F28HS Hardware-Software Interface: Systems Programming

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh

HERIOT
WATT
UNIVERSITY

Semester 2 — 2019/20

Outline

Lecture 1: Introduction to Systems Programming

Introduction to Systems Programming

- This course focuses on **how hardware and systems software work together** to perform a task.
- We take a **programmer-oriented view** and focus on software and hardware issues that are relevant for developing **fast, secure, and portable** code.
- **Performance** is a recurring theme in this course.
- You need to grasp a lot of low-level technical issues in this course.
- In doing so, you become a **"power programmer"**.

Why is this important?

You need to understand issues at the hardware/software interface, in order to

- understand and improve performance and resource consumption of your programs, e.g. by developing cache-friendly code;
- avoid progamming pitfalls, e.g. numerical overflows;
- avoid security holes, e.g. buffer overflows;
- understand details of the compilation and linking process.

Questions to be addressed

For each of these issues we will address several common questions on the hardware/software interface:

- **Optimizing program performance:**
    - ▶ Is a switch statement always more efficient than a sequence of if-else statements?
    - ▶ How much overhead is incurred by a function call?
    - ▶ Is a while loop more efficient than a for loop?
    - ▶ Are pointer references more efficient than array indexes?
    - ▶ Why does our loop run so much faster if we sum into a local variable instead of an argument that is passed by reference?
    - ▶ How can a function run faster when we simply rearrange the parentheses in an arithmetic expression?
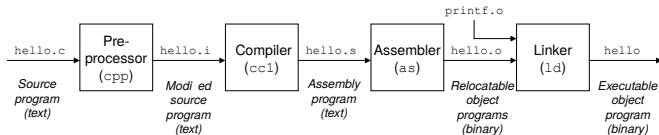
Questions to be addressed

- **Understanding link-time errors:**
  - ▶ What does it mean when the linker reports that it cannot resolve a reference?
  - ▶ What is the difference between a static variable and a global variable?
  - ▶ What happens if you define two global variables in different C files with the same name?
  - ▶ What is the difference between a static library and a dynamic library?
  - ▶ Why does it matter what order we list libraries on the command line?
  - ▶ Why do some linker-related errors not appear until run time?
- **Avoiding security holes:**
  - ▶ How can an attacker exploit a buffer overflow vulnerability?

Compilation of hello world


```
hello.c → Pre-processor (cpp) → hello.i → Compiler (cc1) → hello.s → Assembler (as) → hello.o → Linker (ld) → hello
         Source              Modified       Assembly            Relocatable        Executable
         program             source         program             object             object
         (text)              program        (text)              programs           program
                             (text)                             (binary)           (binary)
                                                    printf.o
```

- We have seen individual phases in the compilation chain so far (e.g. assembly)
- Using `gcc` on top level picks the starting point, depending on the file extension, and generates binary code
- You can view the intermediate files of the compilation using the `gcc` flag `-save-temps`
- This is useful in checking, e.g. which assembler code is generated by the compiler
- We will be using `-D` flags to control the behaviour of the pre-processor on the front end

The Shell

Your window to the system is the **shell**, which is an interpreter for
commands issued to the system:

```
host> echo "Hello_world"
Hello world
host> ls
...
```

The Linux Introduction in F27PX-Praxis gave you an overview of what
you can do in a shell. In this course, we make heavy usage of the
shell. Check the later sections in the on-line Linux Introduction, which
explain some of the more advanced concepts.

Hardware organisation of a typical system
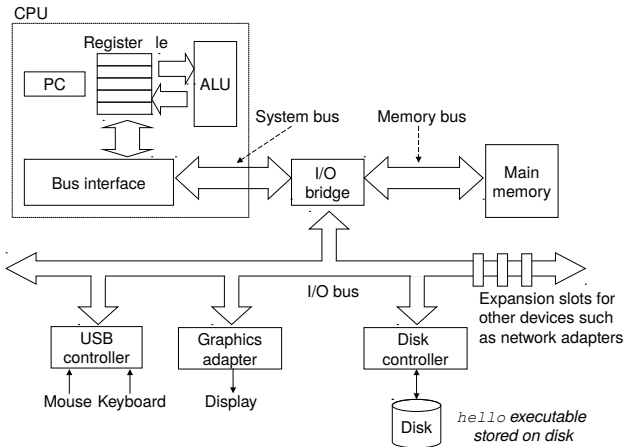
Components

The picture on the previous slide, mentions several important concepts:

- **Processor:** the Central Processing Unit (CPU) is the engine that executes instructions; modern CPUs are complicated in order to provide additional performance (multi-core, pipelining, caches etc);
- **Main Memory:** temporary storage for both program and data; arranged as a sequence of dynamic random access memory (DRAM) chips;
- **Buses** transmit information, as byte streams, between components of the hardware; the Universal Serial Bus (USB) is the most common connection for external devices;
- **I/O devices** are in charge of input/output and represent the interface of the hardware to the external world

The Hello World Program

```c
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```

What happens when we compile and execute this **hello world** program?

Compiling Hello World

When we compile the program by calling

```
gcc -o hello hello.c
```

the compilation chain is executed. Note:

- The source code of Hello World is represented in ASCII characters and stored in a file.
- The contents of the file is just a sequence of bytes
- The **context** determines whether these bytes are interpreted as text or as graphics etc.

When we execute the resulting binary, the next slides show what's happening

```
./hello
```

Compiling Hello World

When we compile the program by calling

```
gcc -o hello hello.c
```

the compilation chain is executed. Note:

- The source code of Hello World is represented in ASCII characters and stored in a file.
- The contents of the file is just a sequence of bytes
- The **context** determines whether these bytes are interpreted as text or as graphics etc.

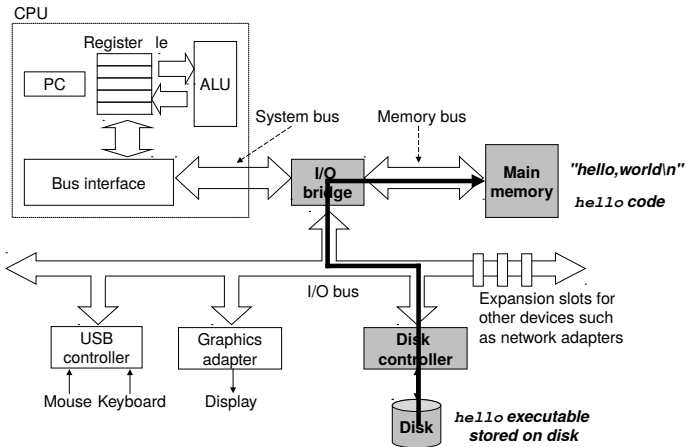When we execute the resulting binary, the next slides show what's happening

```
./hello
```

1. Reading the hello program from the keyboard
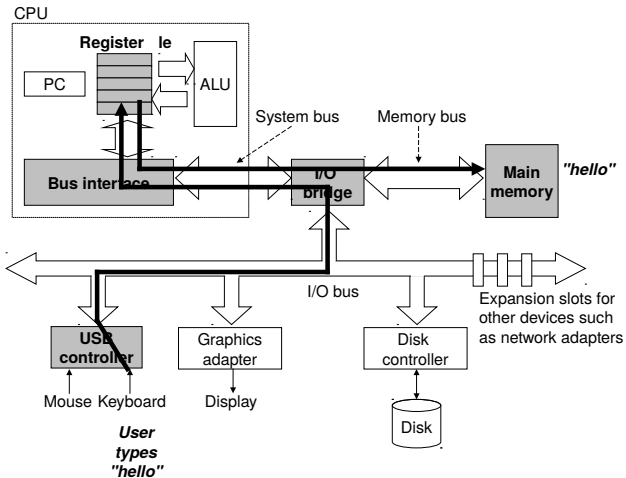
The shell reads ./hello from the keyboard, stores it in memory; then, initiates to load the executable file from disk to memory.
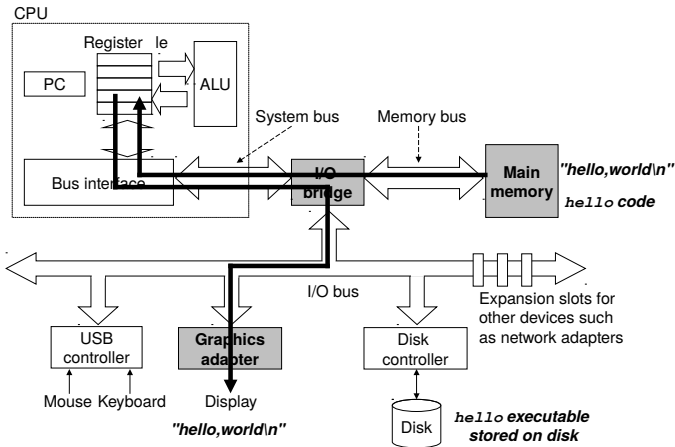
2. Reading the executable from disk to main memory

Using direct memory access (DMA) the data travels from disk directly to memory.

[0]From Bryant and O'Hallaron, Ch 1

# 3. Writing the output string from memory to display

Once the code and data in the hello object file are loaded into memory, the processor begins executing the machine-language instructions in the hello program's main routine.

Caches

- Copying data from memory to the CPU is slow compared to performing an arithmetic or logic operation.
- This difference is called **processor-memory gap** and it is increasing with newer generations of processors.
- Copying data from disk is even slower.
- On the other hand, these slower devices provide more capacity.
- To speed up the computation, smaller faster storage devices called **cache memories** are used.
- These cache memories (or just **caches**) serve as temporary staging areas for information that the processor is likely to need in the near future.

Cache memories
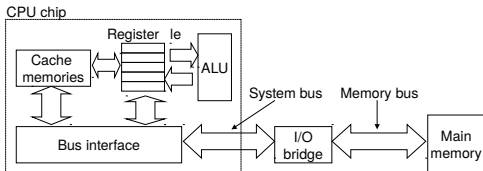
- An **L1 cache** on the processor chip holds tens of thousands of bytes and can be accessed nearly as fast as the register file.
- A larger **L2 cache** with hundreds of thousands to millions of bytes is connected to the processor by a special bus.
- It might take 5 times longer for the process to access the L2 cache than the L1 cache, but this is still 5 to 10 times faster than accessing the main memory.
- The L1 and L2 caches are implemented with a hardware technology known as static random access memory (SRAM). Newer systems even have three levels of cache: L1, L2, and L3.

Caches and Memory Hierarchy
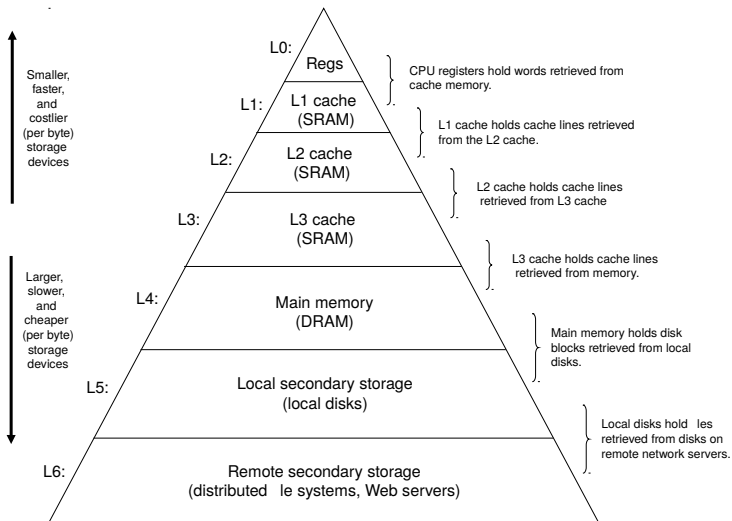
The Role of the Operating System

Application programs			} Software
Operating system			
Processor	Main memory	I/O devices	} Hardware
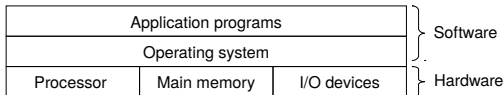
- We can think of the **operating system** as a layer of software interposed between the application program and the hardware.
- All attempts by an application program to manipulate the hardware must go through the operating system.
- This enhances the security of the system, but also generates some overhead.
- In this course we are mainly interested in the **interface between the Software and Hardware layers** in the picture above.

[0] From Bryant and O'Hallaron, Ch 1

Goals of the Operating System

The operating system has two primary purposes:

- to protect the hardware from misuse by runaway applications, and
- to provide applications with simple and uniform mechanisms for manipulating complicated and often wildly different low-level hardware devices.

The operating system achieves both goals via three fundamental abstractions: **processes, virtual memory, and files.**
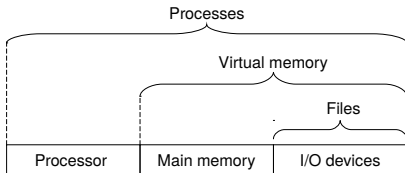
Basic Concepts

In this overview we will cover the following basic concepts:

- Processes
- Threads
- Virtual memory
- Files

Processes

- A **process** is the operating system's abstraction for a running program.
- It provides the illusion of having exclusive access to the entire machine.
- Multiple processes can run concurrently.
- The OS mediates the access to the hardware, and prevents processes from overwriting each other's memory.

Concurrency vs Parallelism vs Threads

- **Concurrent execution** means that the instructions of one process are interleaved with the instructions of another process.
- The operating system performs this interleaving with a mechanism known as context switching.
- The context of a process consists of: the program counter (PC), the register file, and the contents of main memory.
- They appear to run simultaneously, but in reality at each point the CPU is executing just one process' operation.
- On **multi-core** systems, where a CPU contains several independent processors, the two processes can be executed **in parallel**, running on separate cores.
- In this case, both processes are genuinely running simultaneously.
- The main goal of parallelism is to make programs run faster.
- A process can itself consist of multiple **threads**.

Example of Context Switching

This example shows the context switching that is happening between the **shell** process and the hello process, when running our hello world example.
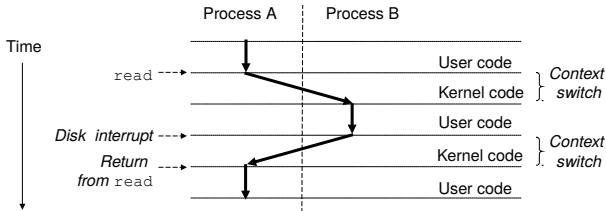
Different Forms of Concurrency

Concurrency can be exploited at different levels:

- **Thread-level concurrency:** A program explicitly creates several threads with independent control flows. Each thread typically represents a large piece of computation. Shared memory, or message passing can be used to exchange data.

- **Instruction-Level Parallelism:** The components of the CPU can be arranged in a way so that the CPU executes several instructions at the same time. For example, while one instruction is performing an ALU operation, the data for the next instruction can be loaded from memory ("pipelining").

- **Single-Instruction, Multiple-Data (SIMD) Parallelism:** Modern processor architectures provide **vector-operations**, that allow to execute an operation such as addition, over a sequence of values ("vectors"), rather than just two values. Graphic cards make heavy use of this form of parallelism to speed-up graphics operations.

Categorizing different processor configurations
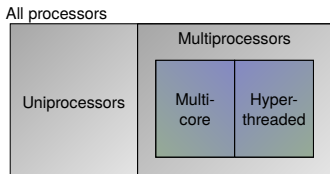
- **Uniprocessors**, with only one CPU, need to context-switch in order to run several processes seemingly at the same time
- **Multiprocessors** replicate certain components of the hardware to genuinely run processes at the same time:
  - **Muticores** replicate the entire CPU, as several "cores", each of can run a process.
  - **Hyperthreaded** machines replicate hardware to store the context of several processes to speed-up context-switching.

Virtual Memory

Virtual memory is an abstraction that provides each process with the illusion that it has exclusive use of the main memory. Each process has the same uniform view of memory, which is known as its virtual address space.
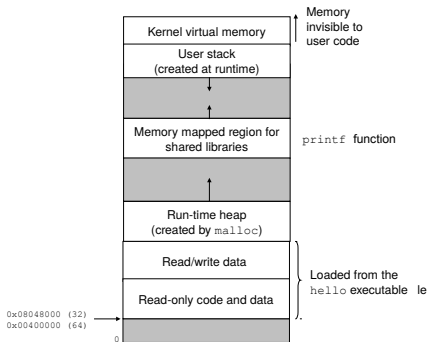
Virtual Memory

The lower region holds the data for the user.

The user space is separated into several areas, with different roles:

- The **code and data area:** contains the progam code and initialised data, starting at a fixed address. The program code is read only, the data is read/write.
- The **heap** contains dynamically allocated data during the execution of the program. In high-level languages, such as Java, any `new` will allocate in the heap. In low-level languages, such as C, you can use the library function `malloc` to dynamically allocate data in the heap.
- The **shared data** section holds dynamically allocated data, managed by shared libraries.
- The **stack** is a dynamic area at the top of the memory, growing downwards. It is used to hold the local data of functions whenever a function is called during program execution.
- The topmost section of the virtual memory is allocated to **kernel virtual memory**, and only accessible to the OS kernel.

Virtual Memory

- **Virtual memory** gives the illusion of a continuous address space, exceeding main memory, with exclusive access.
- It abstracts over the limitations of physical main memory and allows for several parallel threads to access the same address space.
- We will discuss this aspect in more detail in the Lecture on "Memory Hierarchy".

Aside The Linux project

In August 1991, a Finnish graduate student named Linus Torvalds modestly announced a new Unix-like operating system kernel:

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Date: 25 Aug 91 20:57:08 GMT

Hello everybody out there using minix -
I'm doing a (free) operating system (just a hobby, won't be big and
professional like gnu) for 386(486) AT clones. This has been brewing
since April, and is starting to get ready. I'd like any feedback on
things people like/dislike in minix, as my OS resembles it somewhat
(same physical layout of the file-system (due to practical reasons)
among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work.
This implies that I'll get something practical within a few months, and
I'd like to know what features most people would want. Any suggestions
are welcome, but I won't promise I'll implement them :-)

Linus (torvalds@kruuna.helsinki.fi)
```

Files

- A **file** is a sequence of bytes.
- A file can be used to model any I/O device: disk, keyboard, mouse, network connections etc.
- Files can also be used to store data about the hardware (`/proc/` filesystem), or to control the system, e.g. by writing to files.
- Thus, the concept of a **file** is a very powerful abstraction that can be used for many different purposes.

