System Construction

Autumn Semester 2015

Felix Friedrich

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 **lean** systems

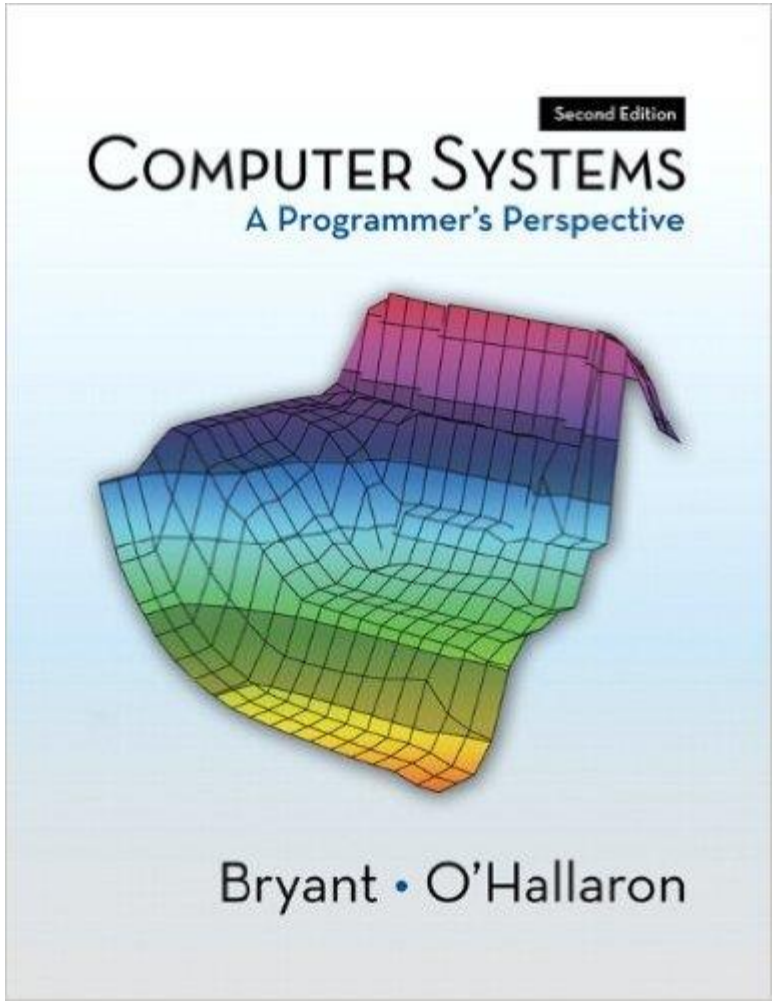
A lot of this course **is about detail.**
A lot of this course is about **bare metal programming**.

Course Concept

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

Prerequisite

- Knowledge corresponding to lectures *Systems Programming* and/or *Operating Systems*
  - *Do you know what a stack-frame is?*
  - *Do you know how an interrupt works?*
  - *Do you know the concept of virtual memory?*
- Good reference for recapitulation: *Computer Systems – A Programmer's Perspective*



Links

- SVN repository

https://svn.inf.ethz.ch/svn/lecturers/vorlesungen/trunk/syscon/2015/shared

- Links on the course homepage

http://lec.inf.ethz.ch/syscon/2015

Background: Co-Design @ ETH

Languages (Pascal Family)

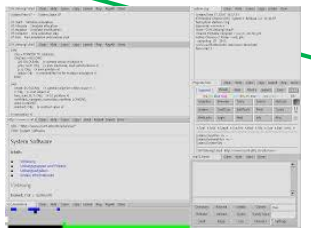
Modula → Oberon → **ActiveOberon** (**+MathOberon**) → **Active Cells**

Oberon → Oberon07

ActiveOberon → Zonnon

Operating / Runtime Systems

Medos → Oberon → Aos → **A2** → **SoC**



Aos → HeliOs → **Minos**

A2 → **LockFree Kernel**

Hardware

Lilith → Ceres → x86 / IA64/ ARM Emulations on Unix / Linux → **TRM (FPGA)**

x86 / IA64/ ARM Emulations on Unix / Linux → RISC (FPGA)

1980 1990 2000 2010

Course Overview

Part1: Contemporary Hardware

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.
- new Now with hands-on lab on Raspberry Pi (2)

Course Overview

Part1: Contemporary Hardware

Case Study 2. A2: Multiprocessor OS

- Universal operating system for symmetric multiprocessors (SMP)
- Based on the co-design of a programming language (Active Oberon) and operating system (A2)
- Topics: Intel SMP Architecture, Multicore Operating System, Scheduling, Synchronisation, Synchronous and Aysynchronous Context Switches, Priority Handling, Memory Handling, Garbage Collection.

Case Study 2a: Lock-free Operating System Kernel

- With hands-on labs on x86ish hardware and Raspberry Pi

Course Overview

Part2: Custom Designed Systems

Case Study 3. RISC: Single-Processor System

- 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 Tuesday 13:15-15:00 (CAB G 57)
  with a break around 14:00

- Exercise Lab Tuesday 15:00 – 17:00 (CAB G 56)
  Guided, open lab, duration normally 2h
  First exercise: **today (15th September)**

- 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 / simplicistic

customizable / inconvenient

economic / unoptimzed

Programming Model
Compiler
Language
Tools
System

sophisticated / complex

universal / overly generic

elaborate / incomprehensible

feature rich / predetermined

optimized / uneconomic

I am about here

Minimal **O**perating **S**ystem

1. CASE STUDY MINOS

Focus Topics

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

*Serial Peripheral Interface,
Universal Asynchronous Receiver Transmitter

Learn to Know the Target Architecture

1.1 HARDWARE

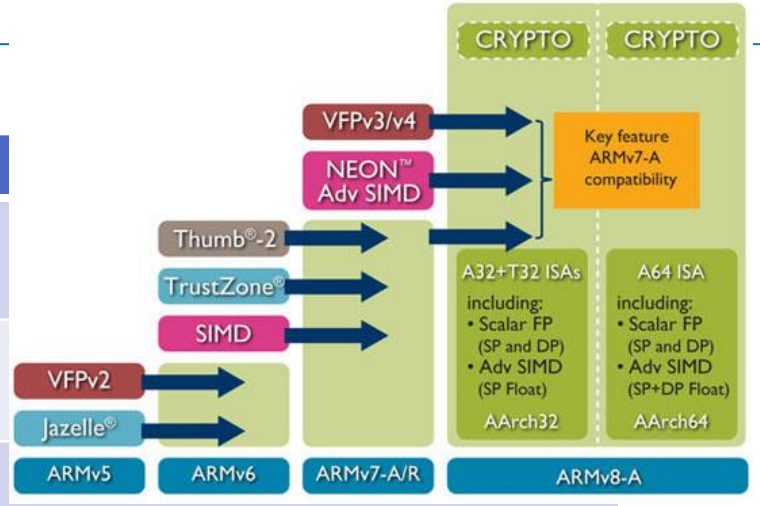
ARM Processor Architecture Family

- 32 bit **R**educed **I**nstruction **S**et **C**omputer architecture by ARM Holdings
  - 1st production 1985 (Acorn Risc Machine at 4MHz)
  - ARM Ltd. today does not sell hardware but (licenses for) chip designs
- StrongARM
  - by DEC & **A**dvanced **R**isc **M**achines.
  - 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:
  ARM families produced for different profiles, such as Application Profile, Realtime Profile and Microcontroller / Low Cost Profile

*http://news.cnet.com/ARMed-for-the-living-room/2100-1006_3-6056729.html
*http://arm.com/about/company-profile/index.php

ARM *Architecture* Versions

Architecture	Features
ARM v1-3	Cache from ARMv2a, 32-bit ISA in 26-bit address space
ARM v4	Pipeline, MMU, 32 bit ISA in 32 bit address space
ARM v4T	16-bit encoded Thumb Instruction Set
ARM v5TE	Enhanced DSP instructions, in particular for audio processing
ARM v5TEJ	Jazelle Technology extension to support Java acceleration technology (documentation restricted)
ARM v6	SIMD instructions, Thumb 2, Multicore, Fast Context Switch Extension
ARM v7	profiles: Cortex- A (applications), -R (real-time), -M (microcontroller)
ARM v8	Supports 64-bit data / addressing (registers). Assembly language overview available (more than 100 pages pure instruction semantics)



