

# System Construction

Autumn Semester 2016

ETH Zürich

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.**

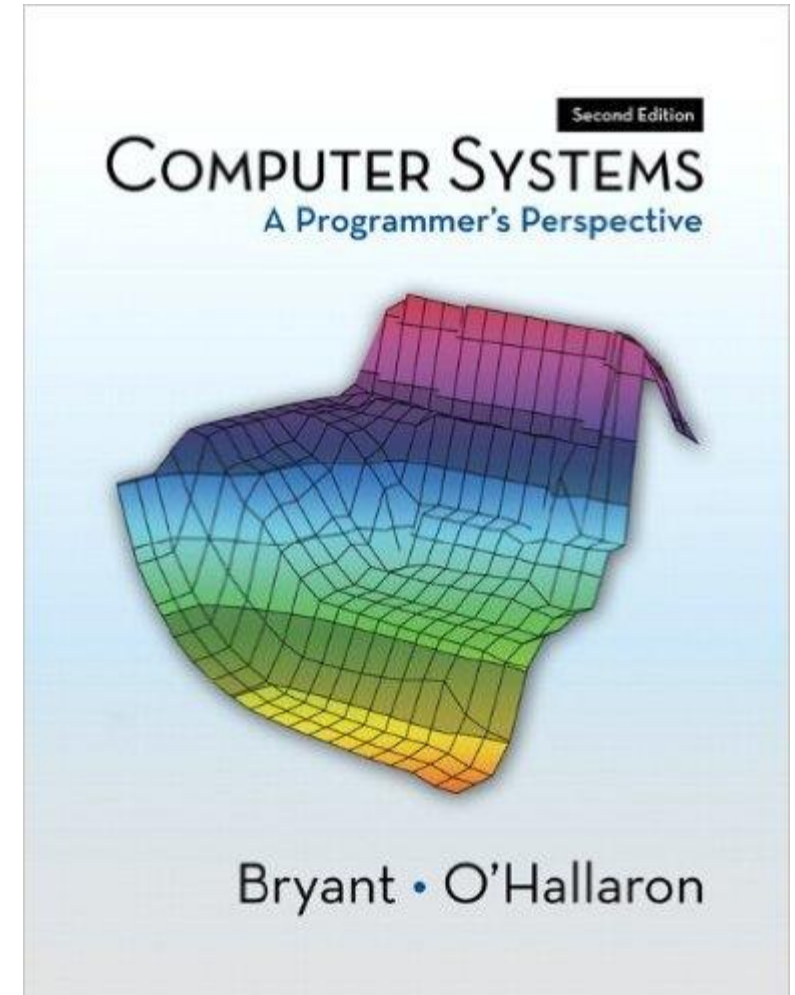
A lot of this course is about **bare metal programming.**

# 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

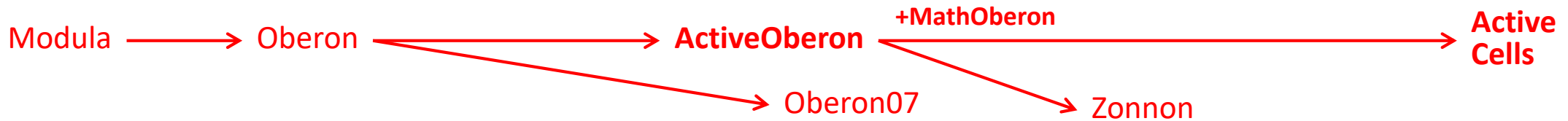
<https://svn.inf.ethz.ch/svn/lecturers/vorlesungen/trunk/syscon/2016/shared>

- Links on the course homepage

<http://lec.inf.ethz.ch/syscon/2016>

# Background: Co-Design @ ETH

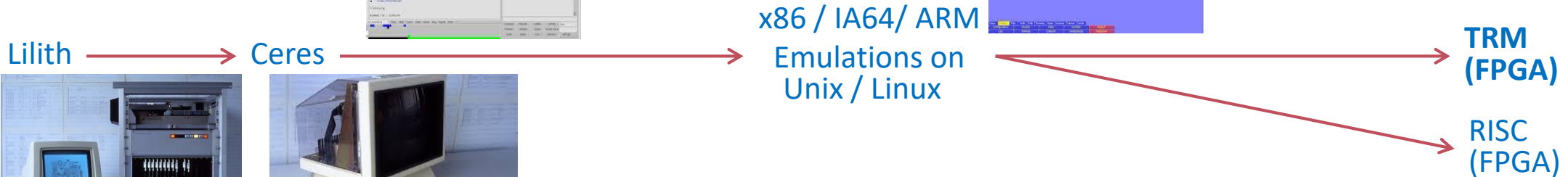
## Languages (Pascal Family)



## Operating / Runtime Systems



## Hardware



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.
- 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.