External Devices

- An important task of the OS/code is to interact with external devices.
- We will see this in detail on the Rpi2
- From the OS point of view, external devices and network connections are files that can be written to and read from.
- When writing to such a special file, the OS sends the data to the corresponding network device
- When reading from such a special file, the OS reads data from the corresponding network device
- This file abstraction simplifies network communication, but is also a source of additional communication overhead.
- Therefore, high performance libraries tend to avoid this "software stack" of implementing file read/write in the OS, but rather directly read to and write from the device (in the same way that we will be using these devices)

A network is another I/O device
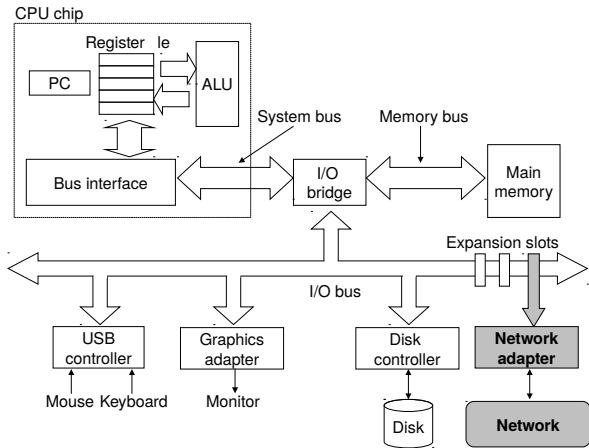
The network can be viewed as just another I/O device.

The Role of Abstraction

- In order to tackle system complexity **abstraction** is a key concept.
- For example, an application program interface (API), abstracts from the internals of an implementation, and only describes its core functionality.
- Java class declaration or C prototypes are programming language features to facilitate abstraction.
- The instruction set architecture abstracts over details of the hardware, so that the same instructions can be used for different realisations of a processor.
- On the level of the operating system, key abstractions are
  - processes (as abstractions of a running program),
  - files (as abstractions of I/O), and
  - virtual memory (as an abstraction of main memory).
- A newer form of abstraction is a **virtual machine**, which abstracts over an entire computer.

Some abstractions provided by a computer system
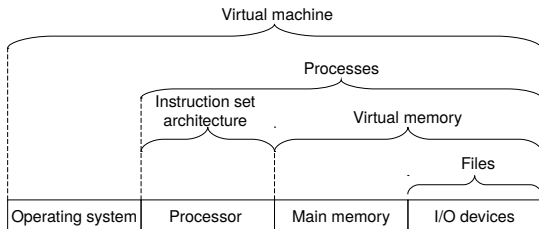
A major theme in computer systems is to provide abstract representations at different levels to hide the complexity of the actual implementations.

Reading List: Systems Programming

📖 David A. Patterson, John L. Hennessy. *"Computer Organization and Design: The Hardware/Software Interface"*, **ARM edition**, Morgan Kaufmann, Apr 2016. ISBN-13: 978-0128017333.

📖 Randal E. Bryant, David R. O'Hallaron *"Computer Systems: A Programmers Perspective"*, 3rd edition, Pearson, 7 Oct 2015. ISBN-13: 978-1292101767.

📖 Bruce Smith *"Raspberry Pi Assembly Language: Raspbian"*, CreateSpace Independent Publishing Platform; 2 edition, 19 Aug 2013. ISBN-13: 978-1492135289.

Other Online Resources

📕 Gordon Henderson *"WiringPi library: GPIO Interface library for the Raspberry Pi,*
http://wiringpi.com/

📕 Valvers *"Bare Metal Programming in C",*
http://www.valvers.com/open-software/raspberry-pi/step01-bare-metal-programming-in-cpt1/

📕 Alex Chadwick, Univ of Cambridge *"Baking Pi",*
https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os

Lecture 2.
Systems Programming with the Raspberry Pi

SoC: System-on-Chip

- A **System-on-Chip** (SoC) integrates all components of a computer or other electronic system into a single chip.
- One of the main advantages of SoCs is their low power consumption.
- Therefore they are often used in embedded devices.
- All versions of the Raspberry Pi are examples of SoCs

Note: In this course we are using the Raspberry Pi 2 Model B. The low-level code will only work with this version.

The Raspberry Pi Foundation: https://www.raspberrypi.org/
UK registered charity 1129409

Raspberry Pi 1 vs 2

The Raspberry Pi version 2 was released on 2nd February 2015. Its components are:

- the BCM2836 SoC (System-on-Chip) by Broadcom
- an ARM-Cortex-A7 CPU with 4 cores (clock frequency: 900MHz)
- 1 GB of DRAM
- a Videocore IV GPU
- 4 USB ports (sharing the one internal port together with the Ethernet connection)
- power supply through a microUSB port

NB: RPi2 is significantly more powerful than RPi1, which used an ARM1176JZ-F single-core at 700MHz clock frequency (as the BCM2835 SoC). However, its network bandwidth is unchanged.

NB: The A-series of the ARM architectures is for "application" usage and therefore more powerful than the M-series, which is mainly for small, embedded systems.

It is possible to *safely* over-clock the processor up to 950 MHz.
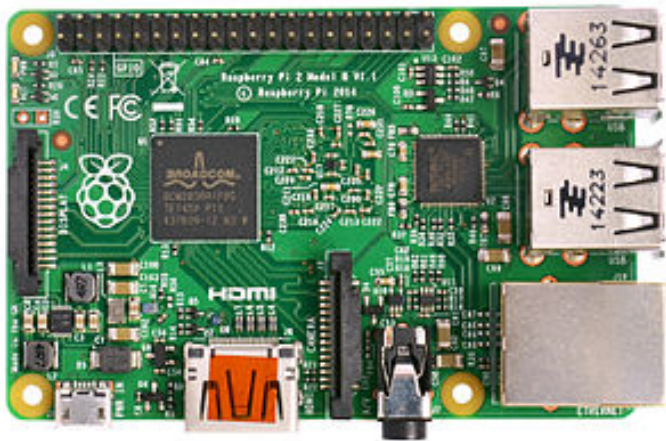
Raspberry Pi 2

Software configuration

- RPi2 supports several major Linux distributions, including: Raspbian (Debian-based), Arch Linux, Ubuntu, etc
- The main system image provided for RPi2 can boot into several of these systems and provides kernels for both ARMv6 (RPi1) and ARMv7 (RPi2)
- The basic software configuration is almost the same as on a standard Linux desktop
- To tune the software/hardware configuration call

```
> sudo raspi-config
```

Updating your software under Raspbian

We are using **Raspbian 7**, which is based on Debian "Wheezy" with a Linux kernel 3.18.

There is a more recent version (2017-01-11) out: Raspbian 8, based on Debian "Jessie" with a Linux kernel 4.4. Highlights:

- Uses `systemd` for starting the system (changes to run-scripts, enabling services).
- Supports OpenGL and 3D graphics acceleration in an experimental driver (enable using the `raspi-config`)

To update the software under Raspbian, do the following:

```
> sudo apt-get update
> sudo apt-get upgrade
> sudo rpi-update
```

To find the package `foo` in the on-line repository, do the following:

```
> sudo apt-cache search foo
```

To install the package `foo` in the on-line repository, do the following:

```
> sudo apt-get install foo
```

Virtualisation

- In this powerful, multi-core configuration, an RPi2 can be used as a server, running several VMs.
- To this end RPi2 under Raspbian runs a **hypervisor** process, mediating hardware access between the VMs.
- Virtualisation is hardware-supported for the ARMv6 and ARMv7 instruction set
- The ARMv7 instruction set includes a richer set of SIMD (single-instruction, multiple-data) instructions (the **NEON** extensions), to use parallelism and speed-up e.g. multi-media applications
- The NEON instruction allow to perform operations on up to 16 8-bit values at the same time, through the processor's support for 64-bit and 128-bit registers
- Performance improvements in the range of $8 - 16\times$ have been reported for multi-media applications
- The usual power consumption of the Ri2 is between $3.5 - -4$ Watt (compared to ca. 2 Watt for the RPi1)

Raspberry Pi 4

Specification:

- **ARMv8, BCM2837B0, ARM Cortex-A72 CPU 64-bit quad-core @ 1.5GHz**
- Up to 1GB, 2GB or 4GB RAM (LPDDR4)
- On board dual-band 802.11.b/g/n/ac wireless LAN
- On board Bluetooth 5.0, low-energy (BLE)
- **Gigabit Ethernet**
- **2 × USB 3.0 ports**, 2 × USB 2.0 ports
- Extended 40-pin GPIO header
- **2 × micro-HDMI ports** (supporting up to 4Kp60)

---

[0]See RPi 4 spec on NewIT pages

CPU Performance Comparison: Hardware

Rechenleistung im Vergleich				
Plattform	**RAM**	**Chip**	**Technologie**	**Architektur**
Raspberry Pi				
Raspberry Pi 1	512 MByte	Broadcom BCM2835	65 nm	ARM1176JZ-F
Raspberry Pi 2	1 GByte LPDDR2	Broadcom BCM2836	28 nm	Cortex A7
Banana Pi				
Banana Pi	1 GByte	AllWinner A20	40 nm	Cortex A7
Banana Pro	1 GByte	AllWinner A20	40 nm	Cortex A7
Banana Pi M2	1 GByte	AllWinner A31S	40 nm	Cortex A7
Andere Single Board Computer (SBC)				
Beaglebone Black	512 MByte	TI Sitara AM3358/9	45 nm	Cortex A8
Hummingboard-i2	1 GByte	Freescale i.MX6 DualLite	40 nm	Cortex A9
Cubox-i4Pro	2 GByte	Freescale i.MX6 Quad	40 nm	Cortex A9
Odroid C1	1 GByte DDR3	Amlogic S805	28 nm	Cortex A5
Smartphones				
Galaxy S3 Mini (GT-I8190)	1 GByte	ST-Ericsson NovaThor U8500	45 nm	Cortex A9
iPhone 5	1 GByte	Apple A6	32 nm high-k metal gate	ARMv7s Swift [Apple]
Spielekonsolen				
Playstation 2	36 MByte	EmotionEngine	250 nm	RISC, basiert auf MIPS R5900
Apple-Computer				
Apple][e	64 KByte	MOS Technology 6502	8000 nm	MOS Technology
Apple Macintosh 128 K	128 KByte	Motorola 68000	3500 nm	CISC
iMac G3	32 MByte	PowerPC 750 G3	260 nm	PowerPC G3
Intel- und AMD-PCs				
No Name PC 1	64 MByte	Pentium II, 300 MHz	350 nm	x86 Intel
No Name PC 2	384 MByte	AMD Duron, 800 MHz	180 nm	AMD Spitfire
Dell Inspiron 7520	8 GByte	Intel Core i7-3632QM	22 nm	Intel Core i7
Hetzner EQ 4 Server	32 GByte	Intel Core i7-3770	22 nm	Intel Core i7

CPU Performance Comparison: Measurements

DMIPS/MHz	Kerne	MHz	DMIPS	Vgl. RPi 1	Vgl. RPi 2
1,25	1	700	875	100%	13%
1,90	4	900	6840	782%	100%
1,90	2	1000	3800	434%	56%
1,90	2	1000	3800	434%	56%
1,90	4	1000	7600	869%	111%
2,00	1	1000	2000	229%	29%
2,50	2	1000	5000	571%	73%
2,50	4	1000	10000	1143%	146%
1,57	4	1500	9420	1077%	138%
2,50	2	1000	5000	571%	73%
3,50	2	1300	9100	1040%	133%
20,34	1	295	6000	686%	88%
0,43	1	1	0,43	0,05%	0,01%
0,23	1	6	1,4	0,16%	0,02%
2,25	1	233	525	60%	8%
0,91	1	300	273,6	31%	4%
2,81	1	800	2250	257%	33%
14,19	4	2200	99750	11400%	1458%
14,19	4	3400	106530	12175%	1557%

CPU Performance Comparison: Measurements

DMIPS/MHz	Kerne	MHz	DMIPS	Vgl. RPi 1	Vgl. RPi 2
1,25	1	700	875	100%	13%
1,90	4	900	6840	782%	100%
1,90	2	1000	3800	434%	56%
1,90	2	1000	3800	434%	56%
1,90	4	1000	7600	869%	111%
2,00	1	1000	2000	229%	29%
2,50	2	1000	5000	571%	73%
2,50	4	1000	10000	1143%	146%
1,57	4	1500	9420	1077%	138%
2,50	2	1000	5000	571%	73%
3,50	2	1300	9100	1040%	133%
20,34	1	295	6000	686%	88%
0,43	1	1	0,43	0,05%	0,01%
0,23	1	6	1,4	0,16%	0,02%
2,25	1	233	525	60%	8%
0,91	1	300	273,6	31%	4%
2,81	1	800	2250	257%	33%
14,19	4	2200	99750	11400%	1458%
14,19	4	3400	106530	12175%	1557%

Note

RPi2 ca. **7.82× faster** than RPi1

Banana Pi M2 is **1.11× faster** than RPi2

Cubox i4Pro is **1.46× faster**
ODroid C1 is **1.38× faster**

Intel i7 PC is **15.5× faster** than RPi2

Network performance comparison: RPi 1 vs RPi 2

- To compare network performance, encrypted data-transfer through scp is used.
- This profits from the quad-core architecture, because one core can be dedicated to encryption, another core to the actual data transfer.
- An increase in network performance by a factor of $2.5\times$ is reported.
- The highest observed bandwidth on the RPi 2 (with overclocking to 1.05 GHz) is 70 Mbit/s.
- The theoretcial peak performance of the LAN-port is ca 90 MBit/s.
- The SunSpider benchmark for rendering web pages, reports up to $5\times$ performance improvement.

