System Construction

Autumn Semester 2019
ETH Zürich
Felix Friedrich, Paul Reed
Goals

- Competence in building custom system software **from scratch**
- Understanding of „how it really works“ behind the scenes **across all levels**
- Knowledge of the approach of fully managed **simple** systems

A lot of this course **is about detail**.
A lot of this course is about **bare metal programming**.
Course Concept

- Discussing elaborated case studies
  - In theory (lectures)
  - and practice (hands-on lab)
- Learning by example vs. presenting topics
Prerequisites

Knowledge corresponding to lectures *Systems Programming* [and Computer Architecture]

- Do you know what a stack-frame is?
- Do you know how an interrupt works?
- Do you know the concept of virtual memory?

Good references for recapitulation:

- Randal E. Bryant, David Richard O'Hallaron, *Computer Systems – A Programmer's Perspective*,
- David A. Patterson, John L. Hennessy *Computer Organization and Design – The Hardware/Software Interface*,
Links

- SVN repository

- Links on the course homepage
  http://lec.inf.ethz.ch/syscon
Some ETH History

1980: Niklaus Wirth develops Lilith, one of the first computers with graphical user interface: bitmap display and mouse

*Lilith* was constructed from 4-bit AMD-Am2900 Slices

Its instruction set was optimized for / codesigned with the intermediate code of the Modula-2 Compiler.

It ran at 7 MHz and had a screen resolution of 704 x 927 pixels.
Some ETH History

1986: A 32-bit processor NS32032 CPU was used to build a new computer *Ceres* together with *its operating system Oberon* that was programmed using the *language Oberon*.

Sources: The Web Site to Remember National Semiconductor's Series 32000 Family, http://www.cpu-ns32k.net/Ceres.html
Some ETH History

1988 Ceres2, based on NS32532 CPU

Sources: The Web Site to Remember National Semiconductor's Series 32000 Family, http://www.cpu-ns32k.net/Ceres.html
Some ETH History

1991 Ceres 3, based on NS32GX32 CPU (cheaper, without MMU)
Used for education at ETH until 1999 (10s of machines)

Sources: The Web Site to Remember National Semiconductor's Series 32000 Family, http://www.cpu-ns32k.net/Ceres.html
Some ETH History

From mid 1990s

Oberon V4 availability as subsystems on Amiga, AtariST, DECStation, HP700, Linux, MacII, PowerMac, RS6000, SiliconGraphics, Solaris 2, Windows

System 3 available on Win3x, Win95NT, Unix (Darwin, PPC Linux, x86 Linux, x86 Solaris), Macintosh (68k, PowerPC), with slim binaries

Native for various platforms.

From 2001: Aos / A2 (Active Oberon)
Background: Co-Design @ ETH

Languages (Pascal Family)
- Modula → Oberon → ActiveOberon → ActiveCells
- Oberon07
- MathOberon

Operating / Runtime Systems
- Medos → Oberon → Aos → A2 → SoC
- BlackBox
- HeliOs → Minos

Hardware
- Lilith → Ceres
- x86 / IA64 / ARM Emulations on Unix / Linux / MacOS
- TRM (FPGA)
- RISC (FPGA)

Timeline:
- 1980
- 1990
- 2000
- 2010
Case Study 1. Minos: Embedded System

- Safety-critical and fault-tolerant monitoring system
- Originally invented for autopilot system for helicopters
- Topics: ARM Architecture, Cross-Development, Object Files and Module Loading, Basic OS Core Tasks (IRQs, MMUs etc.), Minimal Single-Core OS: Scheduling, Device Drivers, Compilation and Runtime Support.

- With hands-on lab on Raspberry Pi (2)
Course Overview
Part 1: Contemporary Hardware

Case Study 2. A2: A lock free Multiprocessor OS kernel

- Universal operating system for symmetric multiprocessors (SMP)
- Based on the co-design of a programming language (Active Oberon) and operating system kernel (A2)
- With hands-on labs on x86ish hardware and Raspberry Pi
Course Overview

Part 2: Custom Designed Systems

Case Study 3. RISC: Single-Processor System [Lectures by Paul Reed]
- RISC single-processor system designed from scratch: hardware on FPGA
- Graphical workstation OS and compiler ("Project Oberon")
- Topics: building a system from scratch, Art of simplicity, Graphical OS, Processor Design.