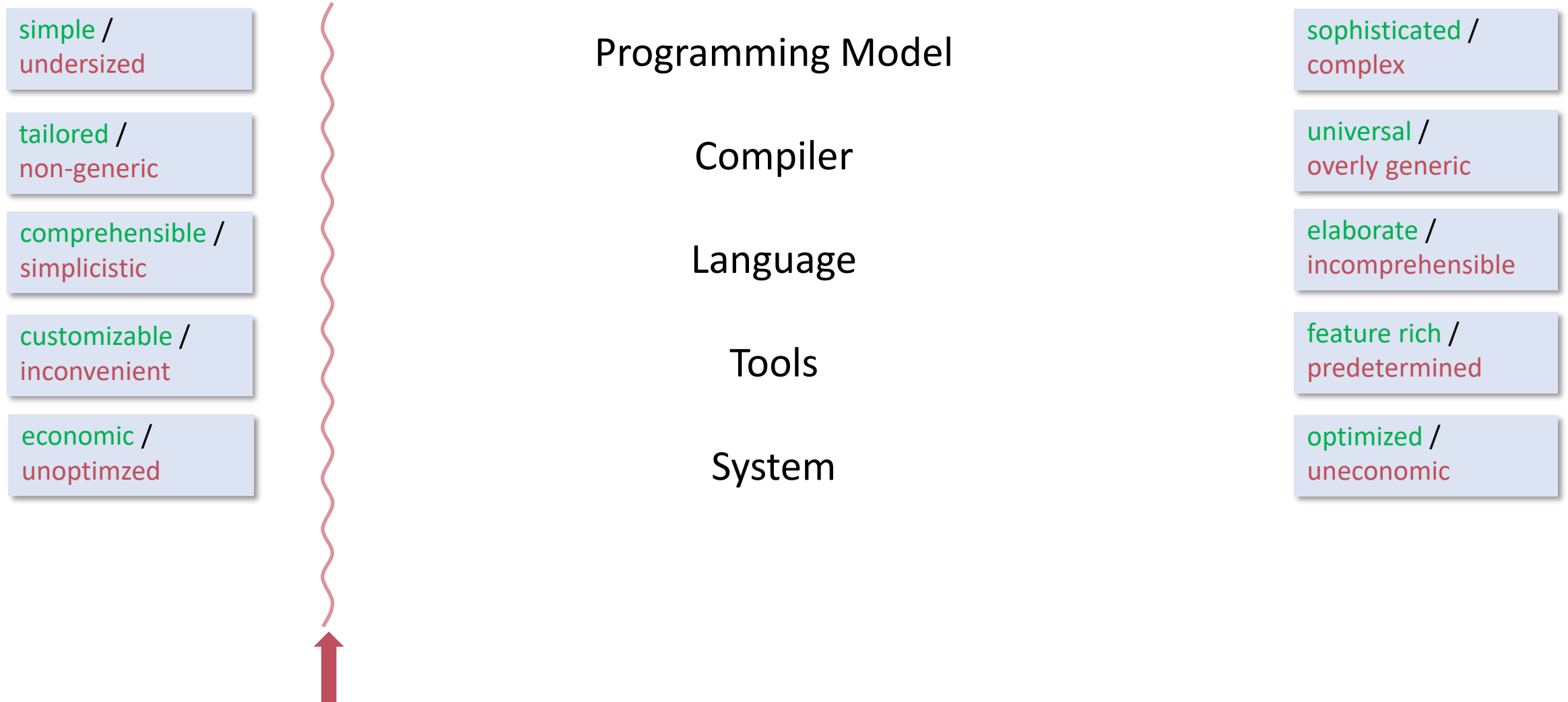
### Case Study 4. Active Cells: Multi-Processor System

- 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



I am about here

**Minimal Operating System**

# **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

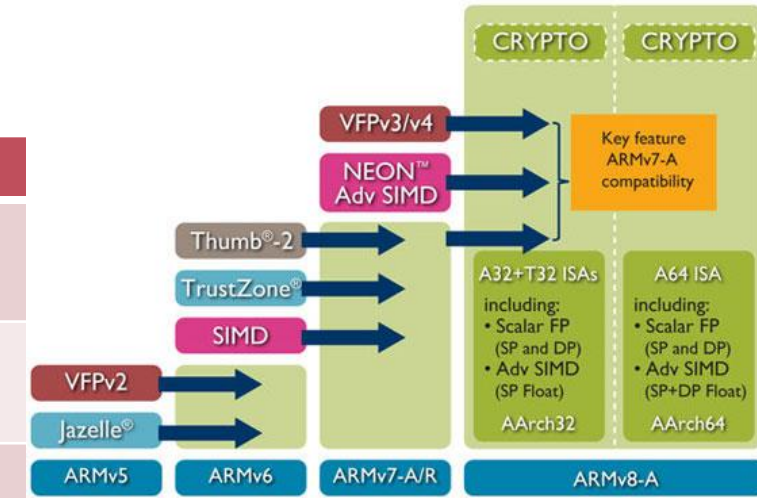


\*[http://news.cnet.com/ARMed-for-the-living-room/2100-1006\\_3-6056729.html](http://news.cnet.com/ARMed-for-the-living-room/2100-1006_3-6056729.html)

\*<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*

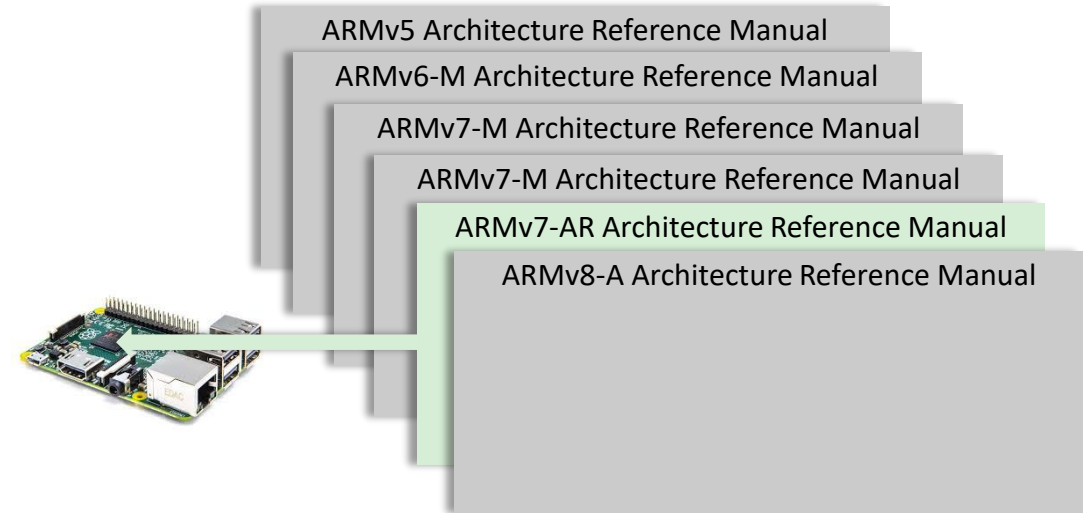
very much simplified & sparse

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



Cortex™-A7 MPCore™  
Technical Reference Manual

- 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.