Network performance Measurements

SCP-Vergleichstest							
ARM Freq	**SDRAM Freq**	**GPU Core Freq**	**Temp**	**SCP-Schreiben**[1]	**%**	**SCP-Lesen**[1]	**%**
Raspberry Pi 2, Raspbian							
900 MHz	450 MHz	250 MHz	53,5° C	52,6 Mbit/s	100,0	54,8 Mbit/s	100
1000 MHz	500 MHz	500 MHz	58,4° C	56,3 Mbit/s	107,0	69,0 Mbit/s	125
1050 MHz	500 MHz	500 MHz	58,4° C	65,6 Mbit/s	124,6	69,0 Mbit/s	126
1100 MHz[2]	500 MHz	500 MHz					
Raspberry Pi 1, Raspbian							
700 MHz	400 MHz	250 MHz	43,3° C	21,1 Mbit/s	40,0	21,1 Mbit/s	38
1000 MHz	600 MHz	250 MHz	51,4° C	36,4 Mbit/s	69,1	33,3 Mbit/s	61
Raspberry Pi 2, Debian Jessie[3]							
900 MHz	450 MHz	250 MHz		47,6 Mbit/s	90,5	52,6 Mbit/s	96
1050 MHz	500 MHz	500 MHz		58,0 Mbit/s	110,1	71,4 Mbit/s	130

[1] Durchschnittswert aus mehreren Durchgängen; [2] Test nicht möglich, da RasPi 2 instabil arbeitet; [3] mit für ARMv8 optimierten Paketen

High-performance Alternatives

- There are several single-board computers that provide a **high-performance** alternative to the RPi.
- These are of interest if you have applications with high computational demands and you want to run it on a low-cost and low-power device.
- It's possible to build for example a **cluster** of such devices as a parallel programming platform: see The Glasgow University Raspberry Pi Cloud
- Here we give an overview of the main **performance characteristics** of three RPi2 alternatives:
  - the CuBox i4Pro by SolidRun
  - the Banana Pi M3 by Sinovoip
  - the Lemaker HiKey by Lemaker

Core Specs of the CuBox i4-Pro

- Freescale i.MX6 (SoC) quad-core, containing an **ARM Cortex A9** (ARMv7 instruction set) with **4 cores**
- GC2000 GPU (supports OpenGL etc)
- 4 GB RAM and a micro-SD card slot
- 10/100/1000 Mb/s Ethernet (max 470Mb/s)
- **WLAN** (802.11b/g/n)
- Bluetooth 4.0
- 1 USB port and eSATA (3Gb/s) interface
- Price: 124*$*

Software

- Debian Linux, Kodi Linux, XBMC Linux

Core Specs of the CuBox i4-Pro

- Freescale i.MX6 (SoC) quad-core, containing an **ARM Cortex A9** (ARMv7 instruction set) with **4 cores**
- GC2000 GPU (supports OpenGL etc)
- 4 GB RAM and a micro-SD card slot
- 10/100/1000 Mb/s Ethernet (max 470Mb/s)
- **WLAN** (802.11b/g/n)
- Bluetooth 4.0
- 1 USB port and eSATA (3Gb/s) interface
- Price: 124*$*

Software

- Debian Linux, Kodi Linux, XBMC Linux

Core Specs of the Banana Pi M3

- Allwinner A83T (SoC) chip, containing an **ARM Cortex-A7** (ARMv7 instruction set) with **8 cores**
- PowerVR SGX544MP1 GPU (supports OpenGL etc)
- 2 GB LPDDR3 RAM plus 8 GB eMMC memory and a micro-SD card slot
- **Gigabit Ethernet**
- **WLAN** (802.11b/g/n)
- Bluetooth 4.0
- 2 USB ports and SATA interface
- 40 GPIO pins (not compatible with RPi2)
- Price: 90€

Software

- BPI-Berryboot (allegedly with GPU support), or Ubuntu Mate

Experiences

- SATA shares the the USB bus connection and is therefore slow
- Problems accessing the on-board micro-phone

Core Specs of the Banana Pi M3

- Allwinner A83T (SoC) chip, containing an **ARM Cortex-A7** (ARMv7 instruction set) with **8 cores**
- PowerVR SGX544MP1 GPU (supports OpenGL etc)
- 2 GB LPDDR3 RAM plus 8 GB eMMC memory and a micro-SD card slot
- **Gigabit Ethernet**
- **WLAN** (802.11b/g/n)
- Bluetooth 4.0
- 2 USB ports and SATA interface
- 40 GPIO pins (not compatible with RPi2)
- Price: 90€

Software

- BPI-Berryboot (allegedly with GPU support), or Ubuntu Mate

Experiences

- SATA shares the the USB bus connection and is therefore slow
- Problems accessing the on-board micro-phone

Core Specs of the Banana Pi M3

- Allwinner A83T (SoC) chip, containing an **ARM Cortex-A7** (ARMv7 instruction set) with **8 cores**
- PowerVR SGX544MP1 GPU (supports OpenGL etc)
- 2 GB LPDDR3 RAM plus 8 GB eMMC memory and a micro-SD card slot
- **Gigabit Ethernet**
- **WLAN** (802.11b/g/n)
- Bluetooth 4.0
- 2 USB ports and SATA interface
- 40 GPIO pins (not compatible with RPi2)
- Price: 90€

Software

- BPI-Berryboot (allegedly with GPU support), or Ubuntu Mate

Experiences

- SATA shares the the USB bus connection and is therefore slow
- Problems accessing the on-board micro-phone

Core Specs of the Lemaker Hikey

- Kirin 620 (SoC) chip with **ARM Cortex A53** and **8 cores**
- ARM Mali450-MP4 (supports OpenGL etc) GPU
- 1 or 2 GB LPDDR3 RAM plus 8 GB eMMC memory and a micro-SD card slot
- **WLAN** (802.11b/g/n)
- Bluetooth 4.1
- 2 USB ports
- 40 GPIO pins (not compatible with RPi2)
- Audio and Video via HDMI connectors
- Board-layout matches the 96-board industrial standard for embedded devices
- Price: 120€

Software

- Android variant (part of 96-board initiative)
- Linaro (specialised Linux version for embedded devices)

Banana Pi M3 and Lemaker Hikey: Specs

	Banana Pi M3	Lemaker Hikey
CPU	A83T ARM Cortex-A7, ARMv7, 8 Kerne, max. 2 GHz	ARM Cortex-A53, ARMv8, 8 Kerne
GPU	PowerVR SGX544MP1 (OpenGL ES 2.0, OpenCL 1.x, DX 9_3)	ARM Mali450-MP4 (OpenGL ES 1.1/2.0, OpenVG 1.1)
RAM	2 GByte LPDDR3	1 oder 2 GByte LPDDR3
Speicher	8 GByte eMMC	8 GByte eMMC
Schnittstellen		
Massenspeicher	Micro-SD-Card, SATA (USB-to-SATA; GL830)	Micro-SD-Card
USB Ports	2 USB 2.0, USB OTG	2 USB 2.0, USB OTG
GPIO	40 Pins (GPIO, UART, I2C, I2S, SPI, PWM, +3.3V, +5V, GND)	40 Pins (GPIO, UART, I2C, SPI, PWM, PCM, SYS_DCIN, +1.8V, +5V, GND); 60 Pins (SDIO, MIPI_DSI, MIPI_CSI)
Netzwerk		
Ethernet	10/100/1000 Mbit/s (Realtek RTL8211E/D)	optional (via USB-Adapter)
WLAN	802.11b/g/n	802.11b/g/n
Bluetooth	Bluetooth 4.0	Bluetooth 4.1 LE
Audio, Video		
Audio Out	3,5mm Klinke, HDMI	HDMI
Audio In	Onboard-Mikrofon	HDMI
Video Out	HDMI 1.4 (HDCP 1.2, max. 1920x1080), MIPI DSI	HDMI 1.4 (max. FHD 1080p), 2 MIPI DSI
Video In	Parallele 8-Bit-Kameraschnittstelle, MIPI CSI	2 MIPI CSI
Sonstiges		
Schalter	Power, Reset, U-Boot	Power/Reset
LEDs	Power, RJ45, benutzerdefiniert	WLAN, Bluetooth, 4 benutzerdefiniert
Strom	Micro-USB, optional 5V-Klinke	8V~18V/3A Klinke
OS	Android, Linux	Android, Linux
Abmessung	92mm x 60mm	85mm x 55mm
Straßenpreis	90 Euro	120 Euro

Raspberry Pi 3 and Lemaker Hikey: Performance

Performance as runtime (of `sysbench` benchmark) and network
bandwidth (using `lperf` benchmark):

	Perf. (runtime) number of threads			Max power	Network bandwidth Ethernet	WLAN
	1	4	8			
Raspberry Pi 2	297s	75s	—			
Raspberry Pi 3	182s	45s	—			45 Mb/s
Cubox i4Pro	296s	75s	—			
Banana Pi M3	159s	40s	21s	1.1A	633 Mb/s	2.4 Mb/s
Lemaker Hikey	**12s**	**3s**	**2s**	1.7A	—	37.3 Mb/s

Summary: In terms of performance, the Lemaker Hikey is the best
choice.

[0]Material from Raspberry Pi Geek 04/2016

Raspberry Pi 3 and Lemaker Hikey: Performance comparison

Benchmark-Ergebnisse			
	1 Thread	4 Threads	8 Threads
Raspberry Pi 3	182 Sekunden	45 Sekunden	–
Banana Pi M3	159 Sekunden	40 Sekunden	21 Sekunden
Lemaker Hikey	12 Sekunden	3 Sekunden	2 Sekunden

To run the (CPU) performance benchmark on the RPi2 do:

```
> sudo apt-get update
> sudo apt-get install sysbench
> sysbench --num-threads=1 --cpu-max-prime=10000 --test=cpu
  run
```

[0] Material from Raspberry Pi Geek 04/2016

Core Specs of Odroid-XU4

- Exynos 5422 (SoC) Octa big.LITTLE ARM with an ARM Cortex-A15 quad-core and an ARM Cortex-A7 quad-core
- Mali-T628 MP6 GPU
- 2 GB LPDDR3 RAM plus eMMC memory and a micro-SD card slot
- **Gigabit Ethernet**
- 1 USB 2.0A and **1 USB 3.0** port
- Video via HDMI connectors
- 40 GPIO pins (not compatible with RPi2)
- Price: 95€

The CPU is the same as in high-end smartphones such as the Samsug Galaxy S5.

The big.LITTLE architecture dynamically switches from (faster) Cortex-A15 to (slower) Cortex-A7 to save power.

Software: Ubuntu 14.04 or Ubuntu 16.04; Android 4.4.4; **OpenMediaVault 2.2.13**, Kali Linux, Debian.

RPi3 vs Odroid-XU4: Specs

Odroid-XU4 vs. Raspberry Pi 3		
	Odroid-XU4	RasPi 3
SoC	Exynos 5422 Octa big.LITTLE ARM	Broadcom BCM2837
CPU	Cortex-A15 (2.0 GHz) Quad-Core und Cortex-A7 Quad-Core	ARM Cortex-A53 Quad-Core (1,2 GHz)
GPU	Mali-T628 MP6	Broadcom Dual Core VideoCore IV
RAM	2 GByte LPDDR3 (933 MHz)	1 GByte LPDDR2 (900 MHz)
Speicher	Micro-SD, eMMC 5.0	Micro-SD
Netzwerk	10/100/1000-Mbit/s-Ethernet	10/100-Mbit/s-Ethernet, WLAN 802.11b/g/n
USB	USB 2.0 A, 2 USB 3.0	4 USB 2.0 (über Hub)
Videoausgang	HDMI	HDMI
Schnittstellen	I2S, I²C, GPIO	SPI, I²C, UART
Größe	83 x 59 x 18 mm	85,6 x 56 x 21 mm
Preis (ca.)	95 Euro	35 Euro

Odroid-XU4

Odroid-XU4 **Open Hardware**
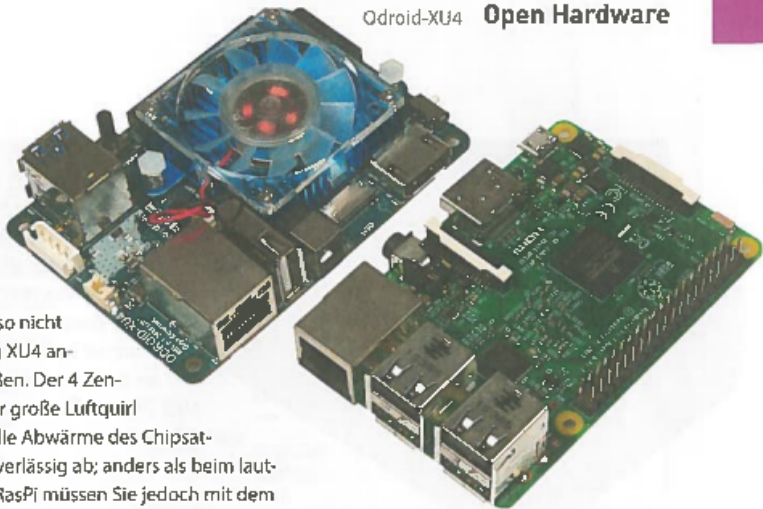
ich also nicht
n den XU4 an-
chließen. Der 4 Zen-
meter große Luftquirl
ührt die Abwärme des Chipsat-
es zuverlässig ab; anders als beim laut-
osen RasPi müssen Sie jedoch mit dem
aufgeräusch leben. Laut Angaben des
erstellers springt der Lüfter jedoch nur

1 In den Dimensionen unterscheiden

Network performance: RPi3 vs Odroid-XU4

| Datenraten im Vergleich	Raspberry Pi 3	Odroid-XU4
Samba		
Datenrate (Upload)	87,80 Mbit/s	418,88 Mbit/s
Datenrate (Download)	89,63 Mbit/s	469,45 Mbit/s
FTP		
Datenrate (Upload)	84,14 Mbit/s	404,15 Mbit/s
Datenrate (Download)	86,18 Mbit/s	439,46 Mbit/s
SSH		
Datenrate (Upload)	86,90 Mbit/s	305,34 Mbit/s
Datenrate (Download)	88,91 Mbit/s	299,59 Mbit/s
Iperf		
Datenrate	94,73 Mbit/s	511,33 Mbit/s
Sämtliche Übertragungsraten über drei Versuche gemittelt.		

Note: Raw network performance is ca. **5× faster** on the ODroid-XU4!

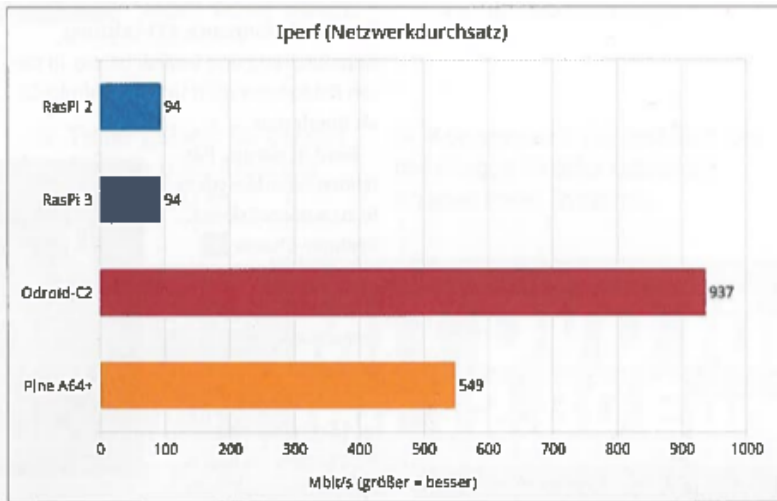# Raspberry Pi 3 and ODroid C2: CPU Performance Comparison
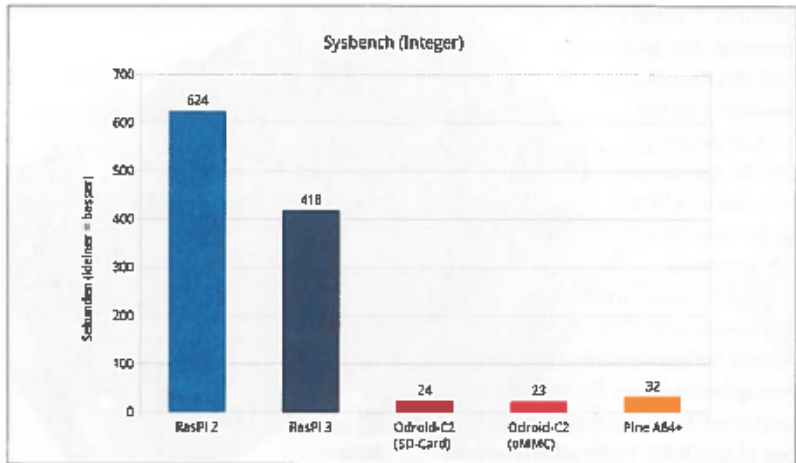
Raspberry Pi 3 and ODroid C2:
I/O Performance Comparison
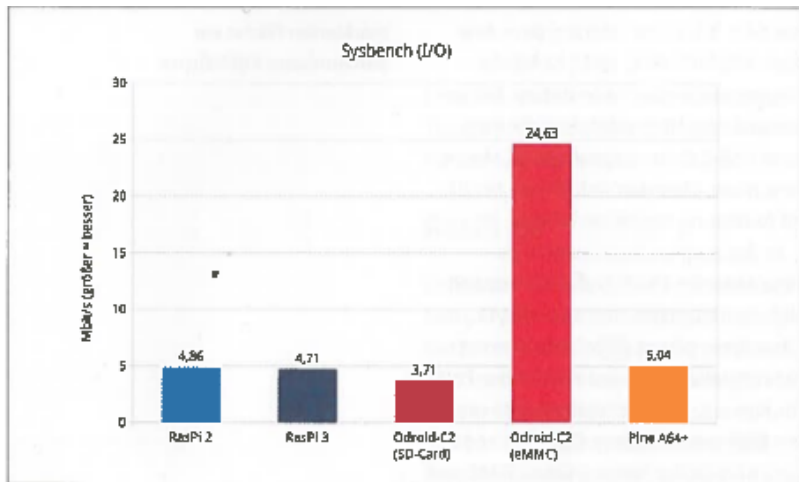
Raspberry Pi 3 and ODroid C2:
Network Performance Comparison

RPi3 vs Odroid-XU4: Experience

- In terms of network-performance, the ODroid-XU4 is much faster.
- It is a good basis for a NAS (Network attached Storage).
- In terms of CPU-performance, the Odroid is slightly faster: Cortex-A15 (2.0 GHz) vs Cortex-A53 (1.2 GHz).
- However, in practice, the **GUI is much slower**.
- Based on the `gtkperf` GUI benchmark, the ODroid is ca. **3×slower**.
- The reason for this difference is more optimisation in the device drivers for RPi's VideoCore IV GPU (compared to ODroid's Mali GPU).
- **Note:** To assess performance and usability, one has to consider the entire software stack, not just the raw performance of the hardware!