Case Study 4. Active Cells: Multi-Processor System
- Special purpose heterogeneous system on a chip (SoC)
- Massively parallel hard- and software architecture based on Message Passing
- Topics: Dataflow-Computing, Tiny Register Machine: Processor Design Principles, Software-/Hardware Codesign, Hybrid Compilation, Hardware Synthesis
Organization

- Lecture Wednesday 13:15-15:00 (CAB H 52)
  with a break around 14:00

- Exercise Lab Wednesday 15:15 – 17:00 (CAB H 52)
  Guided, open lab, duration normally 2h
  First exercise: today (September 25th)

- Oral Examination in examination period after semester (15 minutes).
  Prerequisite: knowledge from both course and lab
Design Decisions: Area of Conflict

Simple / undersized

Tailored / non-generic

Comprehensible / simplistic

Customizable / inconvenient

Economic / unoptimized

Sophisticated / complex

Universal / overly generic

Elaborate / incomprehensible

Feature rich / predetermined

Optimized / uneconomic

Programming Model

Compiler

Language

Tools

System

I am about here
Minimal Operating System

1. CASE STUDY MINOS
Topics

- Hardware platform
- Cross development
- Simple modular OS
- Runtime Support
- Realtime task scheduling
- I/O (SPI)*

*Serial Peripheral Interface,
1.1 HARDWARE

Learn to Know the Target Architecture
ARM Processor Architecture Family

- 32 bit Reduced Instruction Set Computer architecture by ARM Holdings
  - 1st production 1985 (Acorn Risc Machine at 4MHz)
  - ARM Ltd. today does not sell hardware but (licenses and hardware descriptions for) chip designs
- Initial designs used for coprocessors in the 8-bit BBC Micro Computers (Computer Literacy Project in the 1980s)
- First ARM Computer: Archimedes (1987)
- An early prominent example: StrongARM (1995)
  - by DEC, licensing the design from Advanced Risc Machines.
  - XScale implementation by Intel (now Marvell) after DEC take over
- More than 90 percent of the sold mobile phones (since 2007) contain at least one ARM processor (often more)*
  [95% of smart phones, 80% of digital cameras and 35% of all electronic devices*]
- Modular approach (today): ARM families produced for different profiles, such as Application Profile, Realtime Profile and Microcontroller / Low Cost Profile

*http://arm.com/about/company-profile/index.php
Other Contemporary RISC Architectures

Examples

- **MIPS (MIPS Technologies)**
  - Business model similar to that of ARM
  - Architectures MIPS(1|...|V), MIPS(32|64), microMIPS(32|64)

- **AVR (Atmel)**
  - Initially targeted towards microcontrollers
  - Harvard Architecture designed and Implemented by Atmel
  - Families: tinyAVR, megaAVR, AVR32
  - AVR32: mixed 16-/32-bit encoding

- **SPARC (Sun Microsystems)**
  - Available as open-source: e.g. LEON (FPGA)

- **MicroBlaze, PicoBlaze (Xilinx)**
  - Softcore on FPGAs, support integrated in Linux.

- **RISC-V (University of California, Berkeley)**
  - Open Architecture, BSD-licensed
### ARM Architecture Versions

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM v1-3</td>
<td>Cache from ARMv2a, 32-bit ISA in 26-bit address space</td>
</tr>
<tr>
<td>ARM v4</td>
<td>Pipeline, MMU, 32-bit ISA in 32-bit address space</td>
</tr>
<tr>
<td>ARM v4T</td>
<td>16-bit encoded Thumb Instruction Set</td>
</tr>
<tr>
<td>ARM v5TE</td>
<td>Enhanced DSP instructions, in particular for audio processing</td>
</tr>
<tr>
<td>ARM v5TEJ</td>
<td>Jazelle Technology extension to support Java acceleration technology (documentation restricted)</td>
</tr>
<tr>
<td>ARM v6</td>
<td>SIMD instructions, Thumb 2, Multicore, Fast Context Switch Extension</td>
</tr>
<tr>
<td>ARM v7</td>
<td>profiles: Cortex- A (applications), -R (real-time), -M (microcontroller)</td>
</tr>
<tr>
<td>ARM v8</td>
<td>Supports 64-bit data / addressing (registers). ARM 64 base instruction description: more than 500 of 6666 pages of the ARM Architecture Reference Manual</td>
</tr>
</tbody>
</table>