**BCM2835 ARM Peripherals**

# 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)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0													
Data processing immediate shift	cond [1]	0	0	0	opcode			S	Rn			Rd			shift amount			shift	0	Rm																									
Miscellaneous instructions: See Figure 3-3	cond [1]	0	0	0	1	0	x	x	0	x																	0	x				x	x	x											
Data processing register shift [2]	cond [1]	0	0	0	opcode			S	Rn			Rd			Rs			0	shift	1	Rm																								
Miscellaneous instructions: See Figure 3-3	cond [1]	0	0	0	1	0	x	x	0	x																	0	x	x	1	x				x	x	x								
Multiplies, extra load/stores: See Figure 3-2	cond [1]	0	0	0	x			x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	x	x	1	x				x	x	x										
Data processing immediate [2]	cond [1]	0	0	1	opcode			S	Rn			Rd			rotate			immediate																											
Undefined instruction [3]	cond [1]	0	0	1	1	0	x	0	0	x																	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Move immediate to status register	cond [1]	0	0	1	1	0	R	1	0	Mask			SBO			rotate			immediate																										
Load/store immediate offset	cond [1]	0	1	0	P	U	B	W	L	Rn			Rd			immediate																													
Load/store register offset	cond [1]	0	1	1	P	U	B	W	L	Rn			Rd			shift amount			shift	0	Rm																								
Undefined instruction	cond [1]	0	1	1	x																	1	x																						
Undefined instruction [4,7]	1	1	1	1	0	x																	x																						
Load/store multiple	cond [1]	1	0	0	P	U	S	W	L	Rn			register list																																
Undefined instruction [4]	1	1	1	1	1	0	0	x																	x																				
Branch and branch with link	cond [1]	1	0	1	L	24-bit offset														x																									
Branch and branch with link and change to Thumb [4]	1	1	1	1	1	0	1	H	24-bit offset														x																						
Coprocessor load/store and double register transfers [6]	cond [5]	1	1	0	P	U	N	W	L	Rn			CRd			cp_num	8-bit offset																												
Coprocessor data processing	cond [5]	1	1	1	0	opcode1			CRn			CRd			cp_num	opcode2	0	CRm																											
Coprocessor register transfers	cond [5]	1	1	1	0	opcode1			L	CRn			Rd			cp_num	opcode2	1	CRm																										
Software interrupt	cond [1]	1	1	1	1	swi number														x																									
Undefined instruction [4]	1	1	1	1	1	1	1	x																	x																				

shiftable register

conditional execution

8 bit immediates with even rotate

load / store with destination increment

undefined instruction: user extensibility

load / store with multiple registers

branches with 24 bit offset

generic coprocessor instructions

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

# ARM Processor Modes

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

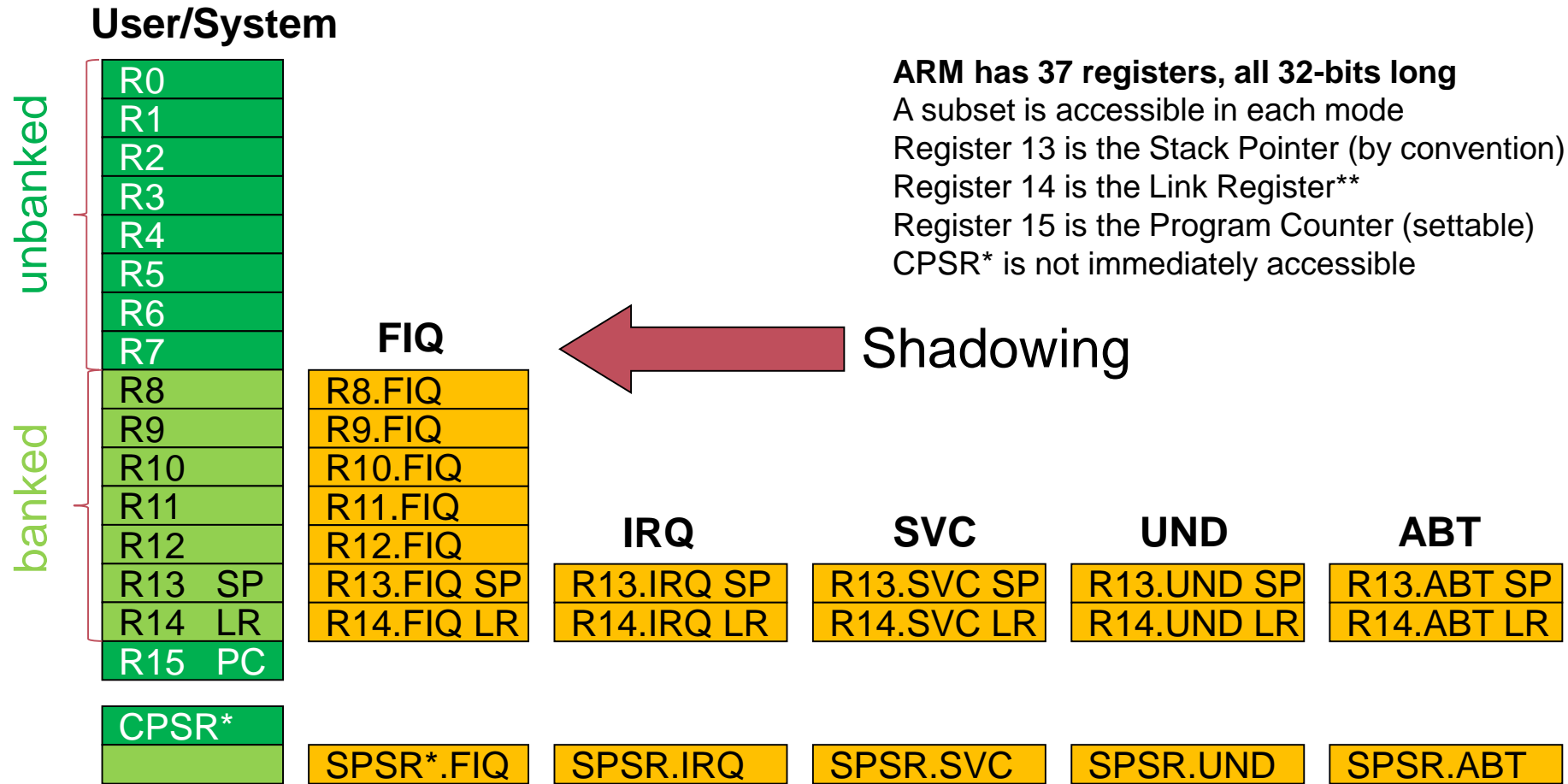
- Each mode has access to **own stack** and a different subset of registers
- Some operations can only be carried out in a privileged mode

