[01] INTRODUCTION

OUTLINE

- Course Summary
- Recap
  - Encodings: Text, Numbers, Data Structures, Instructions
  - A Model Computer and the Fetch-Execute Cycle
  - Some IO Devices, Buses, Interrupts, DMA
- Key Concepts
  - Layering & Multiplexing
  - Synchronous & Asynchronous
  - Latency, Bandwidth, Jitter
  - Caching & Buffering
  - Bottlenecks, 80/20 Rule, Tuning
- Operating Systems
  - What is and is not an Operating System?
  - Evolution of Operating Systems

COURSE SUMMARY

	I	**Protection**
02	Protection	What is the OS protecting?
03	Processes	On what does the OS operate?
04	Scheduling	What does the OS run next?
	II	**Memory Management**
05	Virtual Addressing	How does the OS protect processes from each other?
06	Paging	How to manage virtual addresses from a machine perspective?
07	Segmentation	How to manage virtual addresses from a programmer perspective?

COURSE SUMMARY

	III	**Input/Output**
08	IO Subsystem	How does the OS interact with the outside world?
09	Storage	How does the OS manage persistence for processes?
10	Communications	How does the OS provide communication between processes?
	IV	**Case Studies**
11	Unix	Putting it together I
12	Windows NT	Putting it together II

RECOMMENDED READING

- Tannenbaum A S. (1990) *Structured Computer Organization (3rd Ed.)* Prentice-Hall 1990
- Patterson D and Hennessy J. (1998) *Computer Organization & Design (2rd Ed.)* Morgan Kaufmann
- Bacon J (1997) [and Harris T. (2003)] *Concurrent Systems* [or *Operating Systems*]. Addison Wesley
- Silberschatz A, Peterson J and Galvin P. (1998) *Operating Systems Concepts (5th Ed.)* Addison Wesley
- Leffler, S. (1989). *The Design and Implementation of the 4.3BSD Unix Operating System* Addison-Wesley
- McKusick, M.K., Neville-Neil, G.N. & Watson, R.N.M. (2014) *The Design and Implementation of the FreeBSD Operating System (2nd Ed.)* Pearson Education

RECAP

- Course Summary
- **Recap**
  - **Encodings: Text, Numbers, Data Structures, Instructions**
  - **A Model Computer and the Fetch-Execute Cycle**
  - **Some IO Devices, Buses, Interrupts, DMA**
- Key Concepts
- Operating Systems

TEXT

Two main standards:

- ASCII: 7-bit code holding (American) letters, numbers, punctuation and a few other characters. Regional 8-bit variations. Competitors included EBCDIC (IBM; very odd, not all characters contiguous)
- Unicode: 16-bit code supporting practically all international alphabets and symbols

ASCII used to be widespread default. Unicode now becoming popular (esp. UTF-8):

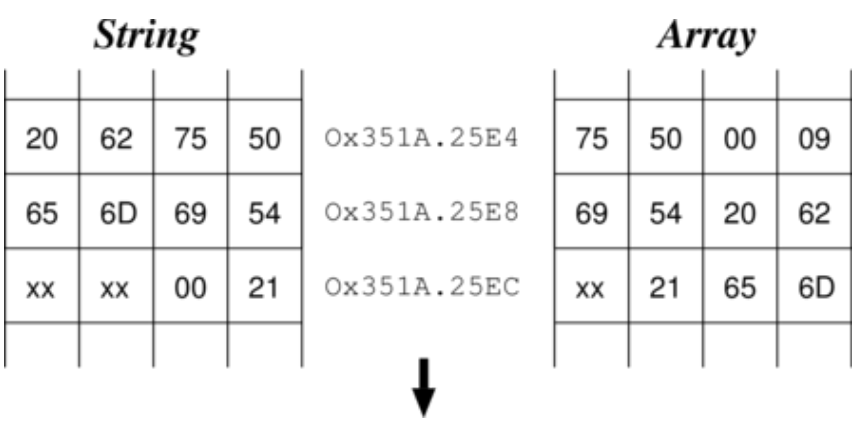
- Unicode 2.0 has 38,885 characters assigned out of potential 65,536
- Commonly use the 8-bit transformation format called UTF-8; superset of ASCII which includes "enough" to get by for Roman languages and regional currency symbols, etc.

UTF-8

Low 128 bits map direct to ASCII which is useful for backward compatibility

Variable length: all other characters are encoded as `<len> <codes>`, where `0xC0 ≤ len ≤ 0xFD` encoding the length, while `0x80 ≤ codes ≤ 0xFD`. Top two bytes unused

Also have UCS-2 (2 byte encoding) which is fixed length, and UCS-4 (4 byte encoding) which is barking mad. Unicode includes directionality, whether text should be displayed right-to-left, or top-to-bottom, etc.



Both ASCII and Unicode are represented in memory as either strings or arrays: e.g. "Pub Time!"