**ARM Processor Families (Microarchitectures)**

very much simplified & sparse

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Product Line / Family (Implementation)</th>
<th>Speed (MIPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv1-ARMv3</td>
<td>ARM1-3, 6</td>
<td>4-28 (@8-33MHz)</td>
</tr>
<tr>
<td>ARMv3</td>
<td>ARM7</td>
<td>18-56 MHz</td>
</tr>
<tr>
<td>ARMv4T, ARMv5TEJ</td>
<td>ARM7TDMI</td>
<td>up to 60</td>
</tr>
<tr>
<td>ARMv4</td>
<td>StrongARM</td>
<td>up to 200 (@200MHz)</td>
</tr>
<tr>
<td>ARMv4</td>
<td>ARM8</td>
<td>up to 84 (@72MHz)</td>
</tr>
<tr>
<td>ARMv4T</td>
<td>ARM9TDMI</td>
<td>200 (@180MHz)</td>
</tr>
<tr>
<td>ARMv5TE(J)</td>
<td>ARM9E</td>
<td>220 (@200MHz)</td>
</tr>
<tr>
<td>ARMv5TE(J)</td>
<td>ARM10E</td>
<td></td>
</tr>
<tr>
<td>ARMv5TE</td>
<td>XScale</td>
<td>up to 1000 @1.25GHz</td>
</tr>
<tr>
<td>ARMv6</td>
<td>ARM11</td>
<td>740</td>
</tr>
<tr>
<td>ARMv6, ARMv7, ARMv8</td>
<td>ARM Cortex</td>
<td>up to 10000 DMIPS (Multicore @2GHz)</td>
</tr>
</tbody>
</table>
ARM Cortex Microarchitectures

- Cortex-A
  - ARM v7-A, ARM v8-A
  - Application profile: typically including luxuries such as MMU support for OSes, ranging up to high performance multicore CPUs with (NEON) SIMD units while power consumption is moderate, newest generation provides 64-bit support

- Cortex-M
  - ARM v6-M, ARM v7-M
  - Microcontroller profile (32bit), Thumb instruction set, very low power consumption, some provide a MPU

- Cortex-R
  - ARM v7-R
  - Realtime profile, tightly coupled memory, deterministic interrupt handling, redundant computation (HW replication for fault tolerance)

ARM Architecture Reference Manuals

describe

- ARM/Thumb instruction sets
- Processor modes and states
- Exception and interrupt model
- System programmer's model, standard coprocessor interface
- Memory model, memory ordering and memory management for different potential implementations
- Optional extensions like Floating Point, SIMD, Security, Virtualization ...

for example required for the implementation of assembler, disassembler, compiler, linker and debugger and for the systems programmer.
ARM Technical System Reference Manuals

describe

- Particular processor implementation of an ARM architecture
- Redundant information from the Architecture manual (e.g. system control processor)
- Additional processor implementation specifics e.g. cache sizes and cache handling, interrupt controller, generic timer

usually required by a system's programmer
System on Chip Implementation Manuals
describe

- Particular implementation of a System on Chip
- Address map:
  physical addresses and bit layout for the registers
- Peripheral components / controllers,
  such as Timers, Interrupt controller, GPIO, USB, SPI, DMA, PWM, UARTs
usually required by a system's programmer.
ARM Instruction Set

consists of

- Data processing instructions
- Branch instructions
- Status register transfer instructions
- **Load and Store** instructions
- Generic Coprocessor instructions
- Exception generating instructions
Some Features
of the ARM Instruction Set

- 32 bit instructions / many in one cycle / 3 operands
- Load / store architecture (no memory operands such as in x86)

```
ldr r11, [fp, #-8]
add r11, r11, #1
str r11, [fp, #-8]
```
Some Features
of the ARM Instruction Set

- 32 bit instructions / many in one cycle / 3 operands
- Load / store architecture (no memory operands such as in x86)

```assembly
ldr r11, [fp, #-8]
add r11, r11, #1
str r11, [fp, #-8]
```

increment a local variable
Some Features of the ARM Instruction Set

- Index optimized instructions (such as pre-/post-indexed addressing)

```
stmdb sp!,{fp,lr} ; store multiple decrease before and update sp
...
ldmia sp!,{fp,pc} ; load multiple increase after and update sp
```
Some Features
of the ARM Instruction Set

