Computer Architecture

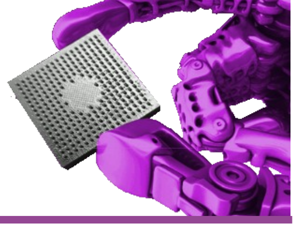
An embedded approach

Module 6

CPU Externals
(connecting it to the real world)

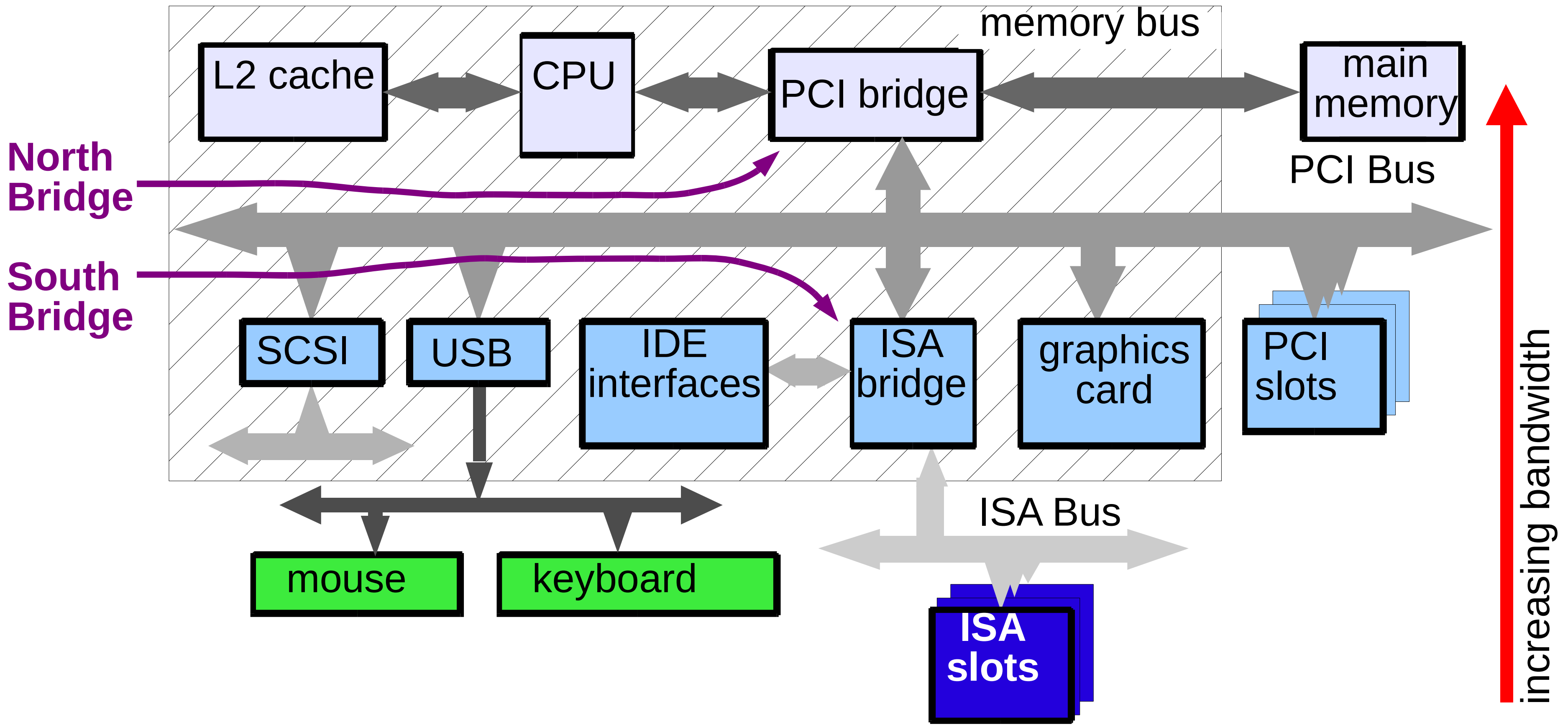
Contents

Bus interfacing

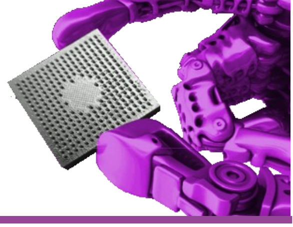


Before we can talk to the real (analogue) world, we need to explore how a CPU device can connect to the other internal units inside a computer.