Since |character| < |machine word size|, need to be careful with endianness. Example is little endian

NUMBERS

n-bit register $b_{n-1}b_{n-2}\ldots b_1b_0$ can represent 2^n different values

- b_{n-1} termed the **Most Significant Bit** (MSB), and b_0 the **Least Significant Bit** (LSB)

Unsigned numbers: treat the obvious way, i.e.,

$$val = b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_1 2^1 + b_0 2^0$$

e.g., $1101_2 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13$

Represents values from 0 to 2^{n-1} inclusive. For large numbers, binary is unwieldy so use hexadecimal (base 16)

- To convert, group bits into groups of 4,
  - e.g., $\texttt{1111101010}_2 = \texttt{0011}|\texttt{1110}|\texttt{1010}_2 = \texttt{3EA}_{16}$
- Often use `0x` prefix to denote hex, e.g., `0x107`
- Can use dot to separate large numbers into 16-bit chunks, e.g., `0x3FF.FFFF`

SIGNED NUMBERS

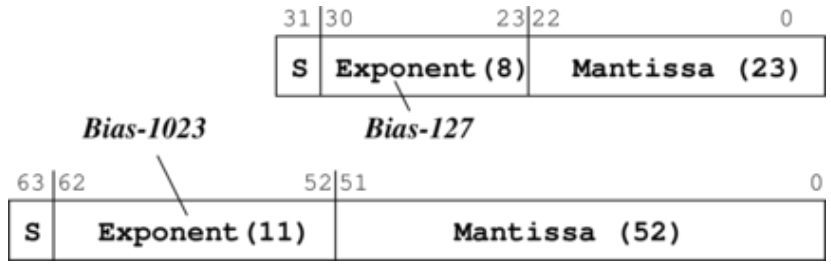
Two main options:

- Sign & magnitude:
  - Top (leftmost) bit flags if negative; remaining bits make value
  - E.g., $\texttt{10011011}_2$ is $\texttt{-0011011}_2$ $\texttt{= -27}$
  - Represents range $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$ and the bonus value $\texttt{-0}$ (!)
- 2's complement:
  - To get $-x$ from x, invert every bit and add 1
  - E.g., $\texttt{+27 = 00011011}_2$, $\texttt{-27 = (11100100}_2$ $\texttt{+ 1) = 11100101}_2$
  - Treat $\texttt{1000...000}_2$ as -2^{n-1}
  - Represents range -2^{n-1} to $+(2^{n-1}-1)$

FLOATING POINT

To handle very large or very small numbers, use scientific notation, e.g., $n = m \times 10^e$, with m the mantissa, e the exponent, e.g., $C = 3.01 \times 10^8$ m/s

For computers, use binary i.e. $n = m \times 2^e$, where m includes a "binary point"



In practice use IEEE floating point with normalised mantissa $m = 1.xx \ldots x_2$ i.e., use $n = (-1)^s((1 + m) \times 2^{e-b})$

IEEE floating point reserves $e = 0$ and $e = \max$:

- m zero: $e = \max$: $\pm\infty$; $e = 0$: ± 0 (!)
- m non-zero: $e = \max$: NaNs; $e = 0$: *denorms*

Normal positive range $[2^{-126}, \sim 2^{128}]$ for single, or $[2^{-1022}, \sim 2^{1024}]$ for double

NB. still only $2^{32}/2^{64}$ values — just spread out

DATA STRUCTURES

Not interpreted by machine — up to programmer (or, compiler) where things go, and how

Fields in records/structures stored as an offset from a base address. In variable size structures, explicitly store addresses (pointers) inside structure, e.g.,

```
datatype rec = node of int * int * rec
             | leaf of int;
val example = node(4, 5, node(6, 7, leaf(8)));
```

If example is stored at address 0x1000:

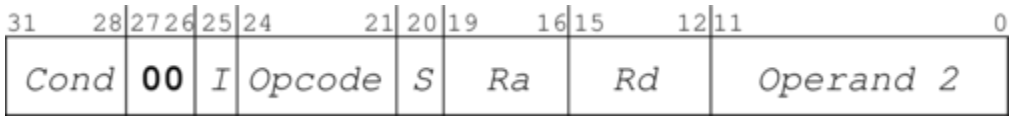
Address	Value	Comment
0x0F30	0xFFFF	Constructor tags for a leaf
0x0F34	8	Integer 8
.	.	.
0x0F3C	0xFFFE	Constructor tag for a node
0x0F40	6	Integer 6
0x0F44	7	Integer 7
0x0F48	0x0F30	Address of inner node
.	.	.
0x1000	0xFFFE	Constructor tag for a node
0x1004	4	Integer 4
0x1008	5	Integer 5
0x100C	0x0F3C	Address of inner node

ENCODING: INSTRUCTIONS

Instructions comprise:

- An **opcode**: specify what to do
- Zero or more **operands**: where to get values
- E.g., `add r1, r2, r3`

