Introduction

COURSE AIMS AND GOALS

Writing real OS code

- There’s a limit to how much you can learn about an OS by reading books.
- Solve common OS problems for yourself
  - Dealing with **concurrency**
  - Dealing with **asynchrony**
  - Tailoring code to specific **hardware**
  - Architecting complete **system** of processes

It’s not Unix

The course avoids using a Unix-like OS:

- Gives a broader outlook on what an OS can be
- Encourages a critical approach to OS designs
- Better understanding of the structure of Unix / Linux / Windows / Android / iOS / …
- Clearer picture of ideas (finally) appearing in older OS designs
  - Or not yet…

What is OS research?

We'll be working with a research OS

- You'll get a sense of how OS research is done
  - Read research papers
  - Relate them to implementations
  - How to evaluate such ideas
- See what constitutes an OS research topic
  - What problems do these ideas solve?
  - How are they motivated, where do they come from?

Key ideas in OS research

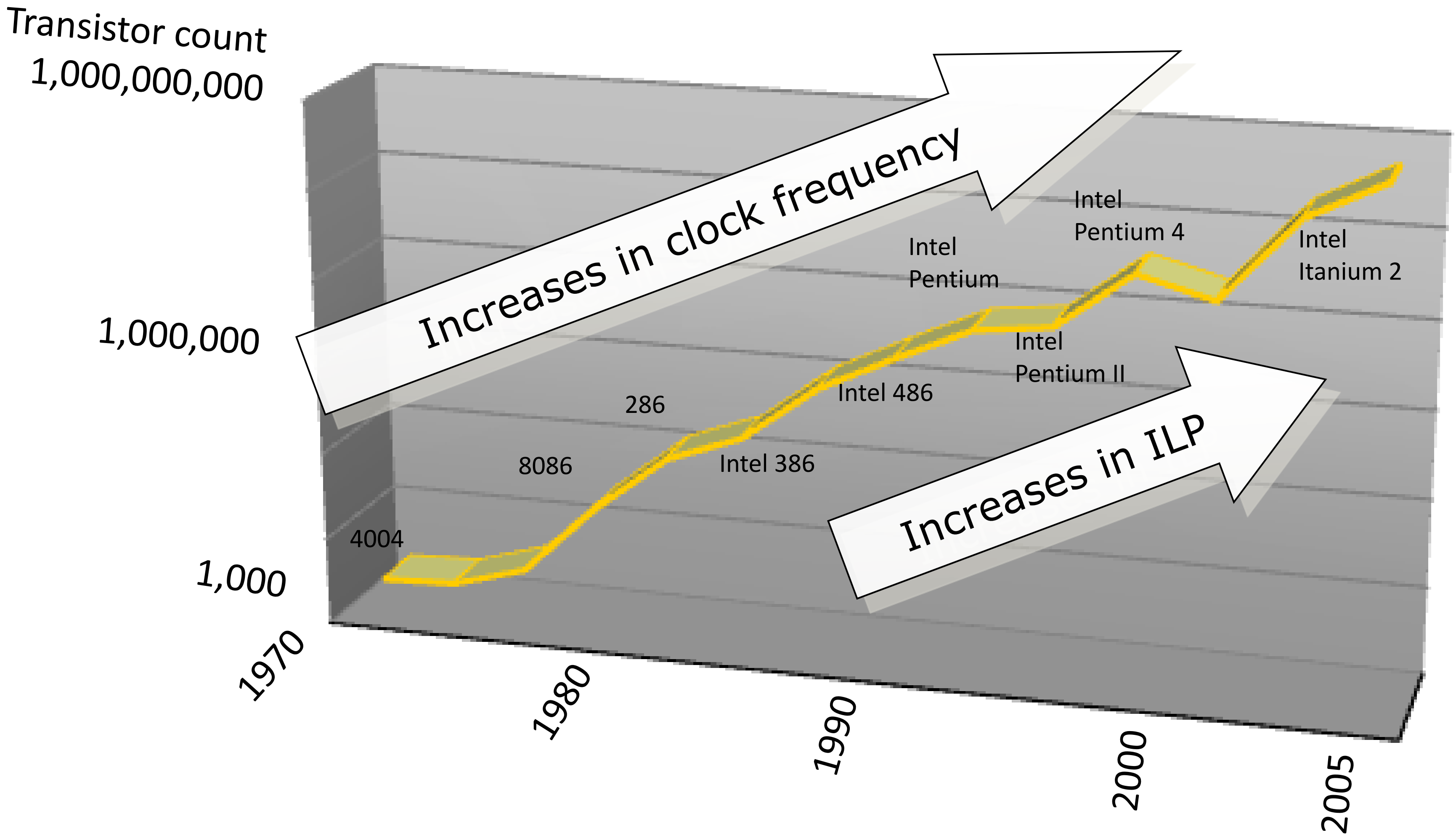
- The OS we use includes a selection of key ideas in OS research
  - You get hands-on experience with the ideas
  - We'll go through the papers on the topic
  - We'll put them into historical context
- Many are not well-known and/or unimplemented in the wider community

MOTIVATION

Why teach non-Unix OS design?

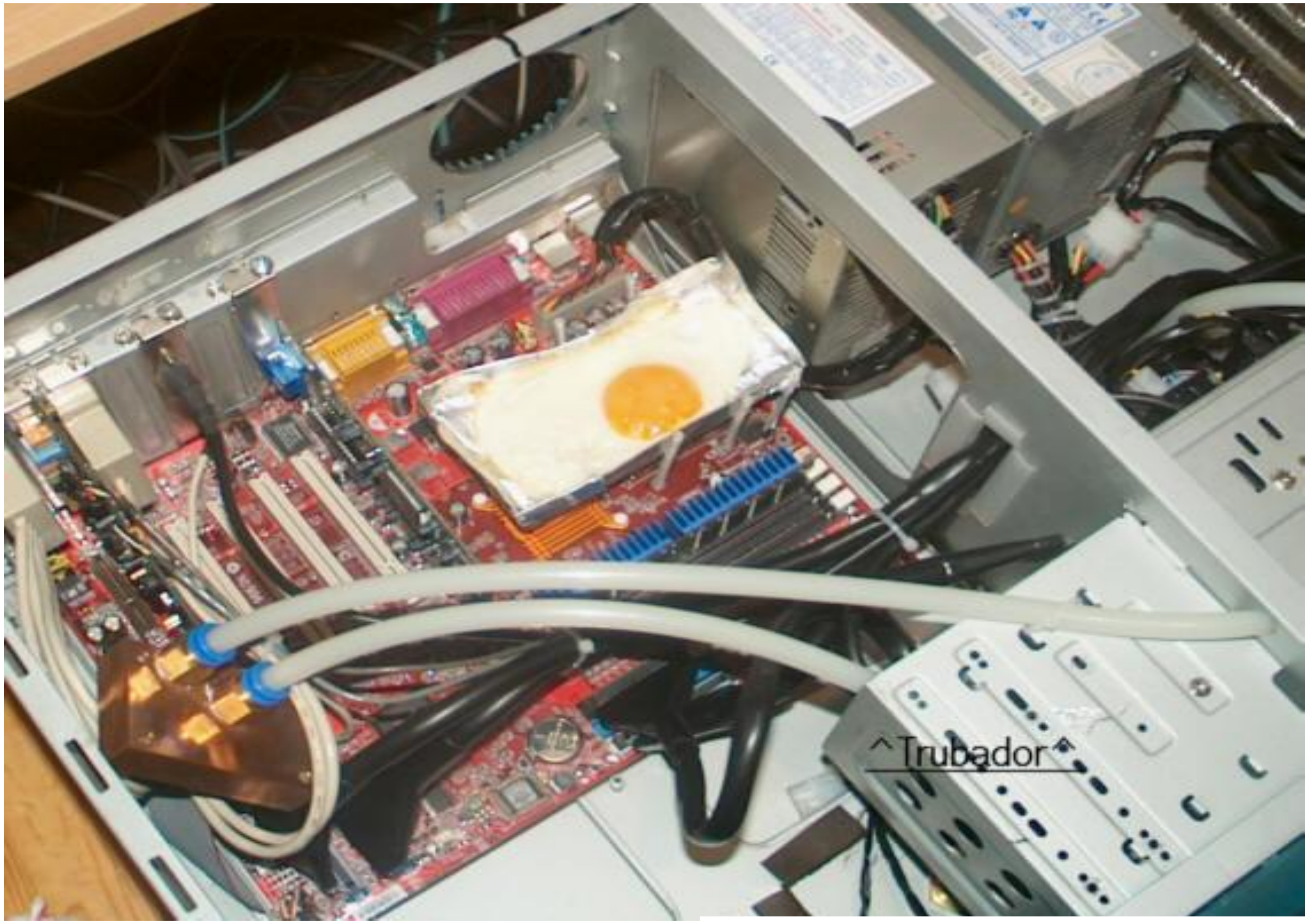
- This is an interesting time for OS research:
  - Multicore
  - Heterogeneous processing
  - System complexity
  - System diversity
- The basic assumptions of Unix/Windows/etc. are being re-evaluated.

