Computer Organisation & Program Execution
2021

Uwe R. Zimmer - The Australian National University
Organization & Contents

Uwe R. Zimmer - The Australian National University
what is offered here?

Fundamentals, Overview & Hands-on Experience of Computer Architecture
who could be interested in this?

anybody who …

… wants to know why and how computer science immediately connects and translates to the physical world.

… would like to see immediate real-world involvement in their work.

… would like to understand what really happens if you run a high level program.
who are these people? – introductions

Ben Swift & Uwe R. Zimmer

Abigail (Abi) Thomas, Ashleigh Johannes, Ben Gray, Brent Schuetze,
Calum Snowdon, Chinmay Garg, Harrison Shoebridge,
Johannes (Johnny) Schmalz, Peter Baker, Ryan Stocks, Septian Razi,
Tom Willingham
how will this all be done?

Lectures:
- 2 x 1.5 hours lectures per week … all the nice stuff
  Monday 13:30, Wednesday 11:30 (both on-line - which is: here)

Laboratories:
- 3 hours per week … all the rough stuff
  time slots: on our web-site – on-campus in CSIT N.xxx or HN Lab.xx laboratories
  - enrolment: https://cs.anu.edu.au/streams/ (opened on Monday)

Resources:
- Course site: http://cs.anu.edu.au/student/comp2300/ … as well as schedules, slides, sources, links to forums, etc. pp. … keep an eye on this page!

Assessment:
- Hurdle lab in week 4 (1%) – a pass here is a hurdle for the course
- Mid-semester exam (13%)
- 3 assignments (12% each)
- Final-exam at the end of the course (50%) – 40/100 is a hurdle for the final exam
Many concepts in this course are in there – *but not all!*

The [Patterson17] provides an excellent general background and a lot of in-depth studies into more specific fields.

References for specific aspects of the course are provided during the course and are found on our web-site.
Computer Organisation & Program Execution 2021

Digital Logic

Uwe R. Zimmer - The Australian National University
References for this chapter

[Patterson17]
David A. Patterson & John L. Hennessy
Computer Organization and Design – The Hardware/Software Interface
Appendix A “The Basics of Logic Design”
ARM edition, Morgan Kaufmann 2017
An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities by George Boole, 1854
Boolean Values & Operators

There are two values:

- e.g. **True** and **False**. (aka “1” and “0”)

Two binary operators on expressions $a, b$:
- $a \lor b$ (aka $a + b$ or “$a$ OR $b$” or SUM)
- $a \land b$ (aka $a \cdot b$ or “$a$ AND $b$” or PRODUCT)

One unary operator on an expression $a$:

- $\bar{a}$ (aka $\neg a$ or $a'$ or “NOT $a$”)

Truth tables:

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$a \lor b$</th>
<th>$a \land b$</th>
<th>$\bar{a}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td></td>
</tr>
</tbody>
</table>
### Axiomatic Boolean Algebra (Whitehead 1898)

<table>
<thead>
<tr>
<th>∨-Laws</th>
<th>∧-Laws</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a \lor a = a )</td>
<td>( a \land a = a )</td>
</tr>
<tr>
<td>( a \lor b = b \lor a )</td>
<td>( a \land b = b \land a ) (commutative)</td>
</tr>
<tr>
<td>( (a \lor b) \lor c = a \lor (b \lor c) )</td>
<td>( (a \land b) \land c = a \land (b \land c) ) (associative)</td>
</tr>
<tr>
<td>( a \lor (a \land b) = a )</td>
<td>( a \land (b \lor c) = (a \land b) \lor (a \land c) ) (distribution)</td>
</tr>
<tr>
<td>( a \land \overline{a} = True )</td>
<td>( a \land \overline{a} = False )</td>
</tr>
</tbody>
</table>

Algebras allow for easier reasoning than truth tables.
### Axiomatic Boolean Algebra (Huntington 1904)

#### ∨-Laws

<table>
<thead>
<tr>
<th>∨-Laws</th>
<th>∧-Laws</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \lor b = b \lor a$</td>
<td>$a \land b = b \land a$</td>
</tr>
<tr>
<td>$a \lor (b \land c) = (a \lor b) \land (a \lor c)$</td>
<td>$a \land (b \lor c) = (a \land b) \lor (a \land c)$</td>
</tr>
<tr>
<td>$a \lor \text{False} = a$</td>
<td>$a \land \text{True} = a$</td>
</tr>
<tr>
<td>$a \lor \overline{a} = \text{True}$</td>
<td>$a \land \overline{a} = \text{False}$</td>
</tr>
</tbody>
</table>

#### Properties

- (commutative)
- (associative)
- (absorption)
- (distribution)
- (identity)
- (constant)
- (inverse)
- DeMorgan
- (double not)
Axiomatic Boolean Algebra

... many other axiomatic formulations of Boolean algebra exist.
### Redundant Boolean Algebra

#### ∨-Laws

<table>
<thead>
<tr>
<th>Law</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \lor a = a$</td>
<td>(redundant)</td>
</tr>
<tr>
<td>$a \lor b = b \lor a$</td>
<td>(commutative)</td>
</tr>
<tr>
<td>$(a \lor b) \lor c = a \lor (b \lor c)$</td>
<td>(associative)</td>
</tr>
<tr>
<td>$a \lor (a \land b) = a$</td>
<td>(absorption)</td>
</tr>
<tr>
<td>$a \lor (b \land c) = (a \lor b) \land (a \lor c)$</td>
<td>(distribution)</td>
</tr>
<tr>
<td>$a \lor \text{False} = a$</td>
<td>(identity)</td>
</tr>
<tr>
<td>$a \lor \text{True} = \text{True}$</td>
<td>(constant)</td>
</tr>
<tr>
<td>$a \lor \overline{a} = \text{True}$</td>
<td>(inverse)</td>
</tr>
<tr>
<td>$\overline{a} \lor b = \overline{a} \land b$</td>
<td>DeMorgan</td>
</tr>
</tbody>
</table>

#### ∧-Laws

<table>
<thead>
<tr>
<th>Law</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \land a = a$</td>
<td>(redundant)</td>
</tr>
<tr>
<td>$a \land b = b \land a$</td>
<td>(commutative)</td>
</tr>
<tr>
<td>$(a \land b) \land c = a \land (b \land c)$</td>
<td>(associative)</td>
</tr>
<tr>
<td>$a \land (a \lor b) = a$</td>
<td>(absorption)</td>
</tr>
<tr>
<td>$a \land (b \lor c) = (a \land b) \lor (a \land c)$</td>
<td>(distribution)</td>
</tr>
<tr>
<td>$a \land \text{True} = a$</td>
<td>(identity)</td>
</tr>
<tr>
<td>$a \land \text{False} = \text{False}$</td>
<td>(constant)</td>
</tr>
<tr>
<td>$a \land \overline{a} = \text{False}$</td>
<td>(inverse)</td>
</tr>
<tr>
<td>$\overline{a} \land b = \overline{a} \lor b$</td>
<td>DeMorgan</td>
</tr>
</tbody>
</table>

### DeMorgan's Theorem

$\overline{a} = a$
Common Boolean operators

Commonly used operators on expressions $a$, $b$ to define boolean algebras:

- $a \lor b$  (aka $a + b$ or “$a$ OR $b$” or SUM)
- $a \land b$  (aka $a \cdot b$ or “$a$ AND $b$” or PRODUCT)
- $\overline{a}$  (aka $\neg a$ or $a’$ or “not $a$”)

Other handy operators:

- $a \rightarrow b = (\overline{a} \lor b)$  (aka “$a$ IMPLIES $b$”)
- $(a = b) = (a \land b) \lor (\overline{a} \land \overline{b})$  (aka “$a$ EQUALS $b$”)
- $a \oplus b = (a \land \overline{b}) \lor (\overline{a} \land b)$  (aka “$a$ EXCLUSIVE-OR $b$” or “$a$ XOR $b$”)
- $\overline{a} \land \overline{b} = (\overline{a} \lor \overline{b})$  (aka “$a$ NOT-AND $b$” or “$a$ NAND $b$”)
- $\overline{a} \lor b = (\overline{a} \land b)$  (aka “$a$ NOT-OR $b$” or “$a$ NOR $b$”)

NAND and NOR are the only sole sufficient boolean operators, i.e. you can reduce any boolean expression to only NAND or only NOR operators.
### All binary Boolean operators

<table>
<thead>
<tr>
<th>Inputs $a, b$</th>
<th>Function $q$</th>
<th>Name</th>
<th>Sum of products</th>
<th>NAND $\neg \land, \neg \lor, \neg \bar{x}$</th>
<th>Don’t cares</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a, b$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$F, F, T, T$</td>
<td>$F, F, F, F$</td>
<td>False</td>
<td>Constant FALSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$F, F, F, T$</td>
<td>$F, F, F, T$</td>
<td>$a \land b$</td>
<td>AND</td>
<td>$a \land b \lor \neg a \lor a \land b$</td>
<td></td>
</tr>
<tr>
<td>$F, F, T, F$</td>
<td>$F, F, T, F$</td>
<td>$\neg a \to b$</td>
<td>NOT-IMPLICATION</td>
<td>$(a \lor \neg b)\lor (a \land b)$</td>
<td></td>
</tr>
<tr>
<td>$F, F, T, T$</td>
<td>$F, F, T, T$</td>
<td>$a$</td>
<td>IDENTITY $a$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$F, T, F, F$</td>
<td>$F, T, F, F$</td>
<td>$\neg b \to a$</td>
<td>NOT-IMPLICATION</td>
<td>$(\neg a \lor b)\lor (\neg a \land b)$</td>
<td></td>
</tr>
<tr>
<td>$F, T, F, T$</td>
<td>$F, T, F, T$</td>
<td>$b$</td>
<td>IDENTITY $b$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$F, T, T, F$</td>
<td>$F, T, T, T$</td>
<td>$a \oplus b$</td>
<td>EXCLUSIVE-OR, XOR</td>
<td>$(a \lor \neg b)\lor (a \land b)$</td>
<td></td>
</tr>
<tr>
<td>$F, T, T, T$</td>
<td>$F, T, T, T$</td>
<td>$a \lor b$</td>
<td>OR</td>
<td>$a \lor a \lor b \lor b$</td>
<td></td>
</tr>
<tr>
<td>$T, F, F, F$</td>
<td>$T, F, F, T$</td>
<td>$\neg a \lor b$</td>
<td>NOT-OR, NOR</td>
<td>$(\neg a \lor b)$</td>
<td></td>
</tr>
<tr>
<td>$T, F, F, T$</td>
<td>$T, F, T, F$</td>
<td>$a = b$</td>
<td>EQUALITY, EQ</td>
<td>$(a \lor b)\lor (a \land b)$</td>
<td></td>
</tr>
<tr>
<td>$T, F, F, T$</td>
<td>$T, F, T, F$</td>
<td>$\neg b$</td>
<td>INVERSE $b$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T, F, T, T$</td>
<td>$T, F, T, T$</td>
<td>$b \to a$</td>
<td>IMPLICATION</td>
<td>$a \lor \neg b$</td>
<td></td>
</tr>
<tr>
<td>$T, T, F, F$</td>
<td>$T, T, F, F$</td>
<td>$\neg a$</td>
<td>INVERSE $a$</td>
<td>$a \lor a$</td>
<td></td>
</tr>
<tr>
<td>$T, T, F, T$</td>
<td>$T, T, F, T$</td>
<td>$a \to b$</td>
<td>IMPLICATION</td>
<td>$a \lor b$</td>
<td></td>
</tr>
<tr>
<td>$T, T, T, F$</td>
<td>$T, T, T, F$</td>
<td>$a \land b$</td>
<td>NOT-AND, NAND</td>
<td>$a \lor \neg b$</td>
<td></td>
</tr>
<tr>
<td>$T, T, T, T$</td>
<td>$T, T, T, T$</td>
<td>True</td>
<td>Constant True</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2021 Uwe R. Zimmer, The Australian National University
Combinational Logic Functions

Logic is reducible/equivalent to pure functions: there are no states!

If the function is combinational then there is only one output for any combination of inputs, e.g. the function can be written out as a truth table:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Output q</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>
**Combinational Logic Functions**

Logic is reducible/equivalent to pure functions: there are no states!

If the function is combinational then there is only one output for any combination of inputs, e.g. the function can be written out as a truth table:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Output q</th>
<th>minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>$\bar{a} \land b \land \bar{c}$</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>$a \land \bar{b} \land \bar{c}$</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>$a \land \bar{b} \land c$</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>$a \land b \land \bar{c}$</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

Sum of minterms: $q = (\bar{a} \land b \land \bar{c}) \lor (a \land \bar{b} \land \bar{c}) \lor (a \land \bar{b} \land c) \lor (a \land b \land \bar{c})$
Logic is reducible/equivalent to pure functions: there are no states!

IF the function is combinational then there is only one output for any combination of inputs, e.g. the function can be written out as a truth table:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Output q</th>
<th>minterms</th>
<th>Simplified minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>a \land b \land \overline{c}</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td></td>
<td>b \land \overline{c}</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>a \land b \land \overline{c}</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td></td>
<td>a \land b \land \overline{c}</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td></td>
<td>a \land b \land \overline{c}</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>a \land b \land \overline{c}</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sum of minterms: $q = (\overline{a} \land b \land \overline{c}) \lor (a \land \overline{b} \land \overline{c}) \lor (a \land b \land \overline{c}) \lor (a \land b \land c) \lor (a \land b \land \overline{c})$

Sum of simplified minterms: $q = (a \land \overline{b}) \lor (b \land \overline{c})$

Simplifications can be done by (automated) algebraic transformations, Karnaugh maps or others.
Logic is reducible/equivalent to pure functions: there are no states!

If the function is combinational then there is only one output for any combination of inputs, e.g. the function can be written out as a truth table:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Output q</th>
<th>minterms</th>
<th>Simplified minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>$\bar{a} \land b \land \bar{c}$</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>$a \land \bar{b} \land \bar{c}$</td>
<td>$a \land \bar{b}$</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>$a \land \bar{b} \land c$</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>$a \land b \land \bar{c}$</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sum of minterms: $q = (\bar{a} \land b \land \bar{c}) \lor (a \land \bar{b} \land \bar{c}) \lor (a \land \bar{b} \land c) \lor (a \land b \land \bar{c})$

Sum of simplified minterms: $q = (a \land \bar{b}) \lor (b \land \bar{c})$

Simplifications can be done by (automated) algebraic transformations, Karnaugh maps or others
Combinational Logic Functions

Logic is reducible/equivalent to pure functions: there are no states!

If the function is combinational then there is only one output for any combination of inputs, e.g. the function can be written out as a truth table:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Output q</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>
Logic is reducible/equivalent to pure functions: there are no states!

If the function is combinational then there is only one output for any combination of inputs, e.g. the function can be written out as a truth table:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Output q</th>
<th>maxterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

maxterms product  \( q = (a \lor b \lor c) \land (a \lor b \lor \overline{c}) \land (a \lor \overline{b} \lor \overline{c}) \land (\overline{a} \lor \overline{b} \lor \overline{c}) \)
Combinational Logic Functions

Logic is reducible/equivalent to pure functions: there are no states!

If the function is combinational then there is only one output for any combination of inputs, e.g. the function can be written out as a truth table:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Output q</th>
<th>maxterms</th>
<th>Simplified maxterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>$a \lor b \lor c$</td>
<td>$a \lor b$</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>$a \lor b \lor \overline{c}$</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>$\overline{b} \lor \overline{c}$</td>
<td>$\overline{b} \lor \overline{c}$</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>$a \lor \overline{b} \lor \overline{c}$</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td></td>
<td>$\overline{b} \lor \overline{c}$</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>$\overline{a} \lor \overline{b} \lor \overline{c}$</td>
<td></td>
</tr>
</tbody>
</table>

Maxterms product \( q = (a \lor b \lor c) \land (a \lor b \lor \overline{c}) \land (a \lor \overline{b} \lor \overline{c}) \land (\overline{a} \lor \overline{b} \lor \overline{c}) \)

Simplified maxterms product \( q = (a \lor b) \land (\overline{b} \lor \overline{c}) \)

Simplifications can be done by (automated) algebraic transformations, Karnaugh maps or others.
Logic is reducible/equivalent to pure functions: there are no states!

If the function is combinational then there is only one output for any combination of inputs, e.g. the function can be written out as a truth table:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Output q</th>
<th>maxterms</th>
<th>Simplified maxterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>$a \lor b \lor c$</td>
<td>$a \lor b$</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>$a \lor b \lor \overline{c}$</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td></td>
<td>$\overline{b} \lor \overline{c}$</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>$a \lor \overline{b} \lor \overline{c}$</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td></td>
<td>$\overline{a} \lor \overline{b} \lor \overline{c}$</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Every combinational function can be written as a **product of sums**!

Simplifications can be done by (automated) algebraic transformations, Karnaugh maps or others.
Symbolic: \( Q = \overline{A \land B} \)

Diagram:

Technology:
Combinational Logic Functions

The logic equivalent to pure functions: there are no states!

If the function is combinational then there is only one output for any combination of inputs, e.g. the function can be written out as a truth table:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Output q</th>
<th>Minterms</th>
<th>Simplified minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>$\overline{a} \land b \land \overline{c}$</td>
<td>$a \land \overline{b}$</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>$a \land \overline{b} \land \overline{c}$</td>
<td>$b \land \overline{c}$</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>$a \land \overline{b} \land c$</td>
<td>$a \land \overline{b}$</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>$a \land b \land \overline{c}$</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sum of minterms: $q = (\overline{a} \land b \land \overline{c}) \lor (a \land \overline{b} \land \overline{c}) \lor (a \land \overline{b} \land c) \lor (a \land b \land \overline{c})$

Sum of simplified minterms: $q = (a \land \overline{b}) \lor (b \land \overline{c})$

Simplifications can be done by (automated) algebraic transformations, Karnaugh maps or others.
**Combinational Logic Functions**

The logic equivalent to pure functions: there are no states!

If the function is combinational then there is only one output for any combination of inputs, e.g. the function can be written out as a truth table:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Output q</th>
<th>Minterms</th>
<th>Simplified minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>$\bar{a} \land b \land \bar{c}$</td>
<td>(a \land \bar{b})</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>(a \land \bar{b} \land \bar{c})</td>
<td>(b \land \bar{c})</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>(a \land \bar{b} \land c)</td>
<td>(a \land \bar{b})</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>(a \land b \land \bar{c})</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>(a \land b \land c)</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>(a \land b \land \bar{c})</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sum of minterms: \(q = (\bar{a} \land b \land \bar{c}) \lor (a \land \bar{b} \land \bar{c}) \lor (a \land \bar{b} \land c) \lor (a \land b \land \bar{c})\)

Sum of simplified minterms: \(q = (a \land \bar{b}) \lor (b \land \bar{c})\)

Simplifications can be done by (automated) algebraic transformations, Karnaugh maps or others.
The logic equivalent to pure functions: there are no states!

If the function is combinational then there is only one output for any combination of inputs, e.g. the function can be written out as a truth table:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Output q</th>
<th>Minterms</th>
<th>Simplified minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>( \overline{a} \wedge b \wedge \overline{c} )</td>
<td>( a \wedge \overline{b} )</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>( a \wedge \overline{b} \wedge \overline{c} )</td>
<td>( a \wedge \overline{b} \wedge \overline{c} )</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>( a \wedge \overline{b} \wedge c )</td>
<td>( a \wedge \overline{b} \wedge c )</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>( a \wedge b \wedge \overline{c} )</td>
<td>( a \wedge b \wedge \overline{c} )</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sum of minterms: \( q = (\overline{a} \wedge b \wedge \overline{c}) \lor (a \wedge \overline{b} \wedge \overline{c}) \lor (a \wedge \overline{b} \wedge c) \lor (a \wedge b \wedge \overline{c}) \)

Sum of simplified minterms: \( q = (a \wedge \overline{b}) \lor (b \wedge \overline{c}) \)

Simplifications can be done by (automated) algebraic transformations, Karnaugh maps or others.
Encrypting a bit vector (whatever it represents) with a secret key:

Assuming the key is random and not used for anything else:

- This is surprisingly secure
- … and extremely fast!
Bit Vectors

Groups of bits could represent:

States, enumeration values, arrays of Booleans, numbers, etc. pp. … or any grouping or combination of the above

Algebraic Types

The form of encoding could be chosen to optimize for:

- **Performance** e.g. minimal decoding effort
- **Redundancy / error detection** e.g. large Hamming distance
- **Safe transitions** e.g. Gray codes
- **Physical mapping** e.g. maps on existing hardware interfaces
- **Compactness** e.g. holds the maximal number of values per memory cell
Encoding

Assuming a type can have 7 different values, many forms of encoding are possible:

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
<th>Single bit</th>
<th>Gray code</th>
<th>Even parity</th>
<th>1-bit error detecting</th>
<th>1-bit error correcting</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Secured</td>
<td>0000001</td>
<td>000</td>
<td>0000</td>
<td>0000000</td>
<td>0000000000</td>
<td>000</td>
</tr>
<tr>
<td>2</td>
<td>Taxi</td>
<td>0000010</td>
<td>001</td>
<td>0011</td>
<td>1110000</td>
<td>000000111</td>
<td>001</td>
</tr>
<tr>
<td>3</td>
<td>Take-off</td>
<td>0001000</td>
<td>011</td>
<td>0101</td>
<td>1001100</td>
<td>000111000</td>
<td>010</td>
</tr>
<tr>
<td>4</td>
<td>Cruising</td>
<td>0001000</td>
<td>010</td>
<td>0110</td>
<td>0111100</td>
<td>000111111</td>
<td>011</td>
</tr>
<tr>
<td>5</td>
<td>Gliding</td>
<td>0010000</td>
<td>110</td>
<td>1001</td>
<td>0101010</td>
<td>111000000</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>Approach</td>
<td>0100000</td>
<td>111</td>
<td>1010</td>
<td>1011010</td>
<td>111000111</td>
<td>101</td>
</tr>
<tr>
<td>7</td>
<td>Landing</td>
<td>1000000</td>
<td>101</td>
<td>1100</td>
<td>1100110</td>
<td>111111000</td>
<td>110</td>
</tr>
</tbody>
</table>

VHDL or Verilog gives you full control over the encoding.
Binary encoding

-Encoding of choice if compactness is essential or you need to add values.

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
\times 2^7 & \times 2^6 & \times 2^5 & \times 2^4 & \times 2^3 & \times 2^2 & \times 2^1 & \times 2^0 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
32 & + & 8 & + & 2 & = 42 \\
\end{array}
\]
**Binary encoding**

Encoding of choice if compactness is essential or you need to add values.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

Binary Hexadecimal Decimal

\[
\begin{align*}
0 \times 2^7 & + 0 \times 2^6 & + 1 \times 2^5 & + 0 \times 2^4 & + 1 \times 2^3 & + 0 \times 2^2 & + 1 \times 2^1 & + 0 \times 2^0 = 42 \\
32 & + 8 & + 2 & = 42
\end{align*}
\]
Binary encoding

Encoding of choice if compactness is essential or you need to add values.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1 0 0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0 0 0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 0 1 0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0 0 0</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 0 1 0</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0 1 0 1 0 0</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0 1 0 1 1 0</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0 0 0</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1 0 0</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1 0 1 0 0 0</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1 0 1 0 1 0</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1 1 0 0 0 0</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1 1 0 0 1 0</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1 1 1 0 0 0</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1 1 1 0 1 1</td>
<td>F</td>
</tr>
</tbody>
</table>

Binary to Hexadecimal Table:

- 0 = 0
- 1 = 1
- 2 = 2
- 3 = 3
- 4 = 4
- 5 = 5
- 6 = 6
- 7 = 7
- 8 = 8
- 9 = 9
- A = A
- B = B
- C = C
- D = D
- E = E
- F = F

Decimal: 32 + 8 + 2 = 42

Binary: 0 0 1 0 1 0

Hexadecimal: 20
## Half Adder

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>S</th>
<th>C</th>
<th>S minterms</th>
<th>C minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\overline{A} \land B$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$A \land \overline{B}$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>$A \land B$</td>
</tr>
</tbody>
</table>
### Half Adder

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>S</th>
<th>C</th>
<th>S minterms</th>
<th>C minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(\overline{A} \land B)</td>
<td>(A \land \overline{B})</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>(A \land \overline{B})</td>
<td>(A \land B)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(A \land B)</td>
<td></td>
</tr>
</tbody>
</table>

\[ S = (A \land \overline{B}) \lor (\overline{A} \land B) \]

\[ C = A \land B \]
## Half Adder

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>S</th>
<th>C</th>
<th>S minterms</th>
<th>C minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\overline{A} \land B$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$A \land \overline{B}$</td>
<td>$A \land B$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ S = (A \land \overline{B}) \lor (\overline{A} \land B) = A \oplus B \]

\[ C = A \land B \]
### Half Adder

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>S</th>
<th>C</th>
<th>S minterms</th>
<th>C minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\overline{A} \land B$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$A \land \overline{B}$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$A \land B$</td>
<td>$A \land B$</td>
</tr>
</tbody>
</table>

\[ S = (A \land \overline{B}) \lor (\overline{A} \land B) = A \oplus B \]

\[ C = A \land B \]
## Full Adder

<table>
<thead>
<tr>
<th>$A_i$</th>
<th>$B_i$</th>
<th>$C_{i-1}$</th>
<th>$S_i$</th>
<th>$C_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</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>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
### Full Adder

<table>
<thead>
<tr>
<th>$A_i$</th>
<th>$B_i$</th>
<th>$C_{i-1}$</th>
<th>$S_i$</th>
<th>$C_i$</th>
<th>$S_i$ minterms</th>
<th>$C_i$ minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\overline{A_i} \land \overline{B_i} \land \overline{C_{i-1}}$</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$A_i \land \overline{B_i} \land \overline{C_{i-1}}$</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\overline{A_i} \land \overline{B_i} \land C_{i-1}$</td>
<td>$A_i \land B_i \land \overline{C_{i-1}}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$A_i \land \overline{B_i} \land C_{i-1}$</td>
<td>$A_i \land B_i \land \overline{C_{i-1}}$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\overline{A_i} \land B_i \land C_{i-1}$</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$\overline{A_i} \land B_i \land C_{i-1}$</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
</tr>
</tbody>
</table>

The output $S_i = (A_i \land \overline{B_i} \land \overline{C_{i-1}}) \lor (\overline{A_i} \land B_i \land \overline{C_{i-1}}) \lor (\overline{A_i} \land \overline{B_i} \land C_{i-1}) \lor (A_i \land B_i \land C_{i-1})$.