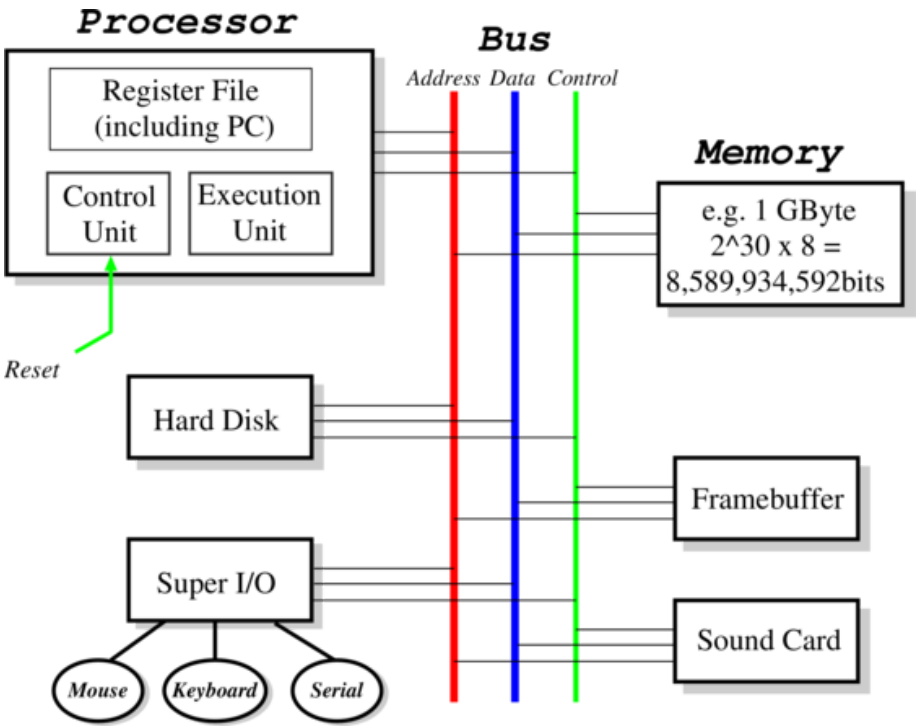
E.g., ARM ALU operations



Range of options:

- Addressing mode (how to interpret operands) either part of opcode, or given explicitly
- Variable length encoding: may give better code density; makes it easier to extend instruction set (!)
- Huffman encoding looks at most probable instructions and assigns them the shortest opcodes; infrequently used instructions get long ones
- But! Makes decoding rather tricky, lose on PC-relative, bad for cache

A MODEL COMPUTER



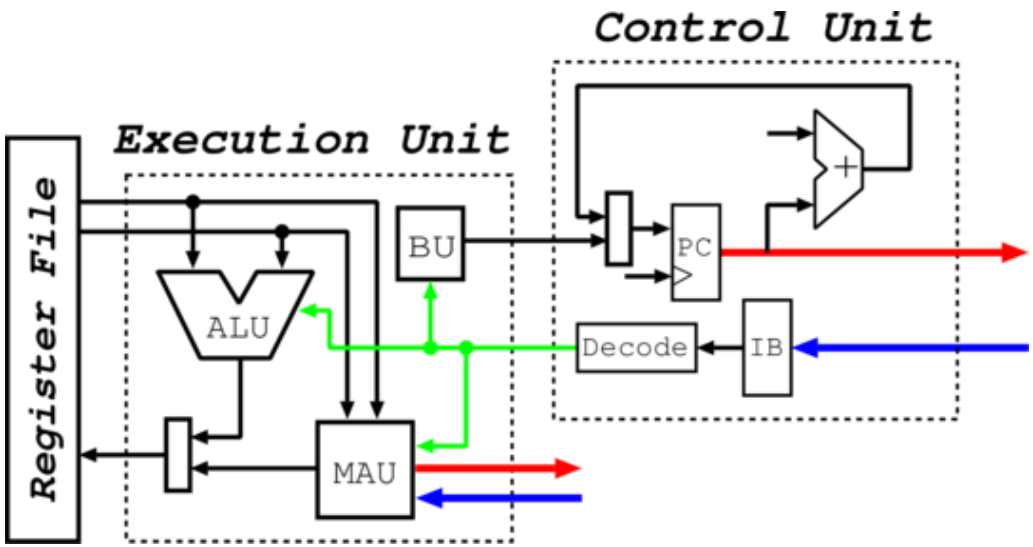
Processor (CPU) executes programs using: **Memory**: stores both programs & data. **Devices**: for input and output. **Bus**: transfers information

Computers operate on information in memory from input devices. Memory is a large byte array that holds any information on which we operate. Computer logically takes values from memory, performs operations, and then stores result back

CPU operates on registers, extremely fast pieces of on-chip memory, usually now 64-bits in size. Modern CPUs have between 8 and 128 registers. Data values are loaded from memory into registers before being operated upon, and results are stored back again

FETCH-EXECUTE CYCLE

CPU fetches & decodes instruction, generating control signals and operand information



Inside **Execution Unit** (EU), control signals select **Functional Unit** (FU) ("instruction class") and operation

If **Arithmetic Logic Unit** (ALU), then read one or two registers, perform operation, and (probably) write back result. If **Branch Unit** (BU), test condition and (maybe) add value to PC. If **Memory Access Unit** (MAU), generate address ("addressing mode") and use bus to read/write value

Repeat

INPUT/OUTPUT DEVICES

Devices connected to processor via a bus (e.g., ISA, PCI, AGP):

- Mouse, Keyboard
- Graphics Card, Sound Card
- Floppy drive, Hard-Disk, CD-Rom
- Network Card, Printer, Modem
- etc.