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

Diagram annotations:

- A red bracket on the left groups the Supervisor, FIQ, IRQ, Abort, and Undef modes under the label "privileged".
- A yellow bracket on the right groups the Supervisor, FIQ, IRQ, Abort, and Undef modes under the label "exceptions".
- A green bracket on the right groups the System and User modes under the label "normal execution".

# ARM Register Set



\* 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 endianness
  - 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.

## 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 and jump back to caller address.

## Caller: cleanup parameters from stack

...

`bl #address`

`stmdb sp!, {fp, lr}`

`mov fp, sp`

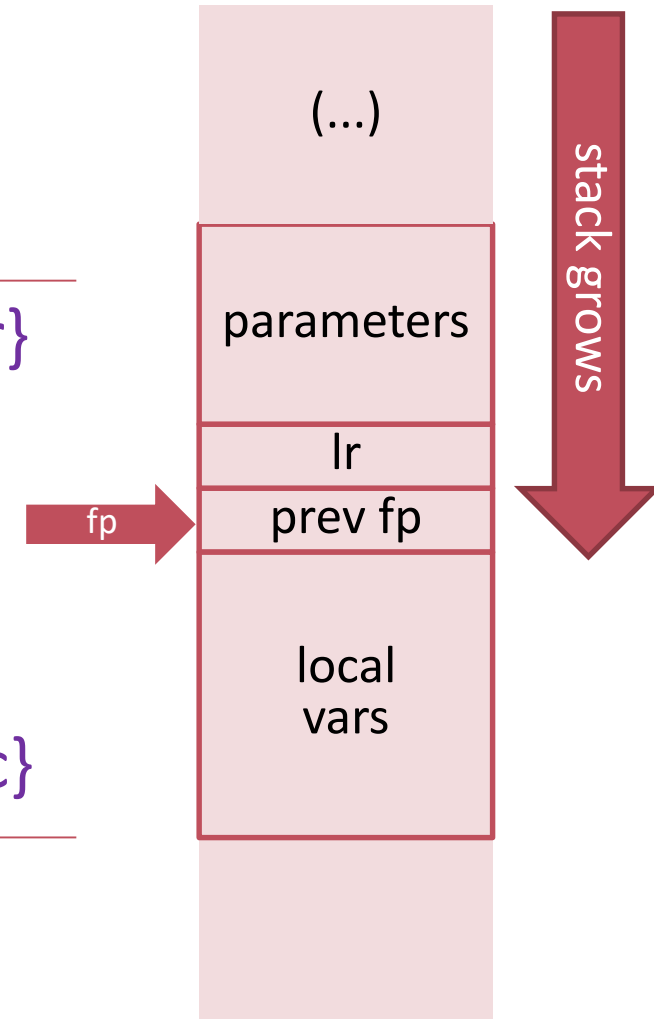
...

`mov sp, fp`

`ldmia sp!, {fp, pc}`

`add sp, sp, #n`

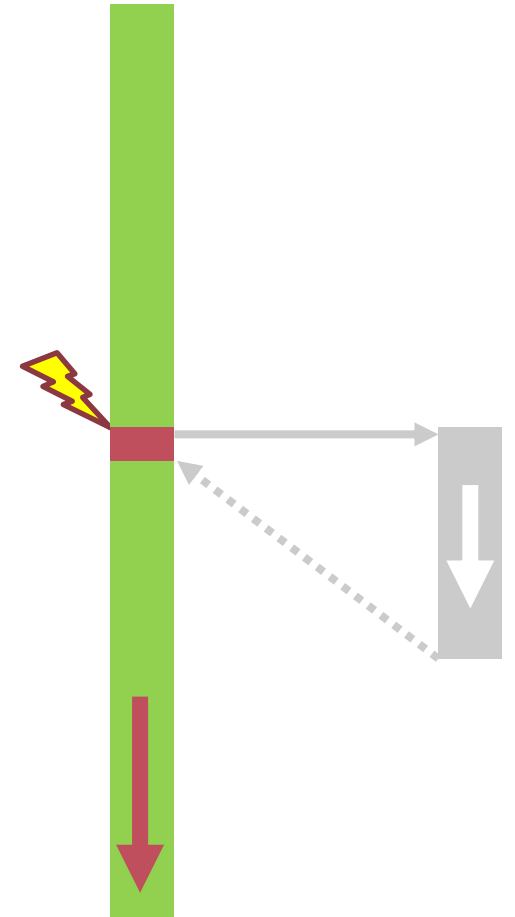
...