- Index optimized instructions (such as pre-/post-indexed addressing)

\[
\text{stmdb sp!},{fp,lr} \ ; \text{ store multiple decrease before and update sp}
\]

\[
\ldmia sp!,{fp,pc} \ ; \text{ load multiple increase after and update sp}
\]
Some Features
of the ARM Instruction Set

- *Predication*: all instructions can be conditionally executed*

```assembly
cmp    r0, #0
swieq #0xa
```

null pointer check

?
Some Features of the ARM Instruction Set

- *Predication*: all instructions can be conditionally executed*

```assembly
cmp  r0, #0
swieq #0xa
```

null pointer check
Impressive Example of Predication

loop: CMP Ri, Rj ; set condition flags
       SUBGT Ri, Ri, Rj ; if i>j then i = i-j;
       SUBLT Rj, Rj, Ri ; if i<j then j = j-i;
       BNE loop ; if i != j then loop
Some Features
of the ARM Instruction Set

Link Register

bl #0x0a0100070

- Shift and rotate in instructions

add r11, fp, r11, lsl #2
Some Features of the ARM Instruction Set

Link Register

- Shift and rotate in instructions

```
bl #0x0a0100070
```

procedure call

```
add r11, fp, r11, lsl #2
```

```
r11 = fp + r11*4
e.g. array access
```
Some Features of the ARM Instruction Set

- PC-relative addressing

  \texttt{ldr \ r0, [pc, \#+24]} ?

- Coprocessor access instructions

  \texttt{mrc \ p15, 0, r11, c6, c0, 0} ?
Some Features of the ARM Instruction Set

- PC-relative addressing

```assembly
ldr r0, [pc, #+24]
```

- Coprocessor access instructions

```assembly
mrc p15, 0, r11, c6, c0, 0
```

load a large constant

setup the mmu
## ARM Instruction Set Encoding (ARM v5)

### Shiftable Register
- ARM Instruction Set Encoding (ARM v5)
- Shiftable register

### 8 bit immediates with even rotate
- 8 bit immediates with even rotate

### Load / Store with destination increment
- Load / store with destination increment

### Undefined instruction: user extensibility
- Undefined instruction: user extensibility

### Load / Store with multiple registers
- Load / store with multiple registers

### Branches with 24 bit offset
- Branches with 24 bit offset

### Generic coprocessor instructions
- Generic coprocessor instructions

---

**From ARM Architecture Reference Manual**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Shift Amount</th>
<th>Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cond [1]</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Misc.</td>
<td>Cond [1]</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data proc.</td>
<td>Cond [1]</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Misc.</td>
<td>Cond [1]</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplies</td>
<td>Cond [1]</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data proc.</td>
<td>Cond [1]</td>
<td>0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undef.</td>
<td>Cond [1]</td>
<td>0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load/store</td>
<td>Cond [1]</td>
<td>0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undef.</td>
<td>Cond [1]</td>
<td>0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undef.</td>
<td>Cond [1]</td>
<td>1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load/store</td>
<td>Cond [1]</td>
<td>1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undef.</td>
<td>Cond [1]</td>
<td>1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>Cond [1]</td>
<td>1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coproc.</td>
<td>Cond [1]</td>
<td>1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undef.</td>
<td>Cond [1]</td>
<td>1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Example Table

<table>
<thead>
<tr>
<th>Condition</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Shift Amount</th>
<th>Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cond [1]</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Misc.</td>
<td>Cond [1]</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data proc.</td>
<td>Cond [1]</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Misc.</td>
<td>Cond [1]</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplies</td>
<td>Cond [1]</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data proc.</td>
<td>Cond [1]</td>
<td>0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undef.</td>
<td>Cond [1]</td>
<td>0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load/store</td>
<td>Cond [1]</td>
<td>0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undef.</td>
<td>Cond [1]</td>
<td>0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undef.</td>
<td>Cond [1]</td>
<td>1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load/store</td>
<td>Cond [1]</td>
<td>1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undef.</td>
<td>Cond [1]</td>
<td>1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>Cond [1]</td>
<td>1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coproc.</td>
<td>Cond [1]</td>
<td>1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undef.</td>
<td>Cond [1]</td>
<td>1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**From ARM Architecture Reference Manual**
Thumb Instruction Set

ARM instruction set complemented by

- **Thumb Instruction Set**
  - 16-bit instructions, 2 operands
  - eight GP registers accessible from most instructions
  - subset in functionality of ARM instruction set
  - targeted for density from C-code (~65% of ARM code size)

- **Thumb2 Instruction Set**
  - extension of Thumb, adds 32 bit instructions to support almost all of ARM ISA (different from ARM instruction set encoding!)
  - design objective: ARM performance with Thumb density
Typical procedure call on ARM

**Caller:** push parameters

use branch and link instruction. Stores the PC of the next instruction into the link register.