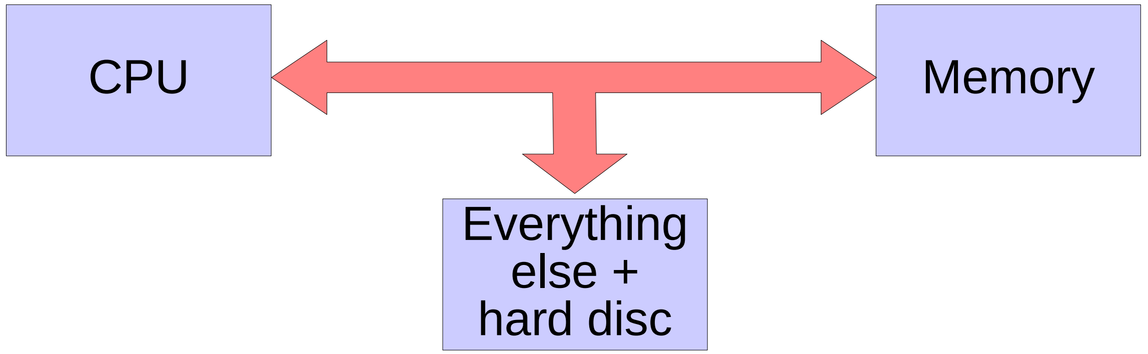
The diagram below shows a PC (x86) from the late 1990s:



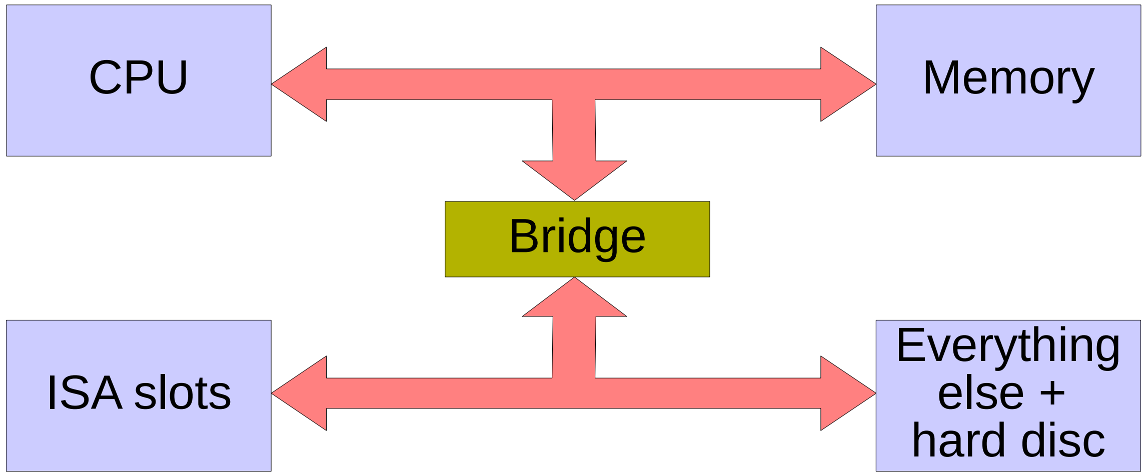
Bus interfacing



We can redraw this a little and show how things are evolving starting from a very basic computer design of maybe 30 years ago:

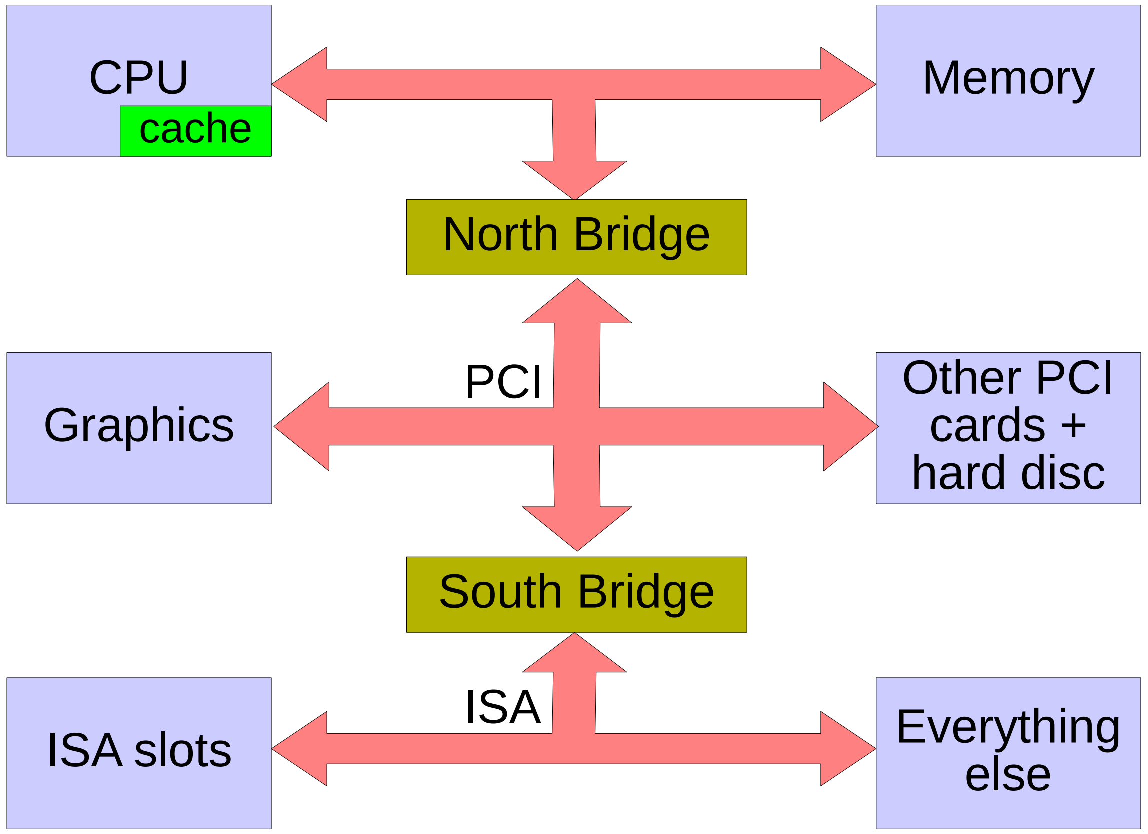


The one bus means a bottleneck (especially when "Everything Else" contains some slow devices)! So we decouple the slow stuff:



Bus interfacing

In actual fact, it's not just fast or slow, there are several speed levels, so we need to layer more carefully, and use a cache to speed up memory access:



But there is more we can do... and as graphics became more important, the graphics chips became faster and faster...

Bus interfacing

With fast graphics, the North Bridge and South Bridge became smart (and now have different names!)