# 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 35

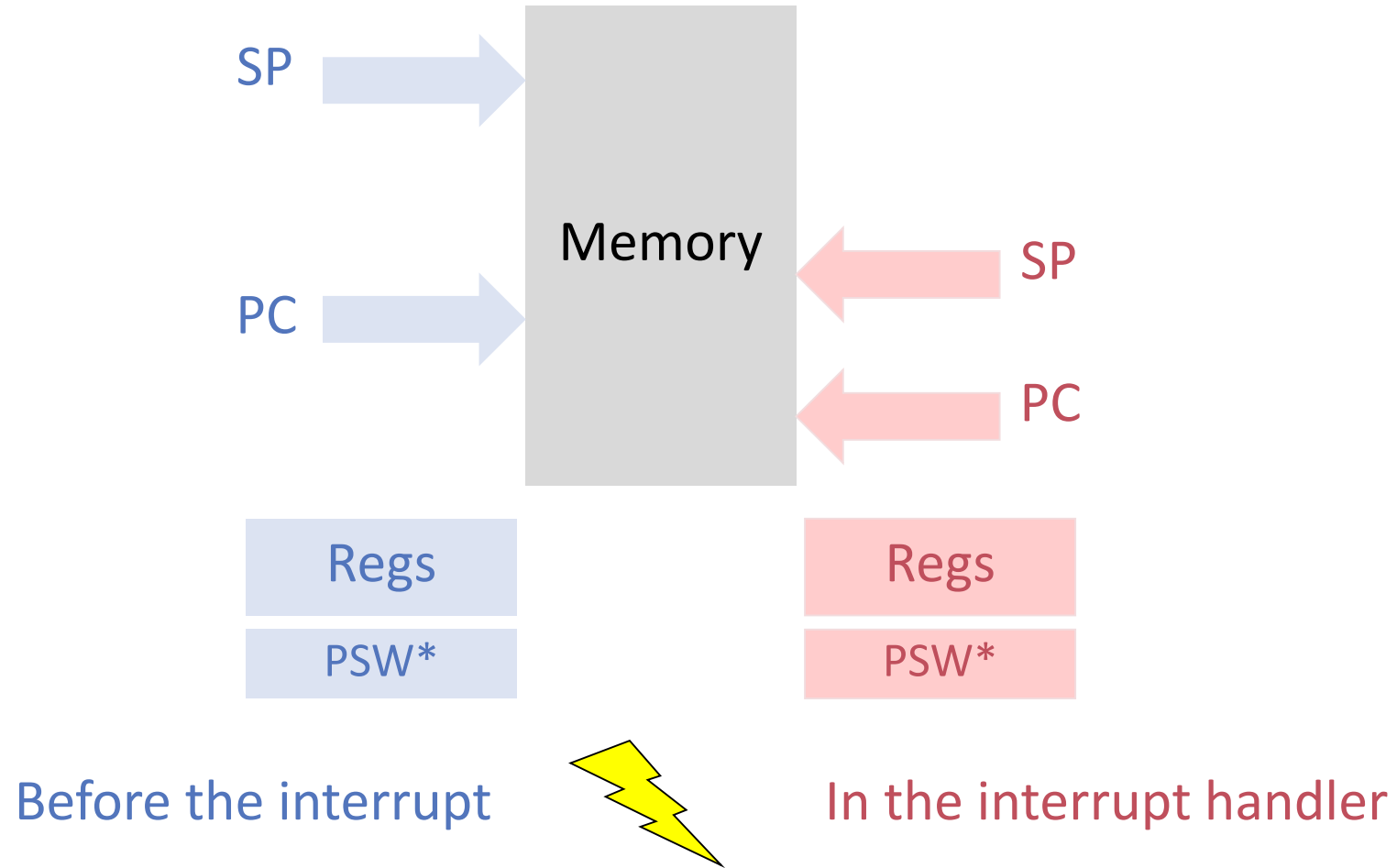
# Exception Table on ARM

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

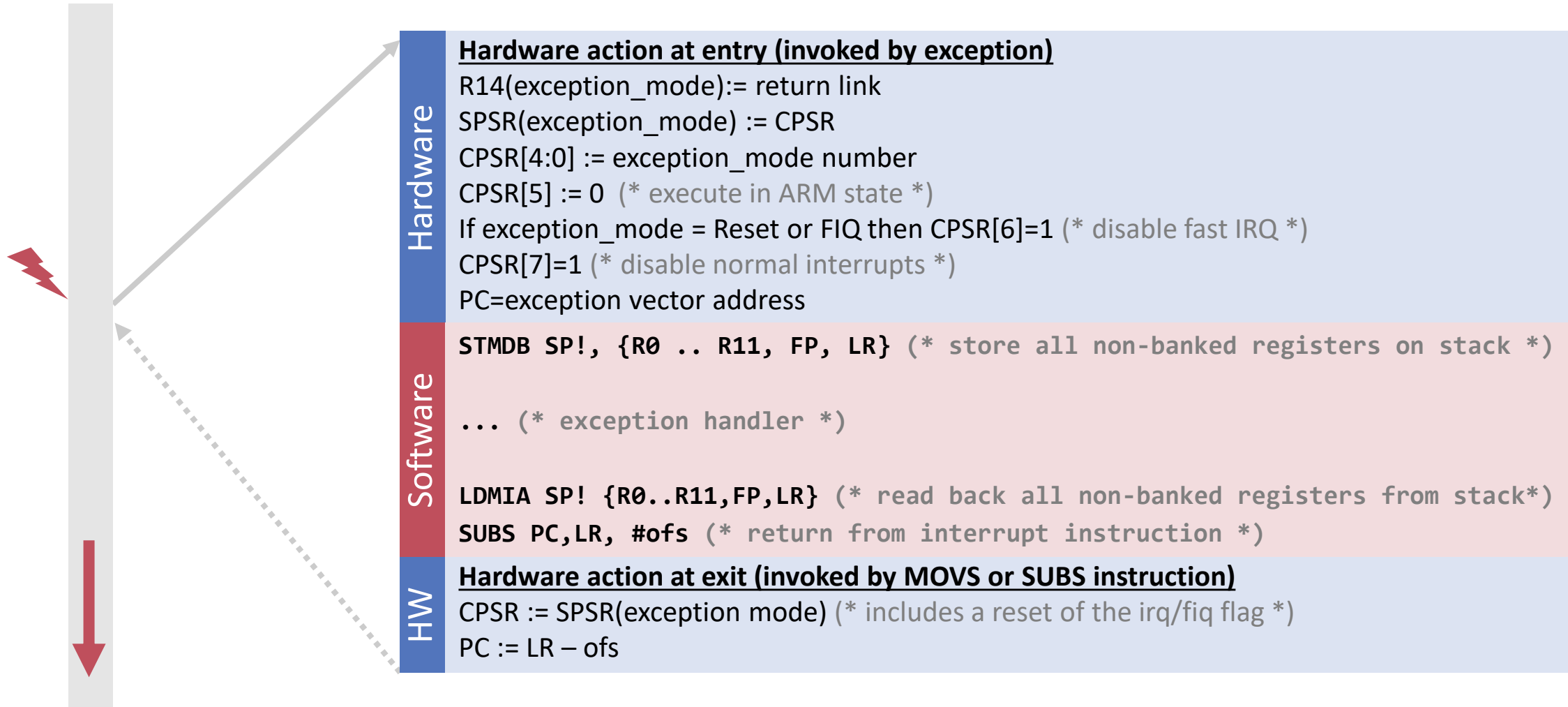
\* alternatively High Vector Address =  $0\text{x}\text{FFFF}0000 + \text{adr}$  (configurable)