---

© 2021 Uwe R. Zimmer, The Australian National University
### Full Adder

<table>
<thead>
<tr>
<th>Ai</th>
<th>Bi</th>
<th>Ci-1</th>
<th>S_i</th>
<th>C_i</th>
<th>S_i minterms</th>
<th>C_i minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( \overline{A_i} \land B_i \land \overline{C_{i-1}} )</td>
<td>( A_i \land B_i \land \overline{C_{i-1}} )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( A_i \land \overline{B_i} \land \overline{C_{i-1}} )</td>
<td>( A_i \land B_i \land \overline{C_{i-1}} )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( \overline{A_i} \land \overline{B_i} \land C_{i-1} )</td>
<td>( A_i \land \overline{B_i} \land C_{i-1} )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( \overline{A_i} \land B_i \land C_{i-1} )</td>
<td>( \overline{A_i} \land B_i \land C_{i-1} )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( A_i \land \overline{B_i} \land C_{i-1} )</td>
<td>( A_i \land B_i \land C_{i-1} )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( A_i \land B_i \land C_{i-1} )</td>
<td>( \overline{A_i} \land B_i \land C_{i-1} )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( A_i \land B_i \land C_{i-1} )</td>
<td>( \overline{A_i} \land B_i \land C_{i-1} )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( A_i \land B_i \land C_{i-1} )</td>
<td>( \overline{A_i} \land B_i \land C_{i-1} )</td>
</tr>
</tbody>
</table>

\[ S_i = (A_i \land \overline{B_i} \land \overline{C_{i-1}}) \lor (\overline{A_i} \land B_i \land \overline{C_{i-1}}) \lor (\overline{A_i} \land \overline{B_i} \land C_{i-1}) \lor (A_i \land B_i \land C_{i-1}) \]

\[ = (((A_i \land \overline{B_i}) \lor (\overline{A_i} \land B_i)) \land C_{i-1}) \lor (((\overline{A_i} \land \overline{B_i}) \lor (A_i \land B_i)) \land C_{i-1}) \]

\[ = ((A_i \oplus B_i) \land C_{i-1}) \lor ((A_i = B_i) \land C_{i-1}) = ((A_i \oplus B_i) \land C_{i-1}) \lor ((\overline{A_i} \oplus B_i) \land \overline{C_{i-1}}) \]

\[ = (A_i \oplus B_i) \oplus C_{i-1} \]
Full Adder

<table>
<thead>
<tr>
<th>$A_i$</th>
<th>$B_i$</th>
<th>$C_{i-1}$</th>
<th>$S_i$</th>
<th>$C_i$</th>
<th>$S_i$ minterms</th>
<th>$C_i$ minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\overline{A}_i \land \overline{B}<em>i \land \overline{C}</em>{i-1}$</td>
<td>$A_i \land B_i \land \overline{C}_{i-1}$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$A_i \land \overline{B}<em>i \land \overline{C}</em>{i-1}$</td>
<td>$A_i \land B_i \land \overline{C}_{i-1}$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\overline{A}_i \land \overline{B}<em>i \land C</em>{i-1}$</td>
<td>$\overline{A}_i \land \overline{B}<em>i \land C</em>{i-1}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$A_i \land \overline{B}<em>i \land C</em>{i-1}$</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\overline{A}<em>i \land B_i \land C</em>{i-1}$</td>
<td>$A_i \land \overline{B}<em>i \land C</em>{i-1}$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$A_i \land \overline{B}<em>i \land C</em>{i-1}$</td>
<td>$\overline{A}_i \land \overline{B}<em>i \land C</em>{i-1}$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
<td>$A_i \land \overline{B}<em>i \land C</em>{i-1}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
</tr>
</tbody>
</table>

$S_i = (A_i \oplus B_i) \oplus C_{i-1}$

$C_i = (A_i \land B_i \land \overline{C}_{i-1}) \lor (\overline{A}_i \land B_i \land C_{i-1}) \lor (A_i \land \overline{B}_i \land C_{i-1}) \lor (A_i \land B_i \land C_{i-1})$
### Full Adder

<table>
<thead>
<tr>
<th>$A_i$</th>
<th>$B_i$</th>
<th>$C_{i-1}$</th>
<th>$S_i$</th>
<th>$S_i$ minterms</th>
<th>$C_i$ minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$\overline{A_i} \land B_i \land \overline{C_{i-1}}$</td>
<td>$A_i \land \overline{B_i} \land \overline{C_{i-1}}$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$A_i \land \overline{B_i} \land \overline{C_{i-1}}$</td>
<td>$A_i \land B_i \land \overline{C_{i-1}}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$A_i \land \overline{B_i} \land C_{i-1}$</td>
<td>$\overline{A_i} \land B_i \land C_{i-1}$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$A_i \land \overline{B_i} \land C_{i-1}$</td>
<td>$A_i \land \overline{B_i} \land C_{i-1}$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
<td>$A_i \land \overline{B_i} \land C_{i-1}$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
<td>$A_i \land \overline{B_i} \land C_{i-1}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
</tr>
</tbody>
</table>

\[ S_i = (A_i \oplus B_i) \oplus C_{i-1} \]

\[ C_i = (A_i \land B_i \land \overline{C_{i-1}}) \lor (\overline{A_i} \land B_i \land C_{i-1}) \lor (A_i \land \overline{B_i} \land C_{i-1}) \lor (A_i \land B_i \land C_{i-1}) \]

\[ = (A_i \land B_i \land \overline{C_{i-1}}) \lor (A_i \land B_i \land C_{i-1}) \lor (\overline{A_i} \land B_i \land C_{i-1}) \lor (A_i \land \overline{B_i} \land C_{i-1}) \]

\[ = (A_i \land B_i) \lor (((\overline{A_i} \land B_i) \lor (A_i \land \overline{B_i})) \land C_{i-1}) \]

\[ = (A_i \land B_i) \lor ((A_i \oplus B_i) \land C_{i-1}) \]
### Full Adder

<table>
<thead>
<tr>
<th>$A_i$</th>
<th>$B_i$</th>
<th>$C_{i-1}$</th>
<th>$S_i$</th>
<th>$C_i$</th>
<th>$S_i$ minterms</th>
<th>$C_i$ minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\overline{A_i} \land B_i \land \overline{C_{i-1}}$</td>
<td>$A_i \land \overline{B_i} \land \overline{C_{i-1}}$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$A_i \land B_i \land \overline{C_{i-1}}$</td>
<td>$\overline{A_i} \land B_i \land C_{i-1}$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\overline{A_i} \land \overline{B_i} \land C_{i-1}$</td>
<td>$A_i \land B_i \land \overline{C_{i-1}}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\overline{A_i} \land B_i \land C_{i-1}$</td>
<td>$\overline{A_i} \land B_i \land C_{i-1}$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\overline{A_i} \land \overline{B_i} \land C_{i-1}$</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
<td>$A_i \land \overline{B_i} \land C_{i-1}$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
<td>$A_i \land B_i \land C_{i-1}$</td>
</tr>
</tbody>
</table>

**$S_i = (A_i \oplus B_i) \oplus C_{i-1}$**

**$C_i = (A_i \land B_i) \lor ((A_i \oplus B_i) \land C_{i-1})$**
Ripple Carry Adder

\[ 2 + 2 = 4 \]
Ripple Carry Adder

\[ 2 + 2 = 4! \]
Ripple Carry Adder

\[
\begin{align*}
2 - 1 &= 1 \\
\end{align*}
\]
Radix complements

Can we define negative numbers such that our adder still works?

\[ x - x = 0 \]

Or: what can you add to 42 in an 8-bit binary representation such that the result will be \(2^8\) (and hence 0 in 8 bits)?

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\quad +
\quad \begin{array}{cccccccc}
\end{array}
\quad =
\quad \begin{array}{cccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[ 42 \]
\[ + \]
\[ -42 \]
\[ = 256 \]
Radix complements

Can we define negative numbers such that our adder still works?

\[ x - x = 0 \]

Or: what can you add to 42 in an 8-bit binary representation such that the result will be \(2^8\) (and hence 0 in 8 bits)?

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
+ & & & & & & & \\
1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
\end{array}
\]

\[ 42 \]

\[ 100000000 \]

\[ 256 \]
Radix complements

Can we define negative numbers such that our adder still works?

\[ x - x = 0 \]

Or: what can you add to 42 in an 8 bit binary representation such that the result will be \(2^8\) (and hence 0 in 8 bits)?

```
0 0 1 0 1 0 1 0  # 42
+               
1 1 0 1 0 1 1 0  # -42
=               
1 0 0 0 0 0 0 0 0  # 256
```

“Invert all bits and add 1”

2’s-complement (as the radix/base is 2)
2’s complements

The 2’s complement encoding interprets the natural binary range $2^{n-1} \ldots 2^n - 1$ as negative numbers $-2^{n-1} \ldots -1$.
2’s complements

The 2’s complement encoding interprets the natural binary range $2^{n-1} \ldots 2^{n} - 1$ as negative numbers $-2^{n-1} \ldots -1$

It’s all in your mind!

<table>
<thead>
<tr>
<th>Natural binary numbers</th>
<th>2^n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1111111111</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0 ...</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2's complement binary numbers</th>
<th>-2^n-1</th>
<th>0</th>
<th>2^n-1-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 0 0 0 ...</td>
<td>0 0 0 0 0 0 0 0 ...</td>
<td>0 1 1 1 1 1 1 1</td>
<td></td>
</tr>
</tbody>
</table>

© 2021 Uwe R. Zimmer, The Australian National University
Ripple Carry Adder

2 - 1 = 1?
Ripple Carry Adder

2 - 1 = 1!

… with an overall carry-flag indicated.
How long does it take until the last carry flag stabilizes?
What distinguishes the red from the green gates?

- Carry-lookahead circuitry
A simple ALU which can ADD, XOR, AND, OR two arguments.
Towards States
(everything up to here was combinational logic)

How do we make operations depends on:

- ... an overflow in the previous operation?
- ... the state of the CPU?
- ... a counter having reached zero?
- ... two arguments having been equal?
- ... etc. pp.

- We need to hold on to some states!
States

<table>
<thead>
<tr>
<th>$\overline{S}$</th>
<th>$\overline{R}$</th>
<th>$Q$</th>
<th>$\overline{Q}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
### States

**Truth Table**

<table>
<thead>
<tr>
<th>$\overline{S}$</th>
<th>$\overline{R}$</th>
<th>$Q$</th>
<th>$\overline{Q}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Q</td>
<td>$\overline{Q}$</td>
</tr>
</tbody>
</table>
States

<table>
<thead>
<tr>
<th>( \bar{S} )</th>
<th>( \bar{R} )</th>
<th>( Q )</th>
<th>( \bar{Q} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>?</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>?</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>?</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>Q</td>
<td>( \bar{Q} )</td>
<td></td>
</tr>
</tbody>
</table>
States

<table>
<thead>
<tr>
<th>$S$</th>
<th>$R$</th>
<th>$Q$</th>
<th>$\overline{Q}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Q</td>
<td>Q</td>
</tr>
</tbody>
</table>
States

\[
\begin{array}{c|c|c|c|c|c}
S & R & Q & \overline{Q} \\
0 & 0 & ? & ? \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 \\
1 & 1 & Q & \overline{Q} \\
\end{array}
\]
States

Assuming \( Q \) as well as \( \overline{Q} \) to be active simultaneously may lead to instability.
“S-R Flip-Flop”

<table>
<thead>
<tr>
<th>$S$</th>
<th>$R$</th>
<th>$Q$</th>
<th>$\overline{Q}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</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>$0$</td>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td>$1$</td>
<td>$1$</td>
<td>$Q$</td>
<td>$\overline{Q}$</td>
</tr>
</tbody>
</table>
## Deriving SR Flip Flops

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>*</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>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>
### Deriving SR Flip Flops

<table>
<thead>
<tr>
<th>$\bar{S}$</th>
<th>$\bar{R}$</th>
<th>$Q$</th>
<th>$Q$ minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>*</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>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>$\bar{S} \land \bar{R} \land Q$</td>
</tr>
</tbody>
</table>
### Deriving SR Flip Flops

<table>
<thead>
<tr>
<th>$\overline{S}$</th>
<th>$\overline{R}$</th>
<th>$Q$</th>
<th>Q minterms</th>
<th>Simplified</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$S \wedge \overline{R} \wedge Q$</td>
<td>$S$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$S \wedge \overline{R} \wedge Q$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$S \wedge \overline{R} \wedge Q$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$S \wedge \overline{R} \wedge Q$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$S \wedge \overline{R} \wedge Q$</td>
<td>$\overline{R} \wedge Q$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$S \wedge \overline{R} \wedge Q$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$S \wedge \overline{R} \wedge Q$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$S \wedge \overline{R} \wedge Q$</td>
<td></td>
</tr>
</tbody>
</table>

$q \Rightarrow Q = S \lor (\overline{R} \wedge Q)$
### Deriving SR Flip Flops

<table>
<thead>
<tr>
<th>$\bar{S}$</th>
<th>$\bar{R}$</th>
<th>$Q$</th>
<th>Q minterms</th>
<th>Simplified</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$S \land \bar{R} \land \bar{Q}$</td>
<td>$S$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$S \land \bar{R} \land Q$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$S \land \bar{R} \land \bar{Q}$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$S \land \bar{R} \land Q$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$S \land \bar{R} \land \bar{Q}$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$S \land \bar{R} \land Q$</td>
<td>$\bar{R} \land Q$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$S \land \bar{R} \land \bar{Q}$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$S \land \bar{R} \land Q$</td>
<td></td>
</tr>
</tbody>
</table>

\[ Q = S \lor (\bar{R} \land Q) = \bar{S} \land \bar{R} \land Q \]
### Deriving SR Flip Flops

<table>
<thead>
<tr>
<th>$S$</th>
<th>$R$</th>
<th>$Q$</th>
<th>Q minterms</th>
<th>Simplified</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$S \land R \land \overline{Q}$</td>
<td>$S$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$S \land R \land Q$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$S \land \overline{R} \land \overline{Q}$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$S \land \overline{R} \land Q$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$S \land \overline{R} \land Q$</td>
<td>$\overline{R} \land Q$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$S \land \overline{R} \land Q$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$S \land \overline{R} \land Q$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$S \land \overline{R} \land Q$</td>
<td></td>
</tr>
</tbody>
</table>

Q = $S \lor (\overline{R} \land Q) = \overline{S} \land \overline{R} \land Q$

\[ Q = S \lor (\overline{R} \land Q) = \overline{S} \land \overline{R} \land Q \]
D Flip-Flop
**D Flip-Flop**

- **Set pre-latch**
  - NAND
  - NAND
  - NAND
  - NAND
  - NAND

- **Reset pre-latch**
  - NAND
  - NAND
  - NAND

- Inputs: D, C
- Outputs: Q, \(\overline{Q}\)
**D Flip-Flop**

The diagram illustrates the operation of a D Flip-Flop using NAND gates. The inputs and outputs are labeled as follows:

- **D** as the data input
- **C** as the clock input
- **Q** as the output
- **Q̅** as the complement of the output

The set pre-latch and reset pre-latch are handled by the NAND gates connected to the clock and data inputs, respectively.

© 2021 Uwe R. Zimmer, The Australian National University
D Flip-Flop

Set pre-latch

Reset pre-latch

© 2021 Uwe R. Zimmer, The Australian National University
D Flip-Flop

\[ D \to Q \]

\[ \overline{Q} \]

\[ C \]

\[ \bar{C} \]

\[ D \]

\[ \bar{D} \]

\[ \text{Set pre-latch} \]

\[ \text{Reset pre-latch} \]
D Flip-Flop
D Flip-Flop
D Flip-Flop

Set pre-latch

Reset pre-latch
Master-Slave JK Flip-Flop
Master-Slave JK Flip-Flop

\[ \begin{align*}
J & \quad S & \quad Q \\
K & \quad R & \quad \bar{Q} \\
C & \quad \bar{C} & \quad \bar{Q}
\end{align*} \]
Master-Slave JK Flip-Flop

\[
\begin{align*}
\text{Master} & : \quad \overline{Q} & = & & \text{NAND} \quad \text{NAND} \\
\text{Slave} & : \quad Q & = & & \text{NAND} \quad \text{NAND}
\end{align*}
\]
Master-Slave JK Flip-Flop

Master is reset on the rising clock edge
Master-Slave JK Flip-Flop
Master-Slave JK Flip-Flop

Slave follows on the falling clock edge
Master-Slave JK Flip-Flop

The decoupling between the two stages makes this flip-flop race free – even in JK-toggle mode.

Slave follows on the falling clock edge.
Could serve as a generic, fast storage inside the CPU (general register)

Or to hold internal states (e.g. ALU overflow) of the CPU which are used by e.g. branching instructions.
Toggle Flip-Flops change state with every clock cycle.
Your controller has many counters which can e.g. be used to delay operations without the need to execute instructions.
You can already build most of the components of a CPU by now.

(The most essential missing component is the sequencer which is a specialized state-machine.)

We will come back to the CPU architectures towards the end of the course.

The next chapter will be about programming a CPU at machine level.
STM32L476 Discovery

Multiplexed 24 bit ΣΔ-DAC converter with stereo power amp

Headphone jack

USB OTG

“9 axis” motion sensor (underneath display):
- 3 axis accelerometer
- 3 axis gyroscope
- 3 axis magnetometer

Current meter to MCU
- 60 nA … 50 mA

Microphone
There is a lot more hardware here than you could possibly master in one semester …

... and you will master a lot more about CPUs at the end the course than you think now.
Summary

Digital Logic

• Boolean Algebra
  • Truth tables and Boolean operations
  • Minterms and simplifying expressions

• Combinational Logic
  • Logic gates
  • Numbers
  • Adders, ALU

• State-oriented Logic
  • Flip-Flops, registers and counters

• CPU Architecture
Hardware/Software Interface

Uwe R. Zimmer - The Australian National University
References for this chapter

[Patterson17]
David A. Patterson & John L. Hennessy
Computer Organization and Design – The Hardware/Software Interface
Chapter 2 “Instructions: Language of the Computer” & Chapter 3 “Arithmetic for Computers”
ARM edition, Morgan Kaufmann 2017
Adding the value of two registers

The CPU will fetch the content of the memory cell which PC is pointing to.

☞ We want the CPU to execute:

\[ r4 := r2 + r3 \]

☞ What to store in this memory cell?
**Hardware/Software Interface**

**Adding the value of two registers**

Register bank

<table>
<thead>
<tr>
<th>r0</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>r4</th>
<th>r5</th>
<th>r6</th>
<th>r7</th>
</tr>
</thead>
</table>

| r8 | r9 | r10 | r11 | r12 | SP | LR | PC |

Status flags

- ALU
- NZCVQ

**ADDS <Rd>, <Rn>, <Rm>**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| 0  | 0  | 0  | 1  | 1  | 0  | 0 | Rm | Rn | Rd |

Op Code           Arguments
Adding the value of two registers

**Register bank**

- **ADD** $r4$, $r2$, $r3$

**Status flags set:**
- N Negative (MSB = 1)
- Z Zero (all bits zero)
- C Carry (carry out)
- V Overflow (sign wrong)

**Assembler**

```
r4 := r2 + r3
```

**Disassembler**
Adding the value of two registers

Register bank

```
r0  r1  r2  r3  r4  r5  r6  r7
```

```
r8  r9  r10 r11 r12 SP  LR  PC
```

Status flags

```
ALU  NZCVQ
```

ANDS <Rdn>, <Rm>

```
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
```

Op Code

Arguments
Adding the value of two registers

Register bank

<table>
<thead>
<tr>
<th>r0</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>r4</th>
<th>r5</th>
<th>r6</th>
<th>r7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r8</th>
<th>r9</th>
<th>r10</th>
<th>r11</th>
<th>r12</th>
<th>SP</th>
<th>LR</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Status flags

ALU

NZCVQ

Disassembler

ASDS r5, r6

Assembler

| 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  |

16#40#  16#35#

r5 := r5 & r6
Adding the value of two registers

Add the value of two registers:

\[ r4 := r2 + r3 \]

Status flags set:
- N Negative (MSB = 1)
- Z Zero (all bits zero)
- C Carry (carry out)
- V Overflow (sign wrong)

Assembler

Disassembler

ADD 4, 2, 3

16#18# 16#D4#

0 0 0 1 1 0 0 0 1 1 0 1 0 1 0 0

© 2021 Uwe R. Zimmer, The Australian National University
ARM v7-M 32 bit add instructions

\[
\begin{align*}
\text{add}\{s\}<c><q> & \{<Rd>,\} <Rn>, <Rm> \{,<\text{shift}>\} \\
\text{adc}\{s\}<c><q> & \{<Rd>,\} <Rn>, <Rm> \{,<\text{shift}>\} \\
\text{add}\{s\}<c><q> & \{<Rd>,\} <Rn>, #<\text{const}} \\
\text{adc}\{s\}<c><q> & \{<Rd>,\} <Rn>, #<\text{const}} \\
\text{qadd}<c><q> & \{<Rd>,\} <Rn>, <Rm>
\end{align*}
\]

- \text{s: sets the flags} based on the result
- \text{c: makes the command conditional. } <c> \text{ can be EQ (equal), NE (not equal), CS (carry set), CC (carry clear), MI (minus), PL (plus), VS (overflow set), VC (overflow clear), HI (unsigned higher), LS (unsigned lower or same), GE (signed greater or equal), LT (signed less), GT (signed greater), LE (signed less or equal), AL (always)}
- \text{q: instruction width. Can be .N for narrow (16 bit) or .W for wide (32 bit)}
- \text{Rd, Rn, Rm: any register, incl. SP, LR and PC (with some restrictions). Result goes to Rn (if no Rd).}
- \text{shift: value of Rm is preprocessed with LSL (logical shift left – fills zeros), LSR (logical shift right – fills zeros), ASR (arithmetic shift right – keeps sign) or ROR (rotate right) followed by the #number of bits to shift/rotate by. There is also a RRX (rotate right by one incl. carry flag)}
- \text{const: an immediate value in the range 0..4095 directly or in the range 0..255 with rotation.
ARM v7-M 32 bit add instructions

```
add{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}
adc{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}
add{s}<c><q> {<Rd>,} <Rn>, #<const>
adc{s}<c><q> {<Rd>,} <Rn>, #<const>
qadd<c><q> {<Rd>,} <Rn>, <Rm>
```

- **s**: sets the flags based on the result
- **c**: makes the command conditional. `<c>` can be EQ (equal), NE (not equal), CS (carry set), CC (carry clear), MI (minus), PL (plus), VS (overflow set), VC (overflow clear), HI (unsigned higher), LS (unsigned lower or same), GE (signed greater or equal), LT (signed less), GT (signed greater), LE (signed less or equal), AL (always)
- **q**: instruction width. Can be .N for narrow (16 bit) or .W for wide (32 bit)
- **Rd, Rn, Rm**: any register, incl. SP, LR and PC (with some restrictions). Result goes to Rn (if no Rd).
- **shift**: value of Rm is preprocessed with LSL (logical shift left – fills zeros), LSR (logical shift right – fills zeros), ASR (arithmetic shift right – keeps sign) or ROR (rotate right) followed by the #number of bits to shift/rotate by. There is also a RRX (rotate right by one incl. carry flag)
- **const**: an immediate value in the range 0..4095 directly or in the range 0..255 with rotation.
**Numeric CPU status flags**

### Natural binary numbers

<table>
<thead>
<tr>
<th>0</th>
<th>a+b</th>
<th>a</th>
<th>b</th>
<th>2^n-1</th>
</tr>
</thead>
</table>

- **Carry**
- Wrap-around or modulo $2^n$

### 2's complement binary numbers

<table>
<thead>
<tr>
<th>-2^n-1</th>
<th>c</th>
<th>a+b</th>
<th>2d</th>
<th>d</th>
<th>0</th>
<th>2c</th>
<th>a</th>
<th>b</th>
<th>2^n-1-1</th>
</tr>
</thead>
</table>

- **Overflow**
- Wrap-around
- **Saturate**

Which of those operations will set which flag?

- **adds**
- **adcs**
- **qadd**
ARM v7-M 32 bit Addition, Subtraction instructions

```
add{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}  ; Rd := Rn + Rm(shifted)
adc{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}  ; Rd := Rn + Rm(shifted) + C
add{s}<c><q> {<Rd>,} <Rn>, #<const>  ; Rd := Rn + #<const>
adc{s}<c><q> {<Rd>,} <Rn>, #<const>  ; Rd := Rn + #<const> + C
qadd<c><q> {<Rd>,} <Rn>, <Rm>  ; Rd := Rn + Rm ; saturated
sub{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}  ; Rd := Rn - Rm(shifted)
sbc{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}  ; Rd := Rn - Rm(shifted) - NOT (C)
rsb{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}  ; Rd := Rm(shifted) - Rn
sub{s}<c><q> {<Rd>,} <Rn>, #<const>  ; Rd := Rn - #<const>
sbc{s}<c><q> {<Rd>,} <Rn>, #<const>  ; Rd := Rn - #<const> - NOT (C)
rsb{s}<c><q> {<Rd>,} <Rn>, #<const>  ; Rd := #<const> - Rn
qsub<c><q> {<Rd>,} Rn, Rm  ; Rd := Rn - Rm ; saturated
```

All instructions operate on 32 bit wide numbers.

... versions for narrower numbers, as well as versions which operate on multiple narrower numbers in parallel exist as well.
64 bit Addition, Subtraction

As your registers are 32 bit wide, you need two steps to add two 64 bit numbers in r3:r2, r5:r4 (with r2 and r4 being the lower 32 bits) to one 64 bit number in r1:r0:

\[
\text{adds} \quad r0, r2, r4 \quad ; \quad r0 := r2 + r4 \quad \text{add least significant words, set flags}
\]

\[
\text{adcs} \quad r1, r3, r5 \quad ; \quad r1 := r3 + r5 + C \quad \text{add most significant words and carry bit}
\]

... and symmetrically if you need a 64 bit subtraction:

\[
\text{subs} \quad r0, r2, r4 \quad ; \quad r0 := r2 - r4 \quad \text{least significant words, set flags}
\]

\[
\text{sbscs} \quad r1, r3, r5 \quad ; \quad r1 := r3 - r5 - \text{NOT} (C) \quad \text{most significant words and carry bit}
\]
ARM v7-M 32bit Boolean (bit-wise) instructions

\[
\begin{align*}
\text{and}\{s\}<c><q> \{<Rd>,\} <Rn>, <Rm> \{,<shift>\}; & \quad Rd := Rn \land Rm^{\text{shifted}} \\
\text{bic}\{s\}<c><q> \{<Rd>,\} <Rn>, <Rm> \{,<shift>\}; & \quad Rd := Rn \lor Rm^{\text{shifted}} \\
\text{orr}\{s\}<c><q> \{<Rd>,\} <Rn>, <Rm> \{,<shift>\}; & \quad Rd := Rn \lor Rm^{\text{shifted}} \\
\text{orn}\{s\}<c><q> \{<Rd>,\} <Rn>, <Rm> \{,<shift>\}; & \quad Rd := Rn \lor const \\
\text{eor}\{s\}<c><q> \{<Rd>,\} <Rn>, <Rm> \{,<shift>\}; & \quad Rd := Rn \lor const \\
\text{cmp}<c><q> <Rn>, <Rm> \{,<shift>\}; & \quad (Rn - Rm^{\text{shifted}}) \rightarrow \text{Flags} \\
\text{cmn}<c><q> <Rn>, <Rm> \{,<shift>\}; & \quad (Rn + Rm^{\text{shifted}}) \rightarrow \text{Flags} \\
\text{tst}<c><q> <Rn>, <Rm> \{,<shift>\}; & \quad (Rn \land Rm^{\text{shifted}}) \rightarrow \text{Flags} \\
\text{teq}<c><q> <Rn>, <Rm> \{,<shift>\}; & \quad (Rn \lor Rm^{\text{shifted}}) \rightarrow \text{Flags} \\
\text{cmp}<c><q> <Rn>, \#<\text{const}>, & \quad (Rn - \text{const}) \rightarrow \text{Flags} \\
\text{cmn}<c><q> <Rn>, \#<\text{const}>, & \quad (Rn + \text{const}) \rightarrow \text{Flags} \\
\text{tst}<c><q> <Rn>, \#<\text{const}>, & \quad (Rn \land \text{const}) \rightarrow \text{Flags} \\
\text{teq}<c><q> <Rn>, \#<\text{const}>, & \quad (Rn \lor \text{const}) \rightarrow \text{Flags}
\end{align*}
\]

This exhausts the simple ALU from chapter 1 …
ARM v7-M Move data inside the CPU

\[
\begin{align*}
\text{mov\{s\}<c><q> } & \text{ <Rd>, <Rm> } & \text{; } & \text{Rd := Rm} \\
\text{mov\{s\}<c><q> } & \text{ <Rd>, #<const> } & \text{; } & \text{Rd := const}
\end{align*}
\]

\[
\begin{align*}
\text{lsr\{s\}<c><q> } & \text{ <Rd>, <Rm>, #<n> } \\
\text{lsr\{s\}<c><q> } & \text{ <Rd>, <Rm>, <Rs> }
\end{align*}
\]

\[
\begin{align*}
\text{asr\{s\}<c><q> } & \text{ <Rd>, <Rm>, #<n> } \\
\text{asr\{s\}<c><q> } & \text{ <Rd>, <Rm>, <Rs> }
\end{align*}
\]

\[
\begin{align*}
\text{lsl\{s\}<c><q> } & \text{ <Rd>, <Rm>, #<n> } \\
\text{lsl\{s\}<c><q> } & \text{ <Rd>, <Rm>, <Rs> }
\end{align*}
\]

\[
\begin{align*}
\text{ror\{s\}<c><q> } & \text{ <Rd>, <Rm>, #<n> } \\
\text{ror\{s\}<c><q> } & \text{ <Rd>, <Rm>, <Rs> }
\end{align*}
\]

\[
\begin{align*}
\text{rrx\{s\}<c><q> } & \text{ <Rd>, <Rm> }
\end{align*}
\]
ARM v7-M Move data inside the CPU

\texttt{mov\{s\}<c><q> <Rd>, <Rm>}; Rd := Rm
\texttt{mov\{s\}<c><q> <Rd>, #<const>}; Rd := const

\texttt{lsr\{s\}<c><q> <Rd>, <Rm>, #<n>};
\texttt{lsr\{s\}<c><q> <Rd>, <Rm>, <Rs>};

\texttt{asr\{s\}<c><q> <Rd>, <Rm>, #<n>};
\texttt{asr\{s\}<c><q> <Rd>, <Rm>, <Rs>};

\texttt{lsl\{s\}<c><q> <Rd>, <Rm>, #<n>};
\texttt{lsl\{s\}<c><q> <Rd>, <Rm>, <Rs>};

\texttt{ror\{s\}<c><q> <Rd>, <Rm>, #<n>};
\texttt{ror\{s\}<c><q> <Rd>, <Rm>, <Rs>};

\texttt{rrx\{s\}<c><q> <Rd>, <Rm>};

\textbf{If this is numbers then ...}

\textbf{Rm/2^n rounded towards \(-\infty\)}

\textbf{for 2's complements}

\textbf{Rm \cdot 2^n}
Simple arithmetic inside the CPU

Calculate:

\[ e := a + b - 2 \times c \]

assuming all types are 32 bit 2’s complement numbers (Integer),

r1 holds a, r2 holds b, r3 holds c, and the results should be in r4.
Calculate:

\[ e := a + b - 2c \]

assuming all types are 32 bit 2’s complement numbers (Integer),
r1 holds a, r2 holds b, r3 holds c, and the results should be in r4.

\[
\begin{align*}
\text{add} & \quad r5, r1, r2 \\
\text{lsl} & \quad r6, r3, \#1 \quad ;\ you\ could\ also\ write: \ mov\ r6, r3, \ lsl\ \#1 \\
\text{sub} & \quad r4, r5, r6
\end{align*}
\]

We need temporary storage (r5, r6) in the process as we didn’t want to over-write the original values. Yet the total number of registers is always limited.
Simple arithmetic inside the CPU

Calculate:

\[ e := a + b - 2\times c \]

assuming all types are 32 bit 2’s complement numbers (Integer),
r1 holds a, r2 holds b, r3 holds c, and the results should be in r4.

\[
\begin{align*}
\text{add} & \quad r5, r1, r2 \\
\text{lsl} & \quad r6, r3, #1 \quad ; \text{you could also write: mov} \ r6, r3, lsl \ #1 \\
\text{sub} & \quad r4, r5, r6
\end{align*}
\]

We need temporary storage (r5, r6) in the process as we didn’t want to over-write the original values. Yet the total number of registers is always limited.