**Callee:** save link register and frame pointer on stack and set new frame pointer.

Execute procedure content

Reset stack pointer and restore frame pointer and jump back to caller address.

**Caller:** cleanup parameters from stack

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bl #address</td>
<td>Branch and link instruction</td>
</tr>
<tr>
<td>stmdb sp!, {fp, lr}</td>
<td>Store parameters on stack</td>
</tr>
<tr>
<td>mov fp, sp</td>
<td>Move frame pointer to sp</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>mov sp, fp</td>
<td>Move stack pointer to fp</td>
</tr>
<tr>
<td>ldmia sp!, {fp, pc}</td>
<td>Load and move instruction</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>add sp, sp, #n</td>
<td>Add stack pointer to sp, offset</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
ARM Processor Modes

ARM from v5 has (at least) seven basic operating modes

- Each mode has access to its **own stack** and a different subset of registers
- Some operations can only be carried out in a privileged mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description / Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervisor</td>
<td>Reset / Software Interrupt</td>
</tr>
<tr>
<td>FIQ</td>
<td>Fast Interrupt</td>
</tr>
<tr>
<td>IRQ</td>
<td>Normal Interrupt</td>
</tr>
<tr>
<td>Abort</td>
<td>Memory Access Violation</td>
</tr>
<tr>
<td>Undef</td>
<td>Undefined Instruction</td>
</tr>
<tr>
<td>System</td>
<td>Privileged Mode with same registers as in User Mode</td>
</tr>
<tr>
<td>User</td>
<td>Regular Application Mode</td>
</tr>
</tbody>
</table>
ARM Register Set

**ARM has 37 registers, all 32-bits long**
A subset is accessible in each mode
Register 13 is the Stack Pointer (by convention)
Register 14 is the Link Register**
Register 15 is the Program Counter (settable)
CPSR* is not immediately accessible

---

*current / saved processor status register, accessible via MSR / MRS instructions
**more than a convention: link register set as side effect of some instructions
Processor Status Register (PSR)

### Condition Codes
- **N** (Negative): result from ALU is negative
- **Z** (Zero): result from ALU is zero
- **C** (Carry): ALU operation is carried out
- **V** (Overflow): ALU operation overflowed

### Interrupt Disable bits
- **I**: Disables IRQ
- **F**: Disables FIQ

### Mode Bits
- Specify processor mode

### T Bit
- **T=0**: Processor in ARM mode
- **T=1**: Processor in Thumb State

### Other bits
- Architecture 5TE(J) and later:
  - **Q** flag: sticky overflow flag for saturating instr.
  - **J** flag: Jazelle state
- Architecture 6 and later:
  - **GE[3:0]**: used by SIMD instructions
  - **E**: controls endianness
  - **A**: controls imprecise data aborts
  - **IT**: controls conditional execution of Thumb2

* reverse cmp/sub meaning compared with x86
Raspberry Pi 2

Raspberry Pi 2 (Model B) will be the hardware used at least in the first 4 weeks lab sessions

- Produced by element14 in the UK (www.element14.com)
- Features
  - Broadcom BCM2836 ARMv7 Quad Core Processor running at 900 MHz
  - 1G RAM
  - 40 PIN GPIO
  - Separate GPU ("Videocore")
  - Peripherals: UART, SPI, USB, 10/100 Ethernet Port (via USB), 4pin Stereo Audio, CSI camera, DSI display, Micro SD Slot
  - Powered from Micro USB port
ARM System Boot

- ARM processors usually starts executing code at adr 0x0
  - e.g. containing a branch instruction to jump over the interrupt vectors
  - usually requires some initial setup of the hardware

- The RPI, however, is **booted from the Video Core CPU (VC):**
  the firmware of the RPI does a lot of things before we get control:
  *kernel-image gets copied to address 0x8000H and branches there*
  No virtual to physical address-translation takes place in the start.