Moore's law: the free lunch



Source: table from http://download.intel.com/pressroom/kits/events/moores_law_40th/MLTimeline.pdf

The power wall



http://www.phys.ncku.edu.tw/~htsu/humor/fry_egg.html

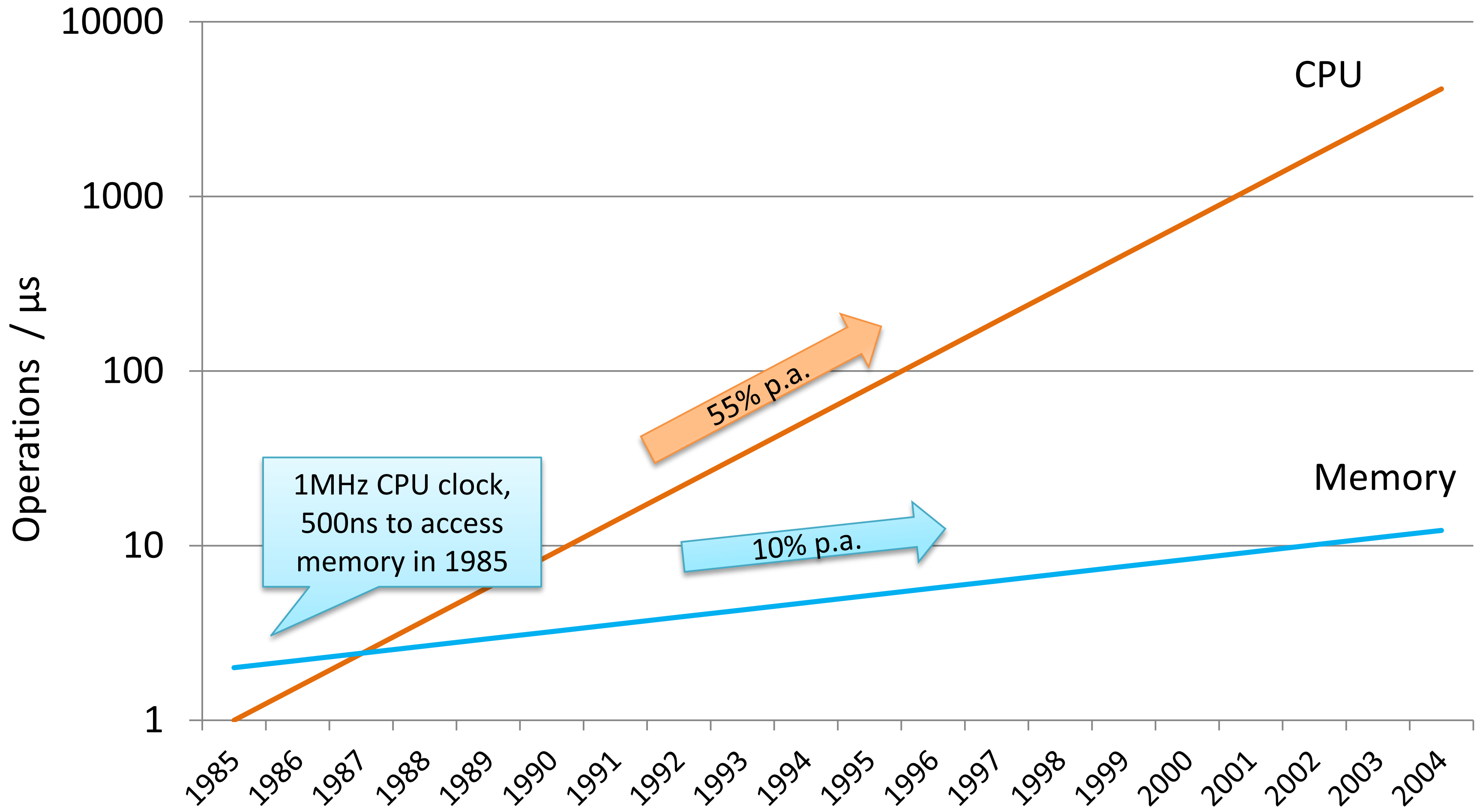
The power wall

- Power dissipation =

  Capacitive load $\times$ Voltage^2 $\times$ Frequency

  - Increase in clock frequency
    $\Rightarrow$ mean more power dissipated
    $\Rightarrow$ more cooling required
  - Decrease in voltage reduces dynamic power but increase the static power leakage
- We’ve reached the practical power limit for cooling commodity microprocessors
  - Can’t increase clock frequency without expensive cooling

The memory wall



The ILP wall

- ILP = “Instruction level parallelism”
- Implicit parallelism between instructions in 1 thread
- Processor can re-order and pipeline instructions, split them into microinstructions, do aggressive branch prediction etc.
  - Requires hardware safeguards to prevent potential errors from out-of-order execution
- Increases execution unit complexity and associated power consumption
  - Diminishing returns
- Serial performance acceleration using ILP has stalled