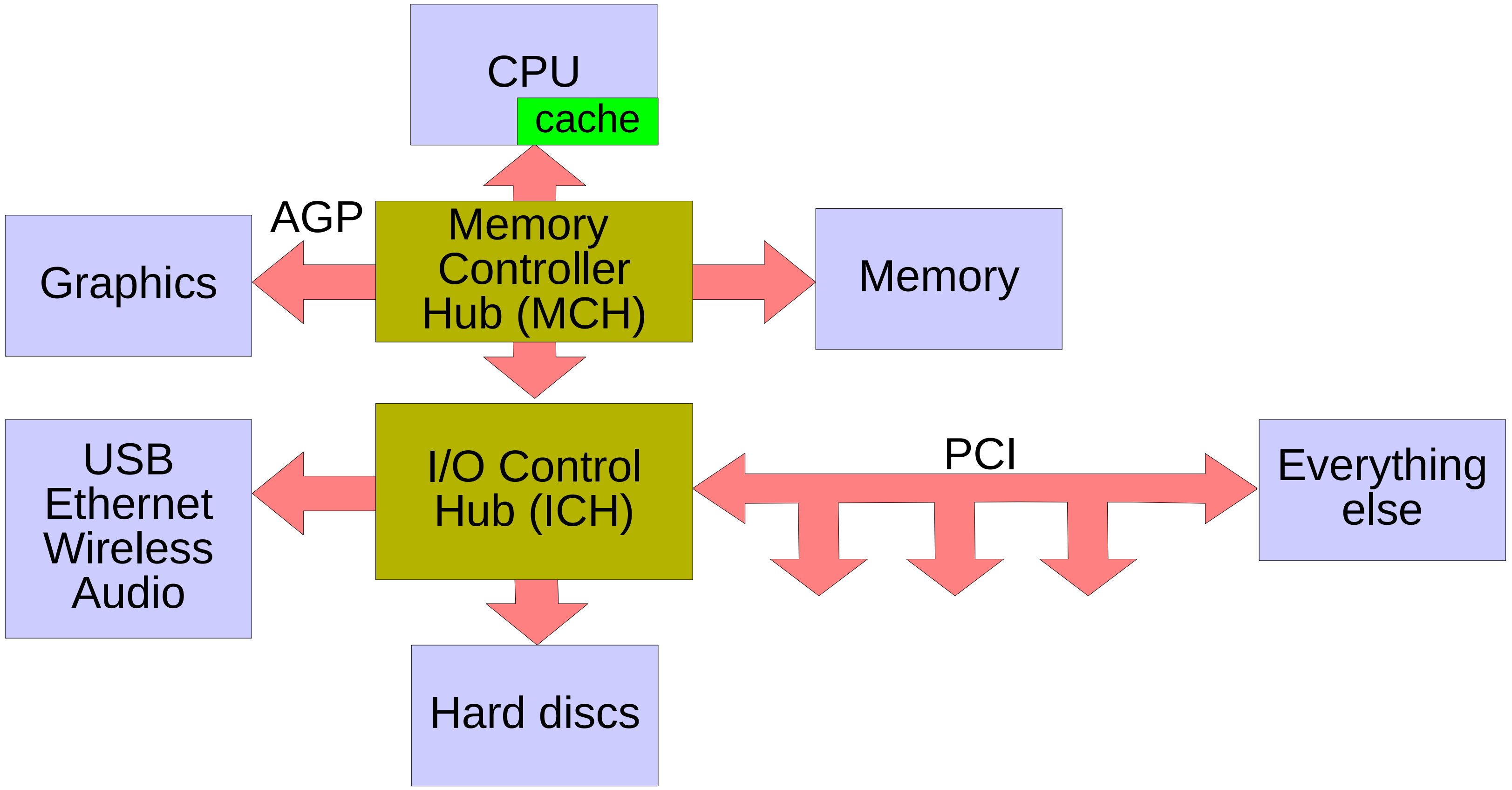
3 GBytes/s

1 GBytes/s

300 MBytes/s

133 MBytes/s

Typical Bandwidths



Bus interfacing

Embedded systems are very similar to what we saw, except:

1. They tend to be a little slower (for reasons of power).

2. Everything will be on a single chip (especially for SoC).

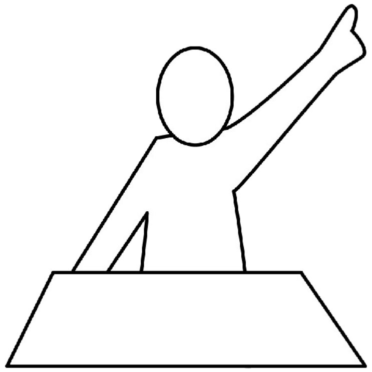
3. Different bus types, instead of AGP, PCI, we might have

AMBA:	advanced microcontroller bus architecture
AHB:	ARM host bus
APB:	ARM peripheral bus
AXI:	no full name was given by ARM but we can guess...
	AMBA eXtension Interface?
	ARM's eXtreme Interface?

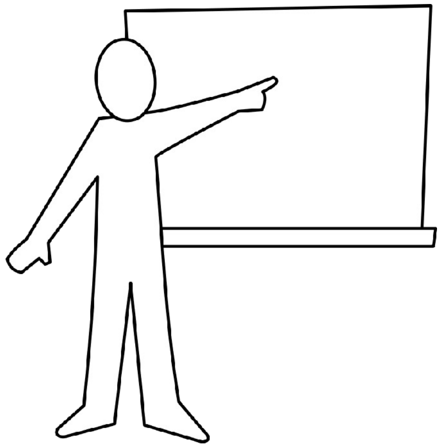
4. Often no hard disc – we prefer flash memory in embedded systems! We also have more built-in peripherals.

5. No MCH, ICH, North Bridge or South Bridge terminology – in embedded systems we design smaller and simpler bridges.

Bus interfacing



So many types of parallel bus!!!
How can I ever learn about so many?