How about we assume that values are no longer needed after this expression:
Simple arithmetic inside the CPU

Calculate:

\[ e := a + b - 2c \]

assuming all types are 32 bit 2’s complement numbers (Integer),
r1 holds a, r2 holds b, r3 holds c, and the results should be in r4.

\[
\begin{align*}
\text{add} & \quad r5, r1, r2 \\
\text{lsl} & \quad r6, r3, #1 \quad ; \text{you could also write: } \text{mov} \ r6, r3, \text{lsl} \ #1 \\
\text{sub} & \quad r4, r5, r6
\end{align*}
\]

We need temporary storage (r5, r6) in the process as we didn’t want to over-write the original values. Yet the total number of registers is always limited.

How about we assume that values are no longer needed after this expression:

\[
\begin{align*}
\text{add} & \quad r1, r1, r2 \\
\text{lsl} & \quad r3, r3, #1 \\
\text{sub} & \quad r4, r1, r3
\end{align*}
\]

... your compiler will know when such side-effects are ok and when not.

Any overflows?
Simple arithmetic inside the CPU

Calculate:

\[ e := a + b - 2\times c \]

We need to check results after each step:

- `adds r1, r1, r2` ; need to check overflow flag
- `lsl r3, r3, #1` ; need to check that the sign did not change
- `subs r4, r1, r3` ; need to check overflow flag again

⚠️ We don’t have the means yet to branch off into different actions in case things go bad … to come soon.
Calculate:

\[ e := a + b - 2c \]

We need to check results after each step:

- **adds** \( r1, r1, r2 \) ; need to check overflow flag
- **lsl** \( r3, r3, #1 \) ; need to check that the sign did not change
- **subs** \( r4, r1, r3 \) ; need to check overflow flag again

We don’t have the means yet to branch off into different actions in case things go bad … to come soon.

Or we use saturation arithmetic and live with the error:

- **qadd** \( r1, r1, r2 \)
- **qadd** \( r3, r3, r3 \)
- **qsub** \( r4, r1, r3 \)

If we know we need to carry on either way, this at least minimizes the local errors.
Cortex-M4 Address Space

Your CPU has 32 bit of address space

\[ 4 \text{ GB} \]

... address space does not equate to physical memory!

Not all memory is equal: Some memory ...

... can be executed
...
... can be written to or read from or both
...
... has side-effects (coffee cups fall over)
...
... has strictly-ordered access
...
... does not physically exist
In its most basic form the value of a register is interpreted as an **address** and the **memory content** there is loaded into another register.
Yet: most data is structured.

... like a group of local variables, a record, an array and any combination of the above …

How to read an entry in an array/record?

In its most basic form the value of a register is interpreted as an **address** and the **memory content** there is loaded into another register.
Most copy operations between CPU and memory follow this basic scheme.
ARM v7-M Move data in and out of the CPU

Immediate addressing

\[
\text{ldr} \langle c \rangle \langle q \rangle \langle Rd \rangle, [\langle Rb \rangle \{, \#+/-\langle \text{offset} \rangle\}]
\]

\[
\text{str} \langle c \rangle \langle q \rangle \langle Rs \rangle, [\langle Rb \rangle \{, \#+/-\langle \text{offset} \rangle\}]
\]

Reads from a potentially offset memory cell with a base register address.
ARM v7-M Move data in and out of the CPU

writes to a potentially offset memory cell with a base register address.

\[
\text{ldr}<c><q> \quad \text{<Rd>}, \quad [\text{<Rb>} \quad \{, \quad \#+/-/\text{offset}\}] \\
\text{str}<c><q> \quad \text{<Rs>}, \quad [\text{<Rb>} \quad \{, \quad \#+/-/\text{offset}\}] \\
\]

Immediate addressing

\[
\text{str} \quad \text{r1, [r4]} \\
\text{str} \quad \text{r1, [r4, #-12]} \\
\]
ARM v7-M Move data in and out of the CPU

Immediate addressing
("Pre-indexed")

\[ \text{ldr} <c> <q> \ <Rd>, [<Rb>, #+/<\text{offset}>]! \]
\[ \text{str} <c> <q> \ <Rs>, [<Rb>, #+/<\text{offset}>]! \]

Reads from an offset memory cell with a base register address and writes the offset address back into the original base register.
**ARM v7-M Move data in and out of the CPU**

Immediate addressing ("Pre-indexed")

\[
\begin{align*}
\textit{ldr}<c><q> & <Rd>, [<Rb>, #+/\!-<offset>]! \\
\textit{str}<c><q> & <Rs>, [<Rb>, #+/\!-<offset>]!
\end{align*}
\]

Writes to an offset memory cell with a base register address and writes the offset address back into the original base register.
ARM v7-M Move data in and out of the CPU

Immediate addressing ("Post-indexed")

\[
\begin{align*}
\text{ldr} & \langle c \rangle < q > \quad \langle \text{Rd} \rangle, \quad [\langle \text{Rb} \rangle], \quad \#/+/-<\text{offset}> \\
\text{str} & \langle c \rangle < q > \quad \langle \text{Rs} \rangle, \quad [\langle \text{Rb} \rangle], \quad \#/+/-<\text{offset}>
\end{align*}
\]

Reads from a memory cell with a base register address and writes the offset address back into the original base register.
**ARM v7-M Move data in and out of the CPU**

- **Immediate addressing** (“Post-indexed”)
  - `ldr<c><q> <Rd>, [<Rb>], #+/<-offset>`
  - `str<c><q> <Rs>, [<Rb>], #+/<-offset>`

Writes to a memory cell with a base register address and writes the offset address back into the original base register.
ARM v7-M Move data in and out of the CPU

Index register addressing

\[ \text{ldr} \<c><q> \ <Rd>, \ [\langle Rb \rangle, \ <Ri> \{, \text{LSL} \ #<shift>\}] \]
\[ \text{str} \<c><q> \ <Rs>, \ [\langle Rb \rangle, \ <Ri> \{, \text{LSL} \ #<shift>\}] \]

Reads from a memory cell with a base register address plus a potentially shifted index register.
ARM v7-M Move data in and out of the CPU

Index register addressing

\[
\text{str } r1, [r4, r3, \text{LSL } #2]
\]

\[
\text{str } r1, [r4, r3]
\]

\[
\text{ldr } <c><q> <Rd>, [<Rb>, <Ri> \{, \text{LSL } #<shift>\}]
\]

\[
\text{str } <c><q> <Rs>, [<Rb>, <Ri> \{, \text{LSL } #<shift>\}]
\]

Writes to a memory cell with a base register address plus a potentially shifted index register.
ARM v7-M Move data in and out of the CPU

**Literal addressing**

```asm
ldr<
   <Rd>, <label>
ldr<
   <Rd>, [PC, #+-<offset>]`
```

Reads from a data area embedded into the code section.

**Note there is no store version.**

```asm
ldr r1, data`
```
ARM v7-M Move data in and out of the CPU

Stores multiple registers into sequential memory addresses. Stores “increment after” and loads “decrement before”.

stmia<c><q> <Rs>{!}, <registers>
ldmdb<c><q> <Rs>{!}, <registers>

stmia r9!, {r1, r3, r4, fp}
ARM v7-M Move data in and out of the CPU

stmia r9!, {r1, r3, r4, fp}
ldmdb r9!, {r1, r3, r4, fp}

Reads multiple registers from sequential memory addresses.
Stores “increment after” and loads “decrement before”.

Multiple registers (positive growing stack)

Note that any register can be use as stack base, i.e. you can have multiple stacks simultaneously.
ARM v7-M Move data in and out of the CPU

Stores multiple registers to sequential memory addresses.
Stores “decrement before” and loads “increment after”.

stmdb<chq> <Rs>{!}, <registers>
ldmia<chq> <Rs>{!}, <registers>

stmdb SP!, {r1, r3, r4, fp}

Negative growing stacks are the de-facto standard in industry.

Multiple registers (negative growing stack)
ARM v7-M Move data in and out of the CPU

stmdb<ch>q <Rs>{}, <registers>
ldmia<ch>q <Rs>{}, <registers>

Reads multiple registers from sequential memory addresses.
Stores “decrement before” and loads “increment after”.

Multiple registers (negative growing stack)
Simple arithmetic in memory

Calculate again:

\[ e := a + b - 2c \]

but now \( a, b, c \) and \( e \) are stored in memory, relative to an address stored in \( FP \) (“Frame Pointer”):

- \( a \) is held at \([fp - 12]\)
- \( b \) at \([fp - 16]\)
- \( c \) at \([fp - 20]\)
- \( e \) at \([fp - 24]\)

In order to do arithmetic we need to load those values into the CPU first and afterwards we need to store the result in memory:

\[
\begin{align*}
\text{ldr} & \quad r1, [fp, #-12] \\
\text{ldr} & \quad r2, [fp, #-16] \\
\text{add} & \quad r1, r1, r2 \\
\text{ldr} & \quad r2, [fp, #-20] \\
\text{lsl} & \quad r2, r2, #1 \\
\text{sub} & \quad r1, r1, r2 \\
\text{str} & \quad r1, [fp, #-24]
\end{align*}
\]

Notice that this time we only used two registers.
Simple arithmetic in memory

Calculate again:

\[ e := a + b - 2c \]

Or in saturation arithmetic:

\begin{verbatim}
  ldr r1, [fp, #-12]
  ldr r2, [fp, #-16]
  qadd r1, r1, r2
  ldr r2, [fp, #-20]
  qadd r2, r2, r2
  qsub r1, r1, r2
  str r1, [fp, #-24]
\end{verbatim}
Simple arithmetic in memory

Calculate again:

\[ e := a + b - 2c \]

Or with overflow checks:

\begin{verbatim}
  ldr  r1, [fp, #-12]
  ldr  r2, [fp, #-16]
  adds r1, r1, r2  ; need to check overflow flag
  ldr  r2, [fp, #-20]
  lsl  r2, r2, #1  ; need to check that the sign did not change
  subs r1, r1, r2  ; need to check overflow flag
  str  r1, [fp, #-24]
\end{verbatim}

✍️ It’s time we learn about branching off into alternative execution paths.
ARM v7-M Branch instructions

\begin{align*}
\text{b}<\text{c}><\text{q}> \quad &\langle\text{label}\rangle \quad ; \text{if } c \quad \text{then} \quad \text{PC} := \text{label} \\
\text{bl}<\text{c}> \quad &\langle\text{label}\rangle \quad ; \text{if } c \quad \text{then} \quad \text{LR} := \text{PC}\_\text{next}; \quad \text{PC} := \text{label} \\
\text{bx}<\text{c}> \quad &\langle\text{Rm}\rangle \quad ; \text{if } c \quad \text{then} \quad \text{PC} := \text{Rm} \\
\text{blx}<\text{c}><\text{q}> \quad &\langle\text{Rm}\rangle \quad ; \text{if } c \quad \text{then} \quad \text{LR} := \text{PC}\_\text{next}; \quad \text{PC} := \text{Rm} \\
\text{cbz}<\text{q}> \quad &\langle\text{Rn}, \langle\text{label}\rangle\rangle \quad ; \text{if } Rn = 0 \quad \text{then} \quad \text{PC} := \text{label} \\
\text{cbnz}<\text{q}> \quad &\langle\text{Rn}, \langle\text{label}\rangle\rangle \quad ; \text{if } Rn \neq 0 \quad \text{then} \quad \text{PC} := \text{label}
\end{align*}

\begin{tabular}{|c|c|c|}
\hline
\(<\text{c}>\) & \text{Meanings} & \text{Flags} \\
\hline
\text{eq} & \text{Equal} & Z = 1 \\
\text{ne} & \text{Not equal} & Z = 0 \\
\text{cs}, \text{hs} & \text{Carry set, Unsigned higher or same} & C = 1 \\
\text{cc}, \text{lo} & \text{Carry clear, Unsigned lower} & C = 0 \\
\text{mi} & \text{Minus, Negative} & N = 1 \\
\text{pl} & \text{Plus, Positive or zero} & N = 0 \\
\text{vs} & \text{Overflow} & V = 1 \\
\text{vc} & \text{No overflow} & V = 0 \\
\text{hi} & \text{Unsigned higher} & C = 1 \land Z = 0 \\
\text{ls} & \text{Unsigned lower or same} & C = 0 \lor Z = 1 \\
\text{ge} & \text{Signed greater or equal} & N = Z \\
\text{lt} & \text{Signed less} & N \neq Z \\
\text{gt} & \text{Signed greater} & Z = 0 \land N = V \\
\text{le} & \text{Signed less or equal} & Z = 1 \lor N \neq V \\
\text{al}, \langle\text{none}\rangle & \text{Always} & \text{any} \\
\hline
\end{tabular}
Simple arithmetic in memory

Calculate again:

\[ e := a + b - 2c \]

Or with overflow checks:

\begin{verbatim}
ldr r1, [fp, #-12]
ldr r2, [fp, #-16]
adds r1, r1, r2
bvs Overflow ; branch if overflow is set
ldr r2, [fp, #-20]
adds r2, r2, r2
bvs Overflow ; branch if overflow is set
subs r1, r1, r2
bvs Overflow ; branch if overflow is set
str r1, [fp, #-24]
\end{verbatim}

\textit{Overflow:}

\begin{verbatim}
svc #5 ; call the operating system or runtime environment with #5
; (assuming that #5 indicates an overflow situation)
\end{verbatim}
Simple arithmetic in memory

Calculate again:

\[ e := a + b - 2c \]

Or with overflow checks:

\[
\begin{align*}
  \text{ldr} & \quad r1, [fp, #-12] \\
  \text{ldr} & \quad r2, [fp, #-16] \\
  \text{adds} & \quad r1, r1, r2 \\
  \text{bvs} & \quad \text{Overflow} \quad ; \text{branch if overflow is set} \\
  \text{ldr} & \quad r2, [fp, #-20] \\
  \text{adds} & \quad r2, r2, r2 \\
  \text{bvs} & \quad \text{Overflow} \quad ; \text{branch if overflow is set} \\
  \text{subs} & \quad r1, r1, r2 \\
  \text{bvs} & \quad \text{Overflow} \quad ; \text{branch if overflow is set} \\
  \text{str} & \quad r1, [fp, #-24]
\end{align*}
\]

... but how do we know where this happened or how to continue operations?

\[ \text{Overflow:} \]

\[
\begin{align*}
  \text{svc} & \quad #5 \quad ; \text{call the operating system or runtime environment with #5} \\
\end{align*}
\]

; (assuming that #5 indicates an overflow situation)
Simple arithmetic in memory

Calculate again:

\[ e := a + b - 2c \]

Or with overflow checks:

\[ \text{ldr } r1, [fp, #-12] \]
\[ \text{ldr } r2, [fp, #-16] \]
\[ \text{adds } r1, r1, r2 \]
\[ \text{blvs Overflow} \quad ; \text{branch if overflow is set; keep next location in LR} \]
\[ \text{ldr } r2, [fp, #-20] \]
\[ \text{adds } r2, r2, r2 \]
\[ \text{blvs Overflow} \quad ; \text{branch if overflow is set; keep next location in LR} \]
\[ \text{subs } r1, r1, r2 \]
\[ \text{blvs Overflow} \quad ; \text{branch if overflow is set; keep next location in LR} \]
\[ \text{str } r1, [fp, #-24] \]

\[ ... \]

\textbf{Overflow:}

\[ ... \]
\[ \text{bx } lr \]
\[ ; \text{... for example writing a log entry with location}\]
\[ ; \text{resume operations - assuming the above did not change LR} \]
ARM v7-M Essential multiplications and divisions

32 bit to 32 bit

\[
\text{mul}\{s\}<c><q> \{<Rd>,\} <Rn>,<Rm> \quad ; \quad Rd := (Rn*Rm)
\]

\[
\text{mla}<c> \quad <Rd>, \quad <Rn>,<Rm>,<Ra> \quad ; \quad Rd := Ra + (Rn*Rm)
\]

\[
\text{mls}<c> \quad <Rd>, \quad <Rn>,<Rm>,<Ra> \quad ; \quad Rd := Ra - (Rn*Rm)
\]

\[
\text{udiv}<c> \quad <Rd>, \quad <Rn>,<Rm> \quad ; \quad Rd := \text{unsigned} \ (Rn/Rm); \text{rounded} \ towards \ 0
\]

\[
\text{sdiv}<c> \quad <Rd>, \quad <Rn>,<Rm> \quad ; \quad Rd := \text{signed} \ (Rn/Rm); \text{rounded} \ towards \ 0
\]

32 bit to 64 bit

\[
\text{umull}<c> \quad <RdLo>,<RdHi>,<Rn>,<Rm> \quad ; \quad RdHi:RdLo := \text{unsigned} \ (Rn*Rm))
\]

\[
\text{umlal}<c><q> \quad <RdLo>,<RdHi>,<Rn>,<Rm> \quad ; \quad RdHi:RdLo := \text{unsigned} \ (RdHi:RdLo + (Rn*Rm))
\]

\[
\text{smull}<c> \quad <RdLo>,<RdHi>,<Rn>,<Rm> \quad ; \quad RdHi:RdLo := \text{signed} \ (Rn*Rm))
\]

\[
\text{smlal}<c> \quad <RdLo>,<RdHi>,<Rn>,<Rm> \quad ; \quad RdHi:RdLo := \text{signed} \ (RdHi:RdLo + (Rn*Rm))
\]

... versions for narrower numbers, as well as versions which operate on multiple narrower numbers in parallel exist as well.
Calculate:

\[ c := a \wedge b \]

\[
\begin{align*}
\text{mov} & \quad r1, \ #7 \quad ; \ a \\
\text{mov} & \quad r2, \ #11 \quad ; \ b \quad ; \ \text{has to be non-negative} \\
\text{mov} & \quad r3, \ #1 \quad ; \ c \\
\text{power:} & \\
\text{cbz} & \quad r2, \ \text{end\_power} \quad ; \ \text{exponent zero?} \\
\text{mul} & \quad r3, \ r1 \\
\text{sub} & \quad r2, \ #1 \\
\text{b} & \quad \text{power} \\
\text{end\_power:} & \\
\text{nop} & \quad ; \ c = a \wedge b
\end{align*}
\]

\[ 7^{11} = 7 \cdot 7 \cdot 7 \cdot 7 \cdot 7 \cdot 7 \cdot 7 \cdot 7 \cdot 7 \cdot 7 \cdot 7 \]

How many iterations?

How many cycles?
More power

Calculate:
\[ c := a \wedge b \]

\begin{verbatim}
    mov r1, #7 ; a
    mov r2, #11 ; b ; has to be non-negative
    mov r3, #1 ; c
    mov r4, r1 ; base a to the powers of two, starting with \( a \wedge 1 \)

    power:
        cbz r2, end_power ; exponent zero?
        tst r2, #0b1 ; right-most bit of exponent set?
        beq skip ; skip this power if not
        mul r3, r4 ; multiply the current power into result
        skip:
            mul r4, r4 ; calculate next power
            lsr r2, #1 ; divide exponent by 2
            b power

    end_power:
        nop ; c = a \wedge b
\end{verbatim}

\[ 7^{11} = 7^8 \cdot 7^2 \cdot 7^1 \]

How many iterations?
How many cycles?
Table based branching

`tbb\langle c\rangle\langle q\rangle [\langle Rn\rangle, \langle Rm\rangle]` ; for tables of offset bytes (8bit)
`tbh\langle c\rangle\langle q\rangle [\langle Rn\rangle, \langle Rm\rangle, lsl \#1]` ; for tables of offset halfwords (16bit)

Common usage for byte (8bit) tables

`tbb [PC, Ri]` ; PC is base of branch table, Ri is index

**Branch_Table:**
`.byte (Case_A - Branch_Table)/2 ; Case_A 8 bit offset`
`.byte (Case_B - Branch_Table)/2 ; Case_B 8 bit offset`
`.byte (Case_C - Branch_Table)/2 ; Case_C 8 bit offset`
`.byte 0x00 ; Padding to re-align with halfword boundaries`

**Case_A:**
```
... ; any instruction sequence
b End_Case ; “break out”
```

**Case_B:**
```
... ; any instruction sequence
b End_Case ; “break out”
```

**Case_C:**
```
... ; any instruction sequence
```

**End_Case:**
Table based branching

\texttt{tbb}<c><q> \[<Rn>, <Rm>] \quad ; \text{for tables of offset bytes (8bit)}
\texttt{tbh}<c><q> \[<Rn>, <Rm>, \text{ls}l \ #1] \quad ; \text{for tables of offset halfwords (16bit)}

Common usage for halfword (16bit) tables

\texttt{tbh} \ [\text{PC, Ri, ls}l \ #1] \quad ; \text{PC used as base of branch table, Ri is index}

\textbf{Branch Table:}

\texttt{.hword} (Case\_A - Branch\_Table)/2 \quad ; \text{Case\_A 16 bit offset}
\texttt{.hword} (Case\_B - Branch\_Table)/2 \quad ; \text{Case\_B 16 bit offset}
\texttt{.hword} (Case\_C - Branch\_Table)/2 \quad ; \text{Case\_C 16 bit offset}

\textbf{Case\_A:}

\texttt{...} \quad ; \text{any instruction sequence}
\texttt{\textbf{b} End\_Case} \quad ; \text{“break out”}

\textbf{Case\_B:}

\texttt{...} \quad ; \text{any instruction sequence}
\texttt{\textbf{b} End\_Case} \quad ; \text{“break out”}

\textbf{Case\_C:}

\texttt{...} \quad ; \text{any instruction sequence}

\textbf{End\_Case:}
## Basic instruction sets

<table>
<thead>
<tr>
<th>Category</th>
<th>Side effects</th>
<th>ARM v7-M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic, Logic</td>
<td>Sets and uses CPU flags</td>
<td>add, adc, qadd, sub, sbc, qsub, rsb, mul, mla, mls, udiv, sdiv, umull, umlal, smull, smlal, and, bic, orr, orn, eor, cmp, cmn, tst, teq</td>
</tr>
<tr>
<td>Move and shift registers</td>
<td></td>
<td>mov, lsr, asr, lsl, ror, rrx</td>
</tr>
<tr>
<td>Branching</td>
<td>Uses CPU flags</td>
<td>b, bl, bx, blx, tbb, tbh</td>
</tr>
<tr>
<td>Load &amp; Store</td>
<td>Effects memory</td>
<td>ldr, str, ldmdb, ldmia, stmia, stmdb</td>
</tr>
</tbody>
</table>
## Basic instruction sets

<table>
<thead>
<tr>
<th>Category</th>
<th>Side effects</th>
<th>ARM v7-M</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Move and shift registers</strong></td>
<td></td>
<td><strong>mov</strong>, <strong>lsr</strong>, <strong>asr</strong>, <strong>lsl</strong>, <strong>ror</strong>, <strong>rrx</strong></td>
</tr>
<tr>
<td><strong>Branching</strong></td>
<td>Uses CPU flags</td>
<td><strong>b</strong>, <strong>bl</strong>, <strong>bx</strong>, <strong>blx</strong>, <strong>tbb</strong>, <strong>tbh</strong></td>
</tr>
<tr>
<td><strong>Load &amp; Store</strong></td>
<td>Effects memory</td>
<td><strong>ldr</strong>, <strong>str</strong>, <strong>ldmdb</strong>, <strong>ldmia</strong>, <strong>stmia</strong>, <strong>stmdb</strong></td>
</tr>
</tbody>
</table>

Instruction sets in the field:

**RISC**: Power, ARM, MIPS, Alpha, SPARK, AVR, PIC, ...

**CISC**: x86, Z80, 6502, 68000, ...

Over 50 billion CPUs on this planet are running ARM instruction sets.
### Basic instruction sets

<table>
<thead>
<tr>
<th>Category</th>
<th>Side effects</th>
<th>ARM v7-M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic, Logic</td>
<td>Sets and uses CPU flags</td>
<td>add, adc, qadd, sub, sbc, qsub, rsb, mul, mla, mls, udiv, sdiv, umull, umlal, smull, smlal, and, bic, orr, orn, eor, cmp, cmn, tst, teq</td>
</tr>
<tr>
<td>Move and shift registers</td>
<td></td>
<td>mov, lsr, asr, lsl, ror, rrx</td>
</tr>
<tr>
<td>Branching</td>
<td>Uses CPU flags</td>
<td>b, bl, bx, blx, tbb, tbh</td>
</tr>
<tr>
<td>Load &amp; Store</td>
<td>Effects memory</td>
<td>ldr, str, ldmdb, ldmia, stmia, stmdb</td>
</tr>
</tbody>
</table>

What’s missing?

- Changing CPU privileges and handling interrupts.
- Synchronizing instructions

Coming in later chapters about concurrency and operating systems.
Hardware/Software Interface

Summary

Hardware/Software Interface

• Instruction formats
  • Register sets
  • Instruction encoding

• Arithmetic / Logic instructions inside the CPU
  • Summation, Subtraction, Multiplication, Division
  • Logic and shift operations

• Load / Store and addressing modes
  • Direct, relative, indexed, and auto-index-increment addressing forms

• Branching
  • Conditional branching and unconditional jumps.
Functions

Uwe R. Zimmer - The Australian National University
Functions

References for this chapter

[Patterson17]
David A. Patterson & John L. Hennessy
Computer Organization and Design – The Hardware/Software Interface
Chapter 2 “Instructions: Language of the Computer”
ARM edition, Morgan Kaufmann 2017
(Greatness from …) Small beginnings

plus_1 :: (Num a) => a -> a
plus_1 x = x + 1

int plus1 (int x) {
    return x + 1;
}

function Plus_1 (x : Integer) return Integer is (x + 1);

def plus1 (x):
    return x + 1;

pure function plus_1 (x)
    int, intent (in) :: x
    int :: plus_1
    plus_1 = x + 1;
end function;

function Plus_1 (x : integer) : integer;
    begin
        Plus_1 := x + 1;
    end;
mov r0, #1
bl Plus_1
mov r4, r0

Plus_1:
add r0, #1
bx lr
...  
...  
...  
...  
...  
...  
...  
...  
...  

```
mov   r0, #1  
bl    Plus_1 
      
Plus_1:  
      
add   r0, #1  
mov   r4, r0  
bx     lr 
```
How is the parameter $x$ passed?

Where do you find the result after the function returns?

Does it work?