Performance comparison: Odroid vs RPi4

Benchmarks

ODROID-N2 : CPU A73 1.800GHz / A53 1.896GHz / DDR4 1.320GHz
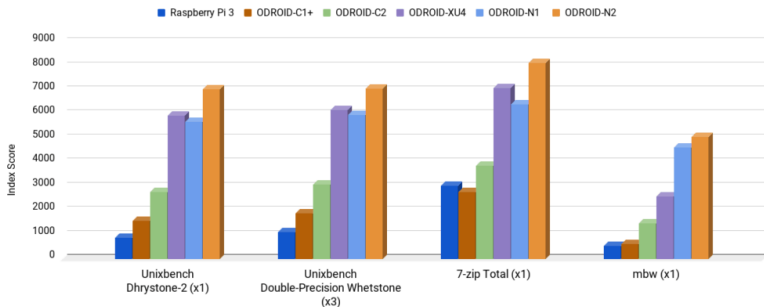
Orange Pi

- Allwinner H3 Soc: Quad-core Cortex-A7 H.265/HEVC 4K
- Mali 400MP2 GPU @ 600MHz
- 1GB DDR3 memory (shared with GPU)
- 8GB EMMC Flash
- 10/100 Ethernet RJ45
- 40 Pins Header,compatible with Raspberry Pi B+
- Runs: Android, Lubuntu, Debian, Raspbian

Beware of stability and performance of the software!

Latest devices

- Raspberry Pi4: available since Fall 2019
- Odroid XU4 or N2: for high performance
- Odroid C2 or HC2: for high bandwith
- Orange Pi H3: cheap but software issues

Summary

- The Raspberry Pi is one of the most widely-used single-board computers.
- The RPi comes in several version (1,2,3); we are using the **Raspberry Pi 2 model B**.
- There is a rich software eco-system for the RPis and excellent, detailed documentation.
- A good high-CPU-performance alternatives is: Lemaker HiKey
- A good high-network-performance alternative is: Odroid-XU4
- Check out the Raspberry Pi projects available online.

Lecture 3:
Memory Hierarchy

Memory Hierarchy: Introduction

- Some fundamental and enduring properties of hardware and software:
    - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
    - The gap between CPU and main memory speed is widening.
    - Well-written programs tend to exhibit good locality.
- These fundamental properties complement each other beautifully.
- They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**.

Memory Hierarchy

- Our view of the main memory so far has been a **flat** one, ie.
- access time to all memory locations is constant.
- In modern architecture this is **not** the case.
- In practice, a memory system is a **hierarchy of storage devices** with different capacities, costs, and access times.
- CPU registers hold the most frequently used data.
- Small, fast cache memories nearby the CPU act as staging areas for a subset of the data and instructions stored in the relatively slow main memory.
- The main memory stages data stored on large, slow disks, which in turn often serve as staging areas for data stored on the disks or tapes of other machines connected by networks

Caches and Memory Hierarchy
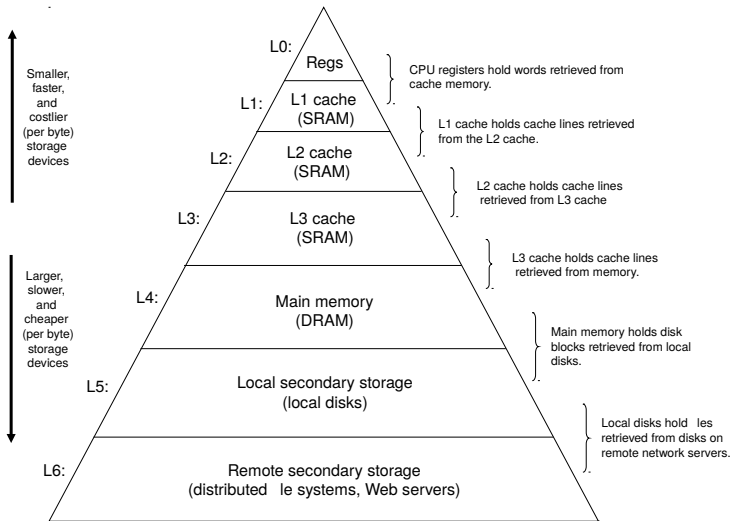
Discussion

As we move from the top of the hierarchy to the bottom, the devices become **slower, larger, and less costly** per byte.

The main idea of a memory hierarchy is that **storage at one level serves as a cache for storage at the next lower level**.

Using the different levels of the memory hierarchy efficiently is crucial to achieving high performance.

Access to levels in the hierarchy can be explicit (for example when using OpenCL to program a graphics card), or implicit (in most other cases).

The importance of the memory hierarchy

- For the programmer this is important because data access times are very different:
    - Register: **0 cycles**
    - Cache: **1–30 cycles**
    - Main memory: **50–200 cycles**
- We want to store data that is frequently accessed **high in the memory hierarchy**

Locality

- **Principle of Locality**: Programs tend to use data and instructions with addresses near or equal to those they have used recently
- **Temporal locality:** Recently referenced items are likely to be referenced again in the near future.
- **Spatial locality:** Items with nearby addresses tend to be referenced close together in time

Locality Example: sum-over-array

```
ulong count; ulong sum;
for (count = 0, sum = 0; count<n; count++)
    sum += arr[count];
res1->count = count;
res1->sum = sum;
res1->avg = sum/count;
}
```

- **Data references**
  - ▸ Reference array elements in succession (stride-1 reference pattern). **spatial locality**
  - ▸ Reference variable sum each iteration. **temporal locality**
- **Instruction references**
  - ▸ Reference instructions in sequence. **spatial locality**
  - ▸ Cycle through loop repeatedly. **spatial locality**

Importance of Locality

Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer!

Which of the following two version of sum-over-matrix has better locality (and performance):

Traversal by rows:

```
int i, j;  ulong  sum;
for (i = 0; i<n; i++)
  for (j = 0; j<n; j++)
    sum += arr[i][j];
```

Traversal by columns:

```
int i, j; ulong  sum;
for (j = 0; j<n; j++)
  for (i = 0; i<n; i++)
    sum += arr[i][j];
```

Caches

- **Cache:** A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Fundamental idea of a memory hierarchy:
    - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level $k + 1$.
- Why do memory hierarchies work?
    - Because of locality, programs tend to access the data at level k more often than they access the data at level $k + 1$.
    - Thus, the storage at level $k + 1$ can be slower, and thus larger and cheaper per bit.
- **Big Idea:** The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

General Cache Concepts

Cache — Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Memory — Larger, slower, cheaper memory viewed as partitioned into "blocks"

General Cache Concepts: Hit

Request: 14

Cache | 8 | 9 | 14 | 3 |

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

Data in block b is needed

Block b is in cache:
Hit!

General Cache Concepts: Miss

Request: 12

Cache | 8 | 5 | 14 | 3 |

12 **Request: 12**

Memory

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

Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

---

[0] From Bryant and O'Hallaron, Ch 6

General Cache Concepts: Miss

Request: 12

Data in block b is needed

Cache

| 8 | 12 | 14 | 3 |

Block b is not in cache:
Miss!

12   Request: 12

Block b is fetched from memory

Memory

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

Block b is stored in cache
- Placement policy:
  determines where b goes
- Replacement policy:
  determines which block
  gets evicted (victim)

---
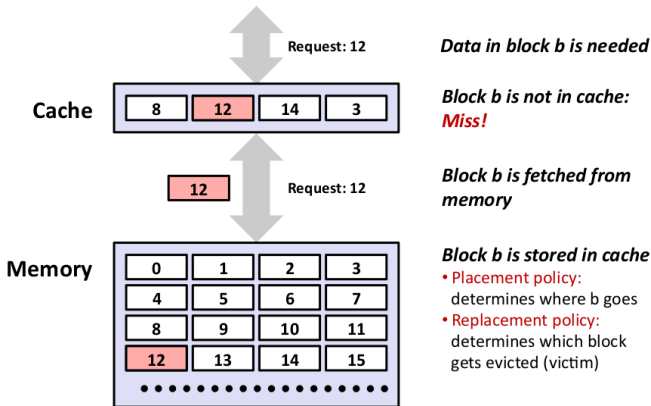
[0]From Bryant and O'Hallaron, Ch 6

Types of Cache Misses

- **Cold (compulsory) miss:**
  - Cold misses occur because the cache is empty.
- **Conflict miss:**
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - ★ E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - ★ E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
- **Capacity miss:**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.

Examples of Caching in the Memory Hierarchy

| Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	64-bytes block	On-Chip L1	1	Hardware
L2 cache	64-bytes block	On/Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

---

[0] From Bryant and O'Hallaron, Ch 6

Summary

- The speed gap between CPU, memory and mass storage continues to widen.
- Well-written programs exhibit a property called locality.
- Memory hierarchies based on caching close the gap by exploiting locality.

Principles of Caches

- Cache memories are small, fast SRAM-based memories managed automatically in hardware.
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory.
- Typical system structure:
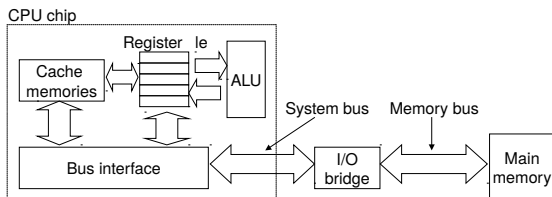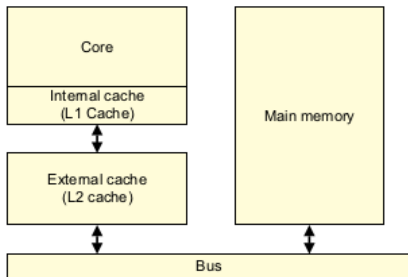
ARM Cortex A7 Cache Hierarchy

Figure 8-1 A basic cache arrangement

A cache is a small, fast block of memory that sits between the core and main memory. It holds copies of items in main memory. Accesses to the cache memory happen significantly faster than those to main memory. Because the cache holds only a subset of the contents of main memory, it must store both the address of the item in main memory and the associated data. Whenever the core wants to read or write a particular address, it will first look for it in the cache. If it finds the address in the cache, it will use the data in the cache, rather than having to perform an access to main memory. This significantly increases the potential performance of the system, by reducing the effect of slow external memory access times. It also reduces the power consumption of the system. NB: In many ARM-based systems, access to external memory will take 10s or 100s of cycles.
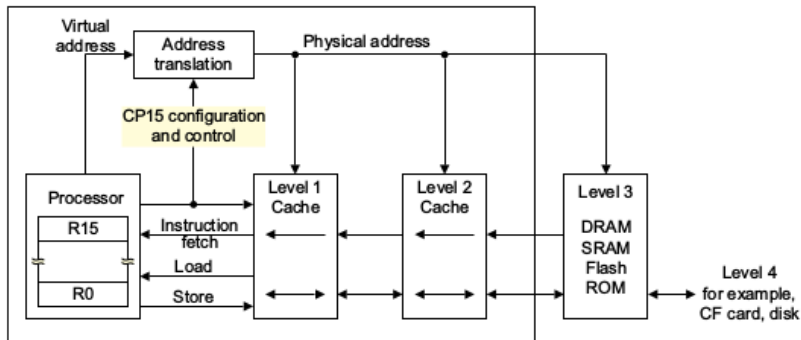
ARMv7-A Memory Hierarchy

Figure A3-6 Multiple levels of cache in a memory hierarchy

See ARM Architcture Reference, Ch A3, Fig A3.6, p.157

Caching policies: direct mapping

- The caching policy determines how to map addresses (and their contents) in main memory to locations in the chache.
- Since the cache is much smaller, several main memory addresses will be mapped to the same cache location.
- The role of the caching policy is to avoid such clashes as much as possible, so that the cache can be used for most memory read/write operations.
- The simplest caching policy is a **direct mapped cache**:
  - ▶ each location in main memory always maps to a single location in the cache
  - ▶ this policy is simple to implement, and therefore requires little hardware
  - ▶ a weakness of the policy is, that if two frequently used memory addresses map to the same cache address, this results in a lot of cache misses ("**cache thrashing**")
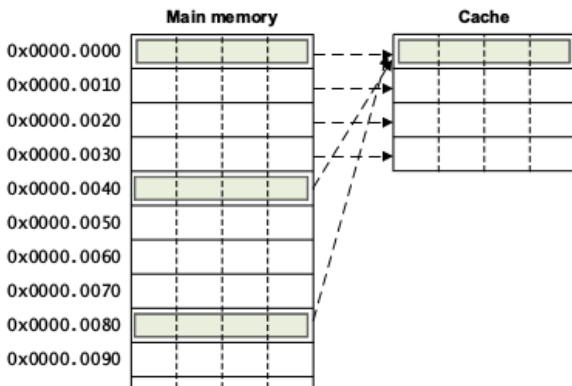
Direct mapped cache

Figure 8-4 Direct mapped cache operation

Caching policies: set-associative

- To eliminate the weakness of the direct-mapped caches, a more flexible **set-associative** cache can be used.
- With this policy, one memory location can map to one of several *ways* in the cache.
- Conceptually, each way represents a slice of the cache.
- Therefore, a main memory address can be mapped to any of these slices in the cache.
- Inside one such slice, however, the location is fixed.
- If the system uses *n* such slices ("ways") it is called an *n*-way associative cache.
- This avoids cache thrashing in cases where no more than *n* frequently used variables (memory locations) occur.