Often two or more stages involved (e.g., IDE, SCSI, RS-232, Centronics, etc.)

Connections may be indirect, e.g.,

- Graphics card (on bus) controls monitor (not on bus)

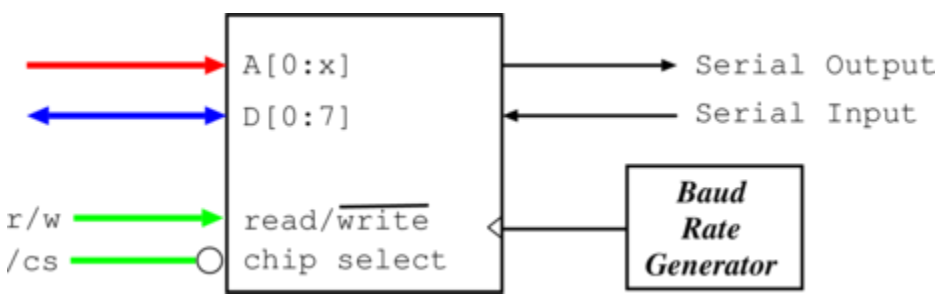
UNIVERSAL ASYNCHRONOUS RECEIVER/TRANSMITTER (UART)

Stores 1 or more bytes internally, converting parallel to serial

- Outputs according to RS-232
- Various baud rates (e.g., 1,200 – 115,200)
- Slow, simple, and **very useful**

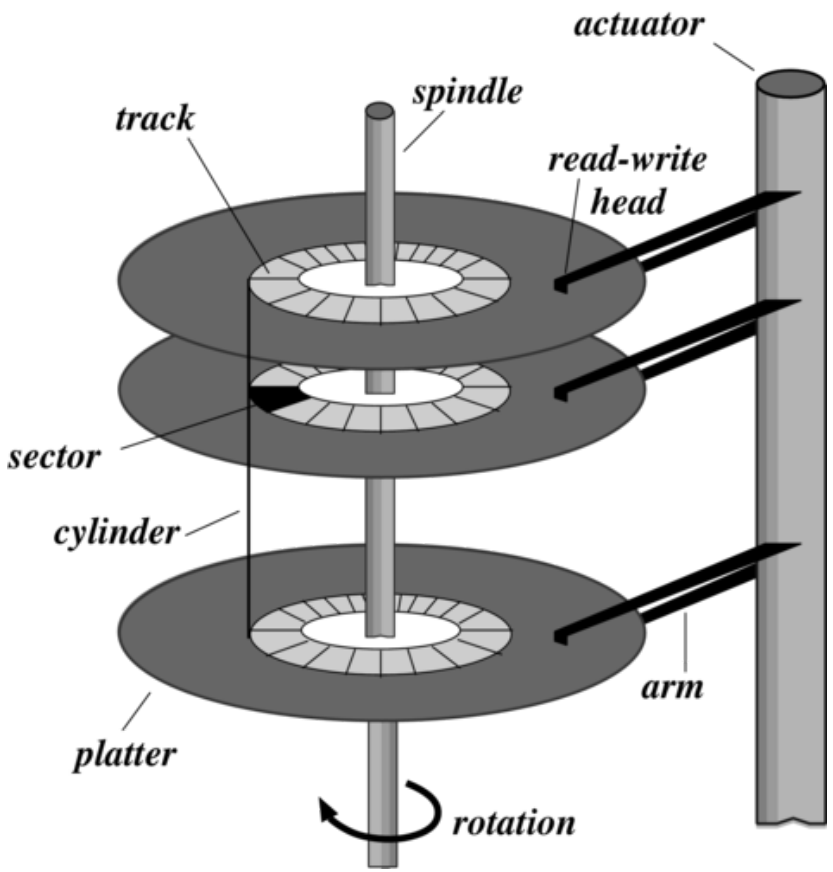
Make up "serial ports" on PC

- Max throughput ~14.4kb/s; variants up to 56kb/s (for modems)
- Connect to terminal (or terminal emulation software) to debug device