Could it be done differently?

```assembly
... 
... 
... 
mov r0, #1 
bl Plus_1 
... 
mov r4, r0 
... 
... 

Plus_1:
  add r0, #1 
bx lr
```
mov r0, #1
add r0, #1
mov r4, r0
...
plus_2 :: (Num a) => a -> a  
plus_2 x = plus_1 $ plus_1 x

int plus2 (int x) {
    return plus1 (plus1 (x));
}

function Plus_2 (x : Integer) return Integer is (Plus_1 (Plus_1 (x)))

def plus2 (x):
    return plus1 (plus1 (x));

pure function plus_2 (x)
    int, intent (in) :: x
    int :: plus_2
    plus_2 = plus_1 (plus_1 (x));
end function;

function Plus_2 (x : integer) : integer;
begin
    Plus_2 := Plus_1 (Plus_1 (x));
end;
What is the value of lr in each case?
What is the value of \( \text{lr} \) in each case?
... we need an example, where a compiler will not just remove all our code!
(Greatness from ...) Small beginnings

fib_fact :: (Num a) => a -> a
fib_fact x = (fib x) + (fact x)

unsigned int fibFact (unsigned int x) {
    return fib (x) + fact (x);
}

function Fib_Fact (x : Natural) return Natural is (Fib (x) + Fact (x));

def fibFact (x):
    return fib (x) + fact (x);

pure function fib_fact (x)
    int, intent (in) :: x
    int :: fib_fact
    fib_fact = fib (x) + fact (x);
end function;

function Fib_Fact (x : cardinal) : cardinal;
begin
    Fib_Fact := Fib (x) + Fact (x);
end;
```
... ...
...
...
...
mov r0, #1
bl Fib_Fact
mov r3, r0
...
... ...
Fib_Fact:
...
...
...
...
bl Fib
mov r4, r0
...
...
bl Fact
add r0, r4
...
...
bx lr
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...(continued)
... where does this lead us?
Functions

Fib_Fact:

```
str lr, [sp, #-4]!
...
...
...
bl Fib_Fact
mov r3, r0
...
...
...
```

Fib:

```
...
...
...
bx lr
```

Fact:

```
...
...
...
bx lr
```

© 2021 Uwe R. Zimmer, The Australian National University
... what if this was holding some information at the time when we were called?
Fib_Fact:

stmdb sp!, {r4, lr}

... ... ...

bl Fib

mov r4, r0

... ...

bl Fact

add r0, r4

... ...

ldmia sp!, {r4, lr}

bx lr

Fib:

... ...

bx lr

Fact:

... ...

bx lr

sp

lr

r4

Fib_Fact
What happens to our parameter $x$ during the function?
Functions

```
mov r0, #1
bl Fib_Fact
mov r3, r0
...
...

Fib_Fact:

stmdb sp!, {r4, lr}
...
sub sp, #4
str r0, [sp]
bl Fib
mov r4, r0
ldr r0, [sp]
bl Fact
add r0, r4
add sp, #4
ldmia sp!, {r4, lr}
bx lr
```

```
Fib:
...
...
...
bl lr
```

```
Fact:
...
...
...
bl lr
```
While addressing via the sp is possible, it may also be complex to keep track of, as the sp may change further.
Keeping a reference to the start of the Stack Frame for this function (with the frame-pointer fp) makes things neater and enables structured access to the dynamic context.
Recursive

unsigned int fib (unsigned int x) {
    switch (x) {
    case 0  : return 0;
    case 1  : return 1;
    default : return fib (x - 1) + fib (x - 2);
    }
}

function Fib (x : Natural) return Natural is
    (case x is
    when 0      => 0,
    when 1      => 1,
    when others => Fib (x - 1) + Fib (x - 2));

unsigned int fact (unsigned int x) {
    if (x == 0) return 1;
    else return x * fact (x - 1);
}

function Fact (x : Natural) return Positive is
    (if x = 0 then 1
    else x * Fact (x - 1));
Functions

Is Fact reentrant?

How high do we stack?

Fact:

```
stmdb    sp!, {fp, lr}
add      fp, sp, #4
sub      sp, #4
str      r0, [fp, #-8]
cmp      r0, #0
bne      Case_Others
mov      r0, #1
b        End_Fact
```

Case_Others:

```
sub      r0, #1
bl       Fact
mov      r1, r0
ldr      r0, [fp, #-8]
```

End_Fact:

```
add      sp, #4
ldmia    sp!, {r4, fp, lr}
```

Fib_Fact:

```
stmdb    sp!, {r4, fp, lr}
add      fp, sp, #8
sub      sp, #4
str      r0, [fp, #-12]
bl       Fib
mov      r4, r0
ldr      r0, [fp, #-12]
```

Where is the last lr stored?

Where is the last lr stored?
A compiler will likely replace such a recursion!

```pascal
function Fact (x : Natural) return Positive is
    fac : Positive := 1;
begin
    for i in 1 .. x loop
        fac := fac * i;
    end loop;
    return fac;
end Fact;
```

```pascal
function Fact (x : Natural) return Positive is
    if x = 0 then 1
    else x * Fact (x - 1)
end if;
```

```pascal
function Fact (x : Natural) return Positive is
    fac : Positive := 1;
begin
    for i in 1 .. x loop
        fac := fac * i;
    end loop;
    return fac;
end Fact;
```
A compiler will likely replace such a recursion!

```c
unsigned int fact (unsigned int x) {
    if (x == 0) return 1;
    else        return x * fact (x - 1);
}
```

```c
unsigned int fact (unsigned int x) {
    int fac = 1;
    for (i = 1, i <= x, i++) {
        fac = fac * i;
    }
    return fac;
}
```
Besides all the inlining, unrolling, flattening, etc.:

Stack operations are still vital for any non-trivial program.
**Functions**

**Fib_Fact:**

- `stmdb sp!, {r4, fp, lr}
- `add fp, sp, #8
- `sub sp, #4
- `str r0, [fp, #-12]
- `bl Fib
- `mov r4, r0
- `ldr r0, [fp, #-12]
- `bl Fact
- `add r0, r0, r4
- `add sp, #4
- `ldmia sp!, {r4, fp, lr}
- `bx lr

**Fib:**

- `...
- `...
- `...
- `bx lr

**Fact:**

- `add r3, r0, #0
- `mov r0, #1
- `beq End_Fact

**Fact_Loop:**

- `mul r0, r3
- `subs r3, #1
- `bne Fact_Loop

**End_Fact:**

- `bx lr

---

Why do we save r4?

But we don’t save r3?

© 2021 Uwe R. Zimmer, The Australian National University
Functions

Fib_Fact:

```
stmdb sp!, {r4, fp, lr}
add fp, sp, #8
sub sp, #4
str r0, [fp, #-12]
bl Fib
mov r4, r0
ldr r0, [fp, #-12]
bl Fact
add r0, r0, r4
add sp, #4
ldmia sp!, {r4, fp, lr}
bx lr
```

Fib:

```
...
...
...
bx lr
```

Fact:

```
add r3, r0, #0
mov r0, #1
beq End_Fact
```

Fact_Loop:

```
mul r0, r3
subs r3, #1
bne Fact_Loop
```

End_Fact:

```
bx lr
```

We keep a copy of r0 here.

But we don’t keep a copy of r0 here!

© 2021 Uwe R. Zimmer, The Australian National University
Functions

Fib_Fact:

```assembly
stmdb sp!, {r4, fp, lr}
add fp, sp, #8
sub sp, #4
str r0, [fp, #-12]
bl Fib
mov r4, r0
ldr r0, [fp, #-12]
bl Fact
add r0, r0, r4
add sp, #4
ldmia sp!, {r4, fp, lr}
```

```assembly
```
There could be **two** further Fib-calls for each call to Fib ...

<table>
<thead>
<tr>
<th><strong>Fib</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{Fib:}</td>
</tr>
</tbody>
</table>
| \begin{align*}
\text{stmdb} &\quad \text{sp!}, \{r4, \text{fp}, \text{lr}\} \\
\text{add} &\quad \text{fp}, \text{sp}, \#8 \\
\text{sub} &\quad \text{sp}, \#4 \\
\text{str} &\quad r0, [\text{fp}, \#-12] \\
\text{bl} &\quad \text{Fib} \\
\text{mov} &\quad r4, r0 \\
\text{ldr} &\quad r0, [\text{fp}, \#-12] \\
\text{bl} &\quad \text{Fact} \\
\text{add} &\quad r0, r0, \#4 \\
\text{ldr} &\quad r0, [\text{fp}, \#-12] \\
\text{sub} &\quad r0, r0, \#2 \\
\text{bl} &\quad \text{Fib} \\
\text{mov} &\quad r4, r0 \\
\text{add} &\quad r0, r4, r0 \\
\end{align*} |
| \textbf{End_Fib:} |
| \begin{align*}
\text{add} &\quad \text{sp}, \#4 \\
\text{ldmia} &\quad \text{sp!}, \{r4, \text{fp}, \text{lr}\} \\
\text{bx} &\quad \text{lr} \\
\end{align*} |

What would be the maximal depth for the stack?

What would the stack look like?
Components / phases of a function call:

- Values (parameters) to be passed to a function.
- Local variables inside a function.
- Values (results) to be returned from a function.

So far we:

- ... passed parameter values in registers (r0 - r3).
- ... called the function (store the return address and jump to the beginning of the function).
- ... pushed the return address, the previous stack frame and used registers (r4 ...).
- ... created a new stack frame (and addressed all local variables relative to this).
- ... grew the stack such that it can hold the local variables.
- ... done the calculations/operations based on the local variables and scratch registers.
- ... passed return values in registers (r0 - r1).
- ... restored the stack pointer (and thus de-allocated all local variables).
- ... popped the return address, the previous stack frame and used registers (r4 ...).
- ... jumped back to the next instruction after the original function call.
- ... used the return values found in r0 - r1.
Components / phases of a function call:

- Values (parameters) to be passed to a function.
- Local variables inside a function.
- Values (results) to be returned from a function.

So far we:
- … passed parameter values in registers (r0 - r3).
- … called the function (store the return address and jump to the beginning of the function).
- … pushed the return address, the previous stack frame and used registers (r4 …).
- … created a new stack frame (and addressed all local variables relative to this).
- … grew the stack such that it can hold the local variables.
- … done the calculations/operations based on the local variables and scratch registers.
- … passed return values in registers (r0 - r1).
- … restored the stack pointer (and thus de-allocated all local variables).
- … popped the return address, the previous stack frame and used registers (r4 …).
- … jumped back to the next instruction after the original function call.
- … used the return values found in r0 - r1.
**Conventions**

**ARM architecture calling practice**

- **r0-r3** are used for parameters.
- **r0-r1** are used for return values.
- **r0-r3** are not expected to be intact after a function call
  
  ... all other registers are expected to be intact!

- If those registers do not suffice, additional parameters and results are passed via the stack.

- There are also memory alignment constraints.
  
  (Mostly due to memory bus constraints)

- Conventions are different in other architectures (e.g. x86, where parameters are generally passed via the stack).

- Why are these conventions architecture related at all?
## Functions

### Parameter passing

**Call by …**

<table>
<thead>
<tr>
<th>Information flow</th>
<th>Access</th>
<th>by copy</th>
<th>by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>by value</td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
<td>by reference (immutable) No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td>out</td>
<td>by result</td>
<td>Calling function expects the parameter value to appear in a specific space at return.</td>
<td>by reference (mutable, no read) No read access from inside the function, write access on return.</td>
</tr>
<tr>
<td>in &amp; out</td>
<td>by value result</td>
<td>Parameter is copied to a local variable and copied back at return.</td>
<td>by reference (mutable) Function can read and write at any time. Outside code shall not write.</td>
</tr>
</tbody>
</table>
### Functions

#### C

Full control over those three modes.

“by value” parameters are local variables.

“in & out, by reference” syntactically as “by pointer value” parameters.

<table>
<thead>
<tr>
<th>Information flow</th>
<th>by value</th>
<th>by reference (immutable)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>in</strong></td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
<td>No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td><strong>out</strong></td>
<td>Calling function expects the parameter value to appear in a specific space at return.</td>
<td></td>
</tr>
<tr>
<td><strong>in &amp; out</strong></td>
<td>Parameter is copied to a local variable and copied back at return.</td>
<td>Function can read and write at any time. Outside code shall not write.</td>
</tr>
</tbody>
</table>
## Access

<table>
<thead>
<tr>
<th>Information flow</th>
<th>by copy</th>
<th>by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>by value</td>
<td>by reference (immutable)</td>
</tr>
<tr>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
<td>No write access is allowed while the function runs (also from outside the function).</td>
<td></td>
</tr>
<tr>
<td>out</td>
<td>by result</td>
<td>by reference (mutable, no read)</td>
</tr>
<tr>
<td>Calling function expects the parameter value to appear in a specific space at return.</td>
<td>No read access from inside the function, write access on return.</td>
<td></td>
</tr>
<tr>
<td>in &amp; out</td>
<td>by value result</td>
<td>by reference (mutable)</td>
</tr>
<tr>
<td>“in &amp; out” parameters are side-effecting and can therefore not exist in a pure functional language.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Functions

### Python

All parameter access is double-indirect (\(\text{G handles}\)).

---

### Access

<table>
<thead>
<tr>
<th>Information flow</th>
<th>by copy</th>
<th>by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>in</strong></td>
<td><strong>by value</strong></td>
<td><strong>by reference (immutable)</strong></td>
</tr>
<tr>
<td></td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
<td>No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td><strong>out</strong></td>
<td><strong>by result</strong></td>
<td><strong>by reference (mutable, no read)</strong></td>
</tr>
<tr>
<td></td>
<td>Calling function expects the parameter value to appear in a specific space at return.</td>
<td>No read access from inside the function, write access on return.</td>
</tr>
<tr>
<td><strong>in &amp; out</strong></td>
<td><strong>by value result</strong></td>
<td><strong>by reference (mutable)</strong></td>
</tr>
<tr>
<td></td>
<td>Parameter is copied to a local variable and copied back at return.</td>
<td>Function can read and write at any time. Outside code shall not write.</td>
</tr>
</tbody>
</table>
# Functions

## Ada

Limited control over “by value result”. “by value” parameters are constants.

<table>
<thead>
<tr>
<th>Information flow</th>
<th>Access</th>
<th>by copy</th>
<th>by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>by value</td>
<td></td>
<td>by reference (immutable)</td>
</tr>
<tr>
<td></td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
<td>No write access is allowed while the function runs (also from outside the function).</td>
<td></td>
</tr>
<tr>
<td>out</td>
<td>by result</td>
<td></td>
<td>by reference (mutable, no read)</td>
</tr>
<tr>
<td></td>
<td>Calling function expects the parameter value to appear in a specific space at return.</td>
<td>No read access from inside the function, write access on return.</td>
<td></td>
</tr>
<tr>
<td>in &amp; out</td>
<td>by value result</td>
<td></td>
<td>by reference (mutable)</td>
</tr>
<tr>
<td></td>
<td>Parameter is copied to a local variable and copied back at return.</td>
<td>Function can read and write at any time. Outside code shall not write.</td>
<td></td>
</tr>
</tbody>
</table>
### Functions

**Assembly**

“By reference” semantics by convention only.

<table>
<thead>
<tr>
<th>Information flow</th>
<th>Access</th>
<th>by copy</th>
<th>by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td><strong>by value</strong></td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
<td><strong>by reference (immutable)</strong> No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td>out</td>
<td><strong>by result</strong></td>
<td>Calling function expects the parameter value to appear in a specific space at return.</td>
<td><strong>by reference (mutable, no read)</strong> No read access from inside the function, write access on return.</td>
</tr>
<tr>
<td>in &amp; out</td>
<td><strong>by value result</strong> Parameter is copied to a local variable and copied back at return.</td>
<td><strong>by reference (mutable)</strong> Function can read and write at any time. Outside code shall not write.</td>
<td></td>
</tr>
</tbody>
</table>
Parameter passing

Call by name

... is conceptually a call-by-value, where the value has not been calculated yet.

Technically a reference to a function is passed and the evaluation of this parameter (function) is left to the called function.

Features:

• Values are only evaluated if and when they are needed.
• Values can change during the life-time of a function (in case of side-effecting functions).
• Values can be stored once calculated (in case of side-effect-free functions).

While this is possible to a degree in most programming languages ...

(even if there is no specific passing mode, you can still pass a reference to a function)

... it is a core concept for functional, lazy evaluation languages, like e.g. Haskell,

and it does find its way back into mainstream languages like C++, .NET languages or Python as anonymous functions (sometimes referred to as $\lambda$-functions or $\lambda$-expressions).
How about using call by reference here?

**Fib_Fact:**
```
stmdb sp!, {r4, fp, lr}
add fp, sp, #4
sub sp, #4
str r0, [fp, #-8]
bl Fib
mov r4, r0
ldr r0, [fp, #-12]
bl Fact
add r0, r0, r4
add sp, #4
ldmia sp!, {r4, fp, lr}
bx lr
```

**Fact:**
```
stmdb sp!, {fp, lr}
add fp, sp, #4
sub sp, #4
str r0, [fp, #-8]
cmp r0, #0
bne Case_Others
mov r0, #1
b End_Fact
```

**Case_Others:**
```
sub r0, #1
bl Fact
add sp, #4
add fp, sp, #8
sub sp, #4
str r0, [fp, #-12]
bl Fib
mov r4, r0
ldr r0, [fp, #-12]
bl Fact
add r0, r0, r4
add sp, #4
ldmia sp!, {r4, fp, lr}
bx lr
```

**End_Fact:**
```
add sp, #4
ldmia sp!, {fp, lr}
bx lr
```
**Fact:**

```plaintext
stmdb sp!, {r5, fp, lr}
add fp, sp, #4
mov r5, r0
ldr r0, [r5]
cmp r0, #0
bne Case_Others
mov r0, #1
b End_Fact
```

**Case_Others:**

```plaintext
sub r0, #1
str r0, [fp, #-12]
bl Fib
```

**End_Fact:**

```plaintext
mov r4, r0
sub r0, fp, #12
bl Fact
add r0, r0, r4
add sp, #4
ldmia sp!, {r4, fp, lr}
```

---

**Fib_Fact:**

```plaintext
stmdb sp!, {r4, fp, lr}
add fp, sp, #8
sub sp, #4
str r0, [fp, #-12]
bl Fib
mov r4, r0
sub r0, fp, #12
bl Fact
add r0, r0, r4
add sp, #4
ldmia sp!, {r4, fp, lr}
```

---

**r5** has been nominated to hold the reference to **x** inside Fact
Functions

Fact:

```
stmdb sp!, {r5, fp, lr}
add fp, sp, #4
mov r5, r0
ldr r0, [r5]
lop r5, #0
bne Case_Others
mov r0, #1
b End_Fact

Case_Others:
sub r0, #1
str r0, [r5]
mov r0, r5
bl Fact
mov r4, r0
sub r5, fp, #12
bl Fact
add r0, r0, r4
add sp, #4
ldmia sp!, {r4, fp, lr}
```

End_Fact:

```
mov sp, fp
ldmia sp!, {r5, fp, lr}
```

We turned Fact into the constant 0.

What did we overlook?

Fib_Fact:

```
stmbd sp!, {r4, fp, lr}
add fp, sp, #8
sub sp, #4
str r0, [fp, #-12]
bl Fib
mov r4, r0
sub r0, fp, #12
bl Fact
add r0, r0, r4
add sp, #4
ldmia sp!, {r4, fp, lr}
bl Fib
mov r4, r0
sub r0, fp, #12
bl Fact
add r0, r0, r4
add sp, #4
ldmia sp!, {r4, fp, lr}
```

G

We turned Fact into the constant 0.

What did we overlook?
What is the value of \( x \) during one execution of \( \text{Fact} \)?

\[\text{Fib}_{-}\text{Fact}:\]
- \( \text{stmdb}\) \( \text{sp}!, \{r4, fp, lr\} \)
- \( \text{add}\) \( fp, sp, #8 \)
- \( \text{sub}\) \( sp, #4 \)
- \( \text{str}\) \( r0, [fp, #-12]\)
- \( \text{bl}\) \( \text{Fact} \)
- \( \text{add}\) \( r0, r0, r4 \)
- \( \text{add}\) \( sp, #4 \)
- \( \text{ldmia}\) \( \text{sp}!, \{r4, fp, lr\} \)
- \( \text{bx}\) \( lr \)

\[\text{Fact}:\]
- \( \text{stmdb}\) \( \text{sp}!, \{r5, fp, lr\} \)
- \( \text{add}\) \( fp, sp, #4 \)
- \( \text{mov}\) \( r5, r0 \)
- \( \text{ldr}\) \( r0, [r5]\)
- \( \text{cmp}\) \( r0, #0 \)
- \( \text{bne}\) \( \text{Case}_{-}\text{Others} \)
- \( \text{mov}\) \( r0, #1 \)
- \( \text{b}\) \( \text{End}_{-}\text{Fact} \)

\[\text{Case}_{-}\text{Others}:\]
- \( \text{sub}\) \( r0, #1 \)
- \( \text{str}\) \( r0, [r5]\)
- \( \text{bl}\) \( \text{Fact} \)
- \( \text{add}\) \( r0, r0, r4 \)
- \( \text{add}\) \( sp, #4 \)
- \( \text{ldmia}\) \( \text{sp}!, \{r4, fp, lr\} \)
- \( \text{bx}\) \( lr \)

\[\text{End}_{-}\text{Fact}:\]
- \( \text{mov}\) \( sp, fp \)
- \( \text{ldmia}\) \( \text{sp}!, \{r5, fp, lr\} \)
- \( \text{bx}\) \( lr \)
## Functions

### Parameter passing

Call by …

<table>
<thead>
<tr>
<th>Information flow</th>
<th>Access</th>
<th>by copy</th>
<th>by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>by value</strong></td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
<td><strong>by reference (immutable)</strong> No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td>in</td>
<td><strong>by result</strong></td>
<td>Calling function expects the parameter value to appear in a specific space at return.</td>
<td></td>
</tr>
<tr>
<td>out</td>
<td></td>
<td></td>
<td>No read access from inside the function, write access on return.</td>
</tr>
<tr>
<td>in &amp; out</td>
<td><strong>by value result</strong></td>
<td>Parameter is copied to a local variable and copied back at return.</td>
<td><strong>by reference (mutable)</strong> Function can read and write at any time. Outside code shall not write.</td>
</tr>
</tbody>
</table>

We should have used either of those modes

Yet we used this mode
### Functions

#### When to use what?

<table>
<thead>
<tr>
<th>Information flow</th>
<th>Access</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>by copy</td>
<td>by value</td>
<td>by reference (immutable)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
<td>No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td>out</td>
<td>by result</td>
<td>Calling function expects the parameter value to appear in a specific space at return.</td>
<td>by reference (mutable, no read)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>No read access from inside the function, write access on return.</td>
</tr>
<tr>
<td>in &amp; out</td>
<td>by value result</td>
<td></td>
<td>by reference (mutable)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Parameter is copied to a local variable and copied back at return.</td>
<td>Function can read and write at any time. Outside code shall not write.</td>
</tr>
</tbody>
</table>
### One-way and by-copy

Those are side-effect-free and hence the resulting scenarios are easy to analyse. Copying large data structures might be time consuming or infeasible. Values can be passed in registers.

<table>
<thead>
<tr>
<th>Information flow</th>
<th>Access</th>
<th>by copy</th>
<th>by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>in</strong></td>
<td>by value</td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
<td>by reference (immutable)</td>
</tr>
<tr>
<td></td>
<td>by result</td>
<td>Calling function expects the parameter value to appear in a specific space at return.</td>
<td>by reference (mutable, no read)</td>
</tr>
<tr>
<td><strong>in &amp; out</strong></td>
<td>by value result</td>
<td>Parameter is copied to a local variable and copied back at return.</td>
<td>by reference (mutable)</td>
</tr>
</tbody>
</table>
# Two-way and by-copy

Still side-effect-free within the function (but not on the outside).

Potentially more convenient as memory space can be reused.

Values can be passed in registers.

<table>
<thead>
<tr>
<th>Information flow</th>
<th>Access</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>by copy</td>
<td>by value</td>
</tr>
<tr>
<td></td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
<td></td>
</tr>
<tr>
<td>out</td>
<td>by result</td>
<td>by reference (immutable)</td>
</tr>
<tr>
<td></td>
<td>Calling function expects the parameter value to appear in a specific space at return.</td>
<td>No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td>in &amp; out</td>
<td>by value result</td>
<td>by reference (mutable, no read)</td>
</tr>
<tr>
<td></td>
<td>Parameter is copied to a local variable and copied back at return.</td>
<td>No read access from inside the function, write access on return.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>by reference (mutable)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Function can read and write at any time. Outside code shall not write.</td>
</tr>
</tbody>
</table>
## Two-way and by-reference

Side-effecting and particular care is required as multiple entities could write on this.

No data has to be replicated.

Values have to be passed in memory.

<table>
<thead>
<tr>
<th>Information flow</th>
<th>Access</th>
<th>by copy</th>
<th>by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>by value</td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
<td>by reference (immutable) No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td>out</td>
<td>by result</td>
<td>Calling function expects the parameter value to appear in a specific space at return.</td>
<td>by reference (mutable, no read) No read access from inside the function, write access on return.</td>
</tr>
<tr>
<td>in &amp; out</td>
<td>by value result</td>
<td>Parameter is copied to a local variable and copied back at return.</td>
<td>by reference (mutable) Function can read and write at any time. Outside code shall not write.</td>
</tr>
</tbody>
</table>
### One-way-Out and by-reference

Side-effect-free, if new memory is allocated on return – cannot be enforced on assembly level (requires compiler).
Values have to be passed in memory.

<table>
<thead>
<tr>
<th>Information flow</th>
<th>Access</th>
<th>by copy</th>
<th>by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td><strong>by value</strong></td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
<td><strong>by reference (immutable)</strong> No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td>out</td>
<td><strong>by result</strong></td>
<td>Calling function expects the parameter value to appear in a specific space at return.</td>
<td><strong>by reference (mutable, no read)</strong> No read access from inside the function, write access on return.</td>
</tr>
<tr>
<td>in &amp; out</td>
<td><strong>by value result</strong></td>
<td>Parameter is copied to a local variable and copied back at return.</td>
<td><strong>by reference (mutable)</strong> Function can read and write at any time. Outside code shall not write.</td>
</tr>
</tbody>
</table>
### One-way-In and by-reference

Side-effect-free – cannot be enforced on assembly level (requires compiler).

No data has to be replicated.

Values have to be passed in memory.

<table>
<thead>
<tr>
<th>Information flow</th>
<th>Access</th>
<th>by copy</th>
<th>by reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>in</strong></td>
<td>by value</td>
<td>Parameter becomes a constant inside the function or is copied into a local variable.</td>
<td>by reference (immutable) No write access is allowed while the function runs (also from outside the function).</td>
</tr>
<tr>
<td><strong>out</strong></td>
<td>by result</td>
<td>Calling function expects the parameter value to appear in a specific space at return.</td>
<td>by reference (mutable, no read) No read access from inside the function, write access on return.</td>
</tr>
<tr>
<td><strong>in &amp; out</strong></td>
<td>by value result</td>
<td>Parameter is copied to a local variable and copied back at return.</td>
<td>by reference (mutable) Function can read and write at any time. Outside code shall not write.</td>
</tr>
</tbody>
</table>
Generic Stack-Frame

Let there be some (global) data on the stack.

Stack-Base (SB) is a static address, always pointing to the ... well.

Stack-Pointer (SP) points to the current top of the stack.

Global variables can also be stored someplace else.
Functions

Generic Stack-Frame

The current code prepared to call a function:
▷ Push parameters on the stack.
Works for any data size (unless the stack overflows) and parameter passing mode.

Types, storage structures and passing modes have to be agreed upon between caller and callee.
Functions

Generic Stack-Frame

The current code prepared to call a function:
- Push parameters on the stack.
Works for any data size (unless the stack overflows) and parameter passing mode.

```plaintext
Types, storage structures and passing modes have to be agreed upon between caller and callee.

- Solved if it’s the same language, compiler and program.