NB: The ARM Cortex A7 uses a 4-way set associative data cache, with cache size of 32kB, and a cache line size of 8 words
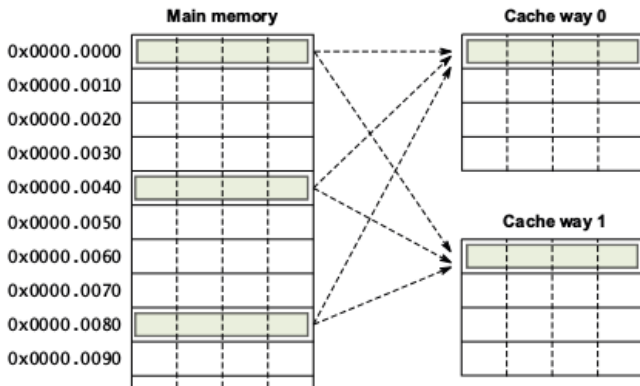
Set-associative cache

Figure 8-6 A 2-way set-associative cache

ARM cache features

Table 8-1 Cache features of Cortex-A series processors

	Processor					
	Cortex-A5	Cortex-A7	Cortex-A8	Cortex-A9	Cortex-A12	Cortex-A15
L2 Cache	External	Integrated	Integrated	External	Integrated	Integrated
L2 Cache size	-	128KB to 1MB[a]	0KB to 1MB[a]	-	256KB to 8MB	512KB to 4MB[a]
Cache Implementation (Data)	PIPT	PIPT	PIPT	PIPT	PIPT	PIPT
Cache Implementation (Instruction)	VIPT	VIPT	VIPT	VIPT	VIPT	PIPT
L1 Cache size (data)[a]	4K to 64K[a]	8KB to 64KB[a]	16/32KB[a]	16KB/32KB/64KB[a]	32KB	32KB
Cache size (Inst)[a]	4K to 64K[a]	8KB to 64KB[a]	16/32KB[a]	16KB/32KB/64KB[a]	32KB or 64KB	32KB
L1 Cache Structure	2-way set associative (Inst) 4-way set associative (Data)	2-way set associative (Inst) 4-way set associative (Data)	4-way set associative	4-way set associative (Inst) 4-way set associative (Data)	4-way set associative (Inst) 4-way set associative (Data)	2-way set associative (Inst) 2-way set associative (Data)
L2 Cache Structure	-	8-way set associative	8-way set associative	-	16-way set associative	16-way associative

ARM cache features

Table 8-1 Cache features of Cortex-A series processors (continued)

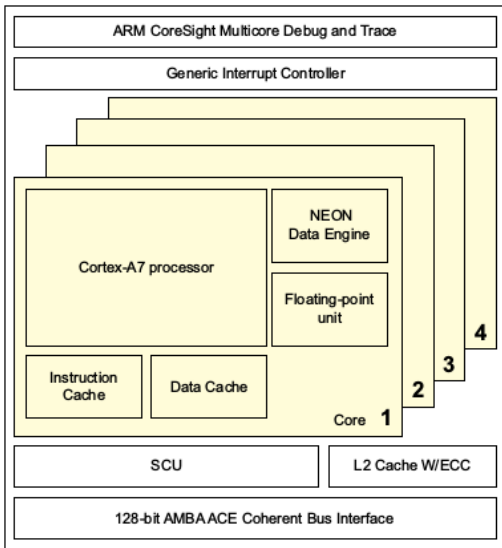|	Processor					
	Cortex-A5	Cortex-A7	Cortex-A8	Cortex-A9	Cortex-A12	Cortex-A15
Cache line (words)	8	8	16	8	-	16
Cache line (bytes)	32	64	64	32	64	64
Error protection	None	None	L2 ECC	None	L1 None, L2 ECC	Optional for L1 and L2

a. Configurable

ARM Cortex A7 Structure

Figure 2-4 Cortex-A7 processor

Example: Cache friendly code

See the background reading material on the web page:
Web aside on blocking in matrix multiplication

Summary: Memory Hierarchy

- In modern architectures the main memory is arranged in a **hierarchy of levels** ("memory hierarchy").
- Levels higher in the hierarchy (close to the processor) have fast access time but small capacity.
- Levels lower in the hierarchy (further from the processor) have slow access time but large capacity.
- Modern systems provide hardware (**caches**) and software (paging; configurable caching policies) support for managing the different levels in the hierarchy.
- The simplest caching policy uses **direct mapping**
- Modern ARM architectures use a more sophisticated **set associative** cache, that reduces "cache thrashing".
- For a programmer it's important to be aware of the impact of **spatial and temporal locality** on the performance of the program.
- Making good use of the cache can reduce runtime by a factor of ca. 3 as in our example of blocked matrix multiplication.

Lecture 5.
Exceptional Control Flow and Signals

What are interrupts and why do we need them?

- In order to deal with internal or external events, **abrupt** changes in control flow are needed.
- Such abrupt changes are also called **exceptional control flow (ECF)**.
- Informally, these are known as **hardware- and software-interrupts**.
- The system needs to take special action in these cases (call interrupt handlers, use non-local jumps)

ECF on different levels

ECF occurs at different levels:

- **hardware level:** e.g. arithmetic overflow events detected by the hardware trigger abrupt control transfers to **exception handlers**
- **operating system:** e.g. the kernel transfers control from one user process to another via **context switches**.
- **application level:** a process can send a signal to another process that abruptly transfers control to a signal handler in the recipient.

In this class we will cover **an overview of ECF with examples from the operating system level.**

Handling ECF on different levels

ECF is dealt with in different ways:

- **hardware level:** call an **interrupt** routine, typ. in Assembler
- **operating system:** call a **signal** handler, typ. in C
- **application level:** call an **exception** handler, e.g. in a Java `catch` block

Exceptions

Definition

An exception is an abrupt change in the control flow in response to some change in the processor's state.
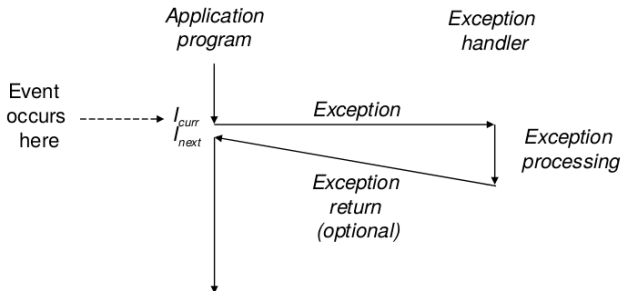
A change in the processor's state (event) triggers an abrupt control transfer (an **exception**) from the application program to an exception handler. After it finishes processing, the handler either returns control to the interrupted program or aborts.

Exceptions (cont'd)

When the processor detects that the event has occurred, it makes an indirect procedure call (the **exception**), through a jump table called an **exception table**, to an operating system subroutine (the **exception handler**) that is specifically designed to process this particular kind of event.

When the exception handler **finishes** processing, one of three things happens, depending on the type of event that caused the exception:

- The handler **returns control to the current instruction**, i.e. the instruction that was executing when the event occurred.

- The handler **returns control to the instruction that would have executed next** had the exception not occurred.

- The handler **aborts** the interrupted program.

Exceptions (cont'd)

When the processor detects that the event has occurred, it makes an indirect procedure call (the **exception**), through a jump table called an **exception table**, to an operating system subroutine (the **exception handler**) that is specifically designed to process this particular kind of event.

When the exception handler **finishes** processing, one of three things happens, depending on the type of event that caused the exception:

- The handler **returns control to the current instruction**, i.e. the instruction that was executing when the event occurred.
- The handler **returns control to the instruction that would have executed next** had the exception not occurred.
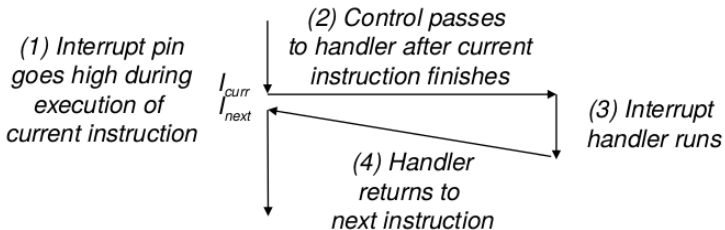- The handler **aborts** the interrupted program.

Interrupt handling

(1) Interrupt pin goes high during execution of current instruction

I_{curr}
I_{next}

(2) Control passes to handler after current instruction finishes

(3) Interrupt handler runs

(4) Handler returns to next instruction

The interrupt handler returns control to the next instruction in the application program's control flow.

Exception Handling

Exception Handling requires close cooperation between software and hardware.

- Each type of possible exception in a system is assigned a unique nonnegative integer **exception number.**
- Some of these numbers are assigned by the designers of the processor. Other numbers are assigned by the designers of the operating system kernel.
- At system **boot time** (when the computer is reset or powered on), the operating system allocates and initializes a jump table called an **exception table**, so that entry *k* contains the address of the handler for exception *k*.
- At **run time** (when the system is executing some program), the processor detects that an event has occurred and determines the corresponding exception number *k*. The processor then triggers the exception by making an **indirect procedure call**, through entry *k* of the exception table, to the corresponding handler.

Exception table
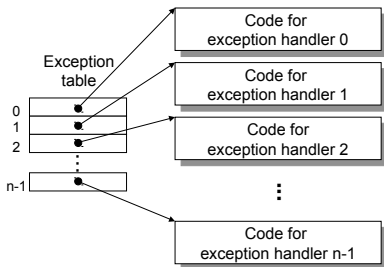
The exception table is a jump table where entry *k* contains the address of the handler code for exception *k*.

Differences between exception handlers and procedure calls

Calling an exception handler is similar to calling a procedure/method, but there are some important differences:

- Depending on the class of exception, the return address is either the current instruction or the next instruction.
- The processor also pushes some additional processor state onto the stack that will be necessary to restart the interrupted program when the handler returns.
- If control is being transferred from a user program to the kernel, all of these items are pushed onto the kernel's stack rather than onto the user's stack.
- Exception handlers run in kernel mode, which means they have complete access to all system resources.

Classes of exceptions

Exceptions can be divided into four classes: interrupts, traps, faults, and aborts:

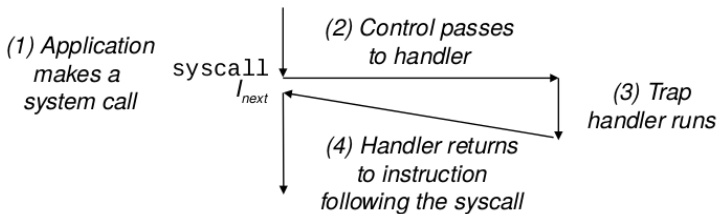| Class	Cause	(A)Sync	Return behavior
Interrupt	Signal from I/O device	Async	Always returns to next instr
Trap	Intentional exception	Sync	Always returns to next instr
Fault	Potent. recoverable error	Sync	Might return to current instr
Abort	Nonrecoverable error	Sync	Never returns

It is useful to distinguish 2 reasons for an exceptional control flow:

- an **exception** is **any** unexpected change in control flow;
  e.g. arithmetic overflow, using an undefined instruction, hardware timer
- an **interrupt** is an unexpected change in control flow triggered by an **external event**;
  e.g. I/O device request, hardware malfunction

Traps and System Calls

- **Traps** are **intentional exceptions** that occur as a result of executing an instruction.
- Traps are often used as an interface between application program and OS kernel.
- **Examples**: reading a file (`read`), creating a new process (`fork`), loading a new program (`execve`), or terminating the current process (`exit`).
- Processors provide a special *"syscall n"* instruction.
- This is exactly the `SWI` instruction on the ARM processor.

Trap Handling

(1) Application makes a system call   syscall   *(2) Control passes to handler*

I_{next}

(3) Trap handler runs

(4) Handler returns to instruction following the syscall

The **trap** handler returns control to the next instruction in the application program's control flow.

Faults

- **Faults** result from **error conditions** that a handler might be able to correct.
- Note that after fault handling, the processor typically reexecutes the same instruction.
- **Example**: page fault exception.
  - Assume an instruction references a virtual address whose corresponding physical page is not in memory.
  - In this case **page fault** is triggered.
  - The fault handler loads the required page into main memory.
  - After that **the same instruction** needs to be executed again.

Fault handling

(1) Current instruction causes a fault I_{curr}

(2) Control passes to handler

(3) Fault handler runs - - - - - - - - - - - - → `abort`

(4) Handler either reexecutes current instruction or aborts.
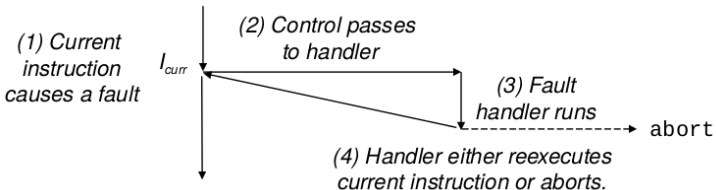
Depending on whether the fault can be repaired or not, the fault handler either reexecutes the faulting instruction or aborts.

Aborts

(1) Fatal hardware error occurs I_{curr} *(2) Control passes to handler*

(3) Abort handler runs   → abort

(4) Handler returns to abort *routine*
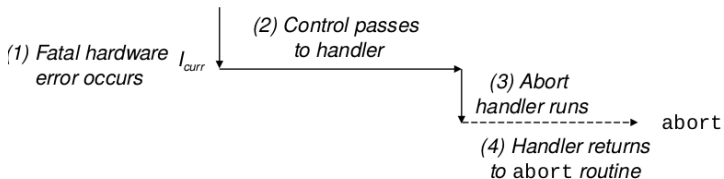
Aborts result from unrecoverable fatal errors, typically hardware errors such as parity errors that occur when DRAM or SRAM bits are corrupted. Abort handlers never return control to the application program.

Common system calls

Number	Name	Description
1	exit	Terminate process
2	fork	Create new process
3	read	Read file
4	write	Write file
5	open	Open file
6	close	Close file
7	waitpi	Wait for child to terminate
11	execve	Load and run program
19	lseek	Go to file offset
20	getpid	Get process ID

---

[0] For a more complete list see Smith, Appendix B "Raspbian System Calls"

Common system calls

Number	Name	Description
27	alarm	Set signal delivery alarm clock
29	pause	Suspend process until signal arrives
37	kill	Send signal to another process
48	signal	Install signal handler
63	dup2	Copy file descriptor
64	getppid	Get parent's process ID
65	getpgrp	Get process group
67	sigaction	Install portable signal handler
90	mmap	Map memory page to file
106	stat	Get information about file

---

[0]For the truly complete list see `/usr/include/sys/syscall.h`

Signal handlers in C

UNIX **signals** are a higher-level software form of exceptional control flow, that allows processes and the kernel to interrupt other processes.

- Signals provide a mechanism for exposing the occurrence of such exceptions to user processes.
- For example, if a process attempts to divide by zero, then the kernel sends it a SIGFPE signal (number 8).
- Other signals correspond to higher-level software events in the kernel or in other user processes.

---

[0] From Bryant and O'Hallaron, Sec 8.5

Signal handlers in C (cont'd)

- For example, if you type a `ctrl-c` (i.e. press the ctrl key and the c key at the same time) while a process is running in the foreground, then the kernel sends a `SIGINT` (number 2) to the foreground process.
- A process can forcibly terminate another process by sending it a `SIGKILL` signal (number 9).
- When a child process terminates or stops, the kernel sends a `SIGCHLD` signal (number 17) to the parent.

Signal handling
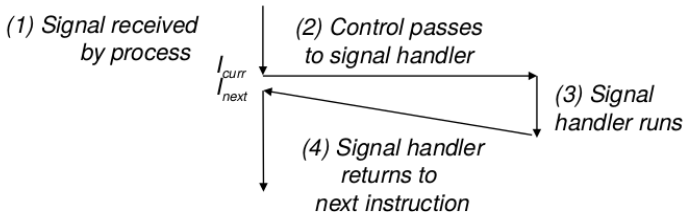
Receipt of a signal triggers a control transfer to a signal handler. After it finishes processing, the handler returns control to the interrupted program.

Example: handling `ctrl-c`

```c
// header files
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void ctrlc_handler(int sig) {
  fprintf(stderr, "Received signum %d; thank you for pressing
      CTRL-C\n", sig);
  exit(1);
}

int main() {
  signal(SIGINT, ctrlc_handler);   // install the signal handler
  while (1) { } ; // infinite loop
}
```

Example: handling `ctrl-c` in more detail

See `signal2.c`

Example: sending SIGALARM by the kernel

```c
/* signal handler, i.e. the fct called when a signal is
    received */
void handler(int sig)
{
    static int beeps = 0;

    printf("BEEP %d\n", beeps+1);
    if (++beeps < 5)
        alarm(1); /* Next SIGALRM will be delivered in 1 second
            */
    else {
        printf("BOOM!\n");
        exit(1);
    }
}

int main() {
    signal(SIGALRM, handler); /* install SIGALRM handler; see:
        man 2 signal */
    alarm(1); /* Next SIGALRM will be delivered in 1s; see: man
```

Timers

- We now want to use timers, i.e. setting up an interrupt in regular intervals.
- The BCM2835 chip as an on-board timer for time-sensitive operations.
- We will explore three ways of achieving this:
  - using C library calls (on top of Raspbian)
  - using assembler-level system calls (to the kernel running inside Raspbian)
  - by directly probing the on-chip timer available on the RPi2
- In this section we will cover **how to use the on-chip timer to implement a simple timeout function in C**

Overview

Features of the different approaches:

- C library calls (on top of Raspbian)
    - are **portable** across hardware and OS
    - require a (system) library for handling the timer
- assembler-level system calls (to the kernel running inside Raspbian)
    - **depend** on the OS, but are **portable** across hardware
    - require a support for software-interrupts in the OS kernel
- directly probing the on-chip timer available on the RPi2
    - **depend** on both hardware and OS
    - the instructions for probing a hardware timer are specific to the hardware

Example: C library functions for controlling timers

getitimer, setitimer - get or set value of an interval timer

```
#include <sys/time.h>

int getitimer(int which, struct itimerval *curr_value);
int setitimer(int which, const struct itimerval *
   new_value,
              struct itimerval *old_value);
```

setitimer sets up an interval timer that issues a signal in an interval specified by the *new_value* argument, with this structure:

```
struct itimerval {
    struct timeval it_interval; /* next value */
    struct timeval it_value;    /* current value */
  };
  struct timeval {
    time_t      tv_sec;          /* seconds */
    suseconds_t tv_usec;         /* microseconds */
  };
```

C library functions for controlling timers

There are three kinds of timers, specified by the *which* argument:

- ITIMER_REAL decrements in real time, and delivers SIGALRM upon expiration.