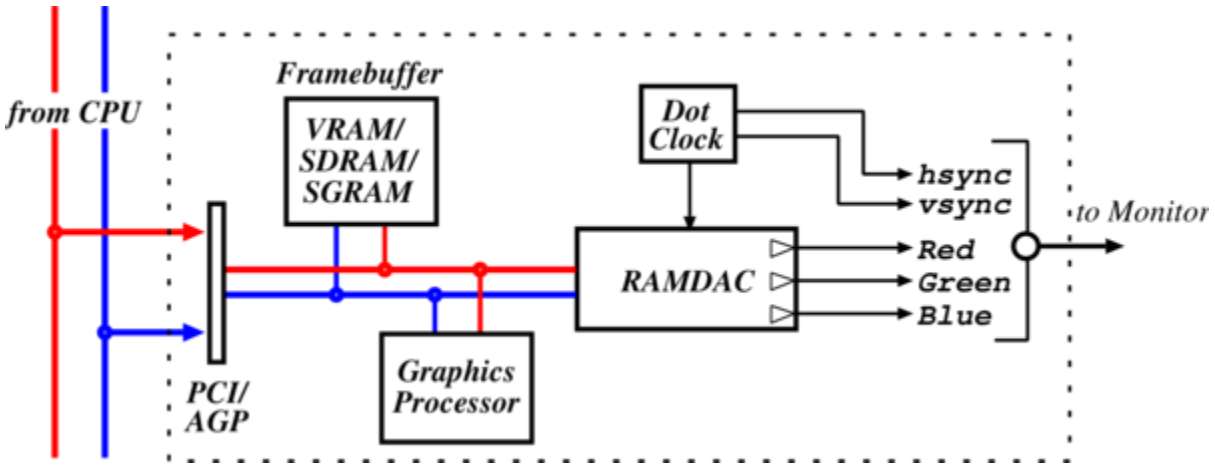


HARD DISKS

Whirling bits of metal, increasingly replaced by **Solid State Devices** (SSDs). Up to around 15,000 rpm, ~2TB per platter, ~2Gb/s



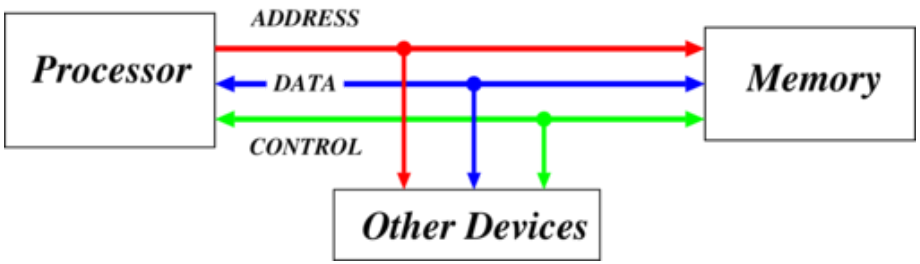
GRAPHICS CARDS



Essentially some RAM (**framebuffer**) and (older) some digital-to-analogue circuitry (**RAMDAC**)

- RAM holds array of pixels: picture elements
- Resolutions e.g., 640x480, 800x600, 1024x768, 1280x1024, 1600x1200, ...
- Depths: 8-bit (LUT), 16-bit (RGB=555), 24-bit (RGB=888), 32-bit (RGBA=888)
- Memory requirement = $x \times y \times \text{depth}$, e.g., 1280x1024 @ 16bpp needs 2560kB
- Full-screen 50Hz video requires 125 MB/s (or ~1Gb/s)

BUSES



Collection of shared communication wires: low cost, versatile but potential bottleneck. Typically comprises **address lines** (determine how many devices on bus), **data lines** (determine how many bits transferred at once) and **control lines**, plus power and ground. Operates in a master-slave manner, e.g.,

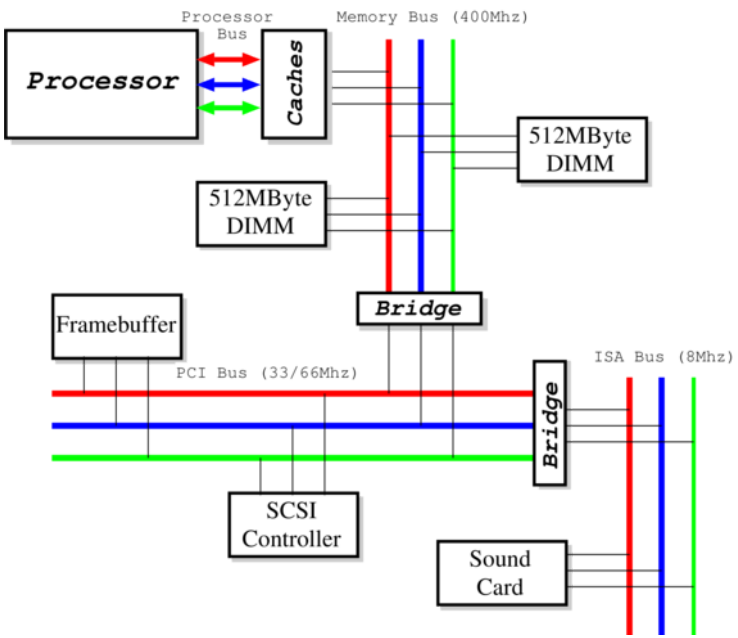
- Master decides to e.g., read some data
- Master puts address onto bus and asserts `read`
- Slave reads address from bus and retrieves data
- Slave puts data onto bus
- Master reads data from bus

Mean we don't need wires everywhere! Also can define bus protocol and then do "plug'n'play"

BUS HIERARCHY

In practice, many different buses with different characteristics, e.g., data width, max number of devices, max length. Most are **synchronous**, i.e. share a clock signal.