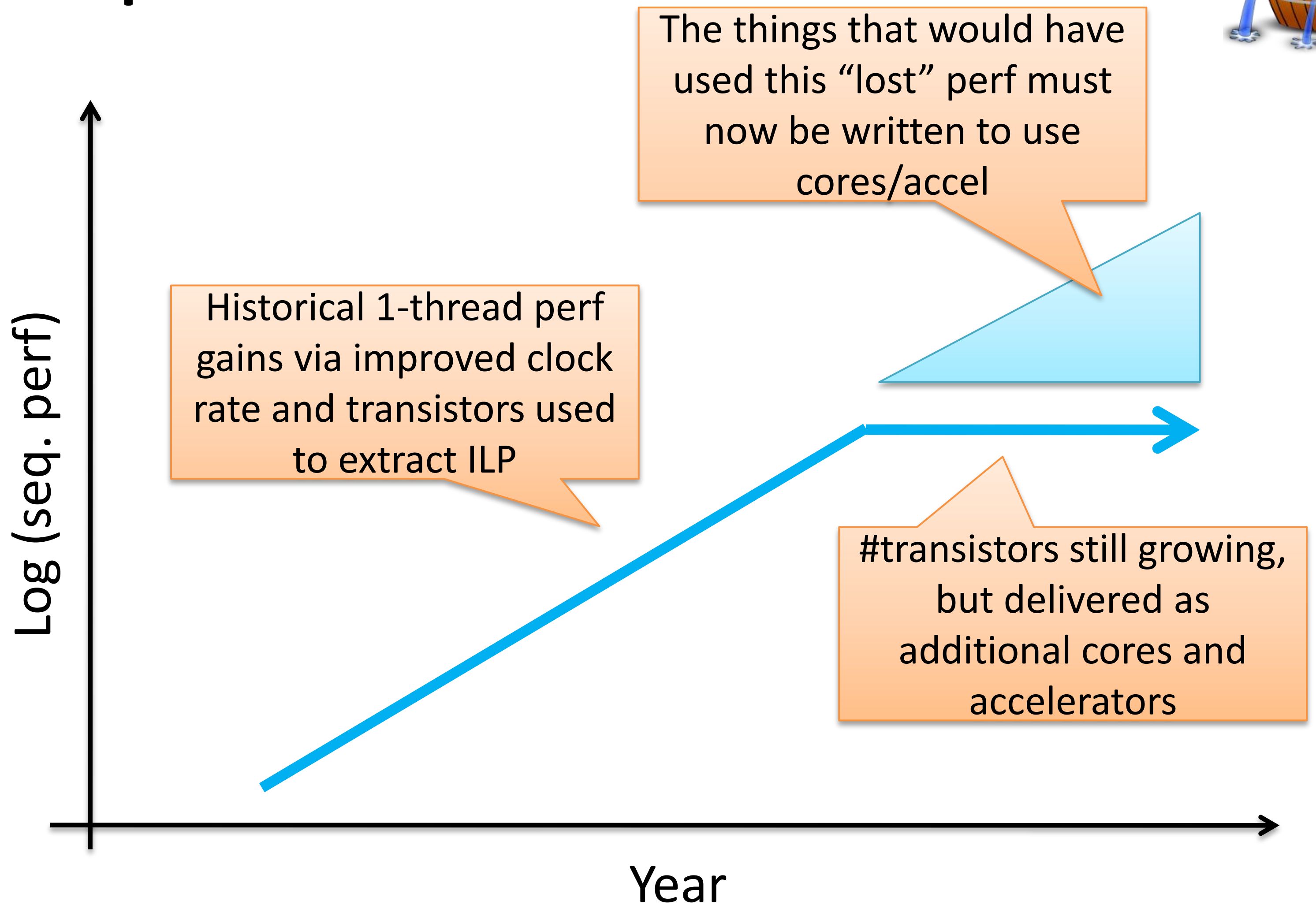
End of the road for serial hardware

- Power wall + ILP wall + memory wall
  = brick wall
  - Power wall ⇒ can’t clock processors any faster
  - Memory wall ⇒ for many workloads performance dominated by memory access times
  - ILP wall ⇒ can’t keep functional units busy while waiting for memory accesses
- There is also a complexity wall, but chip designers don’t like to talk about it…

Multicore processors

- Multiple processor cores per chip
  - This is the future (and present) of computing
- Most multicore chips *so far* are shared memory multiprocessors (SMP)
  - Single physical address space shared by all processors
  - Communication between processors happens through shared variables in memory
  - Hardware typically provides cache coherence

Implications for software


The things that would have used this "lost" perf must now be written to use cores/accel
Historical 1-thread perf gains via improved clock rate and transistors used to extract ILP
#transistors still growing, but delivered as additional cores and accelerators
Log (seq. perf)
Year

And the others?

Trends towards:

- Heterogeneous processing
  - Different types of core on a chip
- System complexity
  - Devices, interconnects, memory system
- System diversity
  - No two systems are alike
  - Very different performance tradeoffs

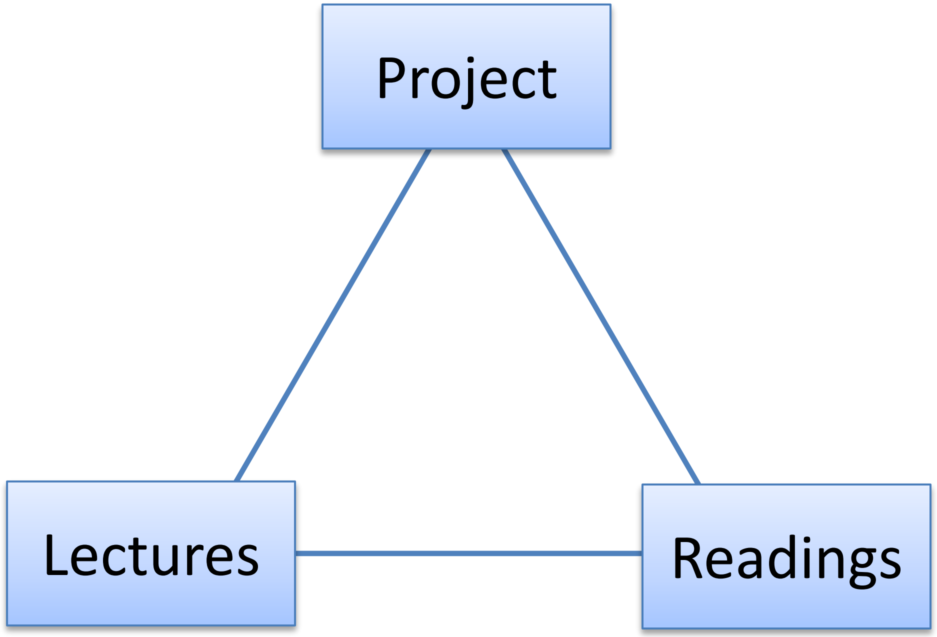
We'll see this all very shortly…

STRUCTURE AND LOGISTICS

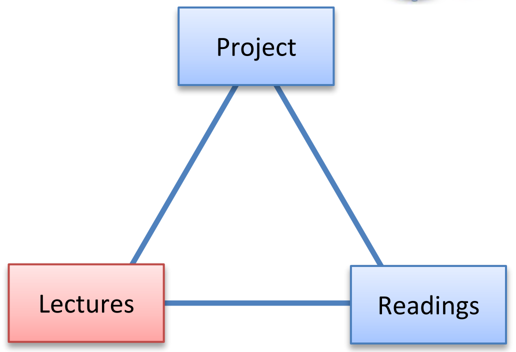
Assumptions

- C Programming
  - Practical labs are mostly in C.
  - You will be writing an OS.
- Computer Architecture
  - Need to know about hardware, registers, DMA,
  - You will be writing a device driver, pager, etc.
- Command line development
  - gcc, emacs/vi, make, etc.
  - Debugging facilities are primitive!