- ITIMER_VIRTUAL decrements only when the process is executing, and delivers SIGVTALRM upon expiration.

- ITIMER_PROF decrements both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

---

[0]See: man getitimer

Programming a C-level signal handler

Signals (or software interrupts) can be programmed on C level by associating a C function with a signal sent by the kernel.
sigaction - examine and change a signal action

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

The sigaction structure is defined as something like:

```
    struct sigaction {
            void     (*sa_handler)(int);
            void     (*sa_sigaction)(int, siginfo_t *,
                 void *);
            sigset_t   sa_mask;
            int        sa_flags;
            void     (*sa_restorer)(void);
```

NB: the `sa_handler` or `sa_sigaction` fields define the action to be performed when the signal with the id `signum` is sent.

[0]See `man sigaction`

Programming Timers using C library calls

We need the following headers:

```c
#include <signal.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <sys/time.h>

// in micro-sec
#define DELAY 250000
```

Programming Timers using C library calls

```c
int main ()
{
 struct sigaction sa;
 struct itimerval timer;

 fprintf(stderr, "configuring a timer with a delay of %d
    micro-seconds ...\n", DELAY);

 /* Install timer_handler as the signal handler for
    SIGALRM. */
 memset (&sa, 0, sizeof (sa));
 sa.sa_handler = &timer_handler;
 sigaction (SIGALRM, &sa, NULL);
```

Calling `sigaction` like this, causes the function `timer_handler` to
be called whenever signal `SIGALRM` arrives.

Programming Timers using C library calls

Now, we need to set-up a timer to send `SIGALRM` every `DELAY` micro-seconds:

```
/* Configure the timer to expire after 250 msec... */
timer.it_value.tv_sec = 0;
timer.it_value.tv_usec = DELAY;
/* ... and every 250 msec after that. */
timer.it_interval.tv_sec = 0;
timer.it_interval.tv_usec = DELAY;
/* Start a real timer. It counts down whenever this
   process is executing. */
setitimer (ITIMER_REAL, &timer, NULL);

/* A busy loop, doing nothing but accepting signals */
while (1) {} ;
}
```

---

[0]Sample source in itimer11.c

Further Reading & Hacking

📖 Randal E. Bryant, David R. O'Hallaron *"Computer Systems: A Programmers Perspective"*,
3rd edition, Pearson, 7 Oct 2015. ISBN-13: 978-1292101767.
Chapter 8: Exceptional Control Flow

📖 David A. Patterson, John L. Hennessy. *"Computer Organization and Design: The Hardware/Software Interface"*,
ARM edition, Morgan Kaufmann, Apr 2016. ISBN-13: 978-0128017333.
Section 4.9: Exceptions

📖 tewart Weiss. *"UNIX Lecture Notes"*
Chapter 5: Interactive Programs and Signals
Department of Computer Science, Hunter College, 2011

Summary

- **Interrupts trigger an exceptional control flow**, to deal with special situations.
- Interrupts can occur at several levels:
  - ▶ hardware level, e.g. to report hardware faults
  - ▶ OS level, e.g. to switch control between processes
  - ▶ application level, e.g. to send signals within or between processes
- The **concept** is the same on all levels: execute a short sequence of code, to deal with the special situation.
- Depending on the source of the interrupt, execution will continue with the same, the next instruction or will be aborted.
- The **mechanisms** how to implement this behaviour are different: in software on application level, in hardware with jumps to entries in the interrupt vector table on hardware level

Lecture 6: Computer Architecture

Classes of Computer Architectures

- There is a wide range of computer architectures from small-scale (embedded) to large-scale (super-computers)
- In this course we focus on **embedded systems**
- A key requirement for these devices is **low power consumption**
- This is also increasingly important for main-stream hardware and even for super-computing
- Embedded devices are found in cars, planes, house-hold devices, network-devices, cell-phones etc
- This is the most rapidly growing market for computer hardware
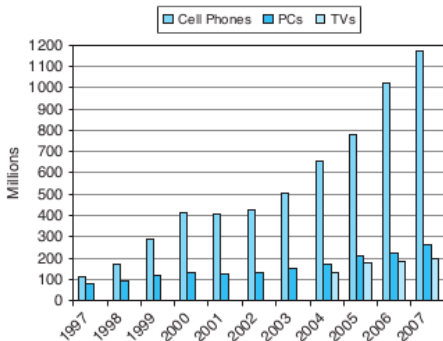
Number of processors produced

FIGURE 1.1 The number of cell phones, personal computers, and televisions manufactured per year between 1997 and 2007. (We have television data only from 2004.) More than a billion new cell phones were shipped in 2006. Cell phones sales exceeded PCs by only a factor of 1.4 in 1997, but the ratio grew to 4.5 in 2007. The total number in use in 2004 is estimated to be about 2.0B televisions, 1.8B cell phones, and 0.8B PCs. As the world population was about 6.4B in 2004, there were approximately one PC, 2.2 cell phones, and 2.5 televisions for every eight people on the planet. A 2006 survey of U.S. families found that they owned on average 12 gadgets, including three TVs, 2 PCs, and other devices such as game consoles, MP3 players, and cell phones.

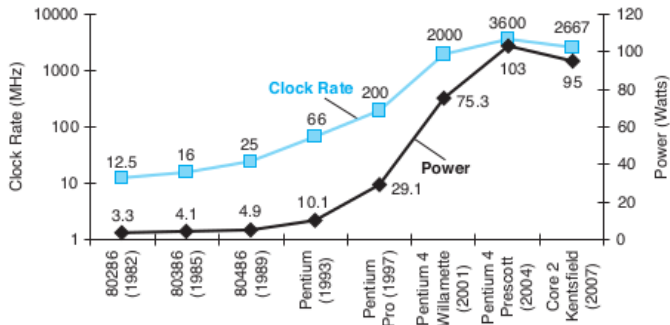# Limitations to further improvements

FIGURE 1.15 Clock rate and Power for Intel x86 microprocessors over eight generations and 25 years. The Pentium 4 made a dramatic jump in clock rate and power but less so in performance. The Prescott thermal problems led to the abandonment of the Pentium 4 line. The Core 2 line reverts to a simpler pipeline with lower clock rates and multiple processors per chip.

Processor Architectures: Introduction

- In this part we take a brief look at the design of processor hardware.
- This view will give you a better understanding of how computers work.
- In particular you will gain a better understanding of issues relevant to **resource consumption.**
- So far we have used a very simple model of a CPU: each instruction is fetched and executed to completion before the next one begins.
- Modern processor architectures use **pipeling** to execute multiple instructions simultaneously ("super-scalar architectures").
- Special measures need to be taken to ensure that the processor computes the same results as it would with sequential execution.

A simple picture of the CPU
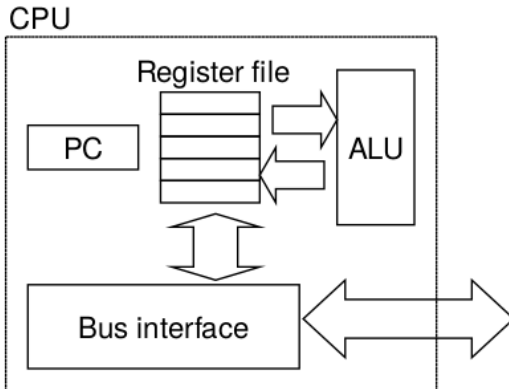
- The ALU executes arithmetic/logic operations with arguments in registers
- Load and store instructions move data between memory and registers

A simple picture of the CPU
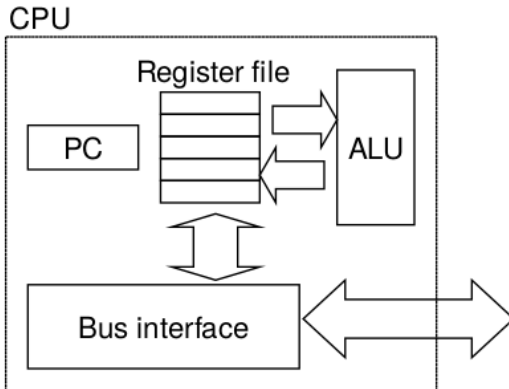
- The ALU executes arithmetic/logic operations with arguments in registers
- Load and store instructions move data between memory and registers

Why should you learn about architecture design?

- It is intellectually interesting and important.
- Understanding how the processor works aids in understanding how the overall computer system works.
- Although few people design processors, many design hardware systems that contain processors.
- You just might work on a processor design.

Stages of executing an assembler instruction

Processing an assembler instruction involves a number of operations:

1. **Fetch:** The fetch stage reads the bytes of an instruction from memory, using the program counter (PC) as the memory address.
2. **Decode:** The decode stage reads up to two operands from the register file.
3. **Execute:** In the execute stage, the arithmetic/logic unit (ALU) either performs the operation specified by the instruction, computes the effective address of a memory reference, or increments or decrements the stack pointer.
4. **Memory:** The memory stage may write data to memory, or it may read data from memory.
5. **Write back:** The write-back stage writes up to two results to the register file.
6. **PC update:** The PC is set to the address of the next instruction.

NB: The processing depends on the instruction, and certain stages may not be used.

Unpipelined computation hardware

Unpipelined Design

Delay: **320 ps**

Throughput: **3.12 GIPS**

(a) Hardware: Unpipelined

300 ps   20 ps

Combinational logic

R e g

Delay = 320 ps
Throughput = 3.12 GIPS

Clock

I1

I2

I3

Time
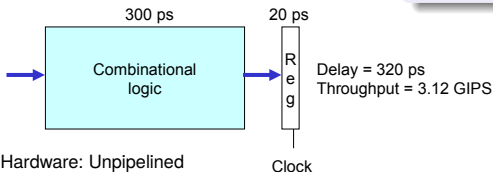
(b) Pipeline diagram

On each 320 ps cycle, the system spends 300 ps evaluating a combinational logic function and 20 ps storing the results in an output register.

[0] From Bryant, Chapter 4

Instruction-level parallelism

- **Key observation:** We can do the different stages of the execution in parallel ("instruction-level parallelism")
- An architecture that allows this kind of parallelism is called **"pipelined"** architecture
- This is a big performance boost: ideally each instruction takes just 1 cycle (as opposed to 5 cycles for the 5 stages of the execution)
- However, the ideal case is often not reached, and modern architecture play clever tricks to get closer to the ideal case: branch prediction, out-of-order execution etc

Three-stage pipelined computation hardware
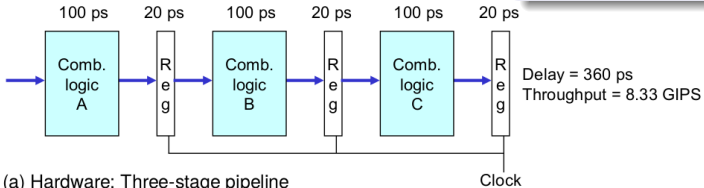
(a) Hardware: Three-stage pipeline
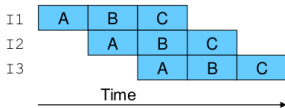
(b) Pipeline diagram

The computation is split into stages A, B, and C. On each 120-ps cycle, each instruction progresses through one stage.

---

[0] From Bryant, Chapter 4

Three-stage pipeline timing
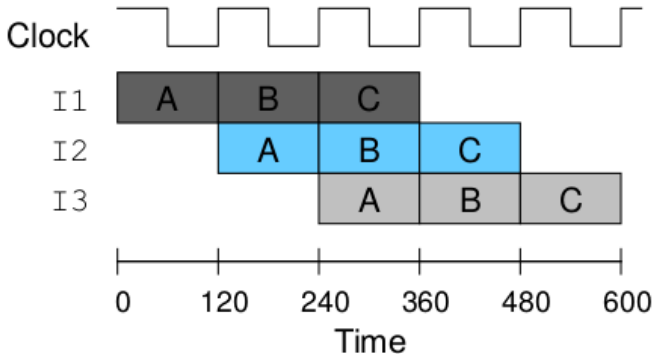
The rising edge of the clock signal controls the movement of instructions from one pipeline stage to the next.

Example: One clock cycle of pipeline operation.
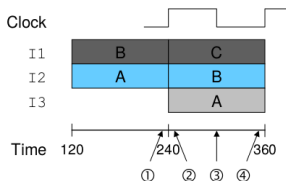
- We now take a closer look on how values are propagated through the pipeline.
- Instruction I1 has completed stage B
- Instruction I2 has completed stage A

Example: One clock cycle of pipeline operation.

① Time = 239
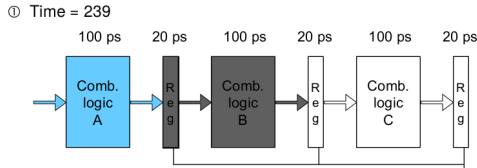
Just **before** clock rise: values have been computed (stage A of instruction I2, stage B of instruction I1), but the pipeline registers have not been updated, yet.

Example: One clock cycle of pipeline operation.
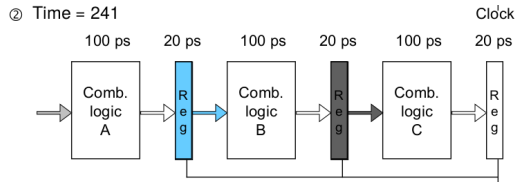
On **clock rise**, inputs are loaded into the pipeline registers.

Example: One clock cycle of pipeline operation.
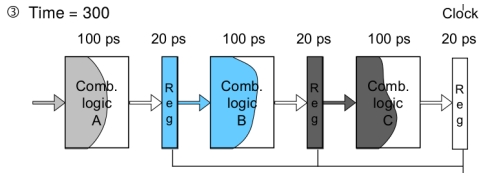
Signals then propagate through the combinational logic (possibly at different rates).

Example: One clock cycle of pipeline operation.
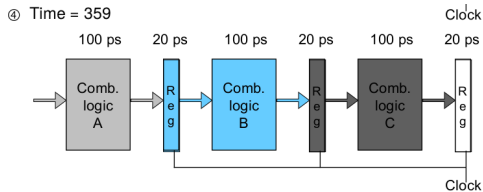
Before time 360, the result values reach the inputs of the pipeline registers, to be propagated at the next rising clock.

---

[0] From Bryant, Chapter 4

Multiple-clock-cycle pipeline diagram
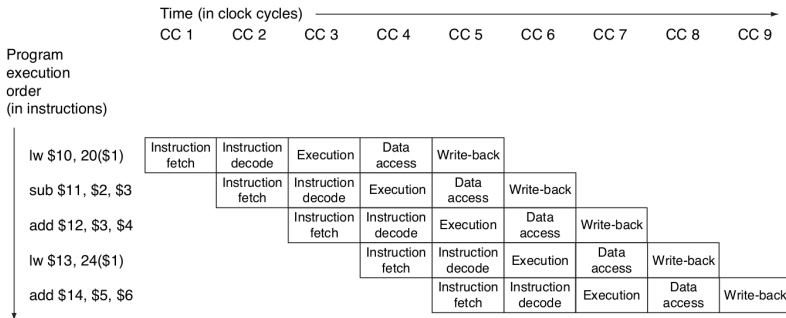
FIGURE 4.44 Traditional multiple-clock-cycle pipeline diagram of five instructions in **Figure 4.43**.

Abstract view of a sequential processor
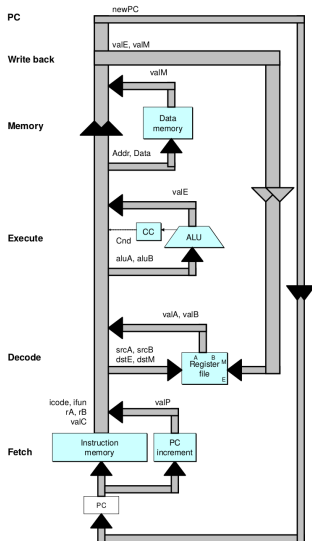
The information processed during execution of an instruction follows a clockwise flow starting with an instruction fetch using the program counter (PC), shown in the lower left-hand corner of the figure.