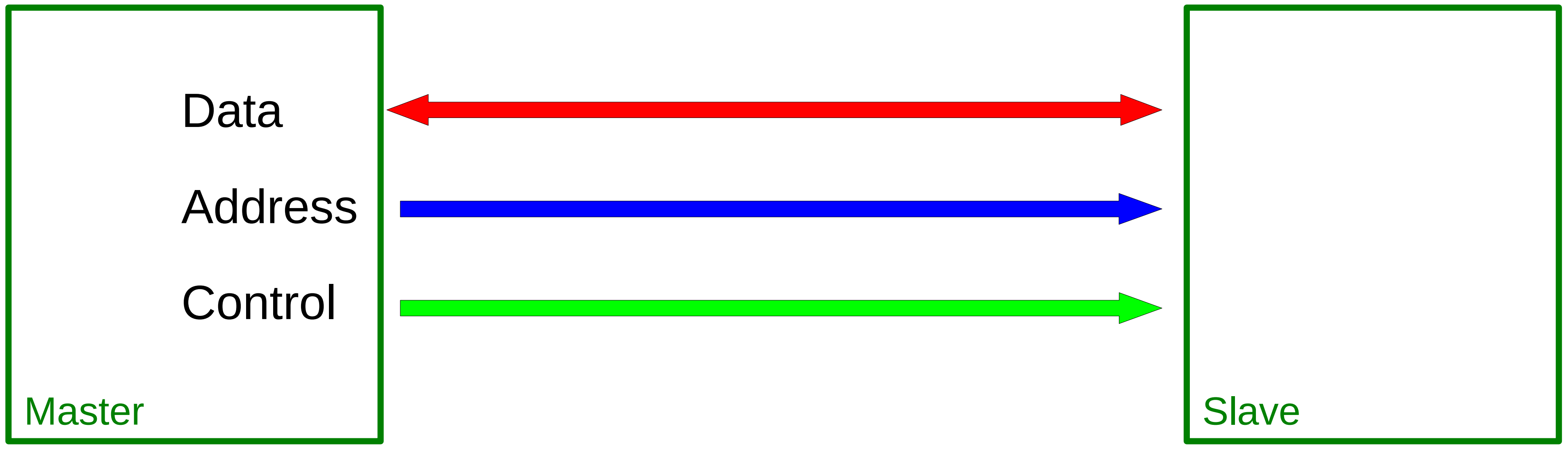


The good news is that, at least for basic operation, they are all pretty much the same.

So let's look at the typical control signals and methods.

Bus interfacing

A parallel bus normally has three components:



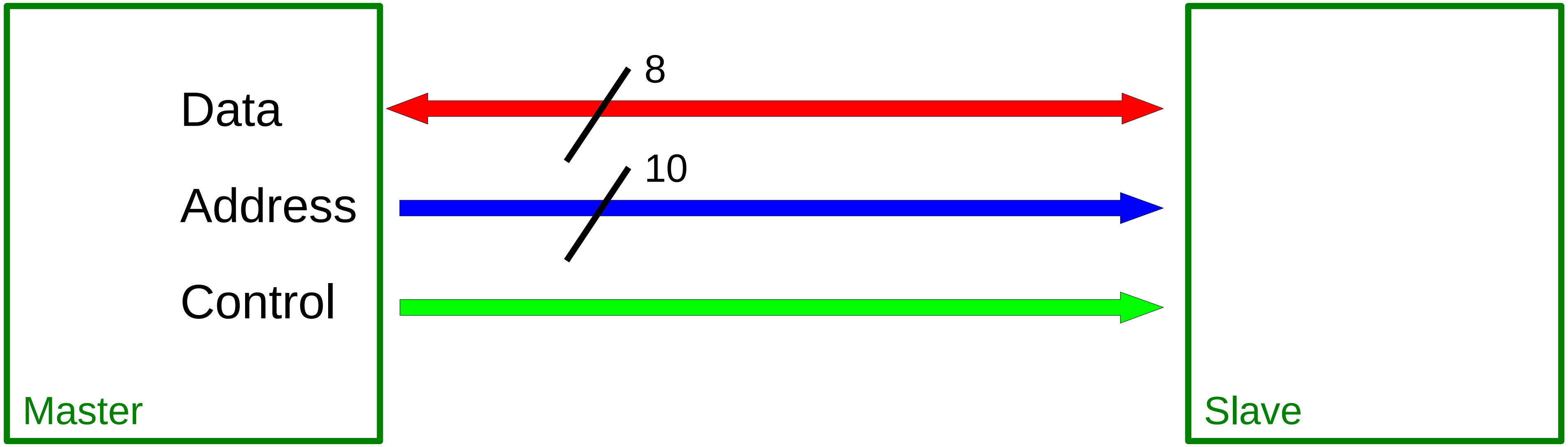
Data can be read or written on the Data bus, and it's the Master (the CPU is almost always the master), that decides *when* to write and *when* to read.

The Master uses the Control bus to tell the Slave what to do.

If the Slave contains different items of information, the Address bus is used by the Master to tell the Slave exactly which item it wants to read from or write to.

Bus interfacing

The buses have different widths:



The width of the bus determines how much data (how many bits) get transferred in each bus cycle.