E.g., with four buses:



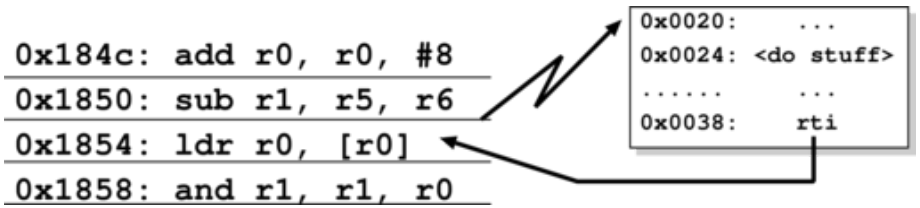
Processor bus: fastest (and widest?), for CPU to talk to cache

Memory bus: to communicate with memory

PCI and **(E)ISA buses**: to communicate with current and legacy devices

Bridges forwards from one side to the other; e.g., to access a device on ISA bus, processor generates magic [physical] address which goes to memory bridge, then to PCI bridge, and then to ISA bridge, and finally to ISA device. Same on the way back

INTERRUPTS



Bus reads and writes are transaction based: CPU requests something and waits until it happens. But, e.g., reading a block of data from a hard-disk might take ~2ms, which could be >5M clock cycles!

Interrupts provide a way to decouple CPU requests from device responses

- CPU uses bus to make a request (e.g., writes some special values to a device)
- Device fetches data while CPU continues doing other stuff
- Device raises an interrupt when it has data
- On interrupt, CPU vectors to handler, reads data from device, and resumes using special instruction, e.g., `rti`

NB. Interrupts happen at any time but are deferred to an instruction boundary. Interrupt handlers must not trash registers, and must know where to resume. CPU thus typically saves values of all (or most) register, restoring with `rti`

DIRECT MEMORY ACCESS (DMA)

Interrupts good but (e.g.) livelock a problem. Even better is a device which can read and write processor memory directly — enter **Direct Memory Access** (DMA). A generic DMA "command" might include:

- Source address
- Source increment / decrement / do nothing
- Sink address
- Sink increment / decrement / do nothing
- Transfer size

Get just one interrupt at end of data transfer. DMA channels may be provided by dedicated DMA controller, or by devices themselves: e.g. a disk controller that passes disk address, memory address and size, and give instruction to read or write. All that's required is that a device can become a bus master. Scatter/Gather DMA chains requests, e.g., of disk reads into set of buffers

Complexities?

SUMMARY

Computers made up of four main parts:

1. Processor (including register file, control unit and execution unit)
2. Memory (caches, RAM, ROM)
3. Devices (disks, graphics cards, etc.)
4. Buses (interrupts, DMA)

Information represented in all sorts of formats:

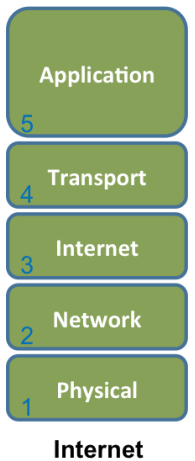
- Strings
- Signed & unsigned integers
- Floating point
- Data structures
- Instructions

KEY CONCEPTS

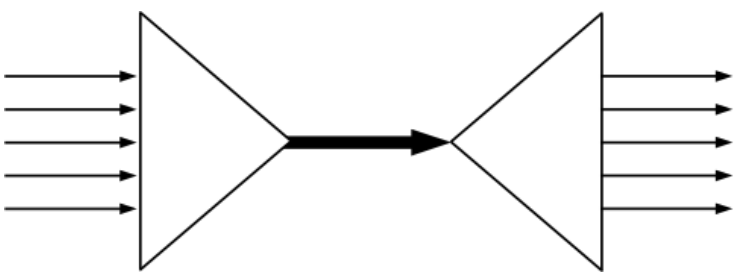
- Course Summary
- Recap
- **Key Concepts**
  - **Layering & Multiplexing**
  - **Synchronous & Asynchronous**
  - **Latency, Bandwidth, Jitter**
  - **Caching & Buffering**
  - **Bottlenecks, 80/20 Rule, Tuning**
- Operating Systems

LAYERING

Layering is a means to manage complexity by controlling interactions between components. Arrange components in a stack, and restrict a component at layer *X* from relying on any other component except the one at layer *X–1* and from providing service to any component except the one at layer *X+1*



MULTIPLEXING



Traditionally a method by which multiple (analogue) signals are combined into a single signal over a shared medium. In this context, any situation where one resource is being consumed by multiple consumers simultaneously