- If the languages or the compilers are different, then standards will be required.
```
Functions (in programming languages) have a context. 

E.g. the surrounding function or the hosting object. 

The caller knows this context and provides it.
Functions (in programming languages) have a context.

E.g. the *surrounding function* or the *hosting object*.

The caller knows this context and provides it.

Some languages will not have a context by default, like C or Assembly.

*(gnu C expands the C standard and provides it though)*
Generic Stack-Frame

The caller also provides a reference to its own stack frame.

This builds a linear chain of calls through the stack.

Will be used e.g. for debugging (stack trace) and exception propagation.

The static and dynamic link might be identical in some cases.
function a (x : Integer) return Integer is
    function b (y : Integer) return Integer is (x + y);
    function c (z : Integer) return Integer is (b (z));
begin
    return c (x);
end a;

a :: Integer -> Integer
a x = c x

where
    b :: Integer -> Integer
    b y = x + y
    c :: Integer -> Integer
    c z = b z
Functions

Static vs. dynamic links

function a (x : Integer) return Integer is
  function b (y : Integer) return Integer is (x + y);
  function c (z : Integer) return Integer is (b (z));
begin
  return c (x);
end a;

a :: Integer -> Integer
a x = c x

where
  b :: Integer -> Integer
  b y = x + y
  c :: Integer -> Integer
  c z = b z

The dynamic and static link for function c are both function a.
Static vs. dynamic links

function a (x : Integer) return Integer is
  function b (y : Integer) return Integer is (x + y);
  function c (z : Integer) return Integer is (b (z));
begin
  return c (x);
end a;

\[
a \colon \text{Integer} \rightarrow \text{Integer}
\]

\[
a \, x = \, c \, x
\]

where

\[
b \colon \text{Integer} \rightarrow \text{Integer}
\]

\[
b \, y = \, x + \, y
\]

\[
c \colon \text{Integer} \rightarrow \text{Integer}
\]

\[
c \, z = \, b \, z
\]

- The \textit{dynamic and static link} for function \(c\) are both function \(a\).

- The caller of function \(b\) is function \(c\).

- Hence exceptions raised in \(b\) are handled first in \(b\), then in \(c\), and then \(a\).
Function $a$ (x : Integer) return Integer is

  function $b$ (y : Integer) return Integer is (x + y);
  function $c$ (z : Integer) return Integer is ($b$ (z));

begin
  return $c$ (x);
end $a$;

The **dynamic link** (prior frame)

- The caller of function $b$ is function $c$.

- Hence exceptions raised in $b$ are handled first in $b$, then in $c$, and then $a$.

The **static link** (context)

- The context for function $b$ is function $a$.

- Hence $b$ can access $x$ (but not $z$).

The **dynamic** and **static link** for function $c$ are both function $a$. 

© 2021 Uwe R. Zimmer, The Australian National University
**Generic Stack-Frame**

The caller also provides a reference to its own stack frame.

This builds a linear chain of calls through the stack.

Will be used e.g. for debugging (stack trace) and exception propagation.
Generic Stack-Frame

The last item to be stored before handing over control is the address of the following instruction.

* The control flow can later return to this address.
Functions

Generic Stack-Frame

Control is handed over to the callee.

The Instruction Pointer (IP), sometimes also called Program Counter (PC) is changed to a new address.

Operations from here are in the control of the callee.
Generic Stack-Frame

A Frame Pointer (FP) is established at the boundary between the caller and the callee.

- Upwards from the FP: data from this function.
- Downwards from the FP: data provided by the previous function.

Saved resources are for instance registers which the callee is planning to use.
Generic Stack-Frame

Local variables are allocated (by moving the stack pointer).

Local variables can be of any size or structure, unless the stack overflows.

The completes a new stack frame.
Generic Stack-Frame

Local variables are allocated (by moving the stack pointer).
Local variables can be of any size or structure, unless the stack overflows.

$\triangleright$ The completes a new stack frame.

While this function is executing, local variables can still be added.

$\triangleright$ Handy, if e.g. the size of a local variable is not yet determined when the function starts.
The next function call will produce the next stack frame.

Variables and parameters from the context stay visible (via the chain of static links).
Functions

Generic Stack-Frame

The next function call will produce the next stack frame.

- Variables and parameters from the context stay visible (via the chain of static links).

Local variables can only be added to the currently executing function.
**Generic Stack-Frame**

The next function call will produce the next stack frame.

Note which variables and parameters are visible.
Generic Stack-Frame

The next function call will produce the next stack frame.

- Note which variables and parameters are visible.

Accessing the context like that can be inefficient!

Compilers may choose other mechanism (e.g. displays, which make all context levels accessible at once).
How fast / complex is the allocation and deallocation of local variables and parameters on the stack?
Functions

Generic Stack-Frame – Caller

Pre_Call:

... ; Allocate/identify space for the parameters
... ; Copy the in and in-out
... ; parameters to this space
... ; Potentially provide links
... ; Provide a return address ("Post_Call")
... ; (usually implicit with the call itself)

☞ Call the function

Post_Call:

... ; Copy the out and in-out parameters
... ; to local variables or registers
... ; Potentially restore the frame pointer
... ; Restore the stack to its previous state
... ; (if the stack has been used)
Functions

Generic Stack-Frame – Callee

Prologue:

... ; Save all registers which are needed
... ; inside this function to the stack
... ; Establish a new frame pointer
... ; while potentially saving the previous fp
... ; Allocate/identify space for local variables
... ; Potentially initialize local variables

Operations, which will use
local and context variables and parameters
(via the FP(s))

Epilogue:

... ; Potentially restore the prior frame pointer
... ; Restore the stack to its state at entry

Return from function
How to keep any memory allocation after function return?
By using an out, by-reference parameter, the link to the newly allocated memory area is kept.

How to keep any memory allocation after function return?
By using an out, by-reference parameter, the link to the newly allocated memory area is kept.

... and a local variable in the calling function can keep this link.
**Generic Stack-Frame – Heap**

- How to keep any memory allocation after function return?

- By using an out, by-reference parameter, the link to the newly allocated memory area is kept.

- ... and a local variable in the calling function can keep this link.

- When to deallocate though?
  - Garbage collection (Java)?
  - Smart pointers (C++)?
  - Reference ownerships (Rust)?
  - Scoped pointers / storage pools (Ada)?
### Functions

**Summary**

**Functions**

- **Framework**
  - Return address
  - Relative addressing

- **Parameter passing modes and mechanisms**
  - Copy versus reference
  - Information flow directions
  - Late evaluation

- **Stackframes**
  - Static and dynamic links
  - Parameters
  - Local variables
Data Structures

Uwe R. Zimmer - The Australian National University
References for this chapter

[Patterson17]
David A. Patterson & John L. Hennessy
Computer Organization and Design – The Hardware/Software Interface
Chapter 2 “Instructions: Language of the Computer”,
Chapter 5 “Large and Fast: Exploiting Memory Hierarchy”
ARM edition, Morgan Kaufmann 2017
Array layout

Elements of equal size sequentially in memory

Array element

Lower array bound

Upper array bound

$x$

$e_{0\_b0}$

$e_{0\_b1}$

$e_{0\_b2}$

$e_{0\_b3}$

$x + 4$

$e_{1\_b0}$

$e_{1\_b1}$

$e_{1\_b2}$

$e_{1\_b3}$

$x + 8$

$e_{2\_b0}$

$e_{2\_b1}$

$e_{2\_b2}$

$e_{2\_b3}$

$x + 12$

$e_{3\_b0}$

$e_{3\_b1}$

$e_{3\_b2}$

$e_{3\_b3}$

$x + 15$
## Array addressing

### Data Structures

<table>
<thead>
<tr>
<th>Index</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>x + 0·es</td>
<td>e0_b0, e0_b1, e0_b2, e0_b3</td>
</tr>
<tr>
<td>x + 1·es</td>
<td>e1_b0, e1_b1, e1_b2, e1_b3</td>
</tr>
<tr>
<td>x + 2·es</td>
<td>e2_b0, e2_b1, e2_b2, e2_b3</td>
</tr>
<tr>
<td>x + 3·es</td>
<td>e3_b0, e3_b1, e3_b2, e3_b3</td>
</tr>
</tbody>
</table>

- **Element size (es)**: 4 bytes
- **Lower array bound**: x + 0·es
- **Upper array bound**: x + 4·es - 1
- **Can arrays always be stored “packed”?**

**Questions:**

- **What happens if array bounds are violated?**
- **Maybe be good to have es = 2^n?**
- **And who is checking that?**
Array addressing via index register

\[ \text{ldr} <c><q> <Rd>, [<Rb>, <Ri> \{, \text{lsl} \ #<shift>\}] \]
Array addressing via index register

**Rb** - Base address

**Ri** - Index (shifted)

Shift works if $es = 2^n$

- Can be handy if **Ri** is actually an index.
- Otherwise **Ri** will be a byte offset.

\[
\text{ldr}<c><q><Rd>, [<Rb>, <Ri>, lsl #<shift>]
\]
Array addressing via element pointer

\[
\begin{array}{c}
  \text{Rp - Array pointer} \\
  \text{Element offset} \\
  \text{Write-back}
\end{array}
\]

\[
\text{Rd - Destination}
\]

\[
\text{ldr}<c><q> \ [\text{Rd}], [\text{Rp}], \#+/-\text{<offset>}
\]
### Array addressing via element pointer

#### Rd - Destination

#### Rp - Array pointer

#### Element offset

#### Write-back

#### Efficient if array has to be worked sequentially

\[ \text{ldr}<c><q> \ <Rd>, \ [<Rp>], \ #+/−<offset> \]
Calculate $\sum x_i$

```c
int sum (unsigned int uints [], unsigned int from, unsigned int to) {
    int i;
    int acc = 0;
    for (i = from; i <= to; i++) {
        acc += uints [i];
    }
    return acc;
}
```

```c
unsigned int uints [100];
unsigned int s;
int i;
for (i = 0; i <= 99; i++) {
    uints [i] = rand () ;
}
s = sum (uints, 0, 99);
```

```c
type Naturals is array (Integer range <>) of Natural;
function Sum (Numbers : Naturals) return Natural is
    Acc : Natural := 0;
begin
    for n of Numbers loop
        Acc := Acc + n;
    end loop;
    return Acc;
end Sum;
```

```c
Numbers : constant Naturals (1 .. 100) :=
    (others => Random (Numbers_Generator));
Sum_of_Numbers : constant Natural := Sum (Numbers);
```
Data Structures

**Calculate** \( \sum x_i \)

```c
int sum (unsigned int uints [], unsigned int from, unsigned int to) {
    int i;
    int acc = 0;
    for (i = from; i <= to; i++) {
        acc += uints [i];
    }
    return acc;
}
```

```c
unsigned int uints [100];
unsigned int s;
int i;
for (i = 0; i <= 99; i++) {
    uints [i] = rand ();
}
s = sum (uints, 0, 99);
```

```c
type Naturals is array (Integer range <>) of Natural;
function Sum (Numbers : Naturals) return Natural is
    Acc : Natural := 0;
begin
    for n of Numbers loop
        Acc := Acc + n;
    end loop;
    return Acc;
end Sum;
```

```c
Numbers : constant Naturals (1 .. 100) :=
    (others => Random (Numbers_Generator));
Sum_of_Numbers : constant Natural := Sum (Numbers);
```

Best of luck with the array bounds!
Arbitrary array indexing

; r0 base address for array a
; r1 from array index
; r2 to array index

```
mov r3, #0    ; sum := 0
mov r4, #4    ; element size is 4 bytes
mov r5, #-1   ; first_element_offset

for_sum:
  cmp r1, r2    ; i > to
  bgt end_for_sum
  mla r6, r1, r4, r5  ; element_offset := (i * 4) - first_element_offset
  ldr r7, [r0, r6]  ; a [i] := [base + element_offset]
  add r3, r7    ; sum := sum + a [i]
  add r1, #1    ; i := i + 1
  b for_sum

end_for_sum:
  mov r0, r3    ; r0 sum over all a [from .. to]
```

Zero-based array indexing

; r0 base address for array a
; r1 from array index
; r2 to array index

```
mov r3, #0    ; sum := 0
mov r4, #4    ; element size is 4 bytes

for_sum:
  cmp r1, r2    ; i > to
  bgt end_for_sum
  mul r5, r1, r4   ; element_offset := (i * 4)
  ldr r6, [r0, r5]   ; a [i] := [base + element_offset]
  add r3, r6    ; sum := sum + a [i]
  add r1, #1    ; i := i + 1
  b for_sum

end_for_sum:
  mov r0, r3    ; r0 sum over all a [from .. to]
```
Replacing multiplication with shifted index register

; r0 base address for array a
; r1 from array index
; r2 to array index

mov r3, #0    ; sum := 0
for_sum:
    cmp r1, r2    ; i > to
    bgt end_for_sum
    ldr r4, [r0, r1, lsl #2] ; a [i] := [base + element_offset]
    add r3, r4    ; sum := sum + a [i]
    add r1, #1    ; i := i + 1
    b for_sum
end_for_sum:
    mov r0, r3    ; r0 sum over all a [from .. to]
Replacing indices with offsets

; r0 base address for array a
; r1 from array index
; r2 to array index

\[
\begin{align*}
\text{lsl} & \quad r1, r1, #2 & \quad ; \text{translate from index to offset} \\
\text{lsl} & \quad r2, r2, #2 & \quad ; \text{translate to index to offset} \\
\text{mov} & \quad r3, #0 & \quad ; \text{sum} := 0
\end{align*}
\]

\textbf{for\_sum:}

\[
\begin{align*}
\text{cmp} & \quad r1, r2 & \quad ; \text{i} > \text{to} \\
\text{bgt} & \quad \text{end\_for\_sum} \\
\text{ldr} & \quad r4, [r0, r1] & \quad ; \text{a}[i] := \text{base} + \text{offset} \\
\text{add} & \quad r3, r4 & \quad ; \text{sum} := \text{sum} + \text{a}[i] \\
\text{add} & \quad r1, #4 & \quad ; \text{offset} := \text{offset} + 4 \\
\text{b} & \quad \text{for\_sum}
\end{align*}
\]

\textbf{end\_for\_sum:}

\[
\begin{align*}
\text{mov} & \quad r0, r3 & \quad ; \text{r0 sum over all a}[\text{from .. to}]
\end{align*}
\]
**Data Structures**

**Assuming non-empty arrays**

; r0 base address for array a
; r1 from array index
; r2 to array index >= from index

<table>
<thead>
<tr>
<th>lsl</th>
<th>r1, r1, #2</th>
<th>; translate from index to offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>lsl</td>
<td>r2, r2, #2</td>
<td>; translate to index to offset</td>
</tr>
<tr>
<td>mov</td>
<td>r3, #0</td>
<td>; sum := 0</td>
</tr>
</tbody>
</table>

```
for_sum:
  ldr  r4, [r0, r1] ; a [i] := [base + offset]
  add  r3, r4      ; sum := sum + a [i]
  add  r1, #4      ; offset := offset + 4
  cmp  r1, r2      ; i <= to
  ble for_sum      ;
end_for_sum:
  mov  r0, r3      ; r0 sum over all a [from .. to]
```
Replacing offsets with addresses

; r0 base address for array a
; r1 from array index
; r2 to array index >= from index

```
lsl r1, r1, #2 ; translate from index to offset
lsl r2, r2, #2 ; translate to index to offset
add r1, r0      ; translate from index to address -> i_addr
add r2, r0      ; translate to index to address -> to_addr
mov r0, #0      ; sum := 0
```

for_sum:
```
ldr r3, [r1], #4 ; a [i] := [i_addr]; i_addr += 4
add r0, r3      ; sum := sum + a [i]
cmp r1, r2      ; i_addr <= to_addr
ble for_sum
```

end_for_sum:
```
; r0 sum over all a [from .. to]
```
Array Slices

numbers = [0, 1, 2, 3, 4, 5]
numbersSlice = numbers [1:3]
# numbersSlice equals [1, 2, 3]

numbers := []int {0, 1, 2, 3, 4, 5}
numbersSlice := numbers [1:3]

type Naturals is array (Integer range <>) of Natural;

Numbers : constant Naturals (-50 .. 50) := (others => Random (Generator));
Numbers_Slice_1 : constant Naturals := Numbers (-10 .. 10);
Numbers_Slice_2 : constant Naturals ( 1 .. 10) := Numbers (11 .. 20);
Numbers_Slice_3 : Naturals := Numbers (-20 .. 50);

begin
  for n of Numbers_Slice_3 loop
    n := n + 1;
  end loop;
end;
Array Slices

```haskell
numbers = [0, 1, 2, 3, 4, 5]
numbersSlice = numbers [1:3]
# numbersSlice equals [1, 2, 3]
```

```haskell
numbers := []int {0, 1, 2, 3, 4, 5}
numbersSlice := numbers [1:3]
```

```haskell
type Naturals is array (Integer range <>) of Natural;
Numbers := constant Naturals (-50 .. 50) := (others => Random (Generator));
Numbers_Slice_1 := constant Naturals := Numbers (-10 .. 10);
Numbers_Slice_2 := constant Naturals ( 1 .. 10) := Numbers ( 11 .. 20);
Numbers_Slice_3 := Naturals := Numbers (-20 .. 50);

begin
  for n of Numbers_Slice_3 loop
    n := n + 1;
  end loop;
end;
```

Are those copy or reference affairs?
Copy array slice

; r0 base address for array a
; r1 from array index
; r2 to array index >= from index
; r3 base address for array b

```
lsl r1, r1, #2  ; translate from index to offset
lsl r2, r2, #2  ; translate to index to offset
add r1, r0      ; translate from index to address -> i_addr
add r2, r0      ; translate to index to address -> to_addr

for_copy:
  ldr r4, [r1], #4  ; a [i]    := [i_addr]; i_addr += 4
  str r4, [r3], #4  ; [j_addr] := a [i]    ; j_addr += 4
  cmp r1, r2       ; i_addr <= to_addr
  ble for_copy

end_for_copy:
  ; b [] := a [from .. to]
```
Copy array slice

; r0 base address for array a
; r1 from array index
; r2 to array index >= from index
; r3 base address for array b

lsl r1, r1, #2 ; translate from index to offset
lsl r2, r2, #2 ; translate to index to offset
add r1, r0 ; translate from index to address -> i_addr
add r2, r0 ; translate to index to address -> to_addr

for_copy:

ldr r4, [r1], #4 ; a [i] := [i_addr]; i_addr += 4
str r4, [r3], #4 ; [j_addr] := a [i] ; j_addr += 4
cmp r1, r2 ; i_addr <= to_addr
ble for_copy

end_for_copy:

; b [] := a [from .. to]

Moving blocks of memory can be done even/much faster with special hardware.

DMA controllers
Summary

Data Structures

• Arrays
  • Structure
  • Alignment
  • Addressing
  • Iterators
  • Copy procedures
Asynchronism

Uwe R. Zimmer - The Australian National University
References for this chapter

[Patterson17]
David A. Patterson & John L. Hennessy
*Computer Organization and Design – The Hardware/Software Interface*
Chapter 4 “The Processor”,
Chapter 6 “Parallel Processors from Client to Cloud”
ARM edition, Morgan Kaufmann 2017
Asynchronism

Why?

How do you handle your communication flow?
Why?

How do you handle your communication flow?

Do you have times when you check certain communication?

Is certain communication interrupting you? – at any time?

Do you assign “importance levels” to your communication channels/sources?
... running its sequence of machine instructions.
Asynchronism

STM32L476 Discovery

CPU

... running its sequence of machine instructions.

MCU

How to interact with all the other devices inside the MCU?
STM32L476 Discovery

- Multiplexed 24 bit \(\Sigma\Delta\)-DAC converter with stereo power amp
- Headphone jack
- USB OTG
- "9 axis" motion sensor (underneath display):
  - 3 axis accelerometer
  - 3 axis gyroscope
  - 3 axis magnetometer
- Current meter to MCU
  - 60 nA ... 50 mA
- Microphone
Asynchronism

STM32L476 Discovery

- Debugger state
- User LEDs
- Reset
- OTG LEDs
- Power
- Over current
- LCD
- Joystick
CPU

... running its sequence of machine instructions.

How to interact with all the other devices inside the

MCU

? 

... and then with all the devices on the board?
STM32L476 Discovery

CPU

… running its sequence of machine instructions.

How to interact with all the other devices inside the MCU?

MCU

… and then with all the devices on the board?

and then with the rest of the world

… which is connected to the board?
Polling

**Sequential machine instructions**

- All external devices need to be “checked” by asking for their status.
- This should usually happen (semi-) regularly.
Polling

Sequential machine instructions

- All external devices need to be “checked” by asking for their status.
- This should usually happen (semi-) regularly.

- This will lead to a loop of polling requests.

- Maximal latencies can be calculated straightforward.
- Simplicity of design (with small number of devices).
- Fastest option with small number of devices (like: one).
- All devices will need to wait their turn
  ... even if this device is the only one with new data!
- The “main” program transforms into one large loop which can be hard to handle in terms of scalable program design.
- Events or data can be missed.
Interrupts

- One or multiple lines wired directly into the sequencer

**Required for:**
- Pre-emptive scheduling, Timer driven actions,
- Transient hardware interactions, ...

- Usually preceded by an external logic (“interrupt controller”) which accumulates and encodes all external requests.

On interrupt (if unmasked):
- CPU stops normal sequencer flow.
- Lookup of interrupt handler’s address
- Current IP and state pushed onto stack.
- IP set to interrupt handler.
We successfully interrupted a sequence of operations ...
Interrupt processing

Interrupt handler

Program

Stack

Code

---

PC

FP

SP

Base

Local variables

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Global variables

---
Interrupt processing

Interrupt handler
Interrupt processing

Interrupt handler

Program

Stack

Code

- PC

Push registers
Declare local variables

Local variables

Registers

- SP

Global variables

Base

- SP

Context

Parameters

Return address

Local variables

Context

Parameters

Global variables

Return address

Local variables
Interrupt processing

Interrupt handler

Push registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time critical code ..
Interrupt processing

Interrupt handler

Push registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time
critical code ..
Remove local variables
Interrupt processing

Interrupt handler

Program

Stack

Code

Push registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time
critical code ..
Remove local variables
Pop registers

Local variables
Return address
Context
Parameters
Local variables
Return address
Context
Parameters
Global variables
Base
## Interrupt processing

### Interrupt handler

- Push registers
- Declare local variables
- Run handler code
  - .. do some I/O ..
  - .. or run some time critical code ..
- Remove local variables
- Pop registers

---

Bahia Honda Rail Bridge (Creative Commons Attribution-ShareAlike 3.0, Photography by MrX at English Wikipedia)
We successfully interrupted a sequence of operations ...  

... and now the trick to get to the other side.
Interrupt processing

Interrupt handler

Program

Stack

Code

PC

SP

FP

Base

Local variables

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Global variables

© 2021 Uwe R. Zimmer, The Australian National University
Interrupt processing

Interrupt handler

Program

Stack

Code

PC

Flags

FP

Local variables

Base

Global variables

Return address

Context

Parameters

- SP

- FP

© 2021 Uwe R. Zimmer, The Australian National University
Interrupt processing

Interrupt handler

Program

Stack

Code

The CPU hardware (!) did that, before anything was changed.
Interrupt processing

Interrupt handler

Program

Stack

Code

- SP

Local variables

Registers

Flags

PC

- Base

Global variables

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Local variables

Push registers

Declare local variables
Interrupt processing

Interrupt handler

Push registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time
critical code ..
Interrupt processing

Interrupt handler

Push registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time
     critical code ..
Remove local variables
Interrupt processing

Interrupt handler

Push registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time critical code ..
Remove local variables
Pop registers
Interrupt processing

Interrupt handler

Push registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time
  critical code ..
Remove local variables
Pop registers
Return from interrupt
Interrupt processing

Interrupt handler

Program

Stack

Code

PC

SP

FP

Local variables
Return address
Context
Parameters
Local variables
Return address
Context
Parameters
Global variables

© 2021 Uwe R. Zimmer, The Australian National University
Interrupt processing

Interrupt handler

LR is loaded with a special value
Interrupt processing

Interrupt handler

Program

Stack

Code

- SP

- FP

- PC

Clear interrupt flag
(Adjust priorities)
(Re-enable interrupt)

Scratch registers
Flags
PC
Local variables
Return address
Context
Parameters
Local variables
Return address
Context
Parameters
Global variables

© 2021 Uwe R. Zimmer, The Australian National University
Interrupt processing

Interrupt handler

- Clear interrupt flag
  (Adjust priorities)
- Re-enable interrupt
- Push other registers
- Declare local variables

Diagram showing stack and register areas including:
- Local variables
- Registers
- Scratch registers
- Flags
- PC
- Return address
- Context
- Parameters
- Global variables
Interrupt processing

Interrupt handler

Program

Stack

- SP
- PC

Local variables
Registers
Scratch registers
Flags

PC

- Local variables
- Return address
- Context
- Parameters
- Global variables

Clear interrupt flag
(Adjust priorities)
(Re-enable interrupt)
Push other registers
Declare local variables
Run handler code
.. do some I/O ..
.. or run some time
critical code ..
Interrupt processing

Interrupt handler

Program

Stack

Code

Clear interrupt flag
(Adjust priorities)
(Re-enable interrupt)
Push other registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time
critical code ..
Remove local variables
Pop other registers
Interrupt processing

Interrupt handler

Clear interrupt flag
(Adjust priorities)
(Re-enable interrupt)
Push other registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time
    critical code ..
Remove local variables
Pop other registers
Return ("bx lr")
Interrupt processing

Interrupt handler

Clear interrupt flag
(Adjust priorities)
(Re-enable interrupt)

Push other registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time
critical code ..
Remove local variables
Pop other registers
Return ("bx lr")
Interrupt handler

Things to consider

- Interrupt handler code can be interrupted as well.
- Are you allowing to interrupt an interrupt handler with an interrupt on the same priority level (e.g. the same interrupt)?
- Can you overrun a stack with interrupt handlers?
Interrupt handler

Things to consider

- Interrupt handler code can be interrupted as well.
- Are you allowing to interrupt an interrupt handler with an interrupt on the same priority level (e.g. the same interrupt)?
- Can you overrun a stack with interrupt handlers?

- Can we have one of those?
Multiple programs

If we can execute interrupt handler code “concurrently” to our “main” program:

Can we then also have multiple “main” programs?
Asynchronism

**Context switch**

Dispatcher

Process 1

- PCB
  - PID
  - \ldots \ldots \ldots \ldots

- Code
  - PC
  - SP
  - FP

- Stack
  - \ldots \ldots \ldots \ldots
  - Local variables
  - Return address
  - Context
  - Parameters
  - \ldots \ldots \ldots \ldots

Process 2

- PCB
  - PID
  - SP
  - \ldots \ldots \ldots \ldots

- Code
  - \ldots \ldots \ldots \ldots
  - Context-switch-variables
  - Registers
  - Flags
  - PC

- Stack
  - \ldots \ldots \ldots \ldots
  - Global variables
  - Local variables
  - Return address
  - Context
  - Parameters
  - \ldots \ldots \ldots \ldots
Context switch

Dispatcher

Process 1

PCB

PID

Code

Stack

- SP

Flags

Local variables

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Global variables

Base

Process 2

PCB

PID

Code

Stack

Context-switch-variables

Registers

Flags

PC

Local variables

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Global variables

Base

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Global variables

Base
Asynchronism

**Context switch**

Process 1

- PCB
  - PID
  - ... ...

- Code
- Stack
  - SP
  - Context-switch-variables
    - Registers
    - Flags
    - PC
    - Local variables
    - Return address
    - Context
    - Parameters
    - Base
    - Global variables

Process 2

- PCB
  - PID
  - SP
  - ... ...

- Code
- Stack
  - Context-switch-variables
    - Registers
    - Flags
    - PC
    - Local variables
    - Return address
    - Context
    - Parameters
    - Base
    - Global variables

Push registers
Declare local variables

Dispatcher
**Context switch**

Dispatcher

- Push registers
- Declare local variables
- Store SP to PCB 1

Process 1

- PCB
  - PID
  - SP
  - ... ...

- Code
  - ... ...

- Stack
  - Context-switch-variables
  - Registers
  - Flags
  - PC
  - Local variables
  - Return address
  - Context
  - Parameters
  - Local variables
  - Return address
  - Context
  - Parameters
  - Global variables

- Base

Process 2

- PCB
  - PID
  - SP
  - ... ...

- Code
  - ... ...

- Stack
  - Context-switch-variables
  - Registers
  - Flags
  - PC
  - Local variables
  - Return address
  - Context
  - Parameters
  - Local variables
  - Return address
  - Context
  - Parameters
  - Global variables

- Base

© 2021 Uwe R. Zimmer, The Australian National University
Asynchronism

Context switch

Dispatcher

Push registers
Declare local variables
Store SP to PCB 1
Scheduler

Process 1

PCB

PID

SP

... ...

Context-switch-variables

Registers

Flags

PC

Local variables

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Global variables

Code

Stack

SP

PC

Process 2

PCB

PID

SP

... ...

Context-switch-variables

Registers

Flags

PC

Local variables

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Global variables

Code

Stack

SP

PC
Context switch

Dispatcher

Push registers
Declare local variables
Store SP to PCB 1
Scheduler
Load SP from PCB 2
Asynchronism

Context switch

Dispatcher

Push registers
Declare local variables
Store SP to PCB 1
Scheduler
Load SP from PCB 2
Remove local variables

Process 1

PCB

PID  SP  …  …  …

Code
Stack

Context-switch-variables

Registers
Flags
PC
Local variables
Return address
Context
Parameters
Local variables
Return address
Context
Parameters
Global variables

Base

Process 2

PCB

PID  …  …  …

Code
Stack

Registers
Flags
PC
Local variables
Return address
Context
Parameters
Local variables
Return address
Context
Parameters
Global variables
**Context switch**

Dispatcher

Push registers
Declare local variables
Store SP to PCB 1
Scheduler
Load SP from PCB 2
Remove local variables
Pop registers

**Process 1**

- PCB
  - PID
  - SP
  - ...
- Code
- Stack
  - Context-switch-variables
  - Registers
  - Flags
  - PC
  - Local variables
  - Return address
  - Context
  - Parameters
  - Local variables
  - Return address
  - Context
  - Parameters
  - Global variables

**Process 2**

- PCB
  - PID
  - ...
- Code
- Stack
  - Context-switch-variables
  - Registers
  - Flags
  - PC
  - Local variables
  - Return address
  - Context
  - Parameters
  - Local variables
  - Return address
  - Context
  - Parameters
  - Global variables

© 2021 Uwe R. Zimmer, The Australian National University
Asynchronism

Context switch

Dispatcher

Push registers
Declare local variables
Store SP to PCB 1
Scheduler
Load SP from PCB 2
Remove local variables
Pop registers
Return from interrupt
Multi-tasking and Contention

Anything else could go wrong?
Multi-tasking and Contention

Anything else could go wrong?

-if there is neither communication nor contention between concurrent parts ... all is easy ... and boring.

What happens if concurrent programs share data?
Assumption 1: every individual base memory cell (word) load and store access is *atomic*

Assumption 2: there is *no* atomic combined load-store access

\[ G : \text{Natural} := 0; \quad \text{-- assumed to be mapped on a 1-word cell in memory} \]

\[
\begin{align*}
\text{task body P1 is} & \quad \text{task body P2 is} & \quad \text{task body P3 is} \\
\text{begin} & \quad \text{begin} & \quad \text{begin} \\
G & := 1 & G & := 2 & G & := 3 \\
G & := G + G; & G & := G + G; & G & := G + G; \\
\text{end P1;} & \quad \text{end P2;} & \quad \text{end P3;} \\
\end{align*}
\]

What is the value of \( G \)?
Shared variables

Atomic load & store operations

Assumption 1: every individual base memory cell (word) load and store access is atomic

Assumption 2: there is no atomic combined load-store access

G: .word 0x00000000