- Only one core runs at that time. (More on this later)
RPI 1 Memory Map

This is for RPI1 (BCM 2835) and wrong for RPI2 (BCM 2836) correct for BCM2836: 3F000000
RPI 2 Memory Map

- Initially the MMU is switched off. No memory translation takes place.
- System memory divided in ARM and VC part, partially shared (e.g. frame buffer)
- ARM's memory mapped registers start from 0x3F000000 -- opposed to reported offset 0x7E000000 in BCM 2835 Manual
General Purpose I/O (GPIO)

- Software controlled processor pins
  - Configurable direction of transfer
  - Configurable connection
    - with internal controller (SPI, MMC, memory controller, ...)
    - with external device

- Pin state settable & gettable
  - High, low

- Forced interrupt on state change
  - On falling/ rising edge
GPIO
Block Diagram (BCM 2835)

pin direction control

internal function selection

output control registers

interrupt control

pull up / down resistor control

input (pin level) registers
### Raspberry Pi 2 GPIO Pinout

<table>
<thead>
<tr>
<th>Name</th>
<th>Pin</th>
<th>Name</th>
<th>Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3V DC</td>
<td>01</td>
<td>DC power 5v</td>
<td>02</td>
</tr>
<tr>
<td>GPIO 02</td>
<td>03</td>
<td>DC power 5v</td>
<td>04</td>
</tr>
<tr>
<td>GPIO 03</td>
<td>05</td>
<td>ground</td>
<td>06</td>
</tr>
<tr>
<td>GPIO 04</td>
<td>07</td>
<td>GPIO 14</td>
<td>08</td>
</tr>
<tr>
<td>ground</td>
<td>09</td>
<td>GPIO 15</td>
<td>10</td>
</tr>
<tr>
<td>GPIO 17</td>
<td>11</td>
<td>GPIO 18</td>
<td>12</td>
</tr>
<tr>
<td>GPIO 27</td>
<td>13</td>
<td>ground</td>
<td>14</td>
</tr>
<tr>
<td>GPIO 22</td>
<td>15</td>
<td>GPIO 23</td>
<td>16</td>
</tr>
<tr>
<td>3.3V DC</td>
<td>17</td>
<td>GPIO 24</td>
<td>18</td>
</tr>
<tr>
<td>GPIO 10</td>
<td>19</td>
<td>ground</td>
<td>20</td>
</tr>
<tr>
<td>GPIO 09</td>
<td>21</td>
<td>GPIO 25</td>
<td>22</td>
</tr>
<tr>
<td>GPIO 11</td>
<td>23</td>
<td>GPIO 08</td>
<td>24</td>
</tr>
<tr>
<td>ground</td>
<td>25</td>
<td>GPIO 07</td>
<td>26</td>
</tr>
<tr>
<td>ID_SD</td>
<td>27</td>
<td>ID_SC</td>
<td>28</td>
</tr>
<tr>
<td>GPIO 05</td>
<td>29</td>
<td>ground</td>
<td>30</td>
</tr>
<tr>
<td>GPIO 06</td>
<td>31</td>
<td>GPIO 12</td>
<td>32</td>
</tr>
<tr>
<td>GPIO 13</td>
<td>33</td>
<td>ground</td>
<td>34</td>
</tr>
<tr>
<td>GPIO 19</td>
<td>35</td>
<td>GPIO 16</td>
<td>36</td>
</tr>
<tr>
<td>GPIO 26</td>
<td>37</td>
<td>GPIO 20</td>
<td>38</td>
</tr>
<tr>
<td>ground</td>
<td>39</td>
<td>GPIO 21</td>
<td>40</td>
</tr>
</tbody>
</table>

Connecting external power with 5v here kills the board!