[http://www.arm.com/products/processors/instruction-set-architectures/]

ARM Processor *Families*

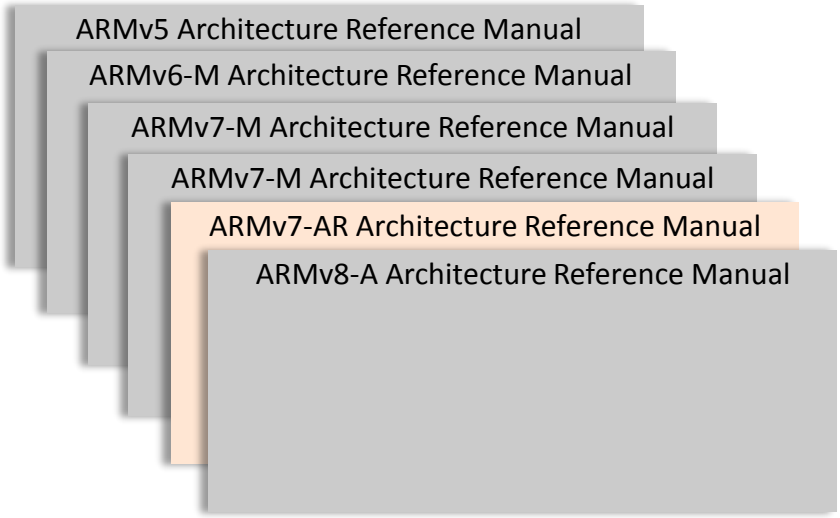
very simplified & sparse

Architecture	Product Line / Family (Implementation)	Speed (MIPS)
ARMv1-ARMv3	ARM1-3, 6	4-28 (@8-33MHz)
ARMv3	ARM7	18-56 MHz
ARMv4T, ARMv5TEJ	ARM7TDMI	up to 60
ARMv4	StrongARM	up to 200 (@200MHz)
ARMv4	ARM8	up to 84 (@72MHz)
ARMv4T	ARM9TDMI	200 (@180MHz)
ARMv5TE(J)	ARM9E	220(@200MHz)
ARMv5TE(J)	ARM10E	
ARMv5TE	XScale	up to 1000 @1.25GHz
ARMv6	ARM11	740
ARMv6, ARMv7, ARMv8	ARM Cortex	up to 2000 (@>1GHz)

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 ...

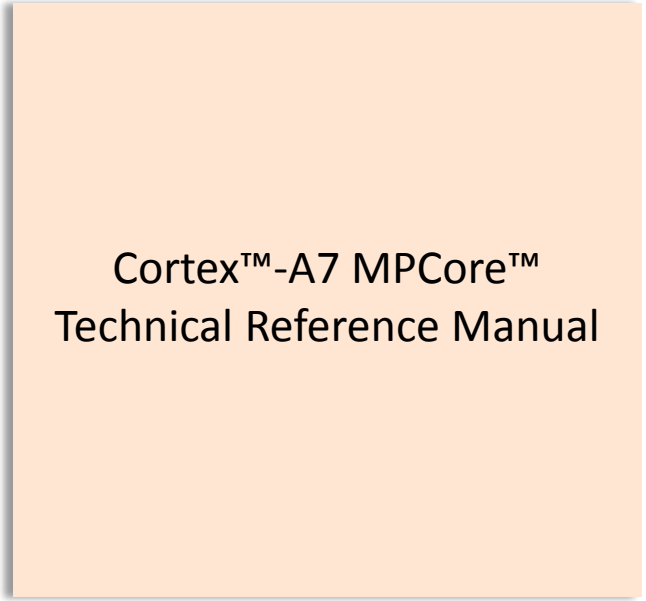
for example required for the implementation of assembler, disassembler, compiler, linker and debugger and for the systems programmer.