\*\* different numbers in Thumb instruction mode

# Context change, schematic

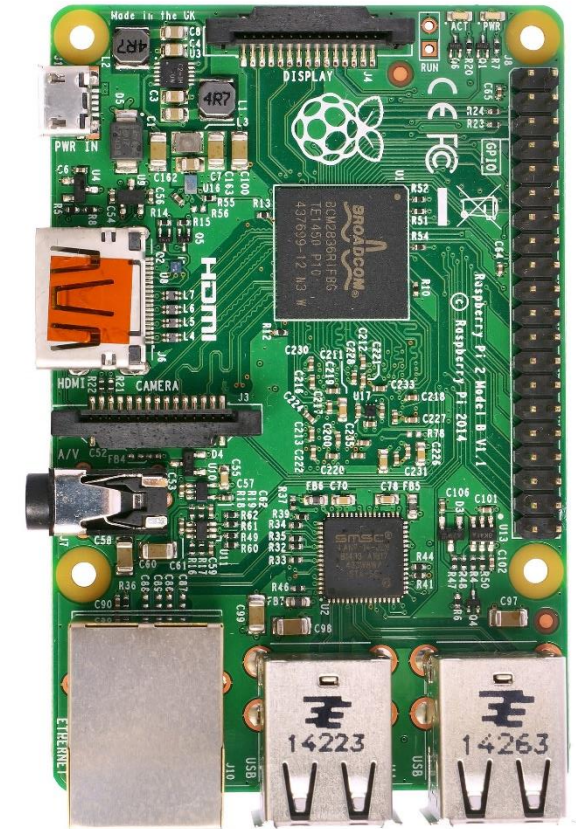


# Exception handling on ARM



# 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](http://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

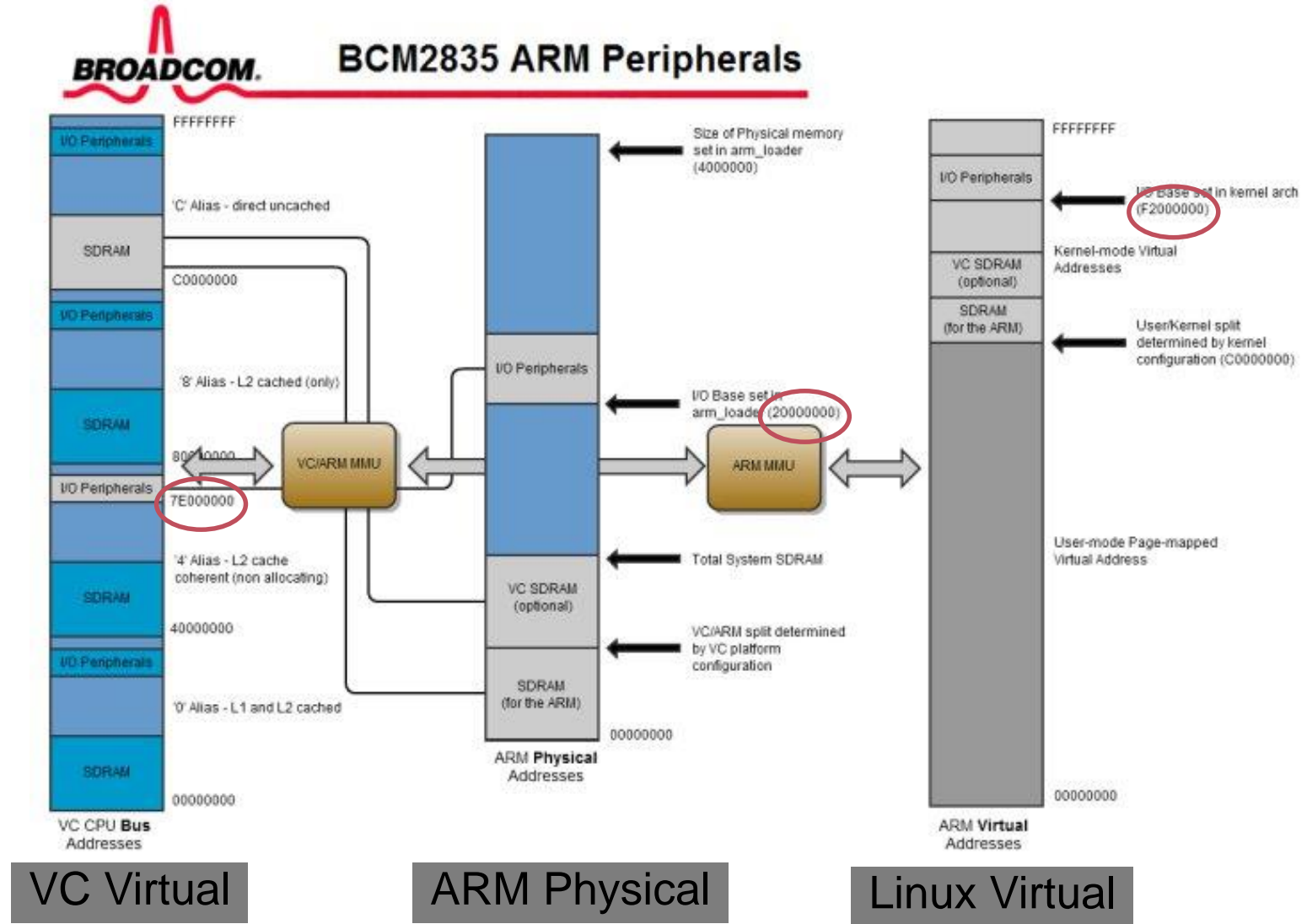


# ARM System Boot

- 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)