In fact the buses are comprised of individual wires connecting individual pins, such as D[0], D[1]... D[7] for an 8-bit data bus, or A[0], A[1], A[2]...A[9] for a 10-bit address bus.

A 10 bit address bus can select 2^{10} locations = 1 kibyte, although there are ways to expand this (e.g. DRAM; see page 319).

Bus interfacing

Remember, in parallel buses, each data bit is transmitted on a separate wire. These are usually 8, 16 or 32 bits wide.

A bus is a common method to exchange information. Different devices can read and write to a bus at different times, but only one device (the master) should write at any instant (drive the bus).

As mentioned, a CPU is usually the master. Otherwise a separate bus arbiter can be used to control the bus.

Driving the bus pulls the wires high or low, reading the bus measures the wire voltage. Devices which are not driving the bus need to be in a high impedance state (high-Z). These are not affecting the bus in any way, allowing it to be driven by other devices.

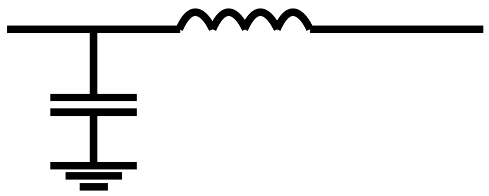
Bus interfacing

When a voltage is applied to a wire, there is a short delay before the entire wire reaches the same voltage, due to its inherent capacitance. This means that once a bus is driven, there is a set-up time needed before any devices can read that bus.

Peripheral devices also need the bus state to remain static for a short time while they read the bus. This is the hold time. These two constraints limit bus cycle speed.

A faster bus needs a lower capacitance (or the devices which drive it must be able to source/sink more current so that the bus charges up quicker).

Bus interfacing



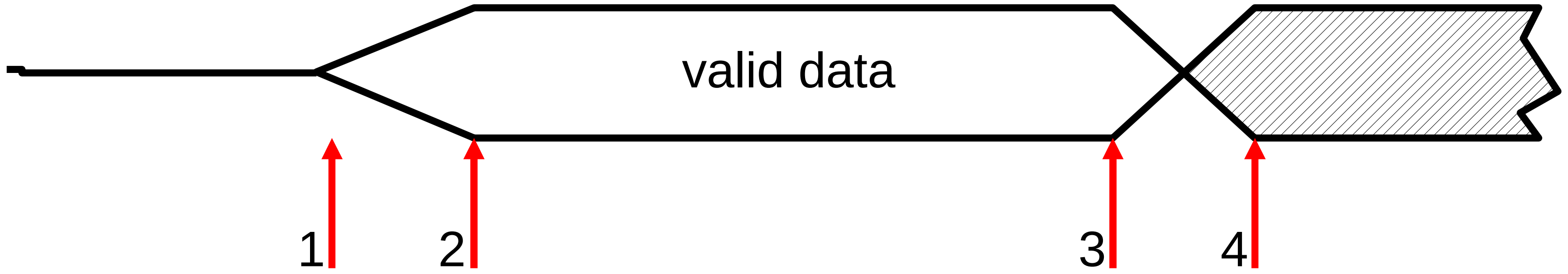
In fact, the equivalent circuit of a bus contains a capacitor and an inductor for each line. The capacitance is increased where there are long buses with many closely-spaced wires, or where there are ground-planes. Capacitance increases the time required to assert a voltage on the bus.

The inductance is caused by closely-spaced wires or convoluted PCB tracks (mutual inductance). This can cause voltage overshoot or voltage reflections.

Everything depends on the length and layout of the bus, and the driving circuitry. It limits the speed at which the bus can reliably transmit data.

Bus interfacing

Here is timing diagram for a synchronous bus:

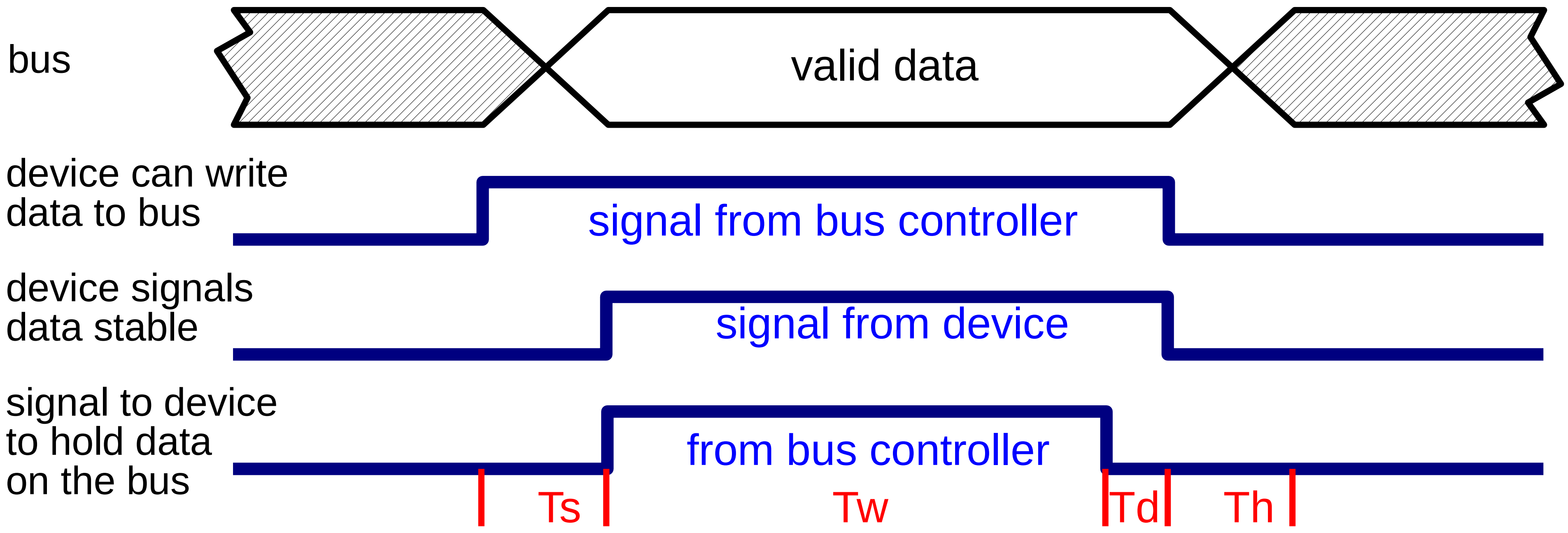


1. a device starts to assert data on the bus
2. the data on the bus has stabilised and can be used
3. the device starts to de-assert the data it is driving
4. bus is now being driven by another device

Think about these questions:
Q. What is on the bus before position 1, and after position 4?
Q. What voltage is on the bus between 1 and 2?
Q. What is happening between 3 and 4?

Bus interfacing

A bus controller must tell a device reading from the bus when data is ready, and that data must remain until the device has finished reading it. This can be through timing, or through a signal from the device to the controller. For a device writing to the bus, the sequence is similar;



bus
valid data
device can write data to bus
signal from bus controller
device signals data stable
signal from device
signal to device to hold data on the bus
from bus controller
Ts
Tw
Td
Th

Most CPUs these days contain an internal bus control unit.

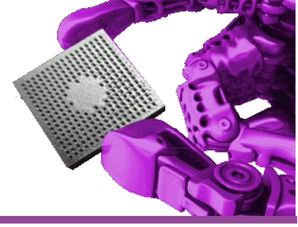
Bus interfacing

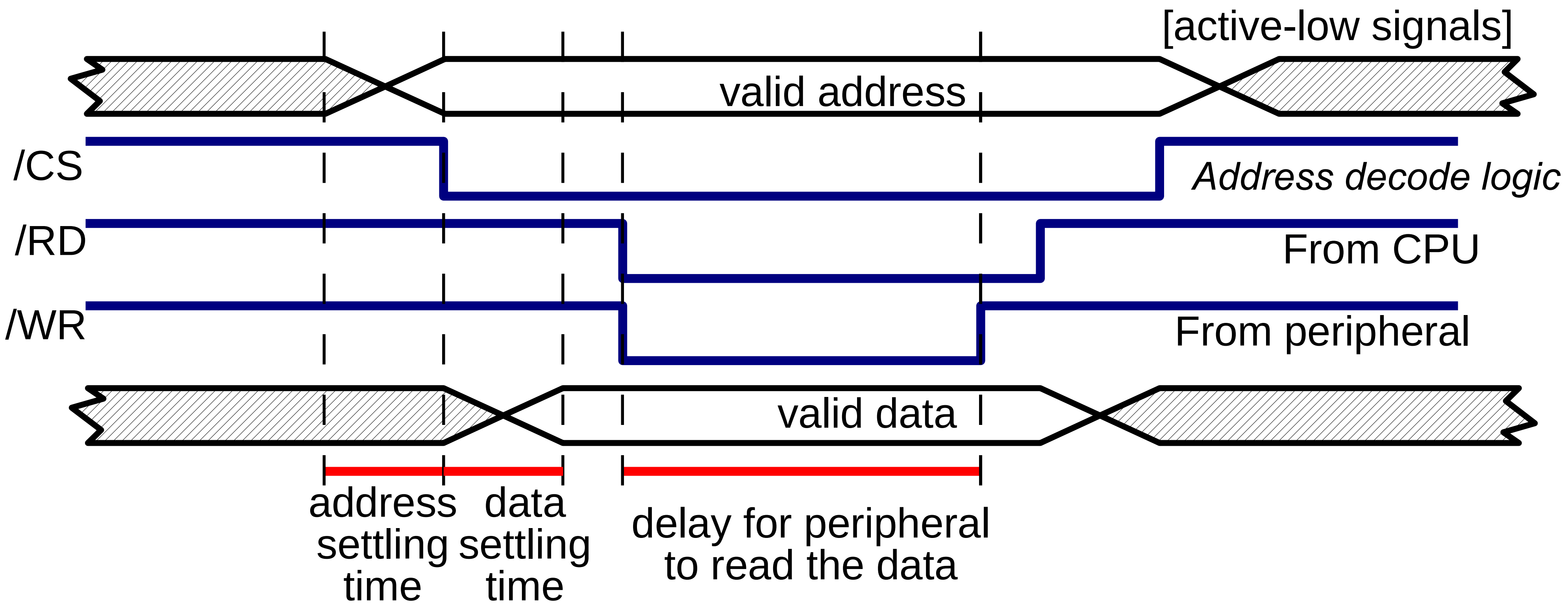
We've only considered the data bus so far; but we often need an address bus to choose which device is being accessed as well as to access devices with multiple internal storage locations (such as RAM).

A possible sequence of events to read from such a peripheral:
1. Controller selects peripheral by placing its address on the address bus (which is used by an address decoder to select the peripheral and often a few address lines are also used to select locations within a peripheral).
2. Controller waits long enough to ensure that the address bus is steady.
3. Controller generates a signal to all peripherals to write (or read).
4. The address-selected peripheral drives its data on to the data bus.
5. (optional the selected peripheral outputs an acknowledge signal).
6. Controller keeps the write signal active until the data bus has been read.
7. Controller f nally de-asserts the write signal.
8. Peripheral de-asserts the acknowledge, and stops driving the data bus.
9. After a short time (hold-off period), the controller can repeat the process.

Many variants exist to this basic scheme....

Bus interfacing

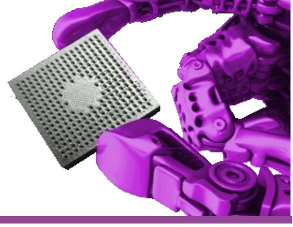




When a CPU writes to a peripheral (asserting the read signal), what happens when that peripheral is too slow to read the data? Either the CPU knows it should extend the data validity by adding wait states, or the peripheral needs to return a bus hold signal to the CPU.

For more on nCS, nRD, nWR, refer to book Section 6.1.1 page 253.

Bus interfacing



Some early 8-bit CPUs reduced pin-count by multiplexing address and data buses. This reduced the number of pins by 7 (-8 because only bus and +1 pin used to indicate whether data or address is being carried).

More modern CPUs can still multiplex data buses: a 16-bit bus can handle 32-bit data in two transfers per 32-bit word.

Some CPUs have different address spaces: these can commonly be data memory, program memory and I/O.

For example, TMS320C50 shares one 16-bit data bus and one 16-bit address bus between 3 address spaces. The address spaces are selected by 3 pins: DS, PS and IS.
Shared buses are also used in ADSP2181 and x86 designs.

Bus interfacing

Synchronous buses are those that make state changes synchronous to a clock, this means, for example, that wait states will be a certain number of clock cycles. These are easiest to control. Unfortunately the bus speed is limited to the slowest device which is connected.

Asynchronous buses use hold and wait signals to control transfers, not number of clock cycles. Each transfer period lasts as long as necessary, and no longer. Obviously these have less wasted time and are more eff cient, but are more diff cult to control.

Synchronous buses include: ISA, EISA (both 8.33MHz), PCI (33MHz or 66MHz), SCSI (5 to 20MHz), AMBA etc...

Few asynchronous buses are used in normal architectures.

Bus interfacing

Bus bandwidth calculation:

The PCI bus available in most PCs is a 32-bit wide bus that runs at 33MHz, so bus bandwidth is 4 bytes×33=132MBytes/second.