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

- 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 decrease after and update sp
```

Some Features

of the ARM Instruction Set

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

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

?

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

- PC-relative addressing

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

?

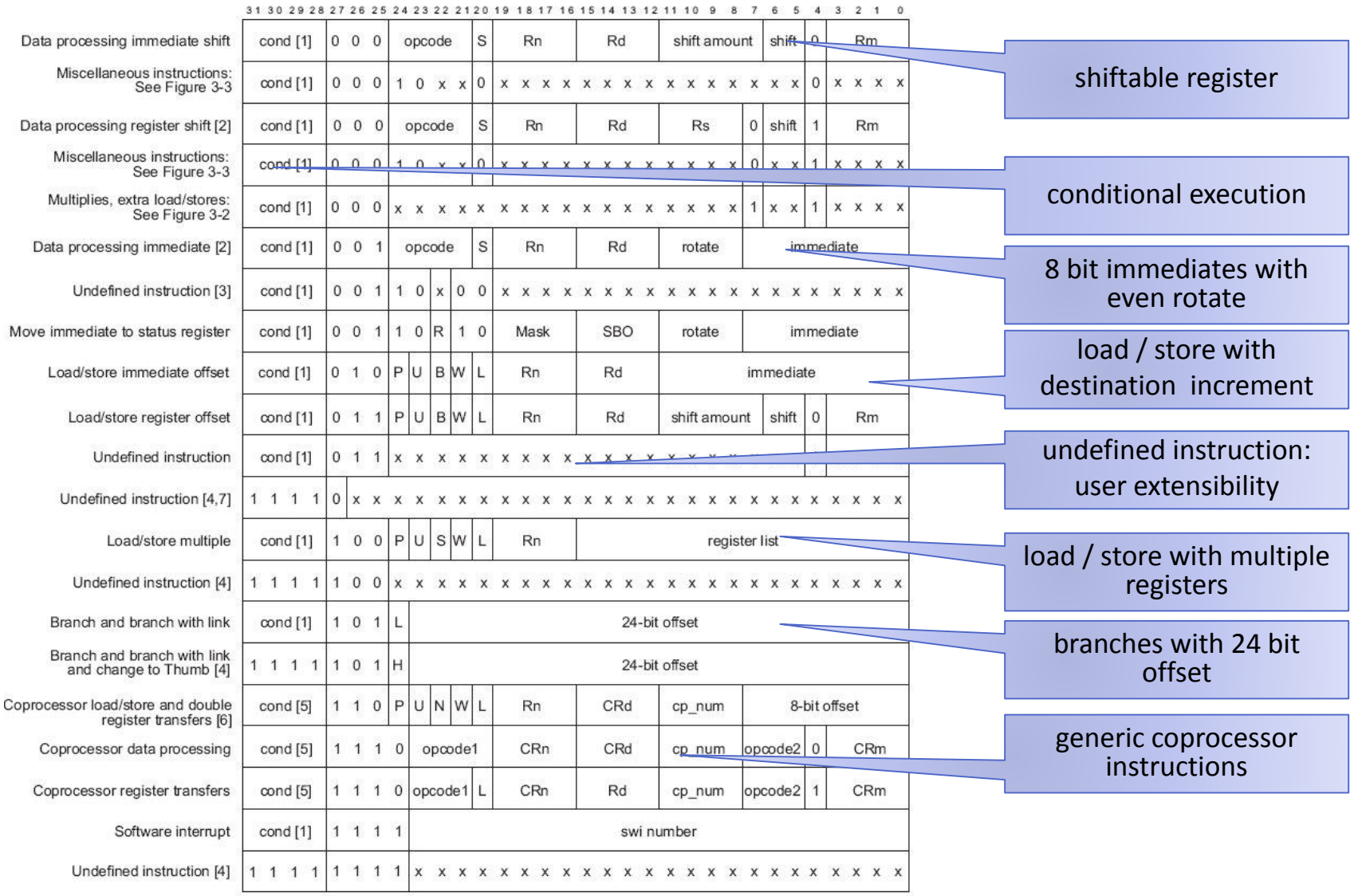
- Coprocessor access instructions

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

?

ARM Instruction Set

Encoding (ARM v5)



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

Other Contemporary RISC Architectures

Examples

- MIPS (MIPS Technologies)
  - Business model similar to that of ARM
  - Architectures MIPS(I|…|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)
- …

ARM Processor Modes

- ARM from v5 has (at least) seven basic operating modes
  - Each mode has access to **own stack** and a different subset of registers
  - Some operations can only be carried out in a privileged mode

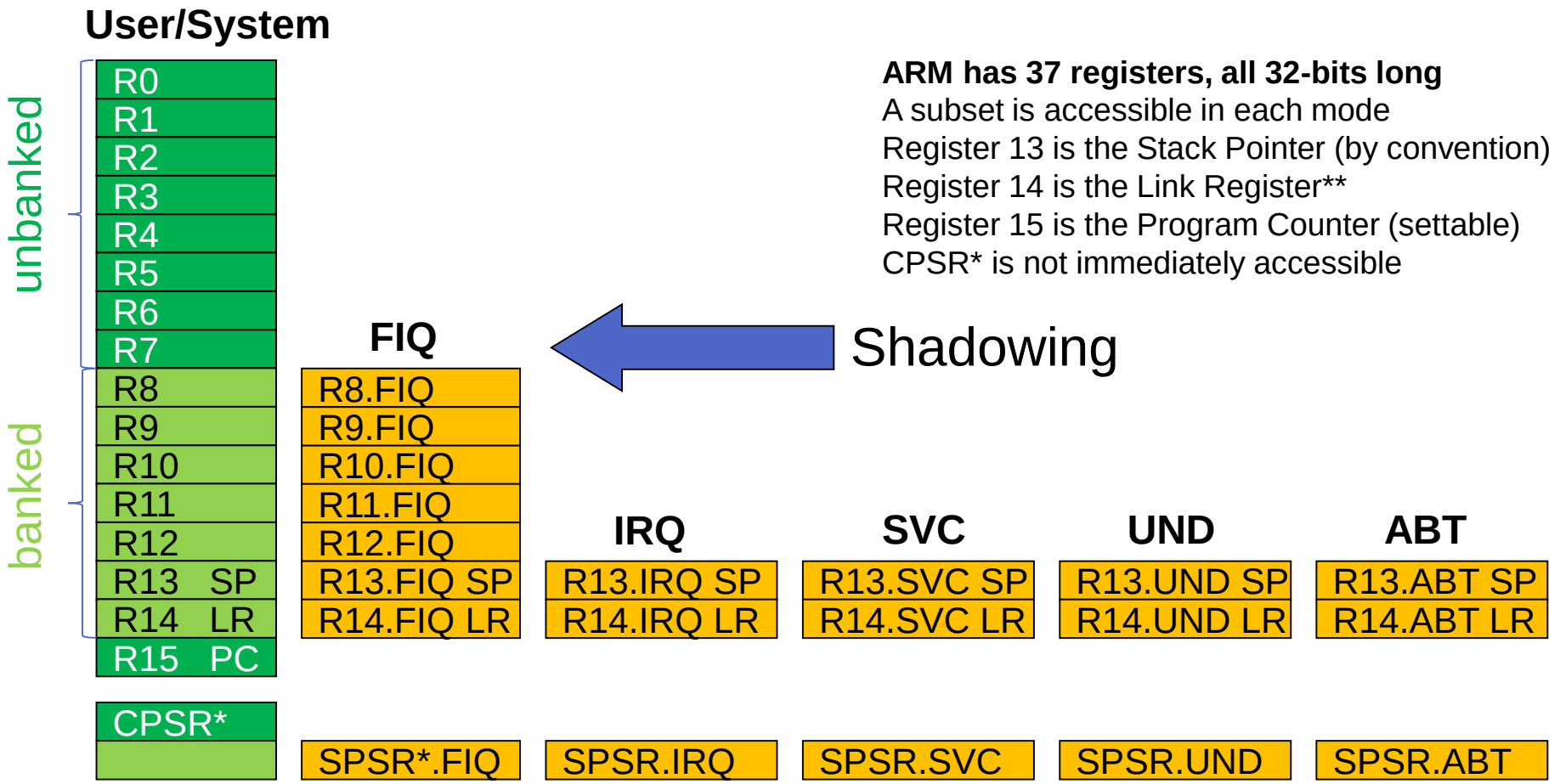
Mode	Description / Cause
Supervisor	Reset / Software Interrupt
FIQ	Fast Interrupt
IRQ	Normal Interrupt
Abort	Memory Access Violation
Undef	Undefined Instruction
System	Privileged Mode with same registers as in User Mode
User	Regular Application Mode

privileged

exceptions

normal execution

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
- Z=Zero result from ALU
- C=ALU operation Carried out *
- V=ALU operation overflowed

Interrupt Disable bits

- I=1: Disables IRQ
- F=1: Disables FIQ

Mode Bits

- Specify processor mode

Other bits

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

T Bit

- T=0: Processor in ARM mode
- T=1: Processor in Thumb State
- Introduced in Architecture 4T

* reverse cmp/sub meaning compared with x86

Typical procedure call on ARM

Caller: push parameters

…

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

bl #address

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

stmdb sp!, {fp, lr}
mov fp, sp

Execute procedure content

…

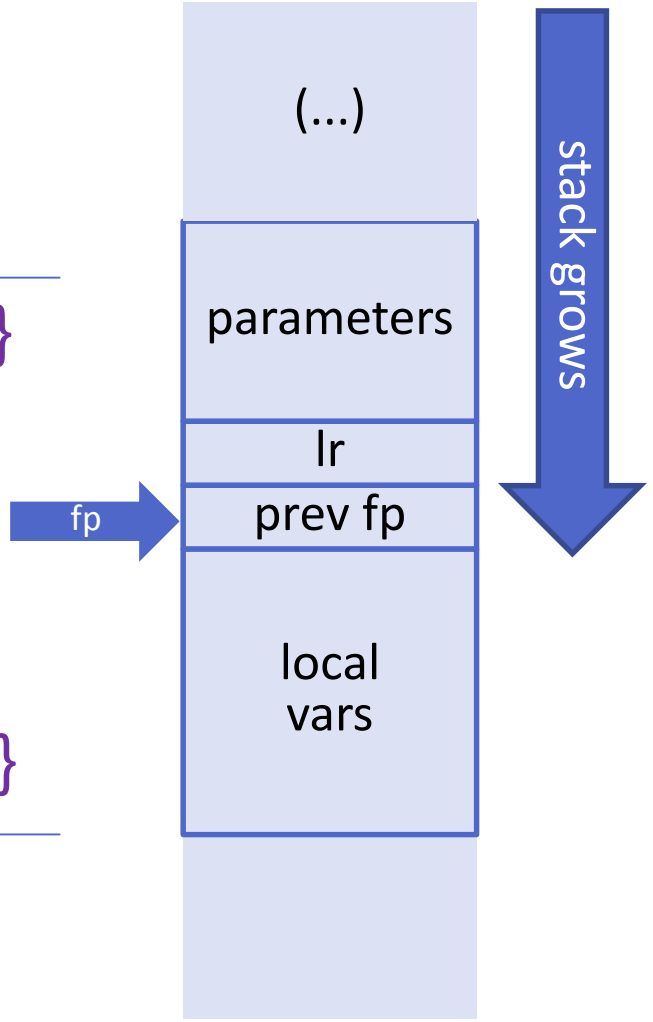