Be careful with the USB TTL Cable (Exercise 2)
## GPIO Register Overview (p. 90)

<table>
<thead>
<tr>
<th>Address</th>
<th>Field Name</th>
<th>Description</th>
<th>Size</th>
<th>Read/Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x 7E20 0000</td>
<td>GPFSSEL0</td>
<td>GPIO Function Select 0</td>
<td>32</td>
<td>R/W</td>
</tr>
<tr>
<td>0x 7E20 0004</td>
<td>GPFSSEL1</td>
<td>GPIO Function Select 1</td>
<td>32</td>
<td>R/W</td>
</tr>
<tr>
<td>0x 7E20 0008</td>
<td>GPFSSEL2</td>
<td>GPIO Function Select 2</td>
<td>32</td>
<td>R/W</td>
</tr>
<tr>
<td>0x 7E20 000C</td>
<td>GPFSSEL3</td>
<td>GPIO Function Select 3</td>
<td>32</td>
<td>R/W</td>
</tr>
<tr>
<td>0x 7E20 0010</td>
<td>GPFSSEL4</td>
<td>GPIO Function Select 4</td>
<td>32</td>
<td>R/W</td>
</tr>
</tbody>
</table>

## GPIO Function Select (p. 92 - 94)

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field Name</th>
<th>Description</th>
<th>Type</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>21-30</td>
<td>GPFSSEL0</td>
<td>Reserved</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>20-27</td>
<td>GPFSSEL1</td>
<td>GPFSSEL - Function Select 19</td>
<td>RW</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000 = GPIO Pin 19 is an input</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>001 = GPIO Pin 19 is an output</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>100 = GPIO Pin 19 takes alternate function 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>101 = GPIO Pin 19 takes alternate function 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>110 = GPIO Pin 19 takes alternate function 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>111 = GPIO Pin 19 takes alternate function 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20-24</td>
<td>GPFSSEL18</td>
<td>GPFSSEL - Function Select 18</td>
<td>RW</td>
<td>0</td>
</tr>
<tr>
<td>23-21</td>
<td>GPFSSEL17</td>
<td>GPFSSEL - Function Select 17</td>
<td>RW</td>
<td>0</td>
</tr>
<tr>
<td>20-18</td>
<td>GPFSSEL16</td>
<td>GPFSSEL - Function Select 16</td>
<td>RW</td>
<td>0</td>
</tr>
<tr>
<td>17-15</td>
<td>GPFSSEL15</td>
<td>GPFSSEL - Function Select 15</td>
<td>RW</td>
<td>0</td>
</tr>
<tr>
<td>14-12</td>
<td>GPFSSEL14</td>
<td>GPFSSEL - Function Select 14</td>
<td>RW</td>
<td>0</td>
</tr>
<tr>
<td>11-9</td>
<td>GPFSSEL13</td>
<td>GPFSSEL - Function Select 13</td>
<td>RW</td>
<td>0</td>
</tr>
<tr>
<td>8-6</td>
<td>GPFSSEL12</td>
<td>GPFSSEL - Function Select 12</td>
<td>RW</td>
<td>0</td>
</tr>
<tr>
<td>5-3</td>
<td>GPFSSEL11</td>
<td>GPFSSEL - Function Select 11</td>
<td>RW</td>
<td>0</td>
</tr>
<tr>
<td>2-0</td>
<td>GPFSSEL10</td>
<td>GPFSSEL - Function Select 10</td>
<td>RW</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6-3 – GPIO Alternate function select register 1

## GPIO Pin Mapping / Alternate Functions (p. 102)

|GPIO13| Low| TXD0| SD6| <reserved>| | | RXD2| SPI1_CLK| RTS1|
|------|----|-----|----|----------| | |     |      |     |
|GPIO14| Low| TXD0| SD6| <reserved>| | | TXD1|       |     |
|GPIO15| Low| RXD0| SD7| <reserved>| | | RXD1|       |     |
|GPIO16| Low| <reserved>| SD8| <reserved>| | | CTS0| SPI1_CE2_N| CTS1|
|GPIO17| Low| <reserved>| SD9| <reserved>| | | RTS0| SPI1_CE1_N| RTS1|
1. Program GPIO Pin Function (in / out / alternate function) by writing corresponding (memory mapped) GPFSEL register. **GPFSELn**: pins $10n$, $\ldots$, $10n + 9$

   Use RMW (Read-Modify-Write) operation in order to keep the other bits

2. Use GPIO Pin
   a. If writing: set corresponding bit in the GPSETn or GPCLRn register
      set pin: **GPSETn**: pins $32n$, $\ldots$, $32n + 31$
      clear pin: **GPCLRn**: pins $32n$, $\ldots$, $32n + 31$
      no RMW (Read – Modify – Write) required.
   b. If reading: read corresponding bit in the GPLEVn register
      **GPLEVn**: pins $32n$, $\ldots$, $32n + 31$
   c. If "alternate function": device acts autonomously. Implement device driver.