```
ldr r4, =G
mov r1, #1
str r1, [r4]
ldr r2, [r4]
ldr r3, [r4]
add r1, r2, r3
str r1, [r4]

ldr r4, =G
mov r1, #2
str r1, [r4]
ldr r2, [r4]
ldr r3, [r4]
add r1, r2, r3
str r1, [r4]

ldr r4, =G
mov r1, #3
str r1, [r4]
ldr r2, [r4]
ldr r3, [r4]
add r1, r2, r3
str r1, [r4]
```

What is the value in memory cell G after all three programs complete?
This is terrible!

Nobody is their right mind would analyse a program like that.

⋯ are we missing something?

⋯ is there an elegant way out?
Asynchronism

Mutual exclusion … or the lack thereof

Count : Integer := 0;

task body Enter is
begin
    for i := 1 .. 100 loop
        Count := Count + 1;
    end loop;
end Enter;

task body Leave is
begin
    for i := 1 .. 100 loop
        Count := Count - 1;
    end loop;
end Leave;

What is the value of Count after both programs complete?
Mutual exclusion ... or the lack thereof

Count: `.word` 0x00000000

```assembly
ldr    r4, =Count
mov    r1, #1

for_enter:
cmp    r1, #100
bgt    end_for_enter

ldr    r2, [r4]
add    r2, #1
str    r2, [r4]
add    r1, #1
b      for_enter

end_for_enter:  
```

```assembly
ldr    r4, =Count
mov    r1, #1

for_leave:
cmp    r1, #100
bgt    end_for_leave

ldr    r2, [r4]
sub    r2, #1
str    r2, [r4]
add    r1, #1
b      for_leave

end_for_leave:  
```

What is the value at address `Count` after both programs complete?
Mutual exclusion … or the lack thereof

Count: .word 0x00000000

ldr r4, =Count
mov r1, #1

for_enter:
  cmp r1, #100
  bgt end_for_enter

  ldr r2, [r4]
  add r2, #1
  str r2, [r4]

add r1, #1
b for_enter

end_for_enter:

for_leave:
  cmp r1, #100
  bgt end_for_leave

  ldr r2, [r4]
  sub r2, #1
  str r2, [r4]

add r1, #1
b for_leave

end_for_leave:

What is the value at address Count after both programs complete?
Mutual exclusion … or the lack thereof

Count: .word 0x00000000

for_enter:
   cmp r1, #100
   bgt end_for_enter

   enter_critical_fail:
      ldrex r2, [r4] ; tag [r4] as exclusive
      add r2, #1
      strex r0, r2, [r4] ; only if untouched
   
      cmp r0, #0
      bne enter_critical_fail
   
      add r1, #1
      b for_enter

end_for_enter:

for_leave:
   cmp r1, #100
   bgt end_for_leave

   leave_critical_fail:
      ldrex r2, [r4] ; tag [r4] as exclusive
      sub r2, #1
      strex r0, r2, [r4] ; only if untouched
   
      cmp r0, #0
      bne leave_critical_fail
   
      add r1, #1
      b for_leave

end_for_leave:

What is the value at address Count after both programs complete?
Mutual exclusion … or the lack thereof

Count: .word 0x00000000

```
ldr  r4, =Count
mov  r1, #1

for_enter:
  cmp  r1, #100
  bgt  end_for_enter

enter_critical_fail:
  ldrex  r2, [r4] ; tag [r4] as exclusive
  add  r2, #1
  strex  r0, r2, [r4] ; only if untouched
  cmp  r0, #0
  bne  enter_critical_fail
  add  r1, #1
  b  for_enter

end_for_enter:
```

Any context switch needs to clear reservations

```
ldr  r4, =Count
mov  r1, #1

for_leave:
  cmp  r1, #100
  bgt  end_for_leave

leave_critical_fail:
  ldrex  r2, [r4] ; tag [r4] as exclusive
  sub  r2, #1
  strex  r0, r2, [r4] ; only if untouched
  cmp  r0, #0
  bne  leave_critical_fail
  add  r1, #1
  b  for_leave

end_for_leave:
```

What is the value at address Count after both programs complete?
Count: .word 0x00000000

\begin{verbatim}
ldr  r4, =Count
mov  r1, #1

for_enter:
cmp  r1, #100
bgt  end_for_enter

ldr  r2, [r4]
add  r2, #1
str  r2, [r4]
add  r1, #1
b    for_enter

end_for_enter:

for_leave:
cmp  r1, #100
bgt  end_for_leave

ldr  r2, [r4]
sub  r2, #1
str  r2, [r4]
add  r1, #1
b    for_leave

end_for_leave:
\end{verbatim}

Negotiate who goes first

Critical section

Indicate critical section completed
for_enter:
cmp r1, #100
bgt end_for_enter
fail_lock_enter:
ldr r0, [r3]
cmp r0, #0
bne fail_lock_enter ; if locked
ldr r2, [r4]
add r2, #1
str r2, [r4]
add r1, #1
b for_enter
end_for_enter:

for_leave:
cmp r1, #100
bgt end_for_leave
fail_lock_leave:
ldr r0, [r3]
cmp r0, #0
bne fail_lock_leave ; if locked
ldr r2, [r4]
sub r2, #1
str r2, [r4]
add r1, #1
b for_leave
end_for_leave:
Count: `.word 0x00000000
Lock: `.word 0x00000000 ; #0 means unlocked

    ldr  r3, =Lock
    ldr  r4, =Count
    mov  r1, #1

for_enter:
    cmp  r1, #100
    bgt  end_for_enter

fail_lock_enter:
    ldr  r0, [r3]
    cmp  r0, #0
    bne  fail_lock_enter ; if locked
    mov  r0, #1          ; lock value
    str  r0, [r3]     ; lock

    ldr  r2, [r4]
    add  r2, #1
    str  r2, [r4]

add  r1, #1
b    for_enter

end_for_enter:

    ldr  r3, =Lock
    ldr  r4, =Count
    mov  r1, #1

for_leave:
    cmp  r1, #100
    bgt  end_for_leave

fail_lock_leave:
    ldr  r0, [r3]
    cmp  r0, #0
    bne  fail_lock_leave ; if locked
    mov  r0, #1          ; lock value
    str  r0, [r3]     ; lock

    ldr  r2, [r4]
    sub  r2, #1
    str  r2, [r4]

add  r1, #1
b    for_leave

end_for_leave:

Critical section

Critical section
Critical section

Any context switch needs to clear reservations

for_enter:
    cmp  r1, #100
    bgt  end_for_enter

fail_lock_enter:
    ldrex  r0, [r3]
    cmp  r0, #0
    bne  fail_lock_enter ; if locked
    mov  r0, #1 ; lock value
    strex  r5, r0, [r3] ; try lock
    cmp  r5, #0
    bne  fail_lock_enter ; if touched
    dmb ; sync memory

ldr  r2, [r4]
add  r2, #1
str  r2, [r4]

add  r1, #1
b  for_enter

end_for_enter:

for_leave:
    cmp  r1, #100
    bgt  end_for_leave

fail_lock_leave:
    ldrex  r0, [r3]
    cmp  r0, #0
    bne  fail_lock_leave ; if locked
    mov  r0, #1 ; lock value
    strex  r5, r0, [r3] ; try lock
    cmp  r5, #0
    bne  fail_lock_leave ; if touched
    dmb ; sync memory

ldr  r2, [r4]
sub  r2, #1
str  r2, [r4]

add  r1, #1
b  for_leave

end_for_leave:
Any context switch needs to clear reservations

Critical section

Critical section
Mutual exclusion: atomic test-and-set operation

```vhdl
type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
  L : Flag;
  begin
    loop
      loop
        [L := C; C := 1];
        exit when L = 0;
        ------ change process
      end loop;
      ------ critical_section_i;
      C := 0;
    end loop;
  end Pi;

task body Pj is
  L : Flag;
  begin
    loop
      loop
        [L := C; C := 1];
        exit when L = 0;
        ------ change process
      end loop;
      ------ critical_section_j;
      C := 0;
    end loop;
  end Pj;
```

Does that work?
Mutual exclusion: atomic test-and-set operation

```pl
type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
    L : Flag;
    begin
        loop
            loop
                [L := C; C := 1];
                exit when L = 0;
            ------ change process
            end loop;
            ------ critical_section_i;
            C := 0;
        end loop;
    end Pi;

task body Pj is
    L : Flag;
    begin
        loop
            loop
                [L := C; C := 1];
                exit when L = 0;
            ------ change process
            end loop;
            ------ critical_section_j;
            C := 0;
        end loop;
    end Pj;
```

- Mutual exclusion!, No deadlock!, No global live-lock!
- Works for any dynamic number of processes.
- Individual starvation possible! Busy waiting loops!
Mutual exclusion: atomic exchange operation

```plaintext
type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
L : Flag := 1;
begin
  loop
    loop
      [Temp := L; L := C; C := Temp];
      exit when L = 0;
      ------ change process
    end loop;
    ------ critical_section_i;
    L := 1; C := 0;
  end loop;
end Pi;

begin
  loop
    loop
      [Temp := L; L := C; C := Temp];
      exit when L = 0;
      ------ change process
    end loop;
    ------ critical_section_i;
    L := 1; C := 0;
  end loop;
end Pj;
```

Does that work?
Mutual exclusion: atomic exchange operation

type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
L : Flag := 1;
begin
loop
loop
[Temp := L; L := C; C := Temp];
exit when L = 0;
-------- change process
end loop;
-------- critical_section_i;
L := 1; C := 0;
end loop;
end Pi;

begin

loop
loop
[Temp := L; L := C; C := Temp];
exit when L = 0;
-------- change process
end loop;
-------- critical_section_i;
L := 1; C := 0;
end loop;
end Pi;

G Mutual exclusion!, No deadlock!, No global live-lock!
G Works for any dynamic number of processes.
G Individual starvation possible! Busy waiting loops!
**Mutual exclusion: memory cell reservation**

```vhdl
type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
L : Flag;
begin
  loop
    loop
      L :^= C; C :^= 1;
      exit when Untouched and L = 0;
      ------ change process
    end loop;
    ------ critical_section_i;
    C := 0;
  end loop;
end Pi;

begin
  loop
    loop
      L :^= C; C :^= 1;
      exit when Untouched and L = 0;
      ------ change process
    end loop;
    ------ critical_section_j;
    C := 0;
  end loop;
end Pj;
```

Does that work?
Mutual exclusion! No deadlock!, No global live-lock!

Works for any dynamic number of processes.

Individual starvation possible! Busy waiting loops!
Critical section

Any context switch needs to clear reservations

Asks for permission

Count: `.word 0x00000000
Lock: `.word 0x00000000 ; #0 means unlocked

for_enter:
  cmp  r1, #100
  bgt  end_for_enter

fail_lock_enter:
  ldrex r0, [r3]
  cmp  r0, #0
  bne  fail_lock_enter ; if locked
  mov  r0, #1          ; lock value
  strex r5, r0, [r3]    ; try lock
  cmp  r5, #0
  bne  fail_lock_enter ; if touched
  dmb ; sync memory

  ldr  r2, [r4]
  add  r2, #1
  str  r2, [r4]
  dmb ; sync memory
  mov  r0, #0          ; unlock value
  str  r0, [r3]        ; unlock
  add  r1, #1
  b    for_enter

end_for_enter:

for_leave:
  cmp  r1, #100
  bgt  end_for_leave

fail_lock_leave:
  ldrex r0, [r3]
  cmp  r0, #0
  bne  fail_lock_leave ; if locked
  mov  r0, #1          ; lock value
  strex r5, r0, [r3]    ; try lock
  cmp  r5, #0
  bne  fail_lock_leave ; if touched
  dmb ; sync memory

  ldr  r2, [r4]
  sub  r2, #1
  str  r2, [r4]
  dmb ; sync memory
  mov  r0, #0          ; unlock value
  str  r0, [r3]        ; unlock
  add  r1, #1
  b    for_leave

end_for_leave:
Mutual exclusion ... or the lack thereof

Count: .word 0x00000000

ldr r4, =Count
mov r1, #1
for_enter:
cmp r1, #100
bgt end_for_enter

enter_critical_fail:
ldrex r2, [r4] ; tag [r4] as exclusive
add r2, #1
strex r0, r2, [r4] ; only if untouched
cmp r0, #0
bne enter_critical_fail
add r1, #1
b for_enter

end_for_enter:

for_leave:
cmp r1, #100
bgt end_for_leave

leave_critical_fail:
ldrex r2, [r4] ; tag [r4] as exclusive
sub r2, #1
strex r0, r2, [r4] ; only if untouched
cmp r0, #0
bne leave_critical_fail
add r1, #1
b for_leave

end_for_leave:

What is the value at address Count after both programs complete?
Beyond atomic hardware operations

Semaphores

Basic definition (Dijkstra 1968)

Assuming the following three conditions on a shared memory cell between processes:

- a set of processes agree on a variable $S$ operating as a flag to indicate synchronization conditions

- an atomic operation $P$ on $S$ — for ‘passeren’ (Dutch for ‘pass’):

  $P(S): [\text{as soon as } S > 0 \text{ then } S := S - 1]$  
  \[\text{this is a potentially delaying operation}\]

- an atomic operation $V$ on $S$ — for ‘vrygeven’ (Dutch for ‘to release’):

  $V(S): [S := S + 1]$

\[\text{then the variable } S \text{ is called a } \text{Semaphore}.\]
Beyond atomic hardware operations

Semaphores

... as supplied by operating systems and runtime environments

• a set of processes $P_1 \ldots P_N$ agree on a variable $S$ operating as a flag to indicate synchronization conditions

• an atomic operation $\text{Wait}$ on $S$: (aka ‘Suspend_Until_True’, ‘sem_wait’, …)

  Process $P_i : \text{Wait} (S)$:
  \[
  \begin{cases}
  \text{if } S > 0 \text{ then } S := S - 1 \\
  \text{else suspend } P_i \text{ on } S
  \end{cases}
  \]

• an atomic operation $\text{Signal}$ on $S$: (aka ‘Set_True’, ‘sem_post’, …)

  Process $P_i : \text{Signal} (S)$:
  \[
  \begin{cases}
  \text{if } \exists P_j \text{ suspended on } S \text{ then release } P_j \\
  \text{else } S := S + 1
  \end{cases}
  \]

then the variable $S$ is called a Semaphore in a scheduling environment.
Beyond atomic hardware operations

Semaphores

Types of semaphores:

- **Binary semaphores**: restricted to [0, 1] or [False, True] resp. Multiple \( V \) (Signal) calls have the same effect than a single call.
  - Atomic hardware operations support binary semaphores.
  - Binary semaphores are sufficient to create all other semaphore forms.
- **General semaphores** (counting semaphores): non-negative number; (range limited by the system) \( P \) and \( V \) increment and decrement the semaphore by one.
- **Quantity semaphores**: The increment (and decrement) value for the semaphore is specified as a parameter with \( P \) and \( V \).

⚠️ All types of semaphores must be initialized:
  - often the number of processes which are allowed inside a critical section, i.e. ‘1’.
Semaphore: `.word 0x00000001`

```
ldr   r3, =Semaphore
...

wait (Semaphore)
...

Critical section
...

signal (Semaphore)
...
```

```
ldr   r3, =Semaphore
...

wait (Semaphore)
...

Critical section
...

signal (Semaphore)
...
```
Semaphore: .word 0x00000001

```assembly
ldr   r3, =Semaphore

wait_1:
  ldr   r0, [r3]
  cmp   r0, #0
  beq   wait_1  ; if Semaphore = 0
  sub   r0, #1  ; dec Semaphore
  str   r0, [r3]  ; update

wait_2:
  ldr   r0, [r3]
  cmp   r0, #0
  beq   wait_2  ; if Semaphore = 0
  sub   r0, #1  ; dec Semaphore
  str   r0, [r3]  ; update

signal (Semaphore)

Critical section

signal (Semaphore)
```

© 2021 Uwe R. Zimmer, The Australian National University
ldr r3, =Semaphore

wait_1:
   ldrex r0, [r3]
   cmp r0, #0  ; if Semaphore = 0
   beq wait_1  ; if Semaphore = 0
   sub r0, #1  ; dec Semaphore
   strex r1, r0, [r3]  ; try update
   cmp r1, #0  ; if touched
   bne wait_1  ; if touched
   dmb  ; sync memory

…  ; sync memory

Semaphore: .word 0x00000001

Any context switch needs to clear reservations

wait_2:
   ldrex r0, [r3]
   cmp r0, #0  ; if Semaphore = 0
   beq wait_2  ; if Semaphore = 0
   sub r0, #1  ; dec Semaphore
   strex r1, r0, [r3]  ; try update
   cmp r1, #0  ; if touched
   bne wait_2  ; if touched
   dmb  ; sync memory

…  ; sync memory

Critical section

signal (Semaphore)

…  ; sync memory

Critical section

…  ; sync memory

signal (Semaphore)
Semaphore: `.word 0x00000001`

```
ldr r3, =Semaphore
...
```

**wait_1:**
```
ldrex r0, [r3]
cmp r0, #0 ; if Semaphore = 0
beq wait_1 ; if Semaphore = 0
sub r0, #1 ; dec Semaphore
strex r1, r0, [r3] ; try update
cmp r1, #0 ; if touched
bne wait_1 ; if touched
dmb ; sync memory
```

```
ldr r0, [r3]
add r0, #1 ; inc Semaphore
str r0, [r3] ; update
```

```
wait_2:
ldrex r0, [r3]
cmp r0, #0 ; if Semaphore = 0
beq wait_2 ; if Semaphore = 0
sub r0, #1 ; dec Semaphore
strex r1, r0, [r3] ; try update
cmp r1, #0 ; if touched
bne wait_2 ; if touched
dmb ; sync memory
```

```
ldr r0, [r3]
add r0, #1 ; inc Semaphore
str r0, [r3] ; update
```

```
... Critical section ...
```

```
... Critical section ...
```

© 2021 Uwe R. Zimmer, The Australian National University
Semaphore: .word 0x00000001

wait_1:
  ldrx r0, [r3]
  cmp r0, #0 ; if Semaphore = 0
  sub r0, #1 ; dec Semaphore
  strex r1, r0, [r3] ; try update
  cmp r1, #0 ; if touched
  bne wait_1 ; sync memory

signal_1:
  ldrx r0, [r3]
  add r0, #1 ; inc Semaphore
  strex r1, r0, [r3] ; try update
  cmp r1, #0
  bne signal_1 ; if touched
  dmb ; sync memory

signal_2:
  ldrx r0, [r3]
  add r0, #1 ; inc Semaphore
  strex r1, r0, [r3] ; try update
  cmp r1, #0
  bne signal_2 ; if touched
  dmb ; sync memory

Any context switch needs to clear reservations.

Semaphore: .word 0x00000001

wait_2:
  ldrx r0, [r3]
  cmp r0, #0 ; if Semaphore = 0
  sub r0, #1 ; dec Semaphore
  strex r1, r0, [r3] ; try update
  cmp r1, #0
  bne wait_2 ; if touched
  dmb ; sync memory

signal_2:
  ldrx r0, [r3]
  add r0, #1 ; inc Semaphore
  strex r1, r0, [r3] ; try update
  cmp r1, #0
  bne signal_2 ; if touched
  dmb ; sync memory

Critical section

Critical section

Any context switch needs to clear reservations.
Semaphores

S : Semaphore := 1;

\begin{verbatim}
  task body Pi is
    begin
      loop
        ------ non_critical_section_i;
        wait (S);
        ------ critical_section_i;
        signal (S);
      end loop;
      end Pi;
  end

  task body Pj is
    begin
      loop
        ------ non_critical_section_j;
        wait (S);
        ------ critical_section_j;
        signal (S);
      end loop;
      end Pj;
  end
\end{verbatim}

Works?
Semaphores

S : Semaphore := 1;

\begin{align*}
&\text{task body } \text{Pi is} \\
&\begin{aligned}
&\text{begin} \\
&\text{loop} \\
&\text{------ non\_critical\_section\_i;} \\
&\text{wait (S);} \\
&\text{------ critical\_section\_i;} \\
&\text{signal (S);} \\
&\text{end loop;}
\end{aligned} \\
&\text{end Pi;}
\end{align*}

\begin{align*}
&\text{task body } \text{Pj is} \\
&\begin{aligned}
&\text{begin} \\
&\text{loop} \\
&\text{------ non\_critical\_section\_j;} \\
&\text{wait (S);} \\
&\text{------ critical\_section\_j;} \\
&\text{signal (S);} \\
&\text{end loop;}
\end{aligned} \\
&\text{end Pj;}
\end{align*}

Mutual exclusion!, No deadlock!, No global live-lock!

Works for any dynamic number of processes

Individual starvation possible!
S1, S2 : Semaphore := 1;

\[
\text{task body } \text{Pi is} \\
\text{begin} \\
\text{loop} \\
\quad \text{----- non_critical_section_i;} \\
\quad \text{wait (S1);} \\
\quad \text{wait (S2);} \\
\quad \text{----- critical_section_i;} \\
\quad \text{signal (S2);} \\
\quad \text{signal (S1);} \\
\quad \text{end loop;} \\
\text{end Pi;} \\
\]

\[
\text{task body } \text{Pj is} \\
\text{begin} \\
\text{loop} \\
\quad \text{----- non_critical_section_j;} \\
\quad \text{wait (S2);} \\
\quad \text{wait (S1);} \\
\quad \text{----- critical_section_j;} \\
\quad \text{signal (S1);} \\
\quad \text{signal (S2);} \\
\quad \text{end loop;} \\
\text{end Pi;} \\
\]

Works too?
Semaphores

S1, S2 : Semaphore := 1;

\begin{verbatim}
  task body Pi is
  begin
    loop
      ------ non_critical_section_i;
      wait (S1);
      wait (S2);
      ------ critical_section_i;
      signal (S2);
      signal (S1);
    end loop;
  end Pi;
\end{verbatim}

\begin{verbatim}
  task body Pj is
  begin
    loop
      ------ non_critical_section_j;
      wait (S2);
      wait (S1);
      ------ critical_section_j;
      signal (S1);
      signal (S2);
    end loop;
  end Pj;
\end{verbatim}

- Mutual exclusion!, No global live-lock!
- Works for any dynamic number of processes.
- Individual starvation possible!
- Deadlock possible!
Semaphores

S1, S2 : Semaphore := 1;

\[
\text{task body } \text{Pi is}
\begin{align*}
\text{begin} \\
\text{loop} \\
\quad \text{------ non\_critical\_section\_i;} \\
\quad \text{wait (S1);} \\
\quad \text{wait (S2);} \\
\quad \text{------ critical\_section\_i;} \\
\quad \text{signal (S2);} \\
\quad \text{signal (S1);} \\
\quad \text{end loop;} \\
\text{end Pi;}
\end{align*}
\]

\[
\text{task body } \text{Pj is}
\begin{align*}
\text{begin} \\
\text{loop} \\
\quad \text{------ non\_critical\_section\_j;} \\
\quad \text{wait (S2);} \\
\quad \text{wait (S1);} \\
\quad \text{------ critical\_section\_j;} \\
\quad \text{signal (S1);} \\
\quad \text{signal (S2);} \\
\quad \text{end loop;} \\
\text{end Pj;}
\end{align*}
\]

- Mutual exclusion!, No global live-lock!
- Works for any dynamic number of processes.
- Individual starvation possible!
- Deadlock possible!

Concurrent programming languages offer higher abstraction and safer synchronization mechanisms.
Asynchronism

Summary

Aynchronism

• Interrupts & Exceptions
  • Concept
  • Hardware/Software interaction
  • Recursive interrupts

• Concurrency & Synchronization
  • Race conditions
  • Synchronization
  • Passing data
Control Structures

Uwe R. Zimmer - The Australian National University
References for this chapter

[Patterson17]
David A. Patterson & John L. Hennessy
Computer Organization and Design – The Hardware/Software Interface
Chapter 2 “Instructions: Language of the Computer”
ARM edition, Morgan Kaufmann 2017
Essential control structures for all imperative programming languages are:

- **Conditionals:** \texttt{if}, \texttt{case}, \texttt{switch}, ...
- **Open Loops:** \texttt{while}, \texttt{repeat}, ...
- **Bound Loops:** \texttt{for}, \texttt{foreach}, \texttt{forall}, ...
- **Procedures and Functions** (already covered)

♫ How do we create those basic control structures in Assembly?

Functional programming languages are based on functions, but also on conditional expressions.

♫ How do those control structures in programming languages translate into Assembly?
Conditionals – IF-ELSE

if Register_1 = Register_2 then
    Register_3 := 1;
else
    Register_3 := 0;
end if;

if (register1 == register2) {
    register3 = 1;
} else {
    register3 = 0;
}

Register_3 := (if Register_1 = Register_2 then 1 else 0);

register_3 register_1 register_2 = case register_1 == register_2 of
    True  -> 1
    False -> 0

if register1 == register2:
    register3 = 1
else:
    register3 = 0
Conditionals – IF-ELSE

if Register_1 = Register_2 then
  Register_3 := 1;
else
  Register_3 := 0;
end if;

if (register1 == register2) {
  register3 = 1;
} else {
  register3 = 0;
}

Register_3 := (if Register_1 = Register_2 then 1 else 0);

register_3 register_1 register_2 = case register_1 == register_2 of
  True  -> 1
  False -> 0

if register1 == register2:
  register3 = 1
else:
  register3 = 0

1. an expression (if)
2. a boolean condition (if)
3. code for True (then)
4. code for False (else)

How do either of those look in assembly?
Conditionals – IF-ELSE

Assuming the values have already been transferred from memory into registers:

```
cmp    r1, r2 ; 1. Instructions to generate status flags
beq    then ; 2. Branch depending on the status flags
mov    r3, #0 ; 4. Instructions for the else branch
b      end_if

then:
  mov    r3, #1 ; 3. Instructions for the then branch
end_if:
```

It seems there are three distinguishable code sections and one status flag condition.

Can we form a general pattern for this?
Conditionals – IF-ELSE

Assuming the values have already been transferred from memory into registers:

```
.macro if condition_code condition then_code else_code
\condition_code
b\condition  then
\else_code
b  end_if

then:
  \then_code

end_if:
  .endm
```
Assuming the values have already been transferred from memory into registers:

```
.macro if condition_code condition then_code else_code
\condition_code
b\condition then
\else_code
b    end_if

then:
\then_code
end_if:
    .endm
```

We might need a lot of those, hence the labels need to be unique to each if-else block.
Conditionals – IF-ELSE

Assuming the values have already been transferred from memory into registers:

```assembly
.macro if condition_code condition then_code else_code
 \condition_code
 b\condition then\@
 \else_code
 b
end_if\@

then\@:
 \then_code

end_if\@:
 .endm
```
Conditionals – IF-ELSE

Assuming the values have already been transferred from memory into registers:

```assembly
.macro if condition_code condition then_code else_code
\condition_code
b\condition then\@
\else_code
b     end_if\@

then\@:
  \then_code

end_if\@:
  .endm
```

We can now write:

```assembly
if “cmp r1, r2”, eq, “mov r3, #1”, “mov r3, #0”
```

... in the general case (with lots of code in each part)
we could create macros for the individual sections as well, so we can e.g. write:

```assembly
if compare_r1_r2, eq, load_1_to_r3, load_0_to_r3
```
Conditionals – IF-ELSE

if Register_1 = Register_2 then
  Register_3 := 1;
else
  Register_3 := 0;
end if;

if (register1 == register2) {
  register3 = 1;
} else {
  register3 = 0;
}

Register_3 := (if Register_1 = Register_2 then 1 else 0);

register_3 register_1 register_2 = case register_1 == register_2 of
  True  -> 1
  False -> 0

if register1 == register2:
  register3 = 1
else:
  register3 = 0
Conditionals – IF-ELSE

if Register_1 = Register_2 then
    Register_3 := 1;
else
    Register_3 := 0;
end if;

if (register1 == register2) {
    register3 = 1;
} else {
    register3 = 0;
}

Register_3 := (if Register_1 = Register_2 then 1 else 0);

register_3 register_1 register_2 = case register_1 == register_2 of
    True  -> 1
    False -> 0

if register1 == register2:
    register3 = 1
else:
    register3 = 0

if “cmp r1, r2”, eq, “mov r3, #1”, “mov r3, #0”
Conditionals – IF-ELSE

if Register_1 = Register_2 then
    Register_3 := 1;
else
    Register_3 := 0;
end if;

if (register1 == register2) {
    register3 = 1;
} else {
    register3 = 0;
}

Register_3 := (if Register_1 = Register_2 then 1 else 0);

register_3 register_1 register_2 = case register_1 == register_2 of
    True  -> 1
    False -> 0

if register1 == register2:
    register3 = 1
else:
    register3 = 0

Computational complexity: $\Theta(1)$
Loops – FOR

for Register_1 in 1..100 loop
  Register_3 := Register_3 + Register_1;
end loop;

for (register1 = 1; register1 <= 100; register1++) {
  register3 += register1;
}

for register1 in range (1, 101):
  register3 += register1

for Register_1 := 1 to 100 do
  Register_3 := Register_3 + Register_1;
end do

do Register_1 = 1, 100
  Register_3 = Register_3 + Register_1
end do

for Register_1 in 1..100 do
  Register_3 += Register_1;

What are the components?
Loops – FOR

for Register_1 in 1..100 loop
    Register_3 := Register_3 + Register_1;
end loop;

for register1 = 1; register1 <= 100; register1++
    register3 += register1;

for register1 in range (1, 101):
    register3 += register1

for Register_1 := 1 to 100 do
    Register_3 := Register_3 + Register_1;
end do

do Register_1 = 1, 100
    Register_3 = Register_3 + Register_1
end do

for Register_1 in 1..100 do
    Register_3 += Register_1;
end for

1. an index
2. a start value
3. an end value
4. code inside loop

How do either of those look in assembly?
Loops – FOR

Assuming the values have already been transferred from memory into registers:

\[
\begin{align*}
\text{mov} & \quad r1, \#1 \quad ; \text{set index to start value} \\
\text{for:} & \\
\text{cmp} & \quad r1, \#100 \quad ; \text{check whether it went beyond its end value} \\
\text{bgt} & \quad \text{end} \_\text{for} \quad ; \text{if so, stop the loop} \\
\text{add} & \quad r3, r1 \quad ; \text{do the work} \\
\text{add} & \quad r1, \#1 \quad ; \text{increment the index} \\
\text{b} & \quad \text{for} \\
\text{end} \_\text{for}: & \\
\end{align*}
\]

We can find the index, the start and end values and the body code.

Can we form a general pattern for this?
Loops – FOR

Assuming the values have already been transferred from memory into registers:

```assembly
.macro for register, from, to, body
    mov   register, #\from
    for@:
        cmp   register, #\to
        bgt   end_for@
        \body
    add   register, #1
    b      for@
    end_for@:
    .endm
```
Loops – FOR

Assuming the values have already been transferred from memory into registers:

```
.macro for register, from, to, body
   mov   \register, #\from
for@:
   cmp   \register, #\to
   bgt   end_for@
\body
   add   \register, #1
   b     for@
end_for@:
   .endm
```

We can now write:

```
for r1, 1, 100 "add r3, r1"
```

... in the general case (with lots of code inside the loop or multiple loops):

```
for r1, 1, 100, loop_body
for r1, 1, 100, "for r2, 1, 100, loop_body"
```
Loops – FOR

for Register_1 in 1..100 loop
    Register_3 := Register_3 + Register_1;
end loop;

for (register1 = 1; register1 <= 100; register1++) {
    register3 += register1;
}

for register1 in range (1, 101):
    register3 += register1

for Register_1 := 1 to 100 do
    Register_3 := Register_3 + Register_1;
end do

do Register_1 = 1, 100
    Register_3 = Register_3 + Register_1
end do

for Register_1 in 1..100 do
    Register_3 += Register_1;

.macro for register, from, to, body
    mov \register, #\from
    for\@:
        cmp \register, #\to
        bgt end_for\@

    \body
    add \register, #1
    b for\@
end_for\@:
.endm
Loops – FOR

for Register_1 in 1..100 loop
    Register_3 := Register_3 + Register_1;
end loop;

for (register1 = 1; register1 <= 100; register1++) {
    register3 += register1;
}

for register1 in range (1, 101):
    register3 += register1

for Register_1 := 1 to 100 do
    Register_3 := Register_3 + Register_1;
end do

for r1, 1, 100 “add r3, r1”

for Register_1 in 1..100 do
    Register_3 += Register_1;
Loops – FOR

for Register_1 in 1..100 loop
    Register_3 := Register_3 + Register_1;
end loop;

for (register1 = 1; register1 <= 100; register1++) {
    register3 += register1;
}

for register1 in range (1, 101):
    register3 += register1

for Register_1 := 1 to 100 do
    Register_3 := Register_3 + Register_1;
end do

do Register_1 = 1, 100
    Register_3 = Register_3 + Register_1
end do

for Register_1 in 1..100 do
    Register_3 += Register_1;

Computational complexity: \( \Theta(n) \)
Loops – WHILE

while Register_1 < 100 loop
    Register_1 := Register_1 ** 2;
end loop;

while (register1 < 100) {
    register1 = register1 * register1;
}

while register1 < 100:
    register1 = register1 ** 2

while Register_1 < 100 do
    Register_1 := Register_1 ** 2;
end do

while Register_1 < 100 do
    Register_1 = Register_1 ** 2
enddo

while (Register_1 < 100) {
    Register_1 = Register_1 ** 2;
}
Loops – WHILE

```plaintext
while Register_1 < 100 loop
    Register_1 := Register_1 ** 2;
end loop;
```

```plaintext
while (register1 < 100) {
    register1 = register1 * register1;
}
```

```plaintext
while register1 < 100:
    register1 = register1 ** 2
```

```plaintext
while Register_1 < 100 do
    Register_1 := Register_1 ** 2;
end do
```

```plaintext
while Register_1 < 100 do
    Register_1 = Register_1 ** 2
enddo
```

```plaintext
while (Register_1 < 100) {
    Register_1 = Register_1 ** 2;
}
```

1. an expression (if)
2. a boolean condition (if)
3. code inside the loop
Loops – WHILE

b while_condition

while:
mul r1, r1 ; 3. Loop body

while_condition:
cmp r1, #100 ; 1. Instructions to generate status flags
blt while ; 2. Branch depending on the status flags

Can we form a general pattern for this?
Loops – WHILE

.macro  while  while_expression, while_condition, body
b   while_condition@

while@:
  \body

while_condition@:
  \while_expression
b\while_condition  while@
.endm

We can now write:

while “cmp  r1, #100”, lt, “mul  r1, r1”

... try to re-write our power functions from the previous chapter with the macros you have now.
Loops – WHILE

while Register_1 < 100 loop
    Register_1 := Register_1 ** 2;
end loop;

while (register1 < 100) {
    register1 = register1 * register1;
}

while register1 < 100:
    register1 = register1 ** 2

enddo

while Register_1 < 100 do
    Register_1 := Register_1 ** 2
enddo

while (Register_1 < 100) {
    Register_1 = Register_1 ** 2;
}

.macro while while_expression, while_condition, body
    while_condition@:
        while_expression
        body
    while_condition@:
        while_expression
        body
.endm
Loops – WHILE

while Register_1 < 100 loop
  Register_1 := Register_1 ** 2;
end loop;

while (register1 < 100) {
  register1 = register1 * register1;
}

while register1 < 100:
  register1 = register1 ** 2

while Register_1 < 100 do
  Register_1 := Register_1 ** 2;
enddo

while Register_1 < 100 do
  Register_1 = Register_1 ** 2
enddo

while (Register_1 < 100) {
  Register_1 = Register_1 ** 2;
}
Loops – WHILE

while Register_1 < 100 loop
  Register_1 := Register_1 ** 2;
end loop;

while (register1 < 100) {
  register1 = register1 * register1;
}

while register1 < 100:
  register1 = register1 ** 2

while Register_1 < 100 do
  Register_1 := Register_1 ** 2;
enddo

while (Register_1 < 100) {
  Register_1 = Register_1 ** 2;
}

Computational complexity: Undefined
Conditionals – CASE (indexed)

type Colour is (Red, Green, Blue);

These values can be represented by (which is also the default in most systems)

for Colour use (
    Red => 0,
    Green => 1,
    Blue => 2);

Assuming that Register_1 is associated with this type, we can then expect a highly efficient implementation of a case construct such as:

    case Register_1 is
        when Red => Register_2 := Register_3;
        when Green => Register_2 := Register_4;
        when Blue => Register_2 := Register_5;
    end case;
A table based branching implementation of:

```plaintext
case Register_1 is
  when Red  => Register_3 := Register_2;
  when Green => Register_4 := Register_2;
  when Blue  => Register_5 := Register_2;
end case;
```

would look like:
Control Structures

Conditionals – CASE (indexed)

tbh [PC, r1, lsl #1] ; PC used as base of branch table, r1 is index

branch_table:
    .hword   (case_red   - branch_table)/2 ; case_red   16 bit offset
    .hword   (case_green - branch_table)/2 ; case_green 16 bit offset
    .hword   (case_blue  - branch_table)/2 ; case_blue  16 bit offset

case_red:
    mov     r3, r2 ; Code for case Red
    b end_case

case_green:
    mov     r4, r2 ; Code for case Green
    b end_case

case_blue:
    mov     r5, r2 ; Code for case Blue

end_case:

The complexity of this operation is $\Theta(1)$, e.g. it is independent of the number of cases!
Can we generate this via a macro automatically in one line?, for instance as:

indexed_case r1, “mov r3, r2”, “mov r4, r2”, “mov r5, r2”
Conditionals – CASE (indexed)

.. yes, but as the number of cases is variable, we need to write this recursively:

```assembly
.macro indexed_case index case_body other_cases:vararg
indexed_case_id \@, \index, "\case_body", \other_cases ; add a unique id
.endm

.macro indexed_case_id id index case_body other_cases:vararg
  tbh [pc, \index, lsl #1]
  branch_table_\id:
    table_entry \id, i, "\case_body", \other_cases ; build up the table entries
    case_entry  \id, i, "\case_body", \other_cases ; add the codes with a label each
  indexed_case_end_\id:
    .endm
```

The parts which are actually producing code are highlighted.

... recursive parts are following on the next page ... hold on to something!
... yes, this is a bit more involved than the previous macros, yet it is here to demonstrate that more complex and dynamic structures can also be macro generated.
case Register_1 is
    when Red => Register_3 := Register_2;
    when Green => Register_4 := Register_2;
    when Blue => Register_5 := Register_2;
end case;

indexed_case r1, "mov r3, r2", "mov r4, r2", "mov r5, r2"
Conditionals – CASE (indexed)

case Register_1 is
  when Red   => Register_3 := Register_2;
  when Green => Register_4 := Register_2;
  when Blue  => Register_5 := Register_2;
end case;

Computational complexity: $\Theta(1)$

Side remark: if you disassemble such a structure, it will look different.  
How and why?

```plaintext
    tbh     [PC, r1, lsl #1]

branch_table:
  .hword (case_red   - branch_table)/2
  .hword (case_green - branch_table)/2
  .hword (case_blue  - branch_table)/2

case_red:
  mov     r3, r2
  b       end_case

case_green:
  mov     r4, r2
  b       end_case

case_Blue:
  mov     r5, r2
end_case:
```
Conditionals – CASE (guarded expressions, list of conditions)

\[
\begin{array}{l}
\text{r0 :: Int -> Int -> Int -> Int} \\
r0 \ r1 \ r2 \ r3 \\
| \ r1 < \ r2 \ = \ r1 \\
| \ r1 > \ r2 \ = \ r2 \\
| \ r1 == r2 \ = \ 0 \\
| \ \text{otherwise} = \text{error “How did I get here?”}
\end{array}
\]

\[
\begin{array}{l}
\text{switch (r1) {} } \\
\quad \text{case 4 : r0 = r1; } \\
\quad \quad \quad \text{break;} \\
\quad \text{case 5 : r0 = r2; } \\
\quad \quad \quad \text{break;} \\
\quad \text{case 6 : r0 = 0; } \\
\}\n\]

\[
\begin{array}{l}
\text{r0 := (if \ r1 < r2 then r1} \\
\quad \text{elsif r1 > r2 then r2} \\
\quad \text{elsif r1 = r2 then 0} \\
\quad \text{else Integer’Invalid);} \\
\end{array}
\]
Conditionals – CASE (guarded expressions, list of conditions)

\[
\begin{align*}
r0 & : \mathbb{Int} \to \mathbb{Int} \to \mathbb{Int} \to \mathbb{Int} \\
r0 \ r1 \ r2 \ r3 \\
| \ r1 < r2 & = r1 \\
| \ r1 > r2 & = r2 \\
| \ r1 == r2 & = 0 \\
| \ otherwise & = \text{error} \ "\text{How did I get here?}" \\
\end{align*}
\]

\[
\text{switch (r1) } \{ \\
\text{case 4 : } r0 = r1; \text{ break; } \\
\text{case 5 : } r0 = r2; \text{ break; } \\
\text{case 6 : } r0 = 0; \\
\} \\
\]

\[
r0 := (\text{if } \ r1 < r2 \ \text{then } r1 \\
\text{elsif } r1 > r2 \ \text{then } r2 \\
\text{elsif } r1 = r2 \ \text{then } 0 \\
\text{else \ } \text{Integer'Invalid});
\]

1. guards
2. guard conditions
3. guard expressions / statements
Conditionals – CASE (guarded expressions, list of conditions)

```
cmp   r1, r2
blt   case_a
cmp   r1, r2
bgt   case_b
cmp   r1, r2
beq   case_c
b     end_case

case_a:
   mov  r0, r1
   b    end_case

case_b:
   mov  r0, r2
   b    end_case

case_c:
   mov  r0, #0
   b    end_case

end_case:
```

1. guards
2. guard conditions
3. guard expressions / statements

Generated by:
```c
case "cmp r1, r2", lt, "mov r0, r1",
    "cmp r1, r2", gt, "mov r0, r2",
    "cmp r1, r2", gt, "mov r0, #0"
```
Control Structures

Conditionals – CASE (indexed)

This is again recursive to handle the variable number of cases:

```
.macro case expression condition case_body other_cases:vararg
  case_id \@, "\expression", \condition, "\case_body", \other_cases
.endm

.macro case_id id expression condition case_body other_cases:vararg
  guards_rec \id, i, "\expression", \condition, "\case_body", \other_cases
  cases_rec \id, i, "\expression", \condition, "\case_body", \other_cases
.endm

end_case_\id:
.endm
```

... The parts which are actually producing code are highlighted.

... and we still need to generate the list of guards, followed by the list of code sections.
Control Structures

Conditionals – CASE (indexed)

... 

.macro guards_rec id case_nr expression condition case_body other_cases:vararg
\expression
b\condition case_\id\()_\case_nr
.ifnb \other_cases
guards_rec \id, \case_nr\()i, \other_cases
.else
b end_case_\id
.endif
.endm

.macro cases_rec id case_nr expression condition case_body other_cases:vararg 
\case_\id\()_\case_nr:
\case_body
b end_case_\id
.ifnb \other_cases
cases_rec \id, \case_nr\()i, \other_cases
.endif
.endm

Keep in mind:
Macro programming is pure textual replacement.
The result is a text which is then translated by the assembler into machine code.
**Control Structures**

**Conditionals – CASE (guarded expressions, list of conditions)**

\[ r0 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]

\[
r0 \ r1 \ r2 \ r3
| \ r1 < \ r2 \ = \ r1
| \ r1 > \ r2 \ = \ r2
| \ r1 == \ r2 \ = \ 0
| \ otherwise = \ \text{error} \ \text{“How did I get here?”}
\]

```plaintext
switch (r1) {
  case 4 : r0 = r1;
         break;
  case 5 : r0 = r2;
         break;
  case 6 : r0 = 0;
}
```

```plaintext
r0 := (if    r1 < r2 then r1
       elsif r1 > r2 then r2
       elsif r1 = r2 then 0
       else Integer’Invalid);
```

```plaintext
case “cmp r1, r2”, lt, “mov r0, r1”,
     “cmp r1, r2”, gt, “mov r0, r2”,
     “cmp r1, r2”, gt, “mov r0, #0”
```
**Conditionals – CASE (guarded expressions, list of conditions)**

\[ r_0 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]

\[
\begin{align*}
r_0 & \; r_1 \; r_2 \; r_3 \\
| \; r_1 < \; r_2 & = r_1 \\
| \; r_1 > \; r_2 & = r_2 \\
| \; r_1 \; == \; r_2 & = 0 \\
| \; \text{otherwise} & = \text{error} \; \text{“How did I get here?”}
\end{align*}
\]

```
switch (r1) {
    case 4 : r0 = r1;
              break;
    case 5 : r0 = r2;
              break;
    case 6 : r0 = 0;
}
```

```
r0 := (if \; r1 < r2 \; then \; r1 \\
              elsif \; r1 > r2 \; then \; r2 \\
              elsif \; r1 \; == \; r2 \; then \; 0 \\
              else \text{Integer’Invalid});
```

**Computational complexity:** \( O(n) \)
Control Structures

if "cmp r1, r2", eq, "mov r3, #1", "mov r3, #0"

for r1, 1, 100 "add r3, r1"
   mov r1, #1
   for:
      cmp r1, #100
      bgt end_for
      add r3, r1
      add r1, #1
   b for
end_for:

while "cmp r1, #100", lt, "mul r1, r1"

b while_condition
while:
   mul r1, r1
while_condition:
   cmp r1, #100
   blt while

if "cmp r1, r2", eq, "mov r3, #1", "mov r3, #0"

for r1, 1, 100 "add r3, r1"
Control Structures

indexed_case r1, "mov r3, r2", "mov r4, r2", "mov r5, r2"

```
tbh [PC, r1, lsl #1]

branch_table:
  .hword (case_red  - branch_table)/2
  .hword (case_green - branch_table)/2
  .hword (case_blue  - branch_table)/2

case_red:
  mov r3, r2
  b end_case

case_green:
  mov r4, r2
  b end_case

case_blue:
  mov r5, r2
  b end_case

end_case:
```

```c

```
indexed_case r1, "mov r3, r2", "mov r4, r2", "mov r5, r2"

```
tbh    [PC, r1, lsl #1]
branch_table:
  .hword (case_red - branch_table)/2
  .hword (case_green - branch_table)/2
  .hword (case_blue - branch_table)/2