Reset stack pointer and restore frame pointer and and jump back to caller address.

mov sp, fp
ldmia sp!, {fp, pc}

Caller: cleanup parameters from stack

add sp, sp, #n

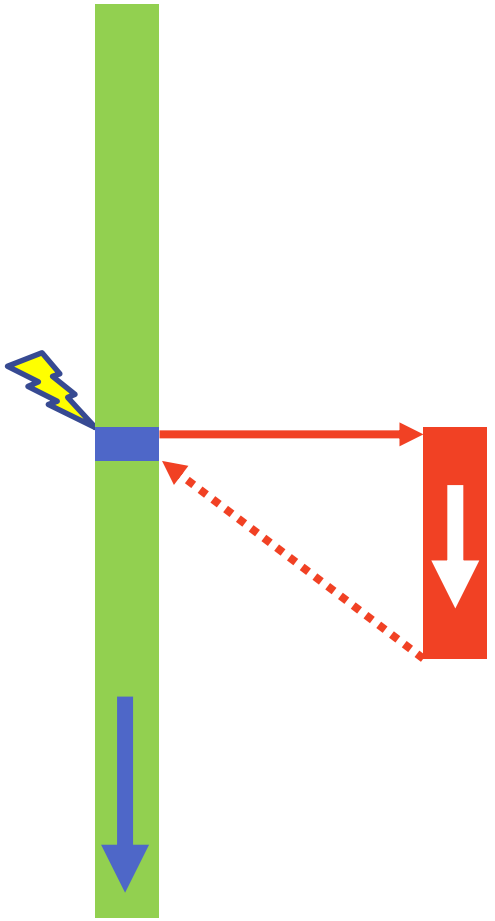
…



Exceptions (General)

Exception = abrupt change in the control flow as a response to some change in the processor's state

- **Interrupt** - asynchronous event triggered by a device signal
- **Trap / Syscall** - intentional exception
- **Fault** - error condition that a handler might be able to correct
- **Abort** - error condition that cannot be corrected



Exception Handling

Involves close interaction between hardware and software.

Exception handling is similar to a procedure call with important differences:

- processor prepares exception handling: save* part of the current processor state before execution of the software exception handler
- assigned to each exception is an exception number, the exception handler's code is accessible via some exception table that is configurable by software
- exception handlers run in a different processor mode with complete access to the system resources.

* in special registers or on the stack – we **will** go into the details for some architectures

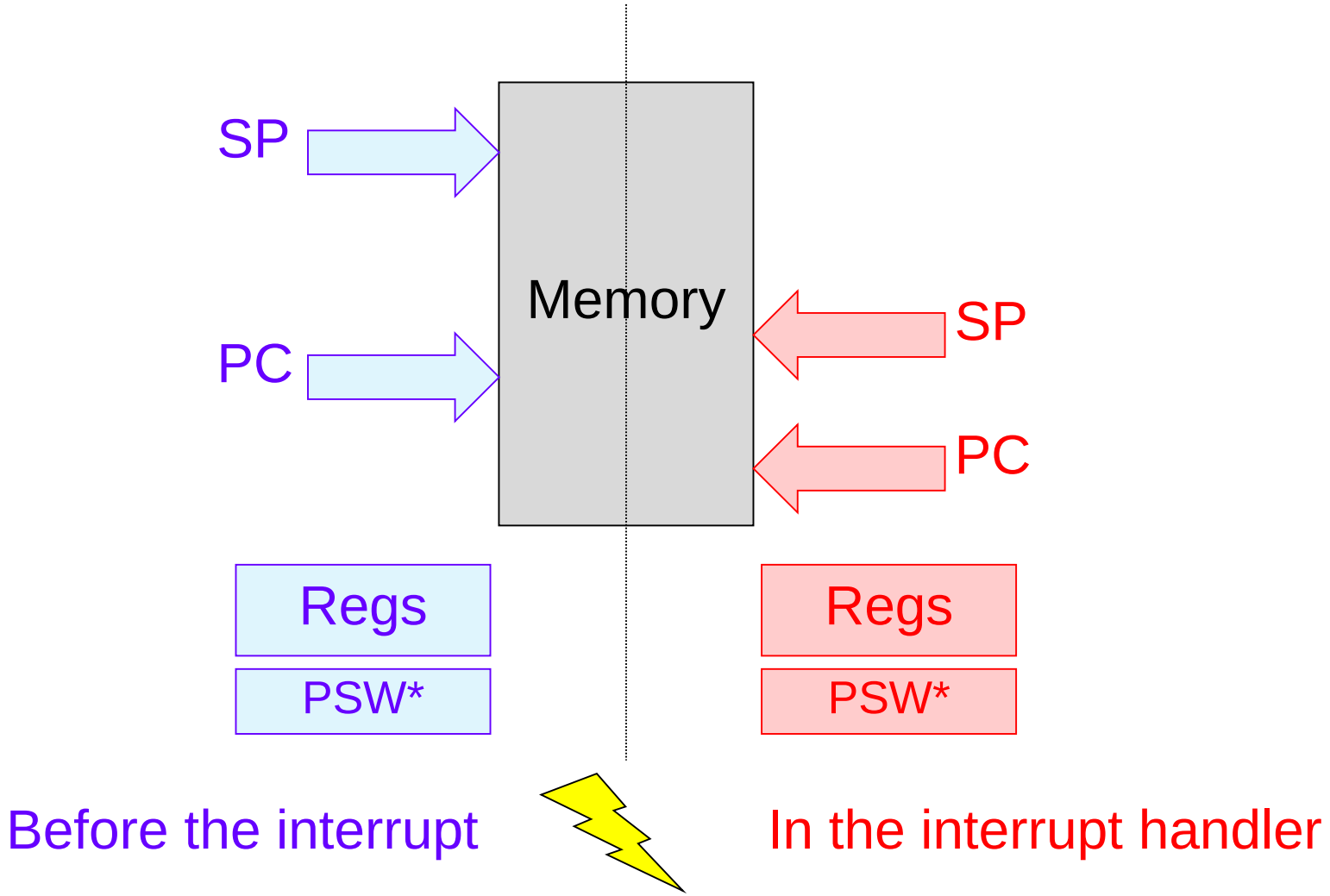
Exception Table on ARM

Type	Mode	Address*	return link(type)**
Reset	Supervisor	`0x0`	`undef`
Undefined Instruction	Undefined	`0x4`	`next instr`
SWI	Supervisor	`0x8`	`next instr`
Prefetch Abort	Abort	`0xC`	`aborted instr +4`
Data Abort	Abort	`0x10`	`aborted instr +8`
Interrupt (IRQ)	IRQ	`0x18`	`next instr +4`
Fast Interrupt (FIQ)	FIRQ	`0x1C`	`next instr +4`

* alternatively High Vector Address = `0xFFFF0000 + adr` (configurable)

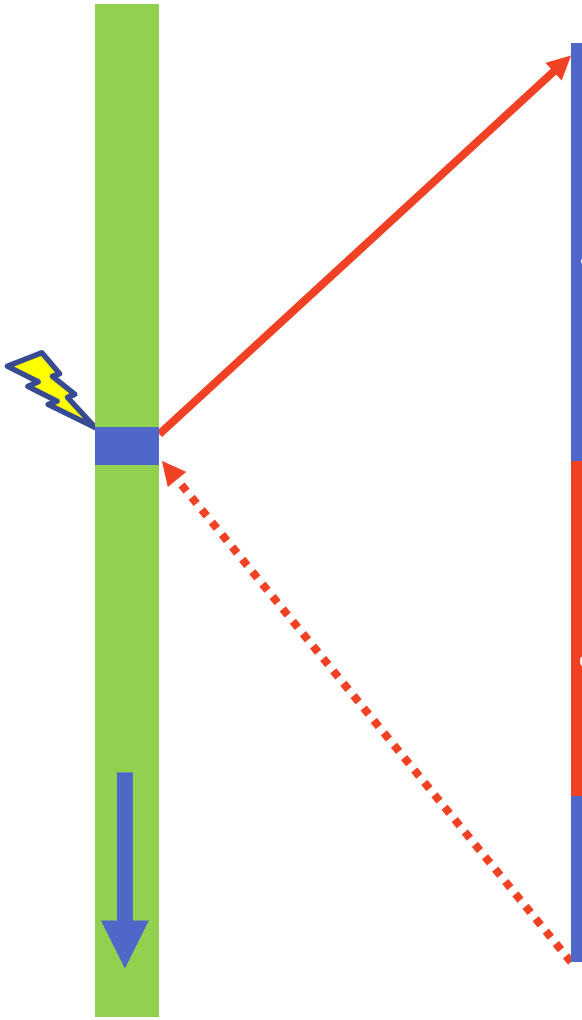
** different numbers in Thumb instruction mode

Context change, schematic



*Processor Status Word

Exception handling on ARM



Hardware

Hardware action at entry (invoked by exception)

R14(exception_mode):= return link
SPSR(exception_mode) := CPSR
CPSR[4:0] := exception_mode number
CPSR[5] := 0 (* execute in ARM state *)
If exception_mode = Reset or FIQ then CPSR[6]=1 (* disable fast IRQ *)
CPSR[7]=1 (* disable normal interrupts *)
PC=exception vector address

Software