Discussion of pipelined execution

The main pipeline **stages** are:

- **Fetch:** Using the program counter register as an address, the instruction memory reads the bytes of an instruction. The PC incrementer computes valP, the incremented program counter.
- **Decode:** The register file has two read ports, A and B, via which register values valA and valB are read simultaneously.
- **Execute:** This uses the arithmetic/logic (ALU) unit for different purposes according to the instruction type: integer operations, memory access, or branch instructions.
- **Memory:** The Data Memory unit reads or writes a word of memory (memory instruction). The instruction and data memories access the same memory locations, but for different purposes.
- **Write back:** The register file has two write ports. Port E is used to write values computed by the ALU, while port M is used to write values read from the data memory.

Abstract view of a pipelined processor
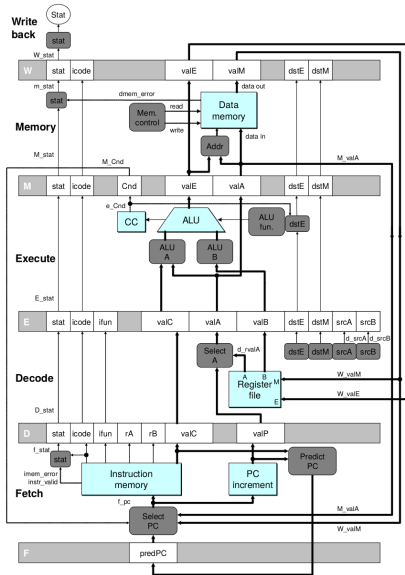
Hardware structure of a pipelined implementation. By inserting pipeline registers between the stages, we create a five-stage pipeline.

[0] From Bryant, Chapter 4

Pipeline registers

The pipeline **registers** are labeled as follows:

- **F** holds a predicted value of the program counter.
- **D** sits between the fetch and decode stages. It holds information about the most recently fetched instruction for processing by the decode stage.
- **E** sits between the decode and execute stages. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage.
- **M** sits between the execute and memory stages. It holds the results of the most recently executed instruction for processing by the memory stage. It also holds information about branch conditions and branch targets for processing conditional jumps.
- **W** sits between the memory stage and the feedback paths that supply the computed results to the register file for writing and the return address to the PC selection logic when completing a return instruction.

Example of instruction flow through pipeline
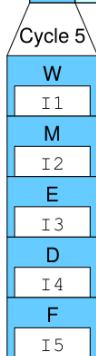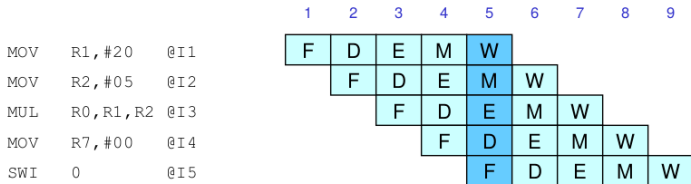
The ARM picture

The pipeline in the BCM2835 SoC for the RPi has 8 pipeline stages:

1. **Fe1:** The first Fetch stage, where the address is sent to memory and an instruction is returned.
2. **Fe2:** Second fetch stage, where the processor tries to predict the destination of a branch.
3. **De:** Decoding the instruction.
4. **Iss:** Register read and instruction issue
5. **Only for ALU operations:**
   1. **Sh:** Perform shift operations as required.
   2. **ALU:** Perform arithmetic/logic operations.
   3. **Sat:** Saturate integer results.
6. **WBi:** Write back of data from any of the above sub-pipelines.

---

[0] See slidesRPiArch and the table in Smith's book

The ARM picture

The pipeline in the BCM2835 SoC for the RPi has 8 pipeline stages:

1. **Fe1:** The first Fetch stage, where the address is sent to memory and an instruction is returned.
2. **Fe2:** Second fetch stage, where the processor tries to predict the destination of a branch.
3. **De:** Decoding the instruction.
4. **Iss:** Register read and instruction issue
5. **Only for Multiply operations:**
   1. **MAC1:** First stage of the multiply-accumulate pipeline.
   2. **MAC2:** Second stage of the multiply-accumulate pipeline.
   3. **MAC3:** Third stage of the multiply-accumulate pipeline.
6. **WBi:** Write back of data from any of the above sub-pipelines.

---

[0] See slidesRPiArch and the table in Smith's book

The ARM picture

The pipeline in the BCM2835 SoC for the RPi has 8 pipeline stages:

1. **Fe1:** The first Fetch stage, where the address is sent to memory and an instruction is returned.
2. **Fe2:** Second fetch stage, where the processor tries to predict the destination of a branch.
3. **De:** Decoding the instruction.
4. **Iss:** Register read and instruction issue
5. **Only for Load/Store operations:**
   1. **ADD:** Address generation stage.
   2. **DC1:** First stage of data cache access.
   3. **DC2:** Second stage of data cache access.
6. **WBi:** Write back of data from any of the above sub-pipelines.

---

[0] See slidesRPiArch and the table in Smith's book

Pipelining and branches

- How can a pipelined architecture deal with conditional branches?
- In this case the processor doesn't know the successor instruction until further down the pipeline.
- To deal with this, modern architectures perform some form of **branch prediction** in hardware.
- There are two forms of branch prediction:
  - static branch prediction always takes the same guess (e.g. guess **always taken**)
  - dynamic branch prediction uses the history of the execution to take better guesses
- Performance is significantly higher when branch predictions are correct
- If they are wrong, the processor needs to stall or inject bubbles into the pipeline

Example: bad branch prediction

```
        .global _start
        .text
_start: EORS R1, R1, R1  @ always 0
        BNE  target      @ Not taken
        MOV  R0, #11      @ fall through
        MOV  R7, #1
        SWI  0
target: MOV  R0, #1
        MOV  R7, #1
        SWI  0
```

Branch prediction: we assume the processor takes an **always taken** policy, i.e. it always assumes that that a branch is taken
NB: the conditional branch (BNE) will never be taken, because exclusive-or with itself always gives 0, i.e. this is a deliberately bad example for the branch predictor

Processing mispredicted branch instructions.
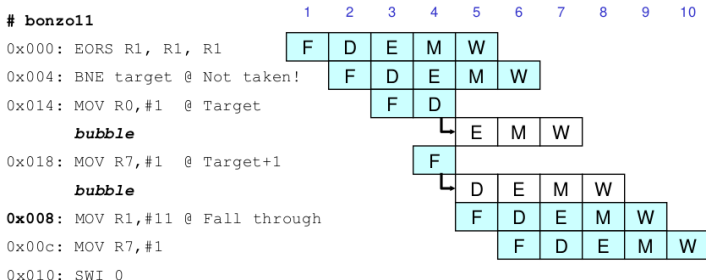

```
# bonzo11
          1   2   3   4   5   6   7   8   9   10
0x000: EORS R1, R1, R1    F   D   E   M   W
0x004: BNE target @ Not taken!   F   D   E   M   W
0x014: MOV R0,#1 @ Target            F   D
       bubble                                E   M   W
0x018: MOV R7,#1 @ Target+1              F
       bubble                                D   E   M   W
0x008: MOV R1,#11 @ Fall through             F   D   E   M   W
0x00c: MOV R7,#1                                 F   D   E   M   W
0x010: SWI 0
```

- Predicting "branch taken", instruction `0x014` is fetched in cycle 3, and instruction `0x018` is fetched in cycle 4.
- In cycle 4 the branch logic detects that the branch is **not** taken
- It therefore abandons the execution of `0x014` and `0x018` by injecting **bubbles** into the pipeline.
- The result will be as expected, but performance is sub-optimal!

[0] Adapted from Bryant, Figure 4.62

Example of bad branch prediction

Code example: `sumav3_asm`

Hazards of Pipelining

- Pipelining complicates the processing of instructions because of:
  - **Control hazards**, where branches are mis-predicted (as we have seen)
  - **Data hazards**, where data dependencies exist between subsequent instructions
- Several ways exist to solve these problems:
  - To deal with **control hazards**, branch prediction is used and, if necessary, partially executed instructions are abandoned.
  - To deal with **data hazards**, bubbles can be injected to delay the execution of instructions, or data in pipeline registers (but not written back) can be forwarded to other stages in the pipeline.
- A lot of the complexities in modern processors is due to deep pipelining, (possibly dynamic) branch prediction, and forwarding of data

For details on pipelining and data hazards, see Bryant & O'Hallaron, *Computer Systems: A Programmer's View*, Chapter 4 (especially Sec 4.4 and 4.5).

Data Hazards

- The branch-prediction example above was a case of a **control hazard**.
- Now we look into a simple example of a **data hazard**.
- Consider the following simple ARM assembler program:

```
ADD    R3, R1, R2    @  R3 = R1 + R2
SUB    R0, R3, R4    @  R0 = R3 - R4
```

- Note, the result from the first instruction, in `R3`, will only become available in the **write-back** (5th) stage
- But, the data in `R3` is needed already in the **decode** (2nd) stage of the second instruction
- Without intervention, this would stall the pipeline, similar to the branch-mis-prediction case
- The solution to this is to introduce **forwarding** (or by-passing) to the hardware of the processor

A Graphical Representation of Forwarding
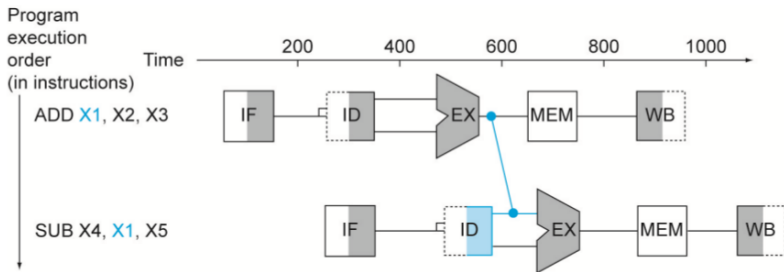
Example: Reordering Code to Avoid Pipeline Stalls

- We have previously examined, how C expressions are compiled to Assembler code. For example, consider this C program fragment:

```
int a, b, c, d, e, f;
a = b + e;
c = b + f;
```

- Knowing about control and data hazards motivates **reordering of code** that should be done by the compiler to avoid pipeline stalls.
- Such reordering is commonly done in the backend of compilers.
- Therefore, the sequence of Assembler instructions might be different from the one you expect.

Data layout and code for a C expression
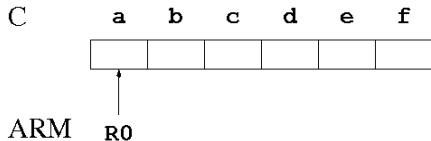
```
C      a   b   c   d   e   f        a = b + e

       |   |   |   |   |   |   |
                                    LDR  R1, [R0, #4]
ARM   R0                            LDR  R2, [R0, #16]
                                    ADD  R3, R1, R2
                                    STR  R3, [R0, #0]
```

Example: Reordering Code to Avoid Pipeline Stalls

Example: Translate the following C expression into Assembler:

```c
int a, b, c, d, e, f;
a = b + e;
c = b + f;
```

Example: We assume the variables are stored in memory, starting
from the location held in register R0. Here is the naive Assembler code:

```
LDR   R1, [R0, #4]     @ load b
LDR   R2, [R0, #16]    @ load e
ADD   R3, R1, R2       @ b + e
STR   R3, [R0, #0]     @ store a
LDR   R4, [R0, #20]    @ load f
ADD   R5, R1, R4       @ b + f
STR   R5, [R0, #12]    @ store c
```

Can you spot the data hazard in this example?

Example: Reordering Code to Avoid Pipeline Stalls

Example: Translate the following C expression into Assembler:

```
int a, b, c, d, e, f;
a = b + e;
c = b + f;
```

Example: We assume the variables are stored in memory, starting from the location held in register `R0`. Here is the naive Assembler code:

```
LDR  R1, [R0, #4]     @ load b
LDR  R2, [R0, #16]    @ load e
ADD  R3, R1, R2       @ b + e
STR  R3, [R0, #0]     @ store a
LDR  R4, [R0, #20]    @ load f
ADD  R5, R1, R4       @ b + f
STR  R5, [R0, #12]    @ store c
```

Can you spot the data hazard in this example?

A Graphical Representation of a Load-Store Hazard

Example: Reordering Code to Avoid Pipeline Stalls

Example: Translate the following C expression into Assembler:

```c
int a, b, c, d, e, f;
a = b + e;
c = b + f;
```

Example: The reordered Assembler code, eliminating the data hazard:

```
LDR  R1, [R0, #4]    @ load b
LDR  R2, [R0, #16]   @ load e
LDR  R4, [R0, #20]   @ load f; moved_up
ADD  R3, R1, R2      @ b + e
STR  R3, [R0, #0]    @ store a
ADD  R5, R1, R4      @ b + f
STR  R5, [R0, #12]   @ store c
```

Moving the third `LDR` instruction upward, makes its result available soon enough to avoid a pipeline stall.

Summary: Processor Architecture and Pipelining

- Modern ("super-scalar") processors can execute several instructions at the same time, by organising the execution of an instruction into several stages and using a **pipeline** structure.
- This exploits **instruction-level** parallelism and boosts performance.
- However, there is a risk of control and data hazards, leading to reduced performance, e.g. due to poor branch prediction
- Knowing these risks, you can develop faster code!
- These code transformations are often done internally by the compiler.

Lecture 10: Revision

A simple picture of the CPU
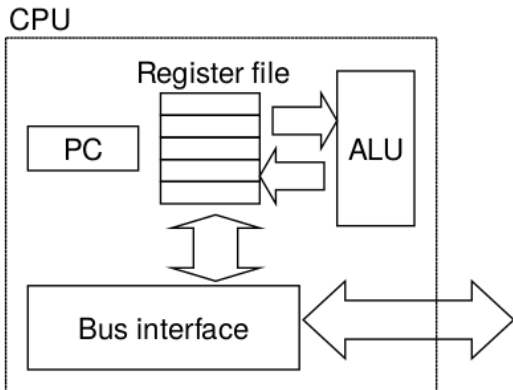
- The ALU executes arithmetic/logic operations with arguments in registers
- Load and store instructions move data between memory and registers

A simple picture of the CPU
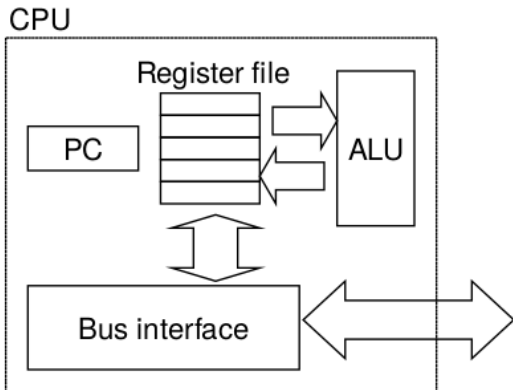
- The ALU executes arithmetic/logic operations with arguments in registers
- Load and store instructions move data between memory and registers

Three-stage pipelined computation hardware

(a) Hardware: Three-stage pipeline

(b) Pipeline diagram
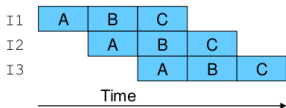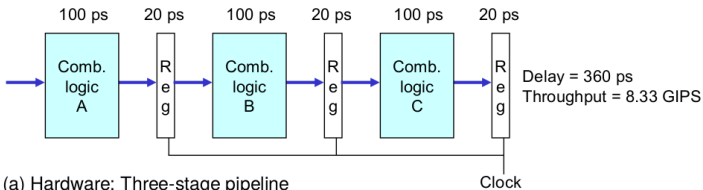
The computation is split into stages A, B, and C. The stages for different instructions can be executed in an overlapping way.

---

[0]From Bryant, Chapter 4

Stages of executing an assembler instruction

Processing an assembler instruction involves a number of operations:

1. **Fetch:** The fetch stage reads the bytes of an instruction from memory, using the program counter (PC) as the memory address.
2. **Decode:** The decode stage reads up to two operands from the register file.
3. **Execute:** In the execute stage, the arithmetic/logic unit (ALU) either performs the operation specified by the instruction, computes the effective address of a memory reference, or increments or decrements the stack pointer.
4. **Memory:** The memory stage may write data to memory, or it may read data from memory.
5. **Write back:** The write-back stage writes up to two results to the register file.
6. **PC update:** The PC is set to the address of the next instruction.