Structure of the course

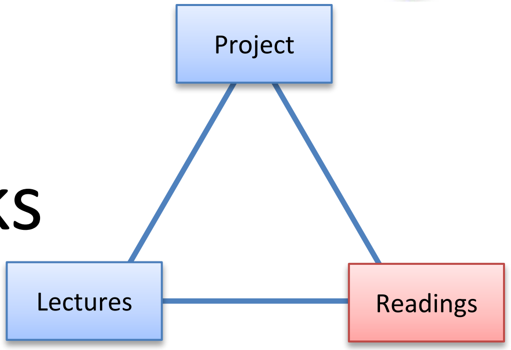


Lectures



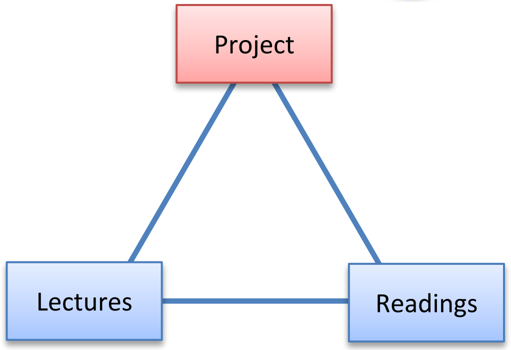
- Cover topics related to current project milestone.
  - Key ideas
  - History / influences
- Explanation / elaboration of readings
- Also Q&A if required

Readings

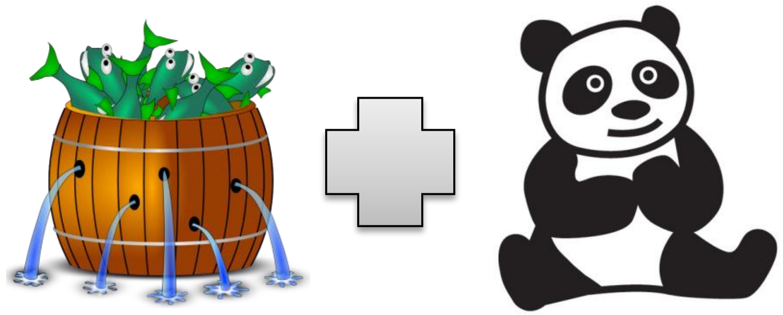


- Most of our material is not (or insufficiently) covered in textbooks
- Readings will be *research papers*
  - Seminal papers in the field
  - Background to the concepts you will deal with in the weeks' project milestone
  - Context for the material covered in the lecture

Project



- Build (most of) an OS
  - based on Barrelfish research OS
  - PandaBoard hardware



Schedule

Intro

- Familiarization with hardware/software
- Boot an intial image, console output, flash LEDS
- Individual, 1 week

Core

- Building the core of an OS over the Barrelfish CPU driver
- Memory, processes, communication, multicore
- Teams of 3-4, 9 weeks

Full OS

- Individual projects based on the core
- Drivers, file system, etc.
- 4 weeks, including writing report

Project milestones

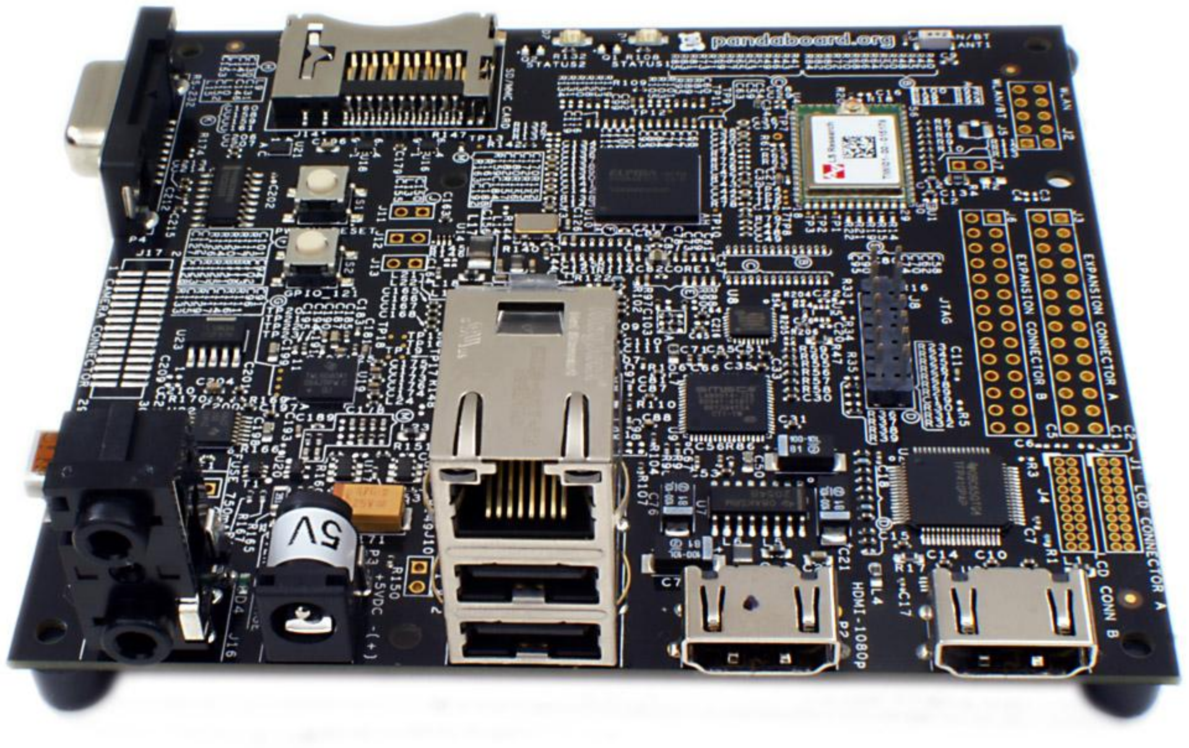
- There are marks for each milestone
- Late milestones incur a penalty
- Beware milestones which build on previous ones!
  - Don’t slip behind

What you need to do.