```
STMDB SP!, {R0 .. R11, FP, LR} (* store all non-banked registers on stack *)

... (* exception handler *)

LDMIA SP! {R0..R11,FP,LR} (* read back all non-banked registers from stack*)
SUBS PC,LR, #ofs (* return from interrupt instruction *)
```

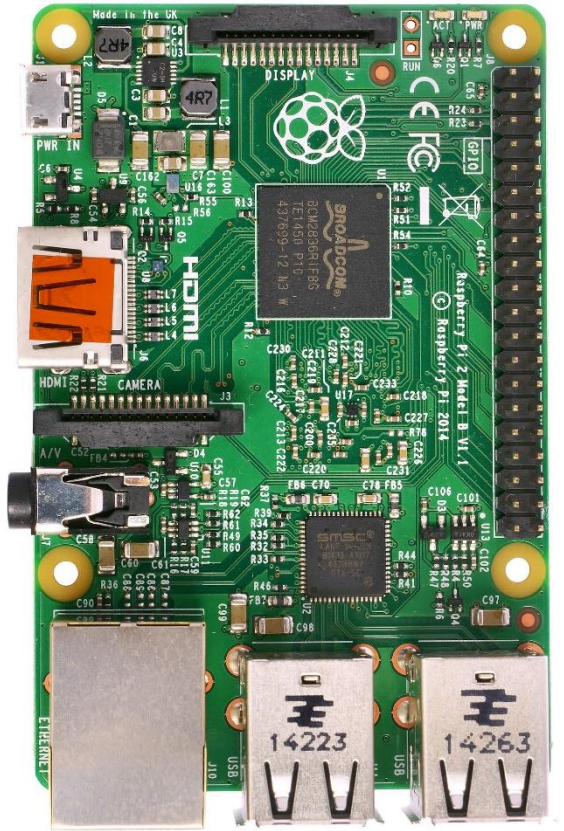
HW

Hardware action at exit (invoked by MOVS or SUBS instruction)

CPSR := SPSR(exception mode) (* includes a reset of the irq/fiq flag *)
PC := LR – ofs

Raspberry Pi 2

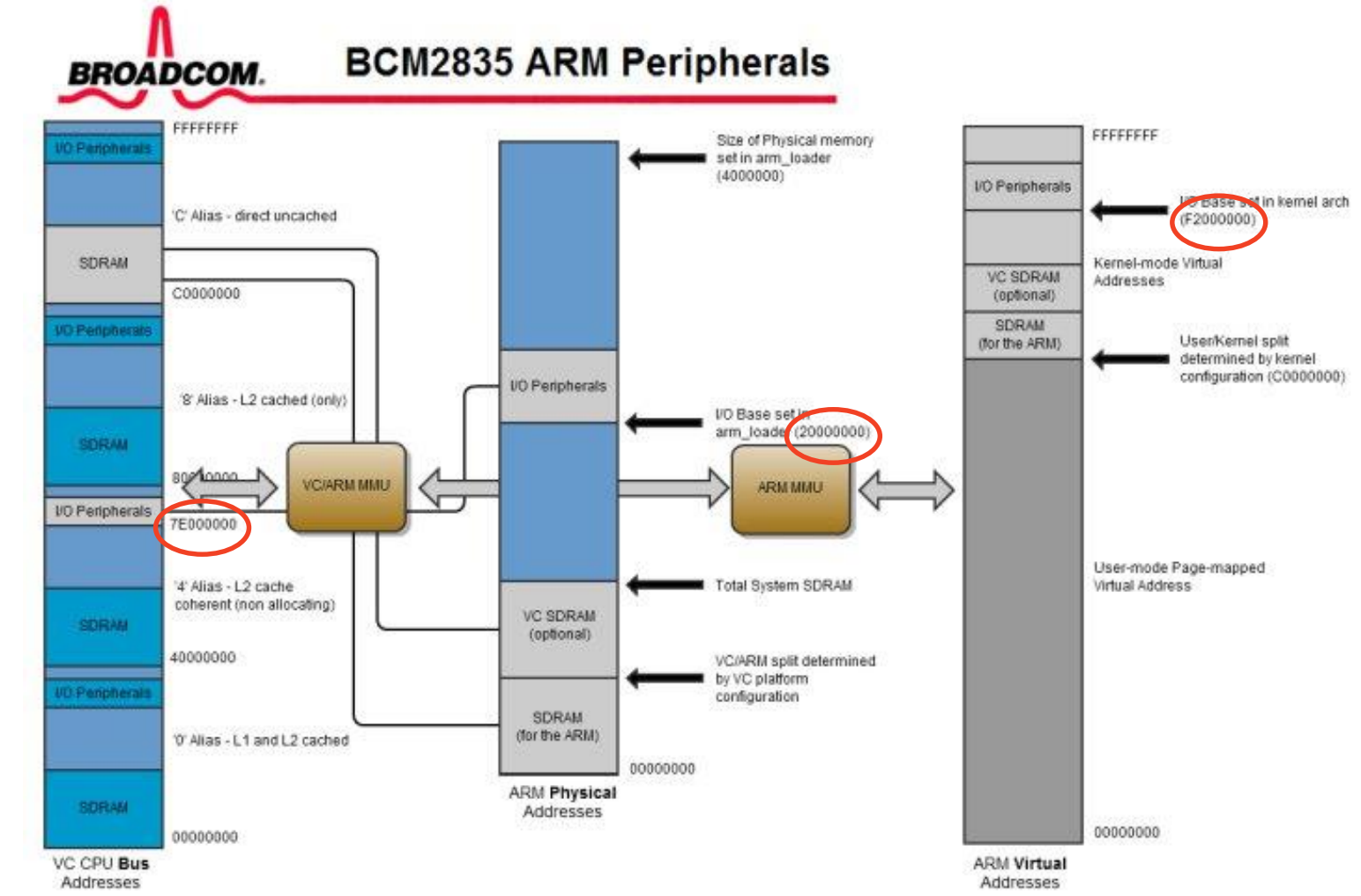
- Raspberry Pi 2 will be the hardware used at least in the first 4 weeks lab sessions
- Produced by element14 in the UK (www.element14.com)
- Features
  - Broadcom BCM2836 ARMv7
    Quad Core Processor running at 900 MHz
  - 1G RAM
  - 40 PIN GPIO
  - Separate GPU ("Videocore")
  - Peripherals: UART, 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 beginning.
- Only one core runs at that time. (More on this later)

RPI 1 Memory Map



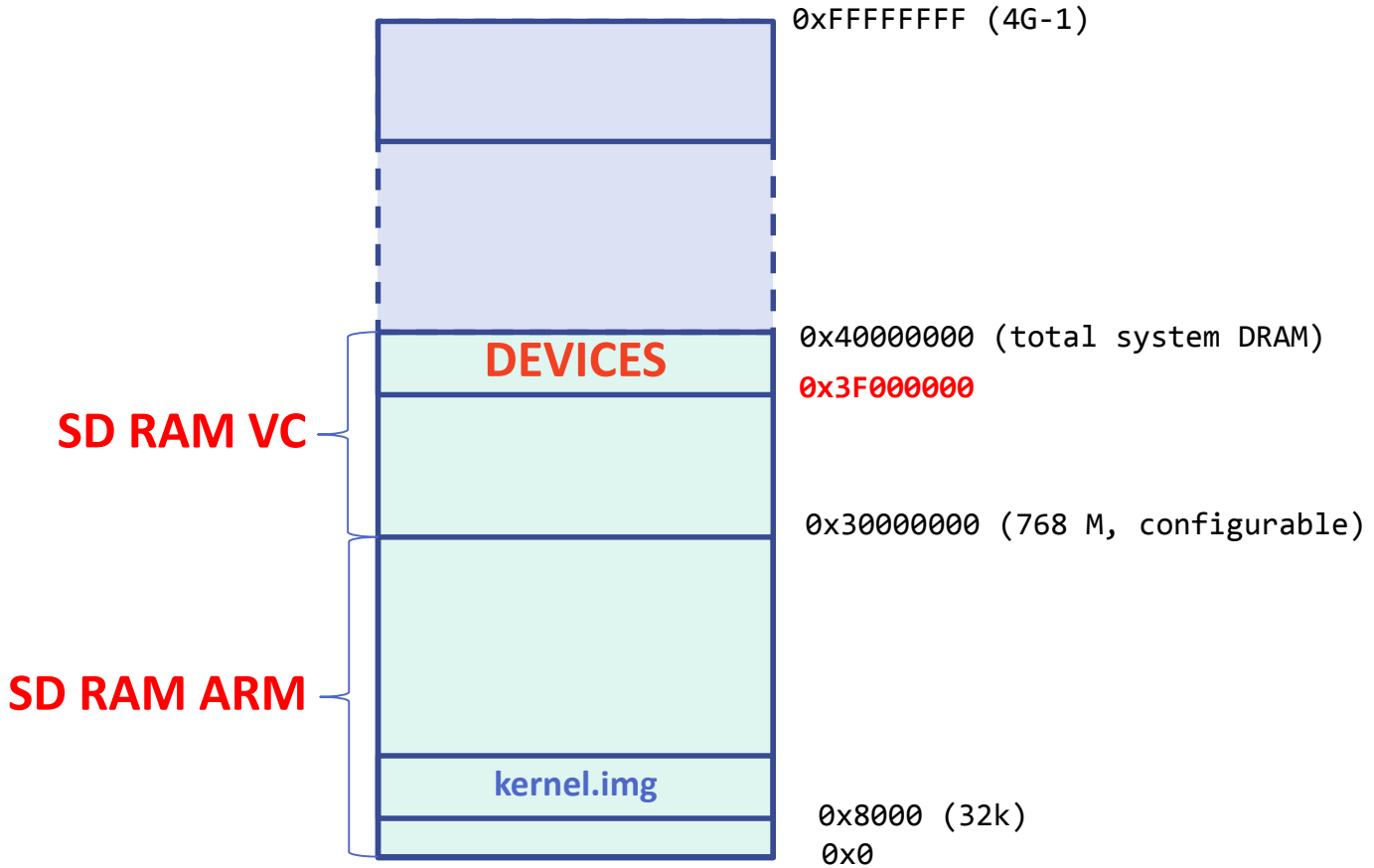
VC Virtual

ARM Physical

Linux Virtual

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

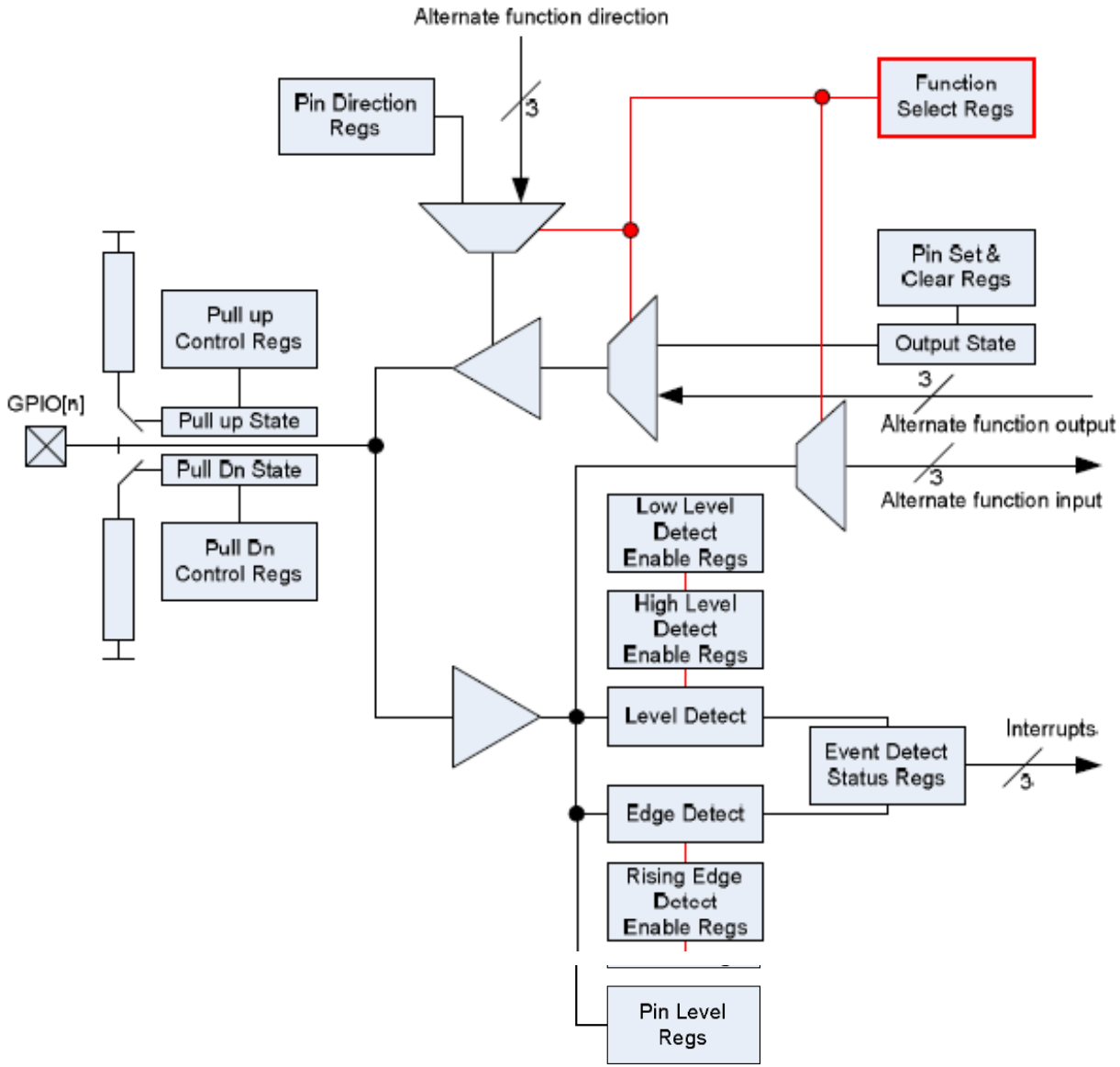


General Purpose I/O (GPIO)

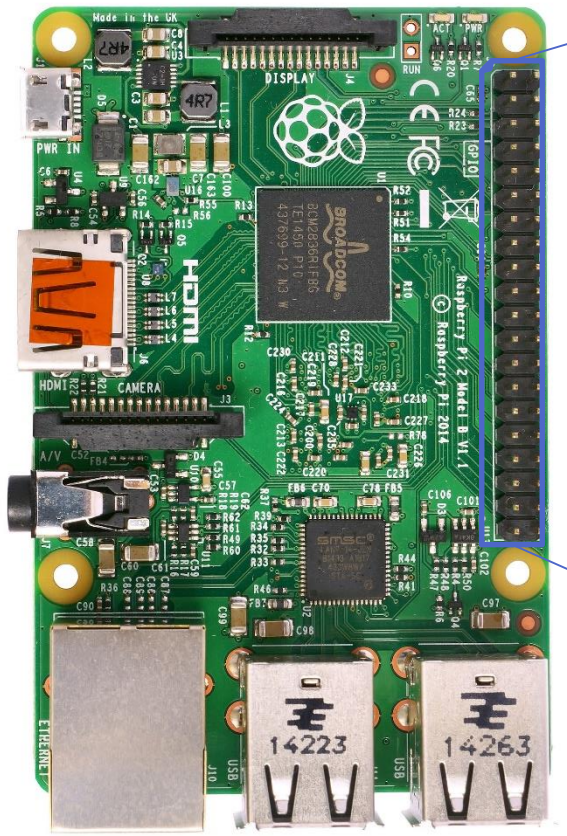
- Software controlled processor pins
  - Configurable direction of transfer
  - Configurable connection
    - → with internal controller (SPI, MMC, memory controller, …)
    - → with external device
- Pin state settable & gettable
  - High, low
- Forced interrupt on state change
  - On falling/ rising edge

GPIO

Block Diagram (BCM 2835)



Raspberry Pi 2 GPIO Pinout