NB: The processing depends on the instruction, and certain stages may not be used.

Pipelining and branches

- How can a pipelined architecture deal with conditional branches?
- In this case the processor doesn't know the successor instruction until further down the pipeline.
- To deal with this, modern architectures perform some form of **branch prediction** in hardware.
- There are two forms of branch prediction:
  - ▶ static branch prediction always takes the same guess (e.g. guess **always taken**)
  - ▶ dynamic branch prediction uses the history of the execution to take better guesses
- Performance is significantly higher when branch predictions are correct
- If they are wrong, the processor needs to stall or inject bubbles into the pipeline

Example: bad branch prediction

```
        .global _start
        .text
_start: MOVS  R1, #0      @ load 0 =>
        LSR   R1, #1      @ LSR yields zero =>
        BNE   target      @ Not taken
        MOV   R0, #0      @ fall through
        MOV   R7, #1
        SWI   0
target: MOV   R0, #1      @ return: branch taken?
        MOV   R7, #1
        SWI   0
```

Branch prediction: we assume the processor takes an **always taken** policy, i.e. it always assumes that that a branch is taken

NB: the conditional branch (BNE) will **NOT** be taken, because the right shift (LSR) will set the zero flag according to the right-most bit, which is **0** in this case. This is a deliberately **bad** example for the branch

Example: good branch prediction

```
        .text
_start: MOVS  R1, #1        @ load 1 =>
        LSR   R1, #1        @ LSR yields one =>
        BNE   target        @ Branch taken
        MOV   R0, #0        @ fall through
        MOV   R7, #1
        SWI   0
target: MOV   R0, #1        @ return: branch taken?
        MOV   R7, #1
        SWI   0
```

Branch prediction: we assume the processor takes an **always taken** policy, i.e. it always assumes that that a branch is taken
NB: now the conditional branch (BNE) **WILL** be taken, because the right shift (LSR) will set the zero flag according to the right-most bit, which is **1** in this case. This is better for the branch predictor and gives better performance.

Performance: good vs bad branch prediction

We now measure the performance of doing these two versions inside two nested loops (`0x10000` iterations, each).
Good Case: branch taken:

```
> hawopi[167](4.2)> as -o bonzo15.o bonzo15.s
> hawopi[168](4.2)> ld -o bonzo15 bonzo15.o
> hawopi[169](4.2)> time ./bonzo15
 real   0m30.091s
 user   0m29.980s
 sys    0m0.000s
> hawopi[170](4.2)> echo $?
 1
```

Performance: good vs bad branch prediction

We now measure the performance of doing these two versions inside two nested loops (`0x10000` iterations, each).
Bad Case: branch NOT taken:

```
> hawopi[171](4.2)> as -o bonzo15.o bonzo15.s
> hawopi[172](4.2)> ld -o bonzo15 bonzo15.o
> hawopi[173](4.2)> time ./bonzo15
 real    0m36.188s
 user    0m34.900s
 sys     0m0.090s
> hawopi[174](4.2)> echo $?
 0
```

NB: a difference in runtime of ca. 16.8%

Processing mispredicted branch instructions.
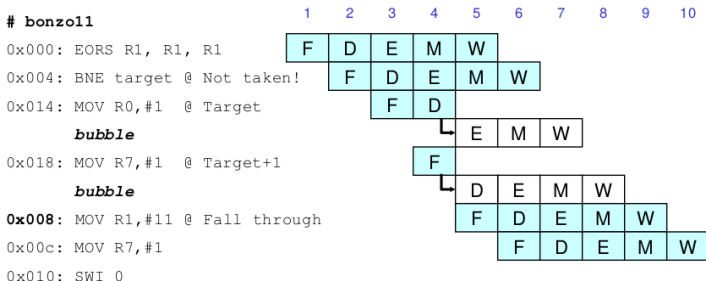

```
# bonzo11                           1   2   3   4   5   6   7   8   9   10
0x000: EORS R1, R1, R1              F   D   E   M   W
0x004: BNE target @ Not taken!          F   D   E   M   W
0x014: MOV R0,#1 @ Target                   F   D
       bubble                                   └─  E   M   W
0x018: MOV R7,#1 @ Target+1                      F
       bubble                                       └─  D   E   M   W
0x008: MOV R1,#11 @ Fall through                     F   D   E   M   W
0x00c: MOV R7,#1                                         F   D   E   M   W
0x010: SWI 0
```

- Predicting "branch taken", instruction $0x014$ is fetched in cycle 3, and instruction $0x018$ is fetched in cycle 4.
- In cycle 4 the branch logic detects that the branch is **not** taken
- It therefore abandons the execution of $0x014$ and $0x018$ by injecting **bubbles** into the pipeline.
- The result will be as expected, but performance is sub-optimal!

⁰Adapted from Bryant, Figure 4.62

The Current Program Status Register (CPSR)

The Current Program Status Register (CPSR) contains flags
(V, Z, N, C) that are set by certain assembler instructions.
For example, the `CMP R0, R1` instruction compares the values of
registers `R0` and `R1` and sets the zero flag (Z) if `R0` and `R1` are equal.

Figure 3-6 shows the bit assignments in the CPSR.
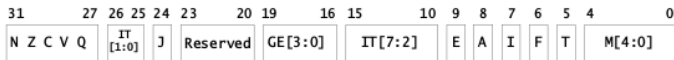
31					27	26 25	24	23		20	19		16	15		10	9	8	7	6	5	4			0
N	Z	C	V	Q		IT[1:0]	J	Reserved			GE[3:0]			IT[7:2]			E	A	I	F	T		M[4:0]		

Figure 3-6 CPSR bits

Caches and Memory Hierarchy

Discussion

As we move from the top of the hierarchy to the bottom, the devices become **slower, larger, and less costly** per byte.

The main idea of a memory hierarchy is that **storage at one level serves as a cache for storage at the next lower level**.

Using the different levels of the memory hierarchy efficiently is crucial to achieving high performance.

Access to levels in the hierarchy can be explicit (for example when using OpenCL to program a graphics card), or implicit (in most other cases).

Importance of Locality

Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer!

Which of the following two version of sum-over-matrix has better locality (and performance):

Traversal by rows:

```
int i, j;  ulong  sum;
for (i = 0; i<n; i++)
  for (j = 0; j<n; j++)
    sum += arr[i][j];
```

Traversal by columns:

```
int i, j; ulong  sum;
for (j = 0; j<n; j++)
  for (i = 0; i<n; i++)
    sum += arr[i][j];
```

The high-level picture
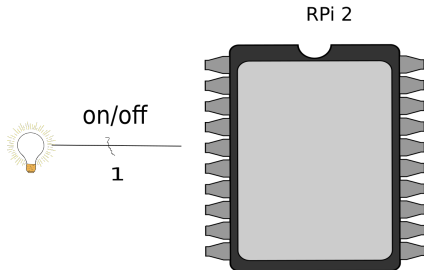
- From the main chip of the RPi2 we want to control an (external) device, here an LED.
- We use one of the **GPIO** pins to connect the device.
- Logically we want to send **1 bit** to this device to turn it **on/off**.

The low-level picture
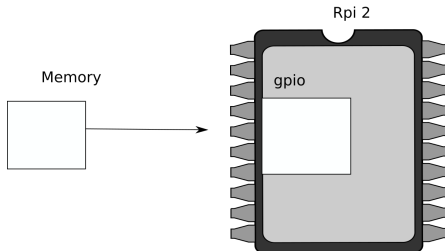
Programmatically we achieve that, by

- memory-mapping the address space of the GPIOs into user-space
- now, we can directly access the device via memory read/writes
- we need to pick-up the meaning of the peripheral registers from the BCM2835 peripherals sheet

BCM2835 GPIO Peripherals

Base adress: 0x3F000000

0	GPFSEL	Pins 0-9	(3-bits per pin)
5		Pins 50-53	
7 8	GPSET	Pins 0-31 Pins 32-53	(1-bit per pin)
10 11	GPCLR	Pins 0-31 Pins 32-53	(1-bit per pin)
13 14	GPLEV	Pins 0-31 Pins 32-53	(1-bit per pin)
...			

The meaning of the registers is (see p90ff of BCM2835 ARM peripherals):

- **GPFSEL**: function select registers (3 bits per pin); set it to 0 for input, 1 for output; 6 more alternate functions available
- **GPSET**: **set** the corresponding pin
- **GPCLR**: **clear** the corresponding pin
- **GPLEV**: return the **value** of the corresponding pin

GPIO Register Assignment

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	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W
0x 7E20 0018	-	Reserved	-	-
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W
0x 7E20 0024	-	Reserved	-	-
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W
0x 7E20 0030	-	Reserved	-	-

The GPIO has 48 32-bit registers (RPi2; 41 for RPi1).

[0]See BCM Peripherals Manual, Chapter 6, Table 6.1

GPIO Register Assignment

GPIO registers (Base address: 0x3F200000)

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL0	0:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL1	1:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL2	2:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL3	3:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL4	4:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL5	5:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
—	6:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSET0	7:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSET1	8:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
—	9:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFCLR0	10:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFCLR1	11:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
—	12:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13

---

[0] See BCM Peripherals, Chapter 6, Table 6.1

Locating the GPFSEL register for pin 47 (ACT)

Bit(s)	Field Name	Description	Type	Reset
31-30	---	Reserved	R	0
29-27	FSEL49	FSEL49 - Function Select 49 000 = GPIO Pin 49 is an input 001 = GPIO Pin 49 is an output 100 = GPIO Pin 49 takes alternate function 0 101 = GPIO Pin 49 takes alternate function 1 110 = GPIO Pin 49 takes alternate function 2 111 = GPIO Pin 49 takes alternate function 3 011 = GPIO Pin 49 takes alternate function 4 010 = GPIO Pin 49 takes alternate function 5	R/W	0
26-24	FSEL48	FSEL48 - Function Select 48	R/W	0
23-21	FSEL47	FSEL47 - Function Select 47	R/W	0
20-18	FSEL46	FSEL46 - Function Select 46	R/W	0
17-15	FSEL45	FSEL45 - Function Select 45	R/W	0
14-12	FSEL44	FSEL44 - Function Select 44	R/W	0
11-9	FSEL43	FSEL43 - Function Select 43	R/W	0
8-6	FSEL42	FSEL42 - Function Select 42	R/W	0
5-3	FSEL41	FSEL41 - Function Select 41	R/W	0
2-0	FSEL40	FSEL40 - Function Select 40	R/W	0

Table 6-6 – GPIO Alternate function select register 4

Accessing a GPIO Pin

- Now we want to control the on-chip LED, called ACT, that normally indicates activity.
- The pin number of this device on the RPi2 is: **47**
- We need to calculate registers and bits corresponding to this pin
- The **GPFSEL** register for pin 47 is **4** (per docu, this register covers pins 40-49 (Tab 6-6, p. 94)
- For each register 3 bits are used to select the function of that pin: bits 0–2 for register 40 etc
- Thus, bits 21–23 cover register 47 (7×3)
- The function that we need to select is OUTPUT, which is encoded as the value 1
- We need to write the value `0x01` into bits 21–23 of register 4

Accessing a GPIO Pin

- Now we want to control the on-chip LED, called ACT, that normally indicates activity.
- The pin number of this device on the RPi2 is: **47**
- We need to calculate registers and bits corresponding to this pin
- The **GPFSEL** register for pin 47 is **4** (per docu, this register covers pins 40-49 (Tab 6-6, p. 94)
- For each register 3 bits are used to select the function of that pin: bits 0–2 for register 40 etc
- Thus, bits 21–23 cover register 47 (7×3)
- The function that we need to select is OUTPUT, which is encoded as the value 1
- We need to write the value $0x01$ into bits 21–23 of register 4

Accessing a GPIO Pin

- Now we want to control the on-chip LED, called ACT, that normally indicates activity.
- The pin number of this device on the RPi2 is: **47**
- We need to calculate registers and bits corresponding to this pin
- The **GPFSEL** register for pin 47 is **4** (per docu, this register covers pins 40-49 (Tab 6-6, p. 94)
- For each register 3 bits are used to select the function of that pin: bits 0–2 for register 40 etc
- Thus, bits 21–23 cover register 47 (7×3)
- The function that we need to select is OUTPUT, which is encoded as the value 1
- We need to write the value $0x01$ into bits 21–23 of register 4

Accessing a GPIO Pin

- Now we want to control the on-chip LED, called ACT, that normally indicates activity.
- The pin number of this device on the RPi2 is: **47**
- We need to calculate registers and bits corresponding to this pin
- The **GPFSEL** register for pin 47 is **4** (per docu, this register covers pins 40-49 (Tab 6-6, p. 94)
- For each register 3 bits are used to select the function of that pin: bits 0–2 for register 40 etc
- Thus, bits 21–23 cover register 47 (7×3)
- The function that we need to select is OUTPUT, which is encoded as the value 1
- We need to write the value $0x01$ into bits 21–23 of register 4

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4

- What's the address of register 4 relative to the base address in `gpio`? **Answer:** `gpio+4`

- How do we read the current value from this register?

- How do we blank out bits 21–23 from this register?

- How do we get the value `0x01` into bits 21–23 of a 32-bit word?

- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
  Answer: `*(gpio+4)`
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value $0x01$ into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value $0x01$ into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
  Answer: `*(gpio + 4) & ~(7 << 21)`
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?

  C code: 7

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

```
28        24        20        16        12        8         4         0
```

- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
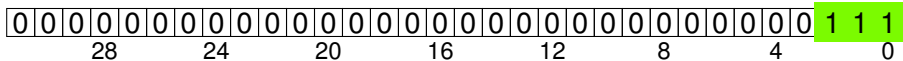- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
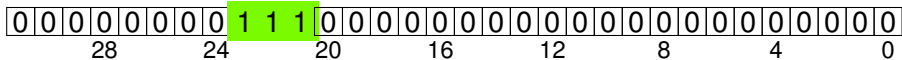- How do we blank out bits 21–23 from this register?

  C code: `7 << 21`

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

```
28      24      20      16      12      8       4       0
```

- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?

C code: `˜(7 << 21)`

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

```
  28        24        20        16        12        8         4         0
```
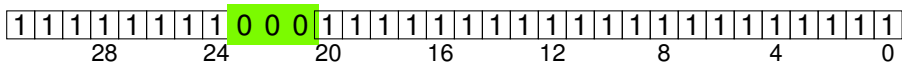
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
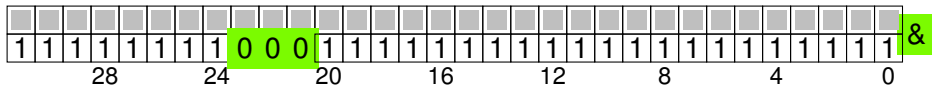
C code: `(*(gpio + 4) & ~(7 << 21))`

- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
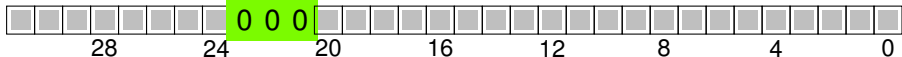
C code: `(*(gpio + 4) & ~(7 << 21))`


```
                    0 0 0
   28        24        20        16        12        8         4         0
```

- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
  Answer: `(1 << 21)`
- How do we put **only these bits** into the contents of register 4?

HERIOT
WATT
UNIVERSITY

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?

`(*(gpio + 4) & ~(7 << 21)) | (1 << 21)`

- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

```
*(gpio + 4) = (*(gpio + 4) & ~(7 << 21)) | (1 << 21)
```

GPIO programming

- The previous slides discussed how to control an LED with a GPIO pin.
- Similar code is used to use a button as an input device, and to read a bit from the right GPIO pin
- For the exam you need to understand the main steps that are needed
- You must be able to perform the above steps to explain, e.g. how to set the mode of a pin
- The LCD device is controlled in a similar way, but always sending 8 bits as the byte to be displayed.
- You should expect specific code questions about GPIO programming, either in C or Assembler

Summary

- Check the detailed tutorial slides about controlling external devices
- Look-up the sample sources (both C and Asm) for the tutorials
- You need to have a **solid understanding of this code** and be able to answer questions about it!
- Focus on the main concepts that we covered in the lectures:
  - Computer architecture, in particular pipelining
  - Memory hierarchy, in particular caching
- You need to be able to explain how these concepts impact performance of some sample programs.
- Be prepared for **small-scale coding questions**