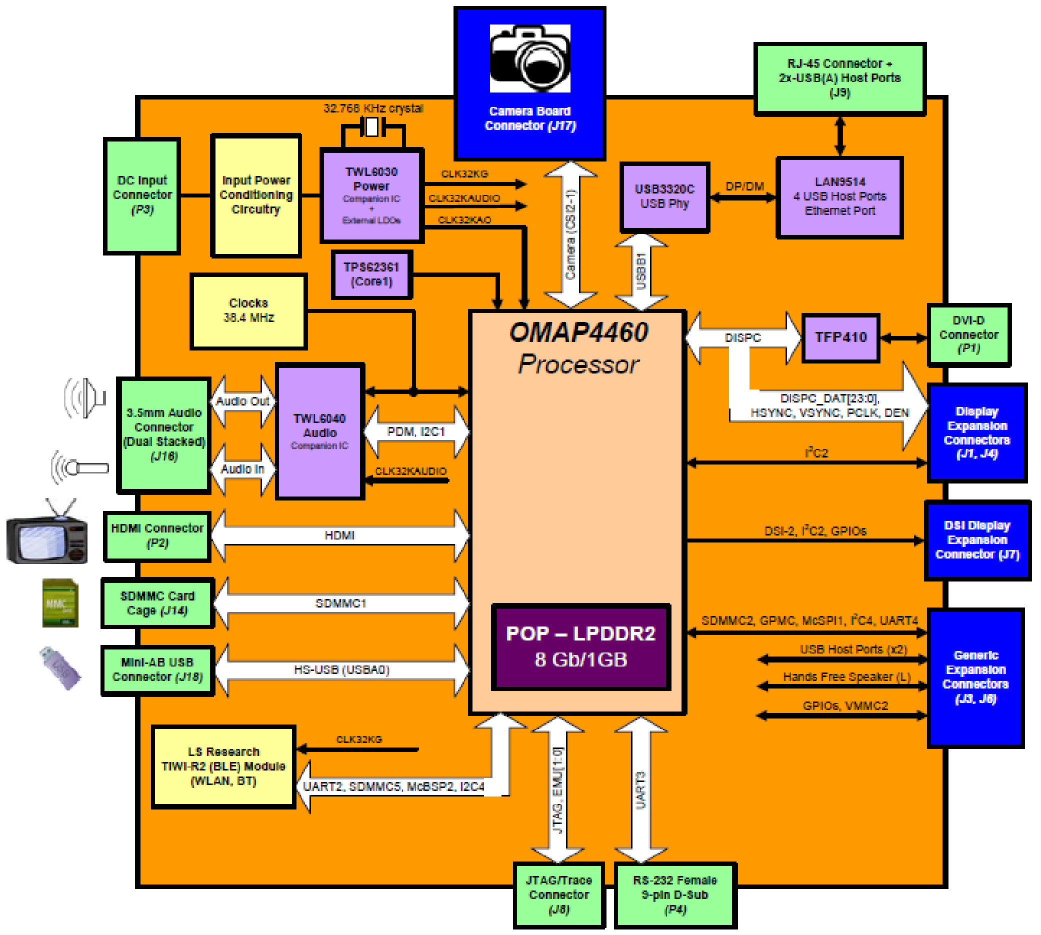
- Attend the introductory lab session!
  - Sign out your Pandaboard kits
  - Bring your ID
  - Verify your board works before leaving
  - Take care of them – they are fragile!
- Form teams of 3-4 for subsequent milestones
  - Inform the teaching assistants
- Deliver your first (individual) milestone
  - Come to the session and demo it

HARDWARE

Hardware: PandaBoard ES

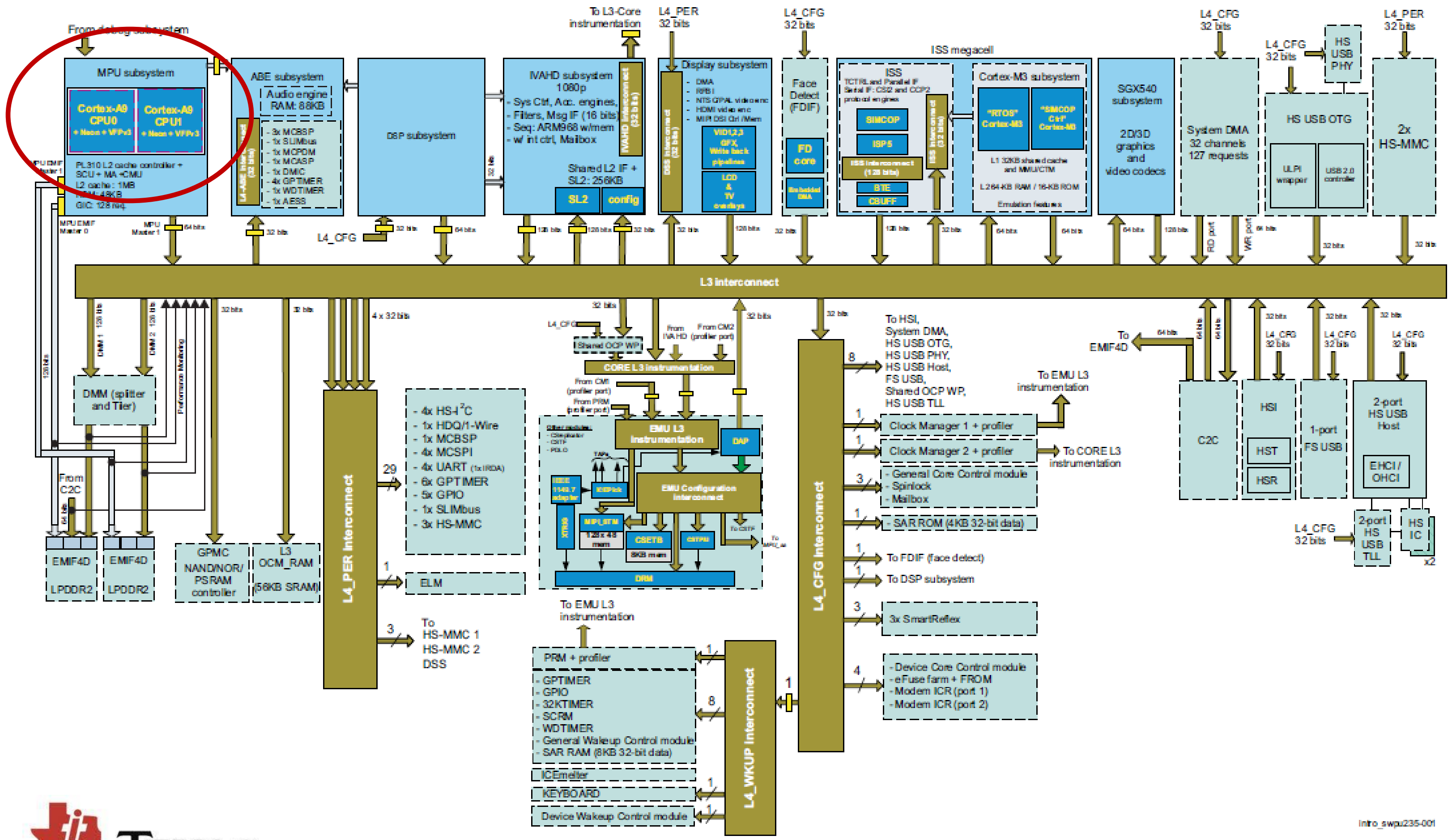


PandaBoard block diagram



Camera Board Connector (J17)
RJ-45 Connector + 2x-USB(A) Host Ports (J9)
32.768 KHz crystal
DC Input Connector (P3)
Input Power Conditioning Circuitry
TWL6030 Power Companion IC + External LDOs
CLK32KG
CLK32KAUDIO
CLK32KAO
Camera (CSI2-1)
USB3320C USB Phy
DP/DM
LAN9514 4 USB Host Ports Ethernet Port
TPS62361 (Core1)
USBB1
Clocks 38.4 MHz
OMAP4460 Processor
DISPC
TFP410
DVI-D Connector (P1)
3.5mm Audio Connector (Dual Stacked) (J16)
Audio Out
Audio In
TWL6040 Audio Companion IC
PDM, I2C1
CLK32KAUDIO
DISPC_DAT[23:0], HSYNC, VSYNC, PCLK, DEN
Display Expansion Connectors (J1, J4)
I²C2
HDMI Connector (P2)
HDMI
DSI-2, I²C2, GPIOs
DSI Display Expansion Connector (J7)
SDMMC Card Cage (J14)
SDMMC1
POP – LPDDR2 8 Gb/1GB
SDMMC2, GPMC, McSPI1, I²C4, UART4
USB Host Ports (x2)
Hands Free Speaker (L)
GPIOs, VMMC2
Generic Expansion Connectors (J3, J6)
Mini-AB USB Connector (J18)
HS-USB (USBA0)
LS Research TiWi-R2 (BLE) Module (WLAN, BT)
CLK32KG
UART2, SDMMC5, McBSP2, I2C4
JTAG, EMU[1:0]
UART3
JTAG/Trace Connector (J8)
RS-232 Female 9-pin D-Sub (P4)