name	pin		pin	name
3.3 V DC	01		02	DC power 5v
GPIO 02	03		04	DC power 5v
GPIO 03	05		06	ground
GPIO 04	07		08	GPIO 14
ground	09		10	GPIO 15
GPIO 17	11		12	GPIO 18
GPIO 27	13		14	ground
GPIO 22	15		16	GPIO 23
3.3V DC	17		18	GPIO 24
GPIO 10	19		20	ground
GPIO 09	21		22	GPIO 25
GPIO 11	23		24	GPIO 08
ground	25		26	GPIO 07
ID_SD	27		28	ID_SC
GPIO 05	29		30	ground
GPIO 06	31		32	GPIO 12
GPIO 13	33		34	ground
GPIO 19	35		36	GPIO 16
GPIO 26	37		38	GPIO 20
ground	39		40	GPIO 21

Documentation Examples

Address	Field Name	Description	Size	Read/ Write
~~0x 7E20 0000~~	~~GPFSEL0~~	~~GPIO Function Select 0~~	~~32~~	~~R/W~~
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1		
0x 7E20 0008	GPFSEL2	GPIO Function Select 2		
0x 7E20 000C	GPFSEL3	GPIO Function Select 3		
0x 7E20 0010	GPFSEL4	GPIO Function Select 4		

Bit(s)	Field Name	Description	Type	Reset
31-30	---	Reserved	R	0
29-27	FSEL19	FSEL19 - Function Select 19 000 = GPIO Pin 19 is an input 001 = GPIO Pin 19 is an output 100 = GPIO Pin 19 takes alternate function 0 101 = GPIO Pin 19 takes alternate function 1 110 = GPIO Pin 19 takes alternate function 2 111 = GPIO Pin 19 takes alternate function 3 011 = GPIO Pin 19 takes alternate function 4 010 = GPIO Pin 19 takes alternate function 5	R/W	0
26-24	FSEL18	FSEL18 - Function Select 18	R/W	0
8-6	FSEL12	FSEL12 - Function Select 12	R/W	0
5-3	FSEL11	FSEL11 - Function Select 11	R/W	0
2-0	FSEL10	FSEL10 - Function Select 10	R/W	0

GPIO14	Low	TXD0	SD6	<reserved>			TXD1
GPIO15	Low	RXD0	SD7	<reserved>			RXD1
GPIO16	Low	<reserved>	SD8	<reserved>	CTS0	SPI1_CE2_N	CTS1