SYNCHRONOUS & ASYNCHRONOUS

Loosely, shared clock (**synchronous**) vs no shared clock (**asynchronous**). In networking, an asynchronous receiver needs to figure out for itself when the transfer starts and ends while a synchronous receiver has a channel over which that's communicated

In the case of Operating Systems, whether two components operate in lock-step: **synchronous IO** means the requester waits until the request is fulfilled before proceeding, while with **asynchronous IO**, the requester proceeds and later handles fulfilment of their request

LATENCY, BANDWIDTH, JITTER

Different metrics of concern to systems designers

Latency: How long something takes. E.g., "This read took 3 ms"

Bandwidth: The rate at which something occurs. E.g., "This disk achieves 2 Gb/s"

Jitter: The variation (statistical dispersal) in latency (frequency). E.g., "Scheduling was periodic with jitter $+/-$ 50μsec"

Be aware whether it is the **absolute** or **relative** value that matters, and whether the **distribution** of values is also of interest

CACHING & BUFFERING

A common system design problem is to handle **impedance mismatch** — a term abused from electrical engineering — where two components are operating at different speeds (latencies, bandwidths). Common approaches are:

- **Caching**, where a small amount of higher-performance storage is used to mask the performance impact of a larger lower-performance component. Relies on **locality** in time (finite resource) and space (non-zero cost)

  E.g., CPU has registers, L1 cache, L2 cache, L3 cache, main memory

- **Buffering**, where memory of some kind is introduced between two components to soak up small, variable imbalances in bandwidth. NB. Doesn't help if one component simply, on average, exceeds the other

  E.g., A hard disk will have on-board memory into which the disk hardware reads data, and from which the OS reads data out

BOTTLENECKS, TUNING, 80/20 RULE

There is typically one resource that is most constrained in a system — the **bottleneck**

Performance optimisation and **tuning** focuses on determining and eliminating bottlenecks

- But often introduces new ones

A **perfectly balanced system** has all resources simultaneously bottlenecked

- Impossible to actually achieve

Often find that **optimising the common case** gets most of the benefit anyway

- Means that measurement is a prerequisite to performance tuning!
- The 80/20 rule — 80% time spent in 20% code
- If you highly optimise a very rare case, it'll make no difference

OPERATING SYSTEMS

- Course Summary
- Recap
- Key Concepts
- **Operating Systems**
  - **What is and is not an Operating System?**
  - **Evolution of Operating Systems**

WHAT IS AN OPERATING SYSTEM?

> *A program controlling the execution of all other programs*

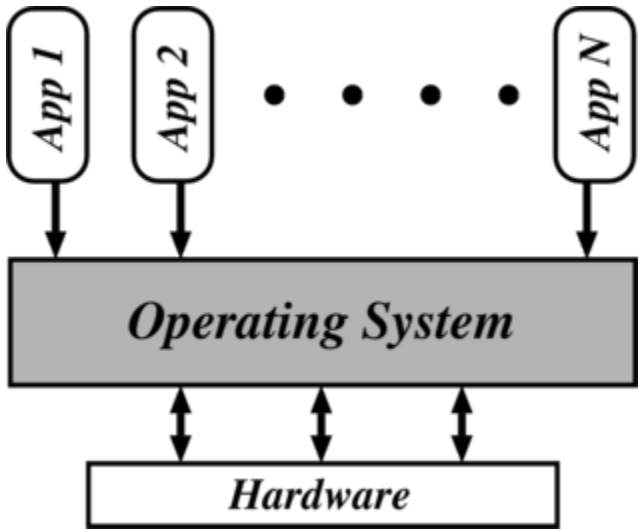
Objectives:

- **Convenience** — hide all the gunk we've just recapped
- **Efficiency** — only does articulation work so minimise overheads
- **Extensibility** — need to evolve to meet changing application demands and resource constraints

There's an analogy to a government: does no useful work, simply legislates on resource use by competing applications with the intent of achieving best function of system (society) through policy

(Also difficult to change and can be imposed on users without consent ;)

WHAT IS NOT AN OPERATING SYSTEM



The Operating System (OS) controls all execution, multiplexes resources between applications, and abstracts away from complexity

Consider the last point particularly — typically involves libraries and tools provided as part of the OS, in addition to a kernel (e.g., `glibc` — but what about language runtime?). Thus no-one really agrees **precisely** what the OS is

For our purposes, focus on **the kernel**

IN THE BEGINNING. . .