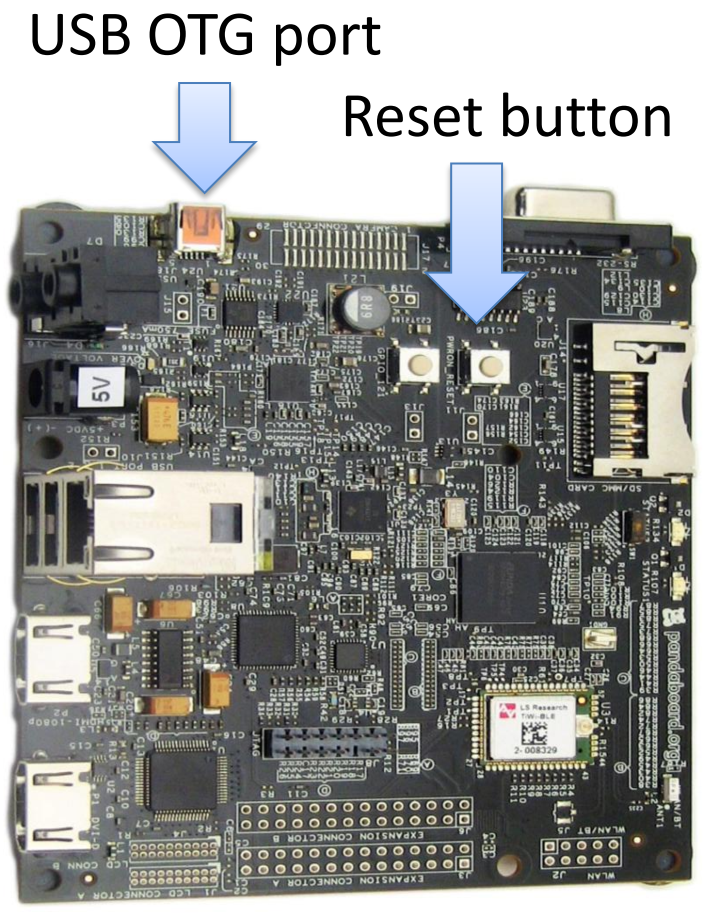
The TI OMAP 4460 SoC



USB booting

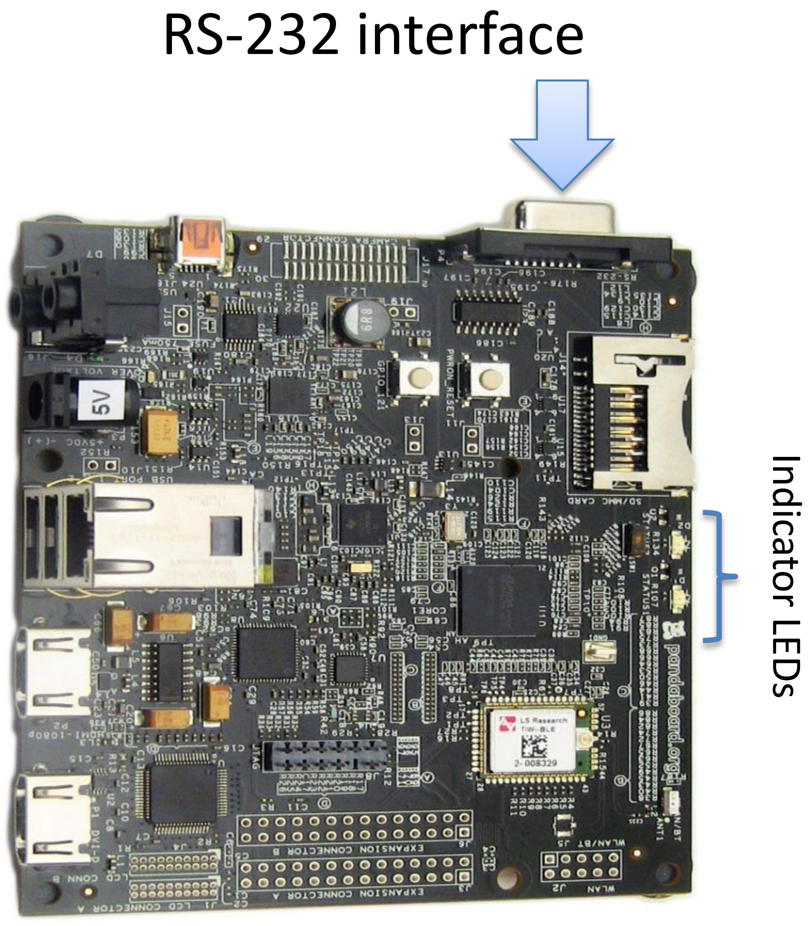
"USB On-The-Go" port:

- Provides power from the host PC
- Host copies code over USB into PandaBoard memory
- Jumps to start of boot image
- No need for network, SD card, etc.



Serial console

- One byte at a time, bidirectional serial line
- Still the best low-level console interface for real OS hacking
- First milestone includes:
  - Very simple console output
  - Flash indicator LED



ARM CPU cores

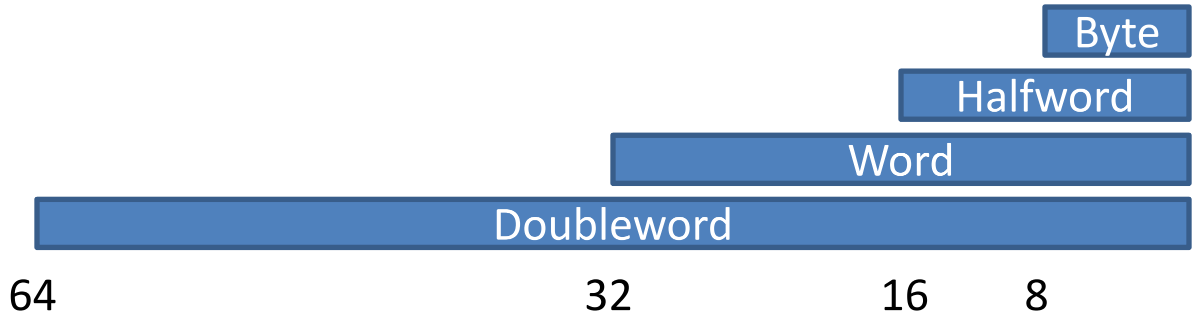
- 2 x ARM Cortex A9 cores
- SMP configuration
- We’ll start with just one core
- We’ll bring up the other one later

There are, in fact, two more ARM Cortex M3 cores.
We’ll ignore these.

(unless you want to play with them in your spare time ☺)

ARM architecture overview

- Data Sizes:
  - 16 bits: Halfword
  - 32 bits: Word
  - 64 bits: Doubleword
- Note: different from Intel/AMD names!