case_red:
  mov   r3, r2
  b     end_case

case_green:
  mov   r4, r2
  b     end_case

case_blue:
  mov   r5, r2

end_case:
```

```
switch r1,
  4, "mov r0, r1",
  5, "mov r0, r2",
  6, "mov r0, #0"
```

```
cmp   r1, #4
  beq  case_a

cmp   r1, #5
  beq  case_b

cmp   r1, #6
  beq  case_c
  b     end_case

case_a:
  mov   r0, r1
  b     end_case

case_b:
  mov   r0, r2
  b     end_case

case_c:
  mov   r0, #0
  b     end_case

end_case:
```
You can form all common sequential control structures
(or generate them via macros if you wish)

(including function calls)
Control Structures

Summary

Control Structures

• Assembler Macros
  • Local labels
  • Recursive macros

• Control Structures in machine code
  • IF
  • WHILE
  • FOR
  • CASEs
References for this chapter

[Patterson17]
David A. Patterson & John L. Hennessy
*Computer Organization and Design – The Hardware/Software Interface*
Chapter 4 “The Processor”,
Chapter 6 “Parallel Processors from Client to Cloud”
ARM edition, Morgan Kaufmann 2017
What is an operating system?
What is an operating system?

1. A virtual machine!

... offering a more comfortable and safer environment

(e.g. memory management and protection, hardware abstraction, process management, inter-process communication, ...)

© 2021 Uwe R. Zimmer, The Australian National University
What is an operating system?

1. A virtual machine!

... offering a more comfortable and safer environment
What is an operating system?

2. A resource manager!

... coordinating access to hardware resources
What is an operating system?

2. A resource manager!

... coordinating access to hardware resources

Operating systems deal with

- processors
- memory
- mass storage
- communication channels
- devices (timers, special purpose processors, peripheral hardware, ...)

and tasks/processes/programs which are applying for access to these resources!
The evolution of operating systems

• in the beginning: single user, single program, single task, serial processing - no OS

• 50s: System monitors / batch processing
  - the monitor ordered the sequence of jobs and triggered their sequential execution

• 50s-60s: Advanced system monitors / batch processing:
  - the monitor is handling interrupts and timers
  - first support for memory protection
  - first implementations of privileged instructions (accessible by the monitor only).

• early 60s: Multiprogramming systems:
  - employ the long device I/O delays for switches to other, runable programs

• early 60s: Multiprogramming, time-sharing systems:
  - assign time-slices to each program and switch regularly

• early 70s: Multitasking systems – multiple developments resulting in UNIX (besides others)

• early 80s: single user, single tasking systems, with emphasis on user interface or APIs.
  MS-DOS, CP/M, MacOS and others first employed ‘small scale’ CPUs (personal computers).

• mid-80s: Distributed/multiprocessor operating systems - modern UNIX systems (SYSV, BSD)
Types of current operating systems

Personal computing systems, workstations, and workgroup servers:

- late 70s: Workstations starting by porting UNIX or VMS to ‘smaller’ computers.
- 80s: PCs starting with almost none of the classical OS-features and services, but with a user-interface (MacOS) and simple device drivers (MS-DOS)

- last 20 years: evolving and expanding into current general purpose OSs, like for instance:
  - Solaris (based on SVR4, BSD, and SunOS)
  - LINUX (open source UNIX re-implementation for x86 processors and others)
  - current Windows (used to be partly based on Windows NT, which is ‘related’ to VMS)
  - MacOS (Mach kernel with BSD Unix and a proprietary user-interface)

- Multiprocessing is supported by all these OSs to some extent.
- None of these OSs are suitable for embedded systems, although trials have been performed.
- None of these OSs are suitable for distributed or real-time systems.
Types of current operating systems

Parallel operating systems

- support for a large number of processors, either:
  - symmetrical: each CPU has a full copy of the operating system
  - asymmetrical: only one CPU carries the full operating system, the others are operated by small operating system stubs to transfer code or tasks.
Types of current operating systems

Distributed operating systems

- all CPUs carry a small kernel operating system for communication services.
- all other OS-services are distributed over available CPUs
- services may migrate
- services can be multiplied in order to
  - guarantee availability (hot stand-by)
  - or to increase throughput (heavy duty servers)
Types of current operating systems

Real-time operating systems

- Fast context switches?
- Small size?
- Quick response to external interrupts?
- Multitasking?
- 'low level' programming interfaces?
- Interprocess communication tools?
- High processor utilization?
Types of current operating systems

Real-time operating systems

- Fast context switches? should be fast anyway
- Small size? should be small anyway
- Quick response to external interrupts? not ‘quick’, but predictable
- Multitasking? often, not always
- ‘low-level’ programming interfaces? needed in many operating systems
- Interprocess communication tools? needed in almost all operating systems
- High processor utilization? fault tolerance builds on redundancy!
Types of current operating systems

Real-time operating systems need to provide...
- the logical correctness of the results as well as
- the correctness of the time, when the results are delivered

Predictability!
(not performance!)

All results are to be delivered just-in-time – not too early, not too late.

Timing constraints are specified in many different ways ...
... often as a response to ‘external’ events
- reactive systems
Types of current operating systems

Embedded operating systems

- usually real-time systems, often hard real-time systems
- very small footprint (often a few kBytes)
- none or limited user-interaction

90-95% of all processors are working here!
Types of current operating systems

- Entertainment system
- Tail gate
- Black Box
- Window control
- Interior lights
- Mirror dimming
- Seat adjustments
- Key identification
- Cross traffic detection
- Hill start assist
- Traction control
- Blindspot detection
- Power regeneration
- Tire pressure sensors
- Night vision
- Image processing
- Speech recognition
- Dashboard
- Driver monitoring
- Engine/motor management
- Start-stop system
- Radar/Lidar sensing
- Adaptive cruise control
- Emergency services call
- Power management
- Automated parking
- Navigation system
- Lane holding
- Alarm system
- Blindspot detection
- Emergency brakes
- Adaptive dampers
- ESC
- ABS
- A/C
- Displays
- Seat heating
- Cylinder deactivation
- Transmission control
- Automated Lights
- Automated Wipers
- HUD
- Steering
- Start-stop system
- Mirror dimming
- Tail gate
- Entertainment system.
Types of current operating systems

Embedded operating systems

- usually real-time systems, often hard real-time systems
- very small footprint (often a few kBytes)
- none or limited user-interaction

90-95% of all processors are working here!

Often over 100 MPUs per car
(and some of them quite high performant)
What is an operating system?

Is there a standard set of features for operating systems?
What is an operating system?

Is there a standard set of features for operating systems?

no:

the term ‘operating system’ covers 4kB microkernels, as well as >1GB installations of desktop general purpose operating systems.
What is an operating system?

Is there a standard set of features for operating systems?

**no:**

the term ‘operating system’ covers 4kB microkernels,
as well as > 1GB installations of desktop general purpose operating systems.

Is there a minimal set of features?
What is an operating system?

Is there a standard set of features for operating systems?

no:

the term ‘operating system’ covers 4kB microkernels,
as well as > 1GB installations of desktop general purpose operating systems.

Is there a minimal set of features?

almost:

memory management, process management and inter-process communication/synchronisation
will be considered essential in most systems
What is an operating system?

Is there a standard set of features for operating systems?

no:
the term ‘operating system’ covers 4kB microkernels,
as well as > 1GB installations of desktop general purpose operating systems.

Is there a minimal set of features?

almost:
memory management, process management and inter-process communication/synchronisation
will be considered essential in most systems

Is there always an explicit operating system?
What is an operating system?

Is there a standard set of features for operating systems?

no:
the term ‘operating system’ covers 4kB microkernels, as well as > 1GB installations of desktop general purpose operating systems.

Is there a minimal set of features?

almost:
memory management, process management and inter-process communication/synchronisation will be considered essential in most systems

Is there always an explicit operating system?

no:
some languages and development systems operate with standalone runtime environments
Typical features of operating systems

Process management:

- Context switch
- Scheduling
- Book keeping (creation, states, cleanup)

context switch:

- needs to...
  - ‘remove’ one process from the CPU while preserving its state
  - choose another process (scheduling)
  - ‘insert’ the new process into the CPU, restoring the CPU state

Some CPUs have hardware support for context switching, otherwise:

- use interrupt mechanism
Typical features of operating systems

Memory management:

- Allocation / Deallocation
- Virtual memory: logical vs. physical addresses, segments, paging, swapping, etc.
- Memory protection (privilege levels, separate virtual memory segments, ...)
- Shared memory

Synchronisation / Inter-process communication

- semaphores, mutexes, cond. variables, channels, mailboxes, MPI, etc. (chapter 4)
  😡 tightly coupled to scheduling / task switching!

Hardware abstraction

- Device drivers
- API
- Protocols, file systems, networking, everything else...
Typical structures of operating systems

Monolithic
(or ‘the big mess...’)

- non-portable
- hard to maintain
- lacks reliability
- all services are in the kernel (on the same privilege level)

but: may reach high efficiency

e.g. most early UNIX systems,
MS-DOS (80s), Windows (all non-NT based versions)
MacOS (until version 9), and many others...
Typical structures of operating systems

Monolithic & Modular

- Modules can be platform independent
- Easier to maintain and to develop
- Reliability is increased
- all services are still in the kernel (on the same privilege level)

may reach high efficiency

e.g. current Linux versions
Monolithic & layered

- easily portable
- significantly easier to maintain
- crashing layers do not necessarily stop the whole OS
- possibly reduced efficiency through many interfaces
- rigorous implementation of the stacked virtual machine perspective on OSs

e.g. some current UNIX implementations (e.g. Solaris) to a certain degree, many research OSs (e.g. ‘THE system’, Dijkstra ‘68)
µKernels & virtual machines

- µkernel implements essential process, memory, and message handling
- all ‘higher’ services are dealt with outside the kernel → no threat for the kernel stability
- significantly easier to maintain
- multiple OSs can be executed at the same time
- µkernel is highly hardware dependent → only the µkernel needs to be ported.
- possibly reduced efficiency through increased communications

  e.g. wide spread concept: as early as the CP/M, VM/370 (’79) or as recent as MacOS X (mach kernel + BSD unix), ...
Typical structures of operating systems

μKernels & client-server models

- μkernel implements essential process, memory, and message handling
- all ‘higher’ services are user level servers
- significantly easier to maintain
- kernel ensures reliable message passing between clients and servers
- highly modular and flexible
- servers can be redundant and easily replaced
- possibly reduced efficiency through increased communications

e.g. current research projects, L4, etc.
Typical structures of operating systems

µKernels & client-server models

- µkernel implements essential process, memory, and message handling
- all ‘higher’ services are user level servers
- significantly easier to maintain
- kernel ensures reliable message passing between clients and servers: locally and through a network
- highly modular and flexible
- servers can be redundant and easily replaced
- possibly reduced efficiency through increased communications

e.g. Java engines, distributed real-time operating systems, current distributed OSs research projects
UNIX features

- Hierarchical file-system (maintained via ‘mount’ and ‘unmount’)
- Universal file-interface applied to files, devices (I/O), as well as IPC
- Dynamic process creation via duplication
- Choice of shells
- Internal structure as well as all APIs are based on ‘C’
- Relatively high degree of portability

UNICS, UNIX, BSD, XENIX, System V, QNX, IRIX, SunOS, Ultrix, Sinix, Mach, Plan 9, NeXTSTEP, AIX, HP-UX, Solaris, NetBSD, FreeBSD, Linux, OPENSTEP, OpenBSD, Darwin, QNX/Neutrino, OS X, QNX RTOS, ... ...
Introduction to processes and threads

**1 CPU per control-flow**

Specific configurations only, e.g.:

- Distributed μcontrollers.
- Physical process control systems: 1 cpu per task, connected via a bus-system.

Process management (scheduling) not required.

Shared memory access need to be coordinated.
Introduction to processes and threads

1 CPU for all control-flows

- OS: emulate one CPU for every control-flow:
  - Multi-tasking operating system
- Support for memory protection essential.
- Process management (scheduling) required.
- Shared memory access need to be coordinated.
Processes

Process ::= Address space + Control flow(s)

Kernel has full knowledge about all processes as well as their states, requirements and currently held resources.
**Threads**

Threads (individual control-flows) can be handled:

- **Inside** the OS:
  - Kernel scheduling.
  - Thread can easily be connected to external events (I/O).

- **Outside** the OS:
  - User-level scheduling.
  - Threads may need to go through their parent process to access I/O.
Symmetric Multiprocessing (SMP)

All CPUs share the same physical address space (and access to resources).

Any process / thread can be executed on any available CPU.
Introduction to processes and threads

Processes ↔ Threads

Also processes can share memory and the specific definition of threads is different in different operating systems and contexts:

- Threads can be regarded as a group of processes, which share some resources (process-hierarchy).
- Due to the overlap in resources, the attributes attached to threads are less than for ‘first-class-citizen-processes’.
- Thread switching and inter-thread communication can be more efficient than switching on process level.
- Scheduling of threads depends on the actual thread implementations:
  - e.g. user-level control-flows, which the kernel has no knowledge about at all.
  - e.g. kernel-level control-flows, which are handled as processes with some restrictions.
Introduction to processes and threads

Process Control Blocks

- Process Id
- Process state:
  {created, ready, executing, blocked, suspended, bored …}
- Scheduling attributes:
  Priorities, deadlines, consumed CPU-time, …
- CPU state: Saved/restored information while context switches (incl. the program counter, stack pointer, …)
- Memory attributes / privileges:
  Memory base, limits, shared areas, …
- Allocated resources / privileges:
  Open and requested devices and files, …

… PCBs (links thereof) are commonly enqueued at a certain state or condition (awaiting access or change in state)
Process states

- **created**: the task is ready to run, but not yet considered by any dispatcher
  - waiting for admission
- **ready**: ready to run
  - waiting for a free CPU
- **running**: holds a CPU and executes
- **blocked**: not ready to run
  - waiting for a resource
Process states

- **created**: the task is ready to run, but not yet considered by any dispatcher waiting for admission
- **ready**: ready to run waiting for a free CPU
- **running**: holds a CPU and executes
- **blocked**: not ready to run waiting for a resource
- **suspended** states: swapped out of main memory (none time critical processes) waiting for main memory space (and other resources)
Process states

- **created**: the task is ready to run, but not yet considered by any dispatcher
  - waiting for admission
- **ready**: ready to run
  - waiting for a free CPU
- **running**: holds a CPU and executes
- **blocked**: not ready to run
  - waiting for a resource
- **suspended** states: swapped out of main memory
  - (none time critical processes)
  - waiting for main memory space (and other resources)

派遣和暂停现在可以是独立的模块
Process states

creation → batch → creation

CPU

pre-emption or cycle done

executing

termination

dispatch

unblock

suspension (swap-out)

unblock

suspension (swap-out)

unblock

suspension (swap-out)

unblock

suspension (swap-out)

unblock

block or synchronize

CPU

creation → batch → creation

pre-emption or cycle done

executing

termination

dispatch

unblock

suspension (swap-out)

unblock

suspension (swap-out)

unblock

suspension (swap-out)

unblock

block or synchronize

CPU
Definition of terms

Time scales of scheduling

Long-term

creation

batch

admit

ready, suspended

swap-in

blocked, suspended

swap-out

blocked

Short-term

pre-emption or cycle done

dispatch

executing

CPU

suspend (swap-out)

unblock

Medium-term

terminate.
Performance scheduling

Requested resource times

Tasks have an average time between instantiations of $T_i$
and a constant computation time of $C_i$
Performance scheduling

First come, first served (FCFS)

As tasks apply *concurrently* for resources, the actual sequence of arrival is non-deterministic. Hence even a deterministic scheduling schema like FCFS can lead to different outcomes.

Waiting time: 0..11, average: 5.9 – Turnaround time: 3..12, average: 8.4
Performance scheduling

First come, first served (FCFS)

<table>
<thead>
<tr>
<th>Job</th>
<th>Arrival Time</th>
<th>Burst Time</th>
<th>Waiting Time</th>
<th>Turnaround Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>12</td>
<td>5</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>16</td>
<td>8</td>
<td>24</td>
</tr>
</tbody>
</table>

**Waiting time:** 0..11, average: 5.4  
**Turnaround time:** 3..12, average: 8.0

In this example:
- the average waiting times vary between 5.4 and 5.9
- the average turnaround times vary between 8.0 and 8.4

Shortest possible maximal turnaround time!
Performance scheduling

Round Robin (RR)

Waiting time: 0..5, average: 1.2 – Turnaround time: 1..20, average: 5.8

- Optimized for swift initial responses.
- “Stretches out” long tasks.
- Bound maximal waiting time! (depended only on the number of tasks)
Summary

Operating Systems

- Operating Systems
  - Concept
  - Categories
  - Architectures

- Processes
  - Definition
  - Relation to architectures
  - Scheduling
Networks

Uwe R. Zimmer - The Australian National University
References for this chapter

[Patterson17]
David A. Patterson & John L. Hennessy
Computer Organization and Design – The Hardware/Software Interface
Chapter 4 “The Processor”,
Chapter 6 “Parallel Processors from Client to Cloud”
ARM edition, Morgan Kaufmann 2017
Network protocols & standards

**OSI network reference model**

Standardized as the *Open Systems Interconnection* (OSI) reference model by the International Standardization Organization (ISO) in 1977

- 7 layer architecture
- Connection oriented

Hardy implemented anywhere in full ...

...but its concepts and terminology are widely used, when describing existing and designing new protocols ...
Network protocols & standards

OSI Network Layers

User data
1: Physical Layer

- **Service**: Transmission of a raw bit stream over a communication channel

- **Functions**: Conversion of bits into electrical or optical signals

- **Examples**: X.21, Ethernet (cable, detectors & amplifiers)
2: Data Link Layer

- **Service**: Reliable transfer of frames over a link
- **Functions**: Synchronization, error correction, flow control
- **Examples**: HDLC (high level data link control protocol), LAP-B (link access procedure, balanced), LAP-D (link access procedure, D-channel), LLC (link level control), …
3: Network Layer

- **Service**: Transfer of packets inside the network
- **Functions**: Routing, addressing, switching, congestion control
- **Examples**: IP, X.25
4: Transport Layer

- **Service**: Transfer of data between hosts
- **Functions**: Connection establishment, management, termination, flow-control, multiplexing, error detection
- **Examples**: TCP, UDP, ISO TP0-TP4
5: Session Layer

- **Service**: Coordination of the dialogue between application programs
- **Functions**: Session establishment, management, termination
- **Examples**: RPC
6: Presentation Layer

- **Service**: Provision of platform independent coding and encryption
- **Functions**: Code conversion, encryption, virtual devices
- **Examples**: ISO code conversion, PGP encryption
7: Application Layer

- **Service:** Network access for application programs
- **Functions:** Application/OS specific
- **Examples:** APIs for mail, ftp, ssh, scp, discovery protocols …
## Network protocols & standards

### OSI
- **Application**
- **Presentation**
- **Session**
- **Transport**
- **Network**
- **Data link**
- **Physical**

### TCP/IP
- **Application**
- **Transport**
- **Network**
  - **IP**
- **Data link**
- **Physical**

### AppleTalk

<table>
<thead>
<tr>
<th>Protocol Type</th>
<th>Protocol Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>AppleTalk Filing Protocol (AFP)</td>
</tr>
<tr>
<td>AT Data Stream Protocol</td>
<td>AT Data Stream Protocol</td>
</tr>
<tr>
<td>AT Session Protocol</td>
<td>AT Session Protocol</td>
</tr>
<tr>
<td>Zone Info Protocol</td>
<td>Zone Info Protocol</td>
</tr>
<tr>
<td>Printer Access Protocol</td>
<td>Printer Access Protocol</td>
</tr>
<tr>
<td>Routing Table Protocol</td>
<td>Routing Table Protocol</td>
</tr>
<tr>
<td>AT Update Based Routing Protocol</td>
<td>AT Update Based Routing Protocol</td>
</tr>
<tr>
<td>Name Binding Prot.</td>
<td>Name Binding Prot.</td>
</tr>
<tr>
<td>AT Transaction Protocol</td>
<td>AT Transaction Protocol</td>
</tr>
<tr>
<td>AT Echo Protocol</td>
<td>AT Echo Protocol</td>
</tr>
<tr>
<td>Datagram Delivery Protocol</td>
<td>Datagram Delivery Protocol</td>
</tr>
<tr>
<td>AppleTalk Address Resolution Protocol</td>
<td>AppleTalk Address Resolution Protocol</td>
</tr>
<tr>
<td>EtherTalk Link Access Protocol</td>
<td>EtherTalk Link Access Protocol</td>
</tr>
<tr>
<td>LocalTalk Link Access Protocol</td>
<td>LocalTalk Link Access Protocol</td>
</tr>
<tr>
<td>TokenTalk Link Access Protocol</td>
<td>TokenTalk Link Access Protocol</td>
</tr>
<tr>
<td>FDDITalk Link Access Protocol</td>
<td>FDDITalk Link Access Protocol</td>
</tr>
<tr>
<td>IEEE 802.3</td>
<td>IEEE 802.3</td>
</tr>
<tr>
<td>LocalTalk</td>
<td>LocalTalk</td>
</tr>
<tr>
<td>Token Ring</td>
<td>Token Ring</td>
</tr>
<tr>
<td>IEEE 802.5</td>
<td>IEEE 802.5</td>
</tr>
<tr>
<td>FDDI</td>
<td>FDDI</td>
</tr>
</tbody>
</table>
Network protocols & standards

OSI

AppleTalk over IP

<table>
<thead>
<tr>
<th>Application</th>
<th>Presentation</th>
<th>Session</th>
<th>Transport</th>
<th>Network</th>
<th>Data link</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IP

Network

Physical

AppleTalk Filing Protocol (AFP)

- AT Data Stream Protocol
- AT Session Protocol
- Zone Info Protocol
- Printer Access Protocol

Routing Table Maintenance Prot.

- AT Update Based Routing Protocol
- Name Binding Protocol
- AT Transaction Protocol
- AT Echo Protocol

Datagram Delivery Protocol (DDP)

AppleTalk Address Resolution Protocol (AARP)

- EtherTalk Link Access Protocol
- LocalTalk Link Access Protocol
- TokenTalk Link Access Protocol
- FDDITalk Link Access Protocol

- IEEE 802.3
- LocalTalk
- Token Ring IEEE 802.5
- FDDI
Serial Peripheral Interface (SPI)

- Used by gazillions of devices ... and it’s not even a formal standard!
- Speed only limited by what both sides can survive.
- Usually push-pull drivers, i.e. fast and reliable, yet not friendly to wrong wiring/programming.

1.8" COLOR TFT LCD display from Adafruit

SanDisk marketing photo
Serial Peripheral Interface (SPI)

Full Duplex, 4-wire, flexible clock rate

- MISO (Master Input Slave Output) connected to MISO (Slave Input Master Output)
- MOSI (Master Output Slave Input) connected to MOSI (Slave Output Master Input)
- SCK (Serial Clock) connected to SCK (Serial Clock)
- NSS (NSS) connected to CS (Chip Select)

**Master**
- Receive shift register
- Transmit shift register
- Clock generator
- Slave selector

**Slave**
- Transmit shift register
- Receive shift register
Network protocols & standards

Serial Peripheral Interface (SPI)

Clock phase and polarity need to be agreed upon.
Network protocols & standards (SPI)

- **Master**
  - Receive shift register
  - Transmit shift register
  - Clock generator
  - Slave selector

- **Slave**
  - Transmit shift register
  - Receive shift register

- **Address and data bus**
  - MOSI
  - MISO
  - SCK
  - NSS

- **Communication controller**
  - CRC controller
  - RXONLY
  - CPOL
  - CPHA
  - DS[0:3]
  - RXCRC
  - RXCRCNEXT
  - RXCRCCL
  - BIDIOE
  - BR[2:0]
  - Internal NSS
  - NSS logic

- **Shift register**
  - Rx FIFO
  - Tx FIFO

- **Read**
  - Write

- 1 shift register?
- FIFOs?
- CRC?
- Data connected to an internal bus?
- DMA?
- Speed?

from STM32L4x6 advanced ARM®-based 32-bit MCUs reference manual: Figure 420 on page 1291

© 2021 Uwe R. Zimmer, The Australian National University
Network protocols & standards (SPI)

Master
- Receive shift register
- Transmit shift register
- Clock generator
- Slave selector

Slave 1
- Receive shift register
- Transmit shift register
- Clock generator
- Slave selector

Slave 2
- Receive shift register
- Transmit shift register
- Clock generator

Slave 3
- Receive shift register
- Transmit shift register

Full duplex with 1 out of x slaves
Network protocols & standards (SPI)

Master
- Receive shift register
- Transmit shift register
- Clock generator
- Slave selector

Slave 1
- Transmit shift register
- Receive shift register
- SCK

Slave 2
- Transmit shift register
- Receive shift register
- SCK

Slave 3
- Transmit shift register
- Receive shift register
- SCK

Master
- MISO
- MOSI
- SCK
- NSS
- CS

Slave
- MISO
- MOSI
- SCK
- NSS
- CS

Concurrent simplex with \( y \) out of \( x \) slaves
Network protocols & standards (SPI)

Master
- Receive shift register
- Transmit shift register
- Clock generator
- Slave selector

Slave 1
- Receive shift register
- Transmit shift register
- Clock generator
- Slave selector

Slave 2
- Receive shift register
- Transmit shift register

Slave 3
- Receive shift register
- Transmit shift register

Master
- MISO
- MOSI
- SCK
- NSS
- CS

Slave
- MISO
- MOSI
- SCK
- NSS
- CS

Concurrent daisy chaining with all slaves
Network protocols & standards

Ethernet / IEEE 802.3

Local area network (LAN) developed by Xerox in the 70’s

- 10 Mbps specification 1.0 by DEC, Intel, & Xerox in 1980.
- First standard as IEEE 802.3 in 1983 (10 Mbps over thick co-ax cables).
- Currently 1 Gbps (802.3ab) copper cable ports used in most desktops and laptops.
- Currently standards up to 100 Gbps (IEEE 802.3ba 2010).
- More than 85% of current LAN lines worldwide (according to the International Data Corporation (IDC)).

Carrier Sense Multiple Access with Collision Detection (CSMA/CD)
Network protocols & standards

**Ethernet / IEEE 802.3**

OSI relation: PHY, MAC, MAC-client

---

<table>
<thead>
<tr>
<th>OSI Network Layers</th>
<th>OSI reference model</th>
<th>IEEE 802.3 reference model</th>
</tr>
</thead>
<tbody>
<tr>
<td>User data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application</td>
<td>Application</td>
<td>MAC-client</td>
</tr>
<tr>
<td>Presentation</td>
<td>Presentation</td>
<td>Media Access (MAC)</td>
</tr>
<tr>
<td>Session</td>
<td>Session</td>
<td>Physical (PHY)</td>
</tr>
<tr>
<td>Transport</td>
<td>Transport</td>
<td></td>
</tr>
<tr>
<td>Network</td>
<td>Network</td>
<td>IEEE 802-specific</td>
</tr>
<tr>
<td>Data link</td>
<td>Data link</td>
<td></td>
</tr>
<tr>
<td>Physical</td>
<td>Physical</td>
<td>Media-specific</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presentation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Session</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transport</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data link</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Physical</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

© 2021 Uwe R. Zimmer, The Australian National University
Network protocols & standards

Ethernet / IEEE 802.3

OSI relation: PHY, MAC, MAC-client
Network protocols & standards

Ethernet / IEEE 802.11

Wireless local area network (WLAN) developed in the 90's

- First standard as IEEE 802.11 in 1997 (1-2 Mbps over 2.4 GHz).
- Typical usage at 54 Mbps over 2.4 GHz carrier at 20 MHz bandwidth.
- Current standards up to 780 Mbps (802.11ac) over 5 GHz carrier at 160 MHz bandwidth.
- Future standards are designed for up to 100 Gbps over 60 GHz carrier.
- Direct relation to IEEE 802.3 and similar OSI layer association.

- **Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA)**
- **Direct-Sequence Spread Spectrum (DSSS)**
Network protocols & standards

Bluetooth

Wireless local area network (WLAN) developed in the 90's with different features than 802.11:

- Lower power consumption.
- Shorter ranges.
- Lower data rates (typically < 1 Mbps).
- Ad-hoc networking (no infrastructure required).

Combinations of 802.11 and Bluetooth OSI layers are possible to achieve the required features set.
Network protocols & standards

Token Ring / IEEE 802.5 / Fibre Distributed Data Interface (FDDI)

- "Token Ring" developed by IBM in the 70's
- IEEE 802.5 standard is modelled after the IBM Token Ring architecture (specifications are slightly different, but basically compatible)
- IBM Token Ring requests are star topology as well as twisted pair cables, while IEEE 802.5 is unspecified in topology and medium
- Fibre Distributed Data Interface combines a token ring architecture with a dual-ring, fibre-optical, physical network.

Unlike CSMA/CD, Token ring is deterministic (with respect to its timing behaviour)

FDDI is deterministic and failure resistant

None of the above is currently used in performance oriented applications.
Network protocols & standards

Fibre Channel

- Developed in the late 80’s.
- ANSI standard since 1994.
- Current standards allow for 16 Gbps per link.

- Allows for three different topologies:
  - **Point-to-point**: 2 addresses
  - **Arbitrated loop** (similar to token ring): 127 addresses deterministic, real-time capable
  - **Switched fabric**: $2^{24}$ addresses, many topologies and concurrent data links possible

- Defines OSI equivalent layers up to the session level.

- Mostly used in storage arrays, but applicable to super-computers and high integrity systems as well.
Network protocols & standards

Fibre Channel

Mapping of Fibre Channel to OSI layers:
Summary

Networks

- Network layer models
  - Open Systems Interconnection (OSI) reference model
- Practical network standards
  - Serial Peripheral Interface (SPI)
  - Ethernet / IEEE 802.3 (CSMA/CD)
  - Tokenring / IEEE 802.5 / FDDI
  - Wireless networks / IEEE 802.11 (CSMA/CA, DSSS)
  - Fibre Channel
Architecture

Uwe R. Zimmer - The Australian National University
References for this chapter

[Patterson17]
David A. Patterson & John L. Hennessy
Computer Organization and Design – The Hardware/Software Interface
Chapter 4 “The Processor”,
Chapter 6 “Parallel Processors from Client to Cloud”
ARM edition, Morgan Kaufmann 2017
Definition: Processor

Hardware origins

18th century machines

L’Ecrivain

1770

Programmable, yet not a computer in today’s definition (not Turing complete)
Definition: Processor

Digital Computers

Hardware origins

- **Patents** by Konrad Zuse (Germany), 1936.
- **First digital computer**: Z1 (Germany), 1937: Relays, programmable via punch tape, clock: 1 Hz, 64 words memory à 22-bit, 2 registers, floating point unit, weight: 1 t.
- **First freely programmable (Turing complete) relays computer**: Z3 (Germany), 1941: 5.3 Hz
- **Atanasoff Berry Computer** (US) 1942: Vacuum tubes, (not Turing complete).
- **Colossus Mark 1** (UK) 1944: Vacuum tubes (not Turing complete).
- **“First Draft of a Report on the EDVAC”** (Electronic Discrete Variable Automatic Computer) by John von Neumann (US), 1945: Influential article about core elements of a computer: **Arithmetic unit**, **control unit** (Sequencer), **memory** (holding data and program), and **I/O**.
- **First high level programming language**: Plankalkül (“Plan Calculus”) by Konrad Zuse, 1945.
- **ENIAC** (Electronic Numerical Integrator And Computer) (US) 1946: Programed by plugboard, **First Turing complete vacuum tubes based computer**, clock: 100 kHz, weight: 27 t on 167 m².
Harvard Architecture

- **Control unit**
  - Concurrently addresses program and data memory and fetches next instruction.
  - Controls next ALU operations and determines the next instruction (based on ALU status).

- **Arithmetic Logic Unit (ALU)**
  - Fetches data from memory.
  - Executes arithmetic/logic operation.
  - Writes data to memory.

- **Input/Output**
- **Program memory**
- **Data memory**
von Neumann Architecture

- Control unit
  - Sequentially addresses program and data memory and fetches next instruction.
  - Controls next ALU operations and determines the next instruction (based on ALU status).

- Arithmetic Logic Unit (ALU)
  - Fetches data from memory.
  - Executes arithmetic/logic operation.
  - Writes data to memory.

- Input/Output

- Memory
  - Program and data is not distinguished
  - Programs can change themselves.
A simple processor (CPU)

- **Decoder/Sequencer**
  Can be a machine in itself which breaks CPU instructions into *concurrent* micro code.

- **Execution Unit / Arithmetic-Logic-Unit (ALU)**
  A collection of transformational logic.

- **Memory**

- **Registers**
  Instruction pointer, stack pointer, general purpose and specialized registers.

- **Flags**
  Indicating the states of the latest calculations.

- **Code/Data management**
  Fetching, Caching, Storing.
Some CPU actions are naturally sequential (e.g. instructions need to be first loaded, then decoded before they can be executed).

More fine grained sequences can be introduced by breaking CPU instructions into micro code.

- Overlapping those sequences in time will lead to the concept of pipelines.
- Same latency, yet higher throughput.
- (Conditional) branches might break the pipelines
- Branch predictors become essential.
Parallel pipelines

Filling parallel pipelines (by alternating incoming commands between pipelines) may employ multiple ALU’s.

- (Conditional) branches might again break the pipelines.
- Interdependencies might limit the degree of concurrency.
- Same latency, yet even higher throughput.
- Compilers need to be aware of the options.
**Pipeline hazards**

**Structural hazard**
- Lack of **hardware** to run operations in parallel,
  - e.g. load a new instruction and load new data in parallel.

**Control hazard**
- A **decision** depends on the previous instruction.
  - e.g. a conditional branch based on the flags from the previous instruction.

**Data hazard**
- Needed **data** is not yet available
  - e.g. the result of an arithmetic operation is needed in the next instruction.
Out of order execution

Breaking the sequence inside each pipeline leads to ‘out of order’ CPU designs.

- Replace pipelines with hardware scheduler.
- Results need to be “re-sequentialized” or possibly discarded.
- “Conditional branch prediction” executes the most likely branch or multiple branches.
- Works better if the presented code sequence has more independent instructions and fewer conditional branches.
- This hardware will require (extensive) code optimization to be fully utilized.
SIMD ALU units

Provides the facility to apply the same instruction to multiple data concurrently. Also referred to as “vector units”.

Examples: Altivec, MMX, SSE[2|3|4], …

requires specialized compilers or programming languages with implicit concurrency.

GPU processing

Graphics processor as a vector unit. Unifying architecture languages are used (OpenCL, CUDA, GPGPU).
Hyper-threading

Emulates multiple virtual CPU cores by means of replication of:

- Register sets
- Sequencer
- Flags
- Interrupt logic

while keeping the “expensive” resources like the ALU central yet accessible by multiple hyper-threads concurrently.

Requires programming languages with implicit or explicit concurrency.

Examples: Intel Pentium 4, Core i5/i7, Xeon, Atom, Sun UltraSPARC T2 (8 threads per core)
Processor Architectures

Multi-core CPUs

Full replication of multiple CPU cores on the same chip package.

- Often combined with hyper-threading and/or multiple other means (as introduced above) on each core.
- Cleanest and most explicit implementation of concurrency on the CPU level.

- Requires synchronized atomic operations.
- Requires programming languages with implicit or explicit concurrency.

Historically the introduction of multi-core CPUs ended the “GHz race” in the early 2000’s.
Virtual memory

Translates logical memory addresses into physical memory addresses and provides memory protection features.

- Does not introduce concurrency by itself.
- Is still essential for concurrent programming as hardware memory protection guarantees memory integrity for individual processes / threads.
Alternative Processor Architectures: Parallax Propeller

- Low cost 32 bit processor ($8)
- 8 cores with 2 kB local memory
- 40 kB shared memory
- No interrupts!
- 8 semaphores

© 2021 Uwe R. Zimmer, The Australian National University
Alternative Processor Architectures: IBM Cell processor (2001)

- 8 cores for specialized high-bandwidth floating point operations and 128 bit registers
- Theoretical 25.6 GFLOPS at 3.2 GHz
- Multiple interconnect topologies
- 64 bit PowerPC core
Multi-CPU systems

Scaling up:

• Multi-CPU on the same memory
  multiple CPUs on same motherboard and memory bus, e.g. servers, workstations

• Multi-CPU with high-speed interconnects
  various supercomputer architectures, e.g. Cray XE6:
  • 12-core AMD Opteron, up to 192 per cabinet (2304 cores)
  • 3D torus interconnect (160 GB/sec capacity, 48 ports per node)

• Cluster computer (Multi-CPU over network)
  multiple computers connected by network interface,
  e.g. Sun Constellation Cluster at ANU:
  • 1492 nodes, each: 2x Quad core Intel Nehalem, 24 GB RAM
  • QDR Infiniband network, 2.6 GB/sec
Summary

Architecture

• History

• Architectures
  • Pipelines
  • Parallel pipelines
  • Out of order execution
  • Vector machines
  • Multi-core CPUs
  • Virtual memory
Summary

Uwe R. Zimmer - The Australian National University
Exam preparations

Helpful

- **Distinguish** central aspects from excursions, examples & implementations.
- **Gain** full understanding of all central aspects.
- Be able to **categorize** any given example under a general theme discussed in the lecture.
- **Explain** to and **discuss** the topics with other (preferably better) students.
- Try whether you can **connect** aspects from different parts of the lecture.

Not helpful

- Remembering the slides word by word.
- Learn the ARM reference manuals page by page.