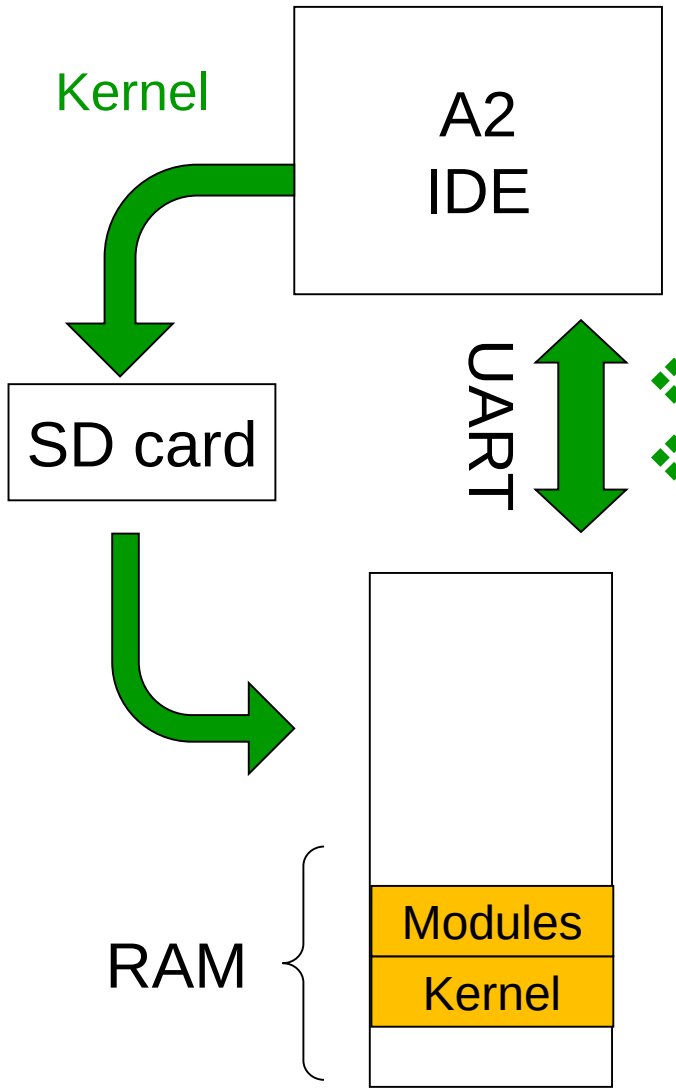
Table 6-3 – GPIO Alternate function select register 1

How to Cross-Develop and Build a System

1.2. CROSS DEVELOPMENT

Cross Development Platform

used in the Exercises



- Program Editor
- Cross Compiler
- Boot Linker
- Dialog

Developer Software

- Download Modules
- Interprete Commands

- Boot from SD Card (VC / ARM on RPI)
- Kernel
- Command Interpreter
- Log

Target Software

Programming Language Oberon

- Pascal family
- Modular with separate compilation
- Strongly typed
  - Static type checking at compile time
  - Runtime (dynamic) support for type guards / tests
- Consequently high level
  - Minimal assembler code (we will use some in the first exercise)
  - Specific low level functions in a Pseudo-Module called SYSTEM

Oberon07

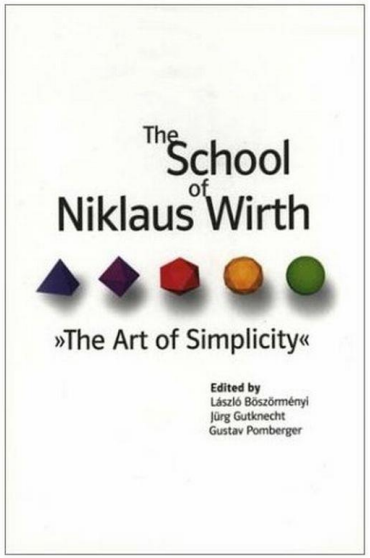
Dialect of Oberon

- Minimal
- Specifically designed for one-pass compilers
  Processor specific functions
- Interrupt procedures
- Pragmatic, predefined functions
- No type OBJECT*, no methods

The compiler used in this course implements Oberon07 as a subset.
Less restrictions apply.

*as opposed to Active Oberon

The art of simplicity



- Most recent Compilers by Prof. N. Wirth

part	size in lines of code
scanner:	300
parser/driver:	1000
types/symbols:	500
generator	1400
	--------
	ca 3k

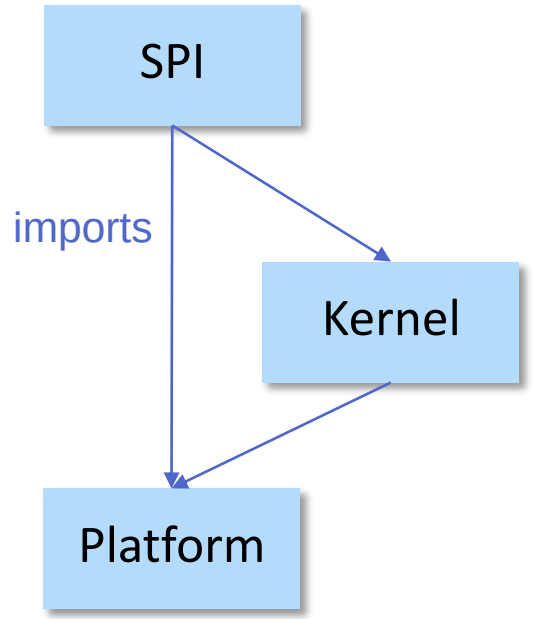
- Fox Compiler, used in the exercises (including all backends and various dialects) ca. 50k lines of code
- gcc / llvm : Millions of lines of code

Example of a Module