ARM Processor modes

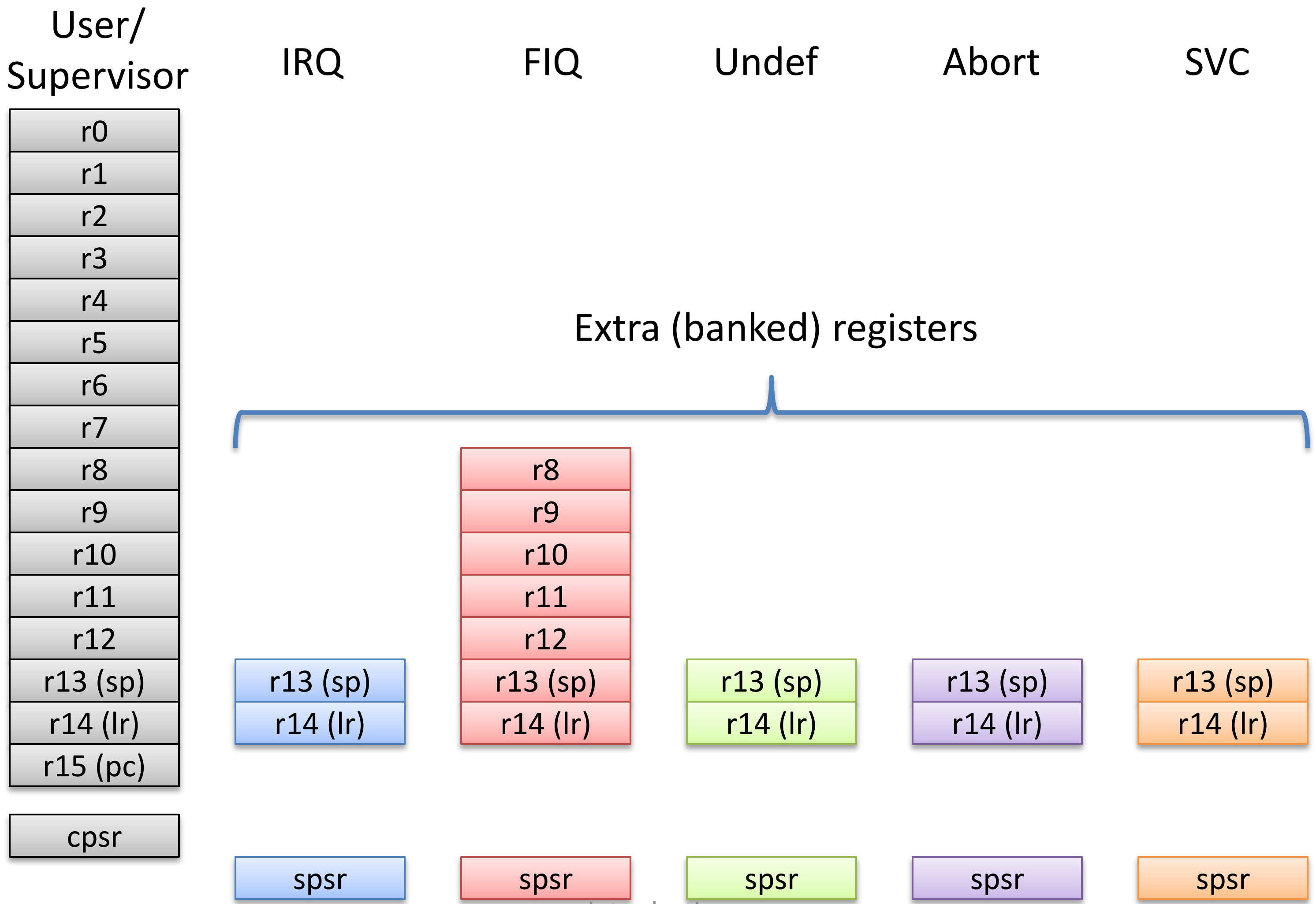
Mode	Description
Supervisor (SVC)	Entered on reset or SWI instruction
FIQ	Fast interrupt
IRQ	Normal interrupt
Abort	Memory access violations
Undef	Undefined instructions
System	Privileged mode, same registers as user mode
User	User mode (regular processes)

Exception modes

Privileged modes

Each mode has its own stack!

ARM register sets
(all 32 bits)



ARM condition codes

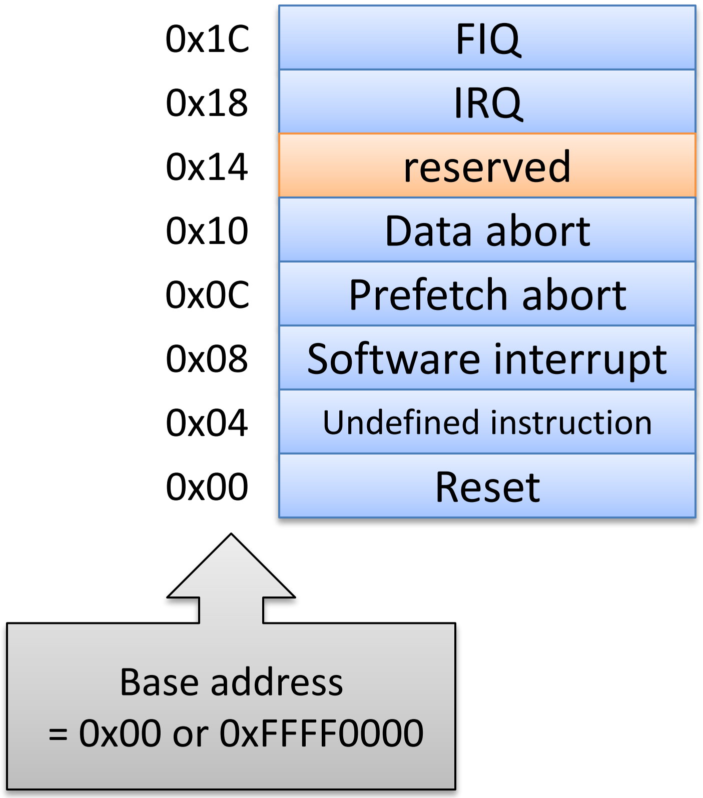
(some of them)

Flag	Description
N	Negative result
Z	Zero result
C	Carry from ALU
V	Overflow from ALU
I	Disable IRQ
F	Disable FIQ
mode	Current mode (4 bits)

Found in the CPSR/SPSR registers

Exception handling

1. CPSR → SPSR$_{\text{mode}}$
2. Set CPSR bits
   - New mode
   - Disable interrupts (if appropriate)
3. Return addr → LR$_{\text{mode}}$
4. Vector addr → PC



0x1C	FIQ
0x18	IRQ
0x14	reserved
0x10	Data abort
0x0C	Prefetch abort
0x08	Software interrupt
0x04	Undefined instruction
0x00	Reset

Base address = 0x00 or 0xFFFF0000

ARM instruction set

- Classic RISC load/store architecture
  - ALU operations are register-register
- All instructions 32 bits long
- Most instructions are conditional
- On-die barrel shifter
- Branches are PC-relative

```
LDR      r0,[r1]
STRNEB   r2,[r3,r4]
B        <label>
SUB      r0,r1,#5
ADD      r2,r3,r3,LSL#2
ADDEQ    r5,r5,r6
```

Calling Conventions

- The **ABI**: **A**pplication **B**inary **I**nterface
- How processor registers are used by software
  - How the compiler uses registers
  - How arguments and results are passed
  - Procedure linkage
- **You** need to set the initial values for a process.