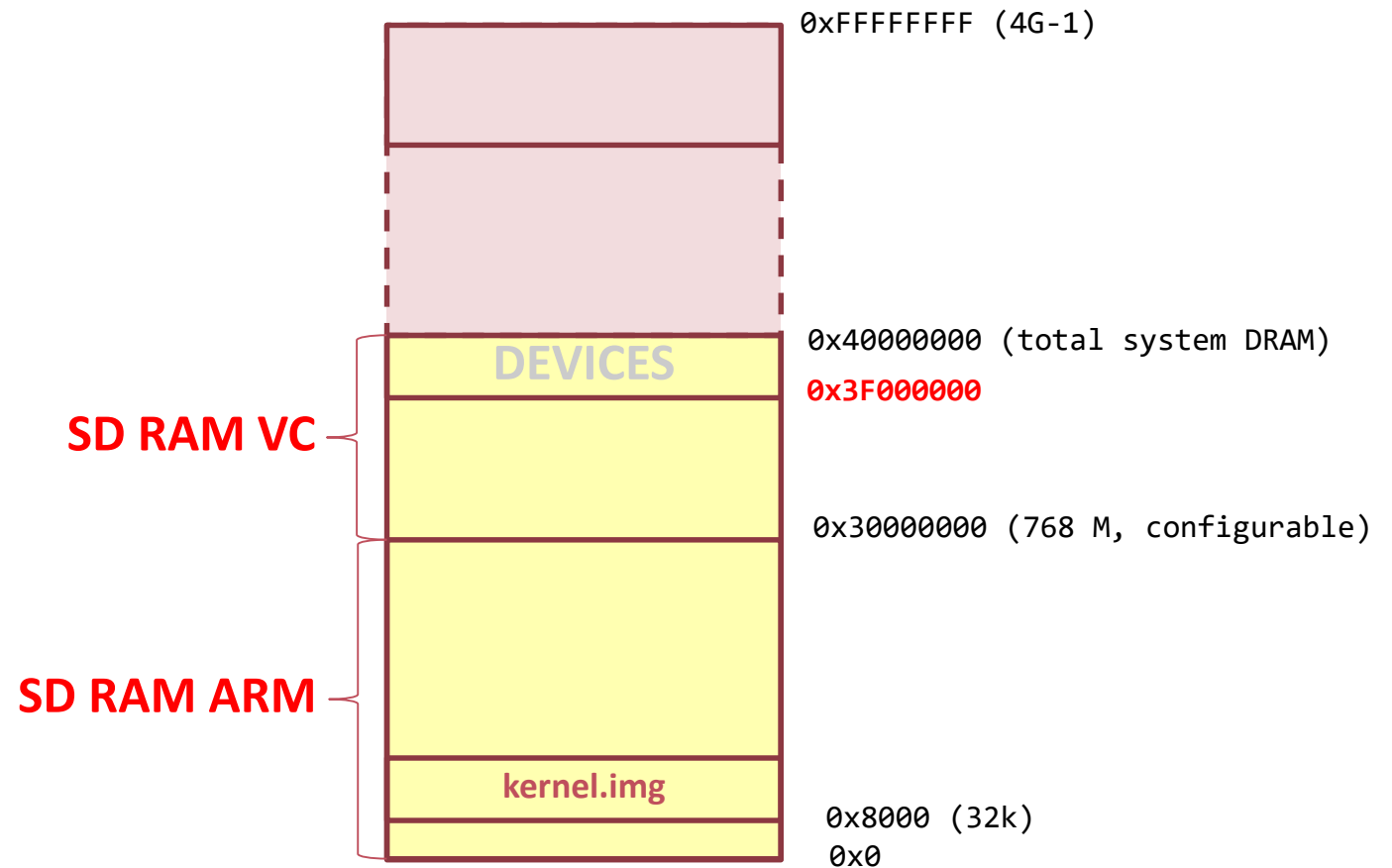


# RPI 1 Memory Map



# 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

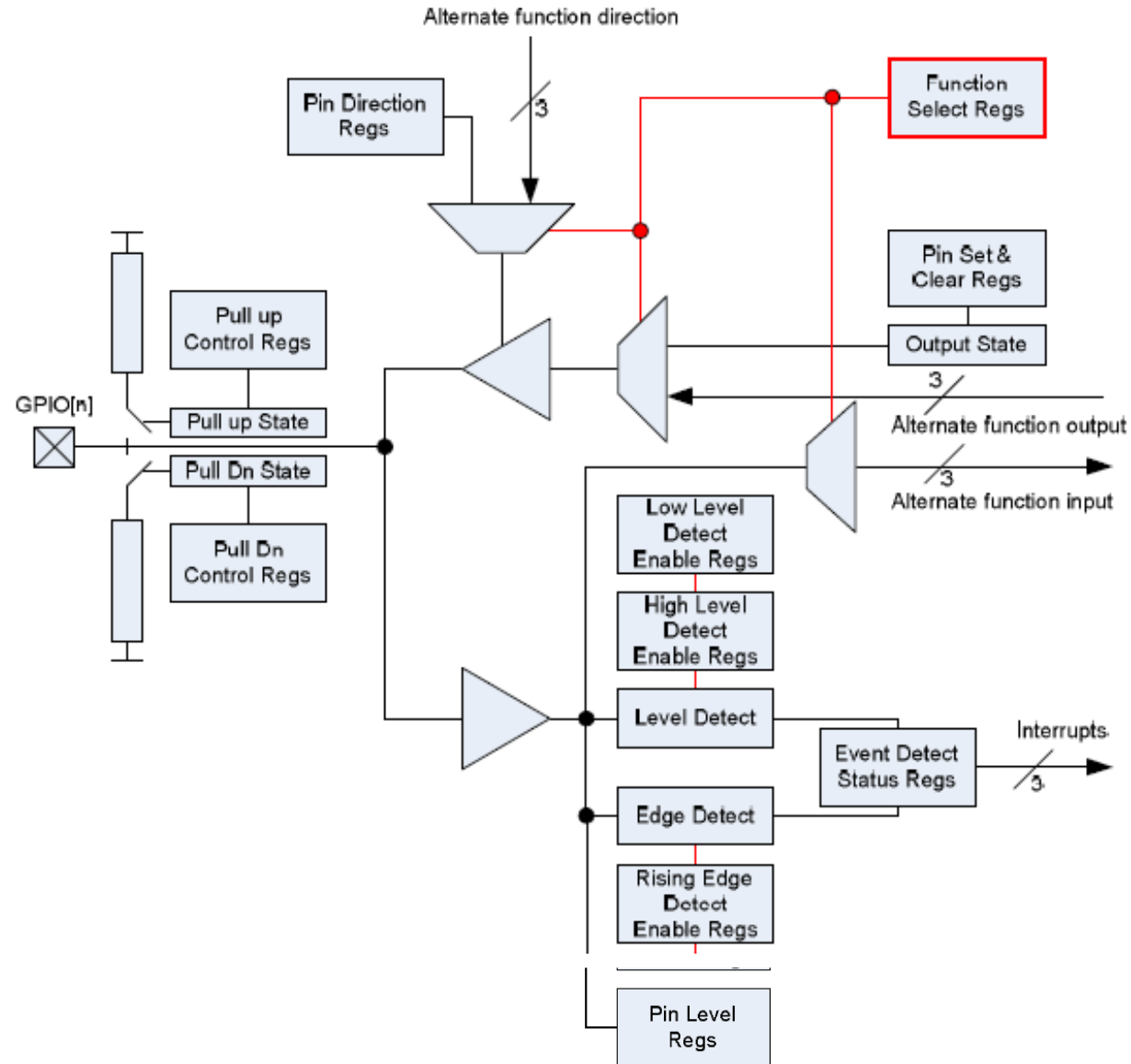


# General Purpose I/O (GPIO)

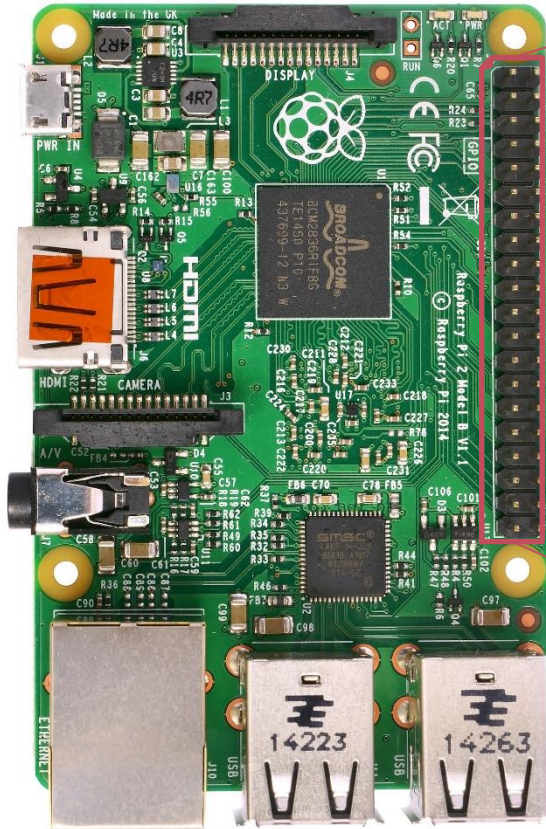
- Software controlled processor pins
  - Configurable direction of transfer
  - Configurable connection
    - with internal controller (SPI, MMC, memory controller, ...)
    - with external device
- Pin state settable & gettable
  - High, low
- Forced interrupt on state change
  - On falling/ rising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

## GPIO Register Overview

Address	Field Name	Description	Size	Read/Write
<del>0x 7E20 0000</del>	<del>GPFSEL0</del>	<del>GPIO Function Select 0</del>	<del>32</del>	<del>R/W</del>
0x 7E20 0000	GPFSEL0	GPIO Function Select 0		
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		

## GPIO Function Select Register Definition

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	RW	0