```
MODULE SPI; (* Raspberry Pi 2 SPI Interface – Bitbanging *)
IMPORT Platform, Kernel;

CONST HalfClock = 100; (* microseconds -- very conservative*)

PROCEDURE SetGPIOs;
BEGIN
   Platform.ClearAndSetBits(Platform.GPFSEL0, {21..29},{21,24});
   Platform.ClearAndSetBits(Platform.GPFSEL1, {0..5},{0,3});
END SetGPIOs;

PROCEDURE Write* (CONST a: ARRAY OF CHAR);
VAR i: LONGINT;
BEGIN
   Kernel.MicroWait(HalfClock);
   Platform.WriteBits(Platform.GPCLR0, SELECT); (* signal select *)
   Kernel.MicroWait(HalfClock);
   FOR i := 0 TO LEN(a)-1 DO
      WriteByte(a[i]);   (* write data, toggling the clock *)
   END;
   Kernel.MicroWait(HalfClock);
   Platform.WriteBits(Platform.GPSET0, SELECT); (* signal deselect *)
END Write;
...

BEGIN
   SetGPIOs;
END SPI;
```

exported procedure: can be used by importing modules

module body: executed first -- and only once -- when module is loaded



Example of a Module

```
MODULE Timer;

IMPORT Kernel,Out;

VAR global: LONGINT; factor: REAL;

    PROCEDURE Start*(VAR ticks: LONGINT);
    BEGIN time := Kernel.GetTicks();
    END Start;

    PROCEDURE Step*(VAR ticks: LONGINT): REAL;
    VAR previous: LONGINT;
    BEGIN previous := ticks; ticks :=  Kernel.GetTicks(); RETURN (ticks-previous)*factor
    END Step;

    PROCEDURE Tick*; BEGIN Start(global); END Tick;

    PROCEDURE Tock*;
    BEGIN Out.String("elapsed seconds: "); Out.Real(Step(global),20); Out.Ln;
    END Tock;

    PROCEDURE Calibrate; BEGIN … END Calibrate;

BEGIN Calibrate();
END Timer.
```

global symbols (variables) in module context

exported procedure without parameters: can be used as command

Inline-Assembly within Modules

```
MODULE MinimalLED;

IMPORT SYSTEM;

PROCEDURE {INITIAL, NOPAF} Entry;
CODE
    ldr r0, [pc, #someNumber - $ - 8]
    mov r1, #0x30
    b end
    someNumber: d32 0x3f000000
    end:
END Entry;

PROCEDURE {FINAL, NOPAF} Exit;
CODE
    end:
    b end
END Exit;

END MinimalLED.
```

Low-level access from Oberon without Assembly

```
IMPORT SYSTEM;


PROCEDURE LetThereBeLight;
CONST GPSET0 = 03F20001CH;
BEGIN
    SYSTEM.PUT(GPSET0, {21});
END LetThereBeLight;
```

SYSTEM.PUT: write to address

Control Structures

- IF

```
IF a = 0 THEN
        (* statement sequence *)
END
```

- WHILE

```
WHILE x<n DO
        (* statement sequence *)
END
```

- REPEAT

```
REPEAT
        (* statement sequence *)
UNTIL x=n;
```

- FOR

```
FOR i := 0 TO 100 DO
        (* statement seq *)
END;
```

Builtin Types

- **BOOLEAN**
  ```
  b := TRUE; IF b THEN END;
  ```
- **CHAR**
  ```
  c := 'a'; c := 0AX;
  ```
- **SHORTINT ⊂ INTEGER ⊂ LONGINT ⊂ HUGEINT**
  ```
  i := SHORT(s); l := 10; h := 010H;
  ```
- **REAL ⊂ LONGREAL**
  ```
  r := 1.0; r := 10E0; d := 1.0D2;
  ```
- **SET**
  ```
  s := {1,2,3}; s := s + {5}: s := s - {5}; s := s *
  {1..6};
  ```
- **ADDRESS, SIZE**

Builtin Functions

- **Increment and decrement**
  `INC(x); DEC(x); INC(x,n); DEC(x,n);`
- **Sets**
  `INCL(set, element); EXCL(set, element);`
- **Assert and Halt**
  `ASSERT(b<0); HALT(100);`
- **Allocation**
  `NEW(x, …);`
- **Shifts**
  `ASH(x,y); LSH(x,y); ROT(x,y);`
- **Conversion**
  `SHORT(x); LONG(x); ORD(ch); CHR(i); ENTIER(r);`
- **Arrays**
  `LEN(x); LEN(x,y); DIM(t);`
- **Misc**
  `ABS(x); MAX(type); MIN(type); ODD(i);`

Oberon Language Features (1)

- Program units
  - MODULE, PROCEDURE (Value, VAR and CONST parameters)
- Data types
  - BOOLEAN, CHAR, SHORTINT, INTEGER, LONGINT, HUGEINT, REAL, LONGREAL, SET, ADDRESS, SIZE
- Structured types
  - ARRAY, RECORD (with type extension), POINTER TO ARRAY, POINTER TO RECORD
- Statements
  - ProcedureCall, Assignments, IF, WHILE, REPEAT, LOOP/EXIT, FOR, CASE, WITH, AWAIT, RETURN, BEGIN ... END

Oberon Predefined Functions

Global

- Functions
  - `ODD(i), ABS(x), LSH(x,n), ASH(x,n), ROT(x,n)`
  - `LEN(a), CAP(c), MAX(t), MIN(t)`
  - `SIZE OF t, SIZEOF(t)`
  - `ADDRESS OF x, ADDRESSOF(x)`
- Type Conversion
  - `CHR(i), ORD(i), ENTIER(f)`
- Proper Procedures
  - `NEW (ptr)`
  - `INC(i), INC(i,n), DEC(i), DEC(i,n)`
  - `ASSERT(b), ASSERT(b,n)`

Predefined Functions

Pseudo Module SYSTEM

- Functions
  - `ARM-specific`
    - `SYSTEM.SP(), SYSTEM.FP(), SYSTEM.LNK()`
- Procedures
  - `SYSTEM.PUT (a, x), SYSTEM.GET (a, x)`
  - ARM-specific
    - `SYSTEM.LDPSR(b,x), SYSTEM.STPSR(b,x)`
    - `SYSTEM.LDCPR(a,b,c), SYSTEM.STCPR(a,b,c), SYSTEM.FLUSH(x)`
    - `SYSTEM.SETSP(x), SYSTEM.SETFP(x)`
- Data Type
  - `SYSTEM.BYTE`

Interrupt Procedures

```
PROCEDURE Handler {INTERRUPT, PCOFFSET=k};
BEGIN (* k is the offset to the next instruction
        cf. table of exceptions *)
END Handler;
```

Special System's Programming Flags and Features

- `PROCEDURE {NOTAG}`
  - Procedure without procedure activation flag
- `PROCEDURE {INITIAL}`
  - Procedure that is linked to the beginning of a kernel
- `CODE … END`
  - special statement block that can contain inline assembler code
- `symbol {ALIGNED(32)}`
  - alignment of a symbol (e.g. variable)
- `symbox {FIXED(0x8000))`
  - pinning of a symbol

Special System's Programming Flags and Features

- `POINTER {UNSAFE} TO ...`
  - Unsafe pointer that is assignment compatible with type ADDRESS
- symbol {UNTRACED}
  - Symbol that is invisible to a Garbage Collector

System Programming with Oberon

Bits

```
PROCEDURE EnableIRQs*;
VAR cpsr: SET;
BEGIN SYSTEM.STPSR( 0, cpsr );
      cpsr := cpsr - {7};
      SYSTEM.LDPSR( 0, cpsr );
END EnableIRQs;
```

- Use built-in type `SET` for bitsets
  - ```
    VAR s: SET;
    INCL(s, 3);   -- include bit 3 in s
    EXCL(s, 4);   -- exclude bit 4 from s
    s := {0,2,5}; -- s consisting of bits 0, 2 and 5 (int value 37)
    s := s + {1,3,5}; -- include bits 1,3,5 in s
    s := s – {1,2,3}; -- exclude bits 1,2,3 from s
    ```
- and / or arithmetic operations and `ODD`
  - ```
    VAR i: LONGINT;
    i := i DIV 10H; -- shift to right by 4
    i := i MOD 10H; -- and with 0FH
    IF ODD(i) THEN – test if bit 0 is set
    i DIV 10000H MOD 100H; --  extract bits 20..27 from i
    ```

System Programming with Oberon

Bytes

- `SYSTEM.PUT32` can be used to write four bytes to a specific memory location
  `SYSTEM.PUT32(Platform.ICMR, s)`
- `SYSTEM.GET` can be used to read from a memory location
  `SYSTEM.GET(Platform.CKEN, reg);`
- `SYSTEM.VAL` can be used for type-cast (e.g. set -> integer)
  `i := SYSTEM.VAL(LONGINT, s);`
- `SYSTEM.BYTE` can be used to transfer generic data

```
PROCEDURE ReadBytes( dev: Device; VAR buf: ARRAY OF SYSTEM.BYTE;
                     offset, len: INTEGER ); …
```