ARM register usage

The ABI assigns a use to each CPU register.

r0	a1
r1	a2
r2	a3
r3	a4
r4	v1
r5	v2
r6	v3
r7	v4
r8	v5
r9	v6
r10	v7
r11	v8
r12	IP
r13	SP
r14	LR
r15	PC

ARM register usage

r0	a1
r1	a2
r2	a3
r3	a4
r4	v1
r5	v2
r6	v3
r7	v4
r8	v5
r9	v6
r10	v7
r11	v8
r12	IP
r13	SP
r14	LR
r15	PC

- **A**rguments (return in r0,r1)
  - Scratch registers within a function
  - Caller saved

ARM register usage

r0	a1
r1	a2
r2	a3
r3	a4
r4	v1
r5	v2
r6	v3
r7	v4
r8	v5
r9	v6
r10	v7
r11	v8
r12	IP
r13	SP
r14	LR
r15	PC

- **V**ariable **R**egisters
  - – Callee saved

ARM register usage

r0	a1
r1	a2
r2	a3
r3	a4
r4	v1
r5	v2
r6	v3
r7	v4
r8	v5
r9	v6
r10	v7
r11	v8
r12	IP
r13	SP
r14	LR
r15	PC

- **I**ntra-**P**rocedure-Call scratch register
- Used by linker during procedure call
  - May be trashed in the process
- Can be used as a scratch register between procedure calls

ARM register usage

r0	a1
r1	a2
r2	a3
r3	a4
r4	v1
r5	v2
r6	v3
r7	v4
r8	v5
r9	v6
r10	v7
r11	v8
r12	IP
r13	SP
r14	LR
r15	PC

- **S**tack **P**ointer.
  - Grows *down*, points to last full slot

ARM register usage

r0	a1
r1	a2
r2	a3
r3	a4
r4	v1
r5	v2
r6	v3
r7	v4
r8	v5
r9	v6
r10	v7
r11	v8
r12	IP
r13	SP
r14	LR
r15	PC

- **L**ink **R**egister
  - Return address for procedure call

ARM register usage

r0	a1
r1	a2
r2	a3
r3	a4
r4	v1
r5	v2
r6	v3
r7	v4
r8	v5
r9	v6
r10	v7
r11	v8
r12	IP
r13	SP
r14	LR
r15	PC

- **P**rogram **C**ounter
  - A normal register!
  - `ADD PC,10` ↔ `B +10`

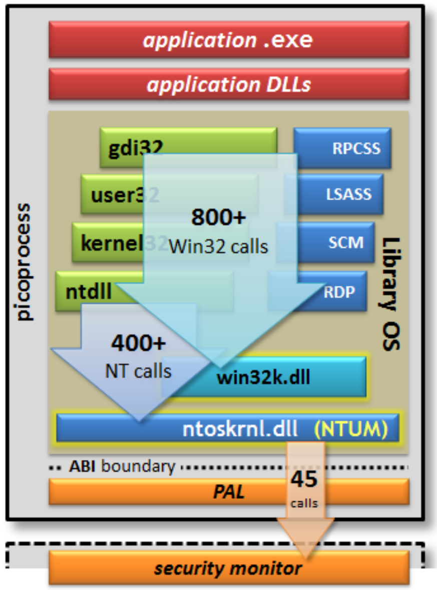
OS: BARRELFISH

What is Barrelfish?

- Open source research operating system
- ETH Zurich
  - Contributions/support: ARM Ltd, Cisco, Hewlett-Packard Enterprise, Huawei, Intel, Microsoft Research, Texas Instruments, University of Washington, VMware, and others.
- Currently supports:
  - 64-bit x86 AMD/Intel
  - Intel MIC/Xeon Phi
  - **ARMv7a**, ARMv8a

What things run on it?

- Many microbenchmarks
- Parallel benchmarks: Parsec, SPLASH-2, NAS
- Webserver
- Databases: SQLite, PostgreSQL
- OpenSSH, other utils
- Virtual machine monitor
  - Linux kernel binary
- Microsoft Office 2010!
  - via Drawbridge



Why use Barrelfish?

- Very different to Linux, BSD, MacOS, Windows
  - The shock of the new
- Smaller OS
  - Easier to understand, easier to hack
- Multikernel approach (see later)
  - Simplifies multicore issues
- Includes lots of other people's research ideas
  - Good illustration of any concepts

Some of the (non-original) ideas in Barrelfish

- Capabilities for resource management (KeyKOS, seL4)
- Minimize shared state (Tornado, K42)
- Upcall processor dispatch (Psyche, Sched. Activations)
- Push policy into user space (Exokernel, Nemesis)
- User-space RPC decoupled from IPIs (URPC)
- Lots of information (Infokernel)
- Single-threaded non-preemptible kernel per core (K42)
- Run drivers in their own domains (µkernels, Xen)
- Fast interprocess communication (LRPC, L4)
- Specify device registers in a little language (Devil)

We'll see many of these later in the course

TOOLCHAIN

Toolchain

- C compiler: **`gcc`**
- Haskell compiler: **`ghc`** ⟵ Huh?
- **`make`**
- Terminal emulator: **`picocom`**
- Boot utility: **`usbboot`**

- Recommended host platform: Ubuntu LTS

Building Barrelfish

```
$ mkdir -p ~/build_milestone_0
$ cd ~/build_milestone_0
$ ~/barrelfish/hake/hake.sh \
    -s ~/barrelfish -a armv7
```

- Builds a Makefile for Barrelfish in the **`build_milestone_0`** directory, based on the source tree in **`~/barrelfish`**, for just the ARMv7 architecture.
- You can look at **`./Makefile`** to see the result (but it can be very large…)

Building Barrelfish

```
$ mkdir –p ~/build_milestone_0
$ cd ~/build_milestone_0
$ ~/barrelfish/hake/hake.sh \
     -s ~/barrelfish -a armv7
$ make armv7_aos_m0_image
$ ls -l armv7_aos_m0_image
```

- Make builds its own directories, dependencies, and other tools as well.
- Result ends up in the top of the build directory.

What’s this Hake thing?
(for the curious)

Scattered through the source tree are files called “**`Hakefile`**”

- Hake concatenates all of these and interprets them as a single, large Haskell expression
- This expression evaluates to the contents of **`./Makefile`**

Why?

- Allows us to build multiple architectures from one source tree at the same time
- Power of a full programming language in expressing configurations
- Still get the efficiency of **`make`** for building

AND FINALLY, SOME ADVICE

Important advice

- You will be judged on the ***design*** of your code
  – Does it work?
  – Handle corner cases, errors, invalid inputs, etc?
  – An operating system runs for a long time.
    Do you leak memory? Etc.
  – Have you thought about issues not explicit in the milestones, but important to a real OS?
- Not all these criteria can be well-specified in advance!
  – Use common-sense in system design
  – You will be graded on issues you can think of and deal with

Important advice

You will be judged on the ***quality*** of your report

- Doxygen and other tools document *lines of code*, **not** the *design of a system*!
- Don’t submit generated docs in place of a report.
- Describe the *choices* you made (and didn’t make)
- Talk about the tradeoffs
- Mention the *difficulties* and *challenges*, and how you overcame them.
- Show you understand how to build a system.

Important advice

- Try to show the depth of your ***understanding***
  - Design principles
  - Why do techniques work?
  - When do they work (or not)?
  - What are the tradeoffs in a problem?
  - What factors affect the tradeoffs?
  - How might this change? How has it changed?

Final advice

- This course is a lot of fun, but a lot of work
  - and we are aware of this!
- It is important not to fall behind
  - If your team is struggling, ask for help.
- If you're good, it's tempting to be clever and cool
  - Resist this temptation!
  - Get the required work done before freestyling.
- We are here to help you!

Good luck and have fun!