal/processor/external).
- Capacity (units: Kilobyte $= 1024$ byte, KiloHz $= 1000$ Hz; mega, giga etc.; bits vs. bytes).
- Addressable units (what’s an addressable unit?), word (what’s a word?).
- Access (Sequential/direct/random/associative)
- Access time (how computed); memory cycle time (definition).
- Physical types (Semi-conductor/magnetic surface/optical/magneto-optical).
- Volatile/non-volatile.
- Erasable/non-erasable.
- SRAM/DRAM (how physically done).
**Internal Memory (Cont.)**

- Core memory (only, what it is).
- Notions: RAM/ROM/PROM/EPROM/EEPROM/Flash;
  Read only/read mostly (just the definitions).
- SRAM/DRAM (how physically done).
- Core memory (only, what it is).
- Memory hierarchy (which technologies?; why not only SRAM used?)

**Cache**

- Why?
- Two level cache (L1/L2).
- Blocks stored in it (in order to reduce number of tags).
- Hit/miss.
- Split vs. unified cache.
- Direct mapped/associative/set associative cache:
  - Tag – line/set – word (concrete calculations; as in coursework);
  - Four replacement algorithms (concrete examples).
  - Write policies.

**More on Cache**

Assume Block size is 16 bytes. Then in decimal representation:

- Byte 0 - 15 go to first block.
- Byte 16 - 31 to second block
- Byte 32 - 47 to third block
- etc.

If we have 4 lines and direct mapping then

- Block 0,1,2,3 go to line 0,1,2,3 respectively.
  (Block 0 is byte 0 - 15; block 1 is byte 16 - 31 etc. if blocksize = 16 byte).
- Block 4,5,6,7 go to line 0,1,2,3 respectively.
- Block 8,9,10,11 go to line 0,1,2,3 respectively.
- etc.

If we have set-associative mapping, 4 sets with 2 lines each (ie. 8 lines or 8 blocks in total) then:

- Block 0,1,2,3 go to set 0,1,2,3 respectively.
- Block 4,5,6,7 go to set 0,1,2,3 respectively.
- Block 8,9,10,11 go to set 0,1,2,3 respectively.
- etc.

If addressable unit is word, count in words instead of bytes.
Stacks

- What is a stack?
- Stack machines.
- Stack addressing (write a small program).
- Implementation of stacks (how does a concrete implementation look like?).

8. CPU Instruction Sets, Addressing Modes

- Notions: assembly language, assembler.
- Basic types of instructions (data processing, data transfer etc; examples for each).
- Flags (N/Z/C/O-flag)
- Operator, operands
- Number of addresses.

CPU Instruction Sets (Cont.)

- Description (eg.: How is the address calculated in relative addressing mode?)
- Which operand is loaded in “Load register indirect R3#”, if R3 = 10, memory location 10 has content 20?
- No need to know names for fancy addressing modes (vi.3 - 5, Pentium II addressing modes)
- Assembly language program for computing sum of elements of an array; try out similar examples.
9. I/O, Interrupts

- Groups of I/O devices (input; output; input and output; storage).
- I/O modules (definition; tasks).
- Three control methods: programmed I/O, interrupt driven I/O, DMA. How do they work (roughly)?
- Traps vs. interrupts, both cases of exceptions.
- How are traps/interrupts detected (traps caused by the CPU; interrupts tested at the end of the execution cycle)?
- Identification of origin of interrupts (4 methods and brief characterization).

10. CPU Structure

- Three parts: ALU, CU, registers. Basic diagram, tasks, basic behaviour.
- User-visible/control registers (which ones?). MDR.
- Basic flow of data (don’t learn slide 10-6 by heart, but maybe structure on slide 10.5; if you need to fill in data paths for one instruction, follow the pattern of the lecture and fill in the lines required).
  - How would flow of data work when fetching an operand with e.g. REGISTER INDIRECT R2 (or some addressing mode not dealt with in the lecture directly).
- Hard-wired control vs. micro-programming (diagrams of the CU, distinction).
- Microprogramming.
11. High Performance Microprocessor Architectures

- Only the principal idea of pipelining.