	FSEL18	FSEL18 - Function Select 18	RW	0
	FSEL12	FSEL12 - Function Select 12	RW	0
	FSEL11	FSEL11 - Function Select 11	RW	0
	FSEL10	FSEL10 - Function Select 10	RW	0

## GPIO Pin Mapping / Alternate Functions

GPIO Pin	Mode	Alternate Function	Alternate Function	Alternate Function
GPIO13	Low	TXD0	SD5	<reserved>
GPIO14	Low	TXD0	SD6	<reserved>
GPIO15	Low	RXD0	SD7	<reserved>
GPIO16	Low	<reserved>	SD8	<reserved>
GPIO17	Low	<reserved>	SD9	<reserved>

Table 6-3 – GPIO Alternate function select register 1

# GPIO Setup (RPI2)

1. Program GPIO Pin Function (in / out / alternate function) by writing corresponding (memory mapped) GPFSEL register.  
GPFSELn: pins  $10*n .. 10*n+9$   
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  $32*n .. 32*n+31$   
clear pin: GPCLRn: pins  $32*n .. 32*n+31$   
no RMW required.
  - b. If reading: read corresponding bit in the GPLEVn register  
GPLEVn: pins  $32*n ... 32*n+1$
  - c. If "alternate function": device acts autonomously. Implement device driver.