First stored-program machine (EDSAC, 1949-1955), operated "open shop": user = programmer = operator. All programming in machine code (no assembly mnemonics). Users sign up for blocks of time to do development, debugging, etc. To reduce costs, hire a separate (relatively unskilled) operator: management happy, everyone else hates it. Also reduces "interactivity" so CPU utilisation reduces

BATCH SYSTEMS

Introduction of tape drives allow batching of jobs:

- Programmers put jobs on cards as before
- All cards read onto a tape
- Operator carries input tape to computer
- Results written to output tape
- Output tape taken to printer

SPOOLING SYSTEMS

Even better: spooling systems

- Spool jobs to tape for input to CPU, on a slower device not connected to CPU
- Interrupt driven IO
- Magnetic disk to cache input tape
- Fire operator

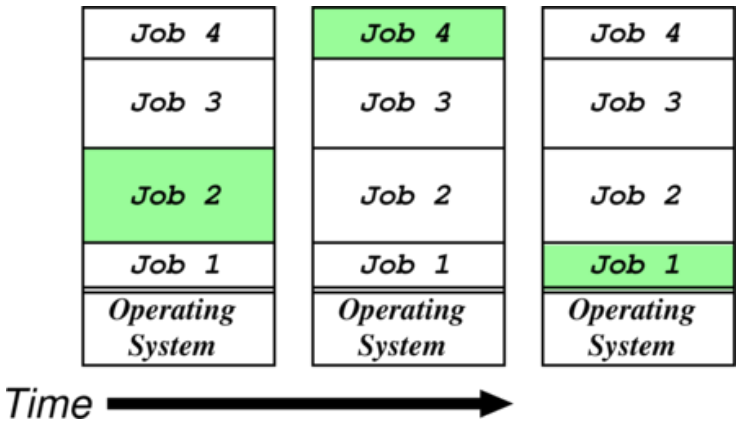
Computer now has a resident monitor:

- Initial control is in monitor, which reads job and transfers control
- End of job, control transfers back to monitor

Monitor now schedules jobs, so need job control language to separate jobs on tape and to allow jobs to pass control to the monitor on completion, e.g. FMS had `$JOB`, `$FTN`, `$RUN`, `$DATA`, `$END`, with `$FTN` being optional for assembly programs

But! Need to "trust" the job will give control back to monitor, and devices still slow compared to CPU though...

MULTI-PROGRAMMING



Use memory to cache jobs from disk, meaning >1 job active (resident) simultaneously

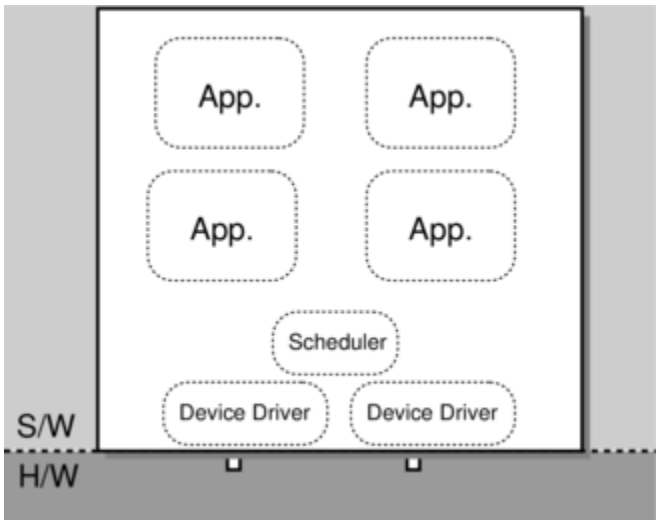
Two stage scheduling: 1. select jobs to load: **job scheduling**; 2. select resident job to run: **CPU scheduling**. End up with one job computing while another waits for IO, causes competition for CPU and space in main memory

Batch Multi-Programming: extension of batch system to allow more than one job to be resident simultaneously

Users wanting more interaction leads to time-sharing:

- E.g., CTSS (first, in 1961), TSO, Unix, VMS, Windows NT, ...
- Use timesharing to develop code, then batch to run: give each user a teletype/terminal; interrupt on `return`; OS reads line and creates new job

MONOLITHIC OPERATING SYSTEMS



Oldest kind of OS structure ("modern" examples are DOS, original MacOS)

Applications and OS bound in a big lump, without clear interfaces. All OS provides is a simple abstraction layer, making it easier to write applications

Problem is, applications can trash the OS, other applications, lock the CPU, abuse IO, etc. Doesn't provide useful fault containment. Need a better solution...

OPERATING SYSTEM FUNCTIONS

Regardless of structure, OS needs to securely multiplex resources, i.e. to protect applications while sharing physical resources. Many OS design decisions are about where this line is drawn

Also usually want to abstract away from grungy harware, i.e. OS provides a virtual machine to:

- Share CPU (in time) and provide each application with a virtual processor
- Allocate and protect memory, and provide applications with their own virtual address space
- Present a set of (relatively) hardware independent virtual devices
- Divide up storage space by using filing systems

Remainder of this part of the course will look at each of the above areas in turn

SUMMARY

- Course Summary
- Recap
  - Encodings: Text, Numbers, Data Structures, Instructions
  - A Model Computer and the Fetch-Execute Cycle
  - Some IO Devices, Buses, Interrupts, DMA
- Key Concepts
  - Layering & Multiplexing
  - Synchronous & Asynchronous
  - Latency, Bandwidth, Jitter
  - Caching & Buffering
  - Bottlenecks, 80/20 Rule, Tuning
- Operating Systems
  - What is and is not an Operating System?
  - Evolution of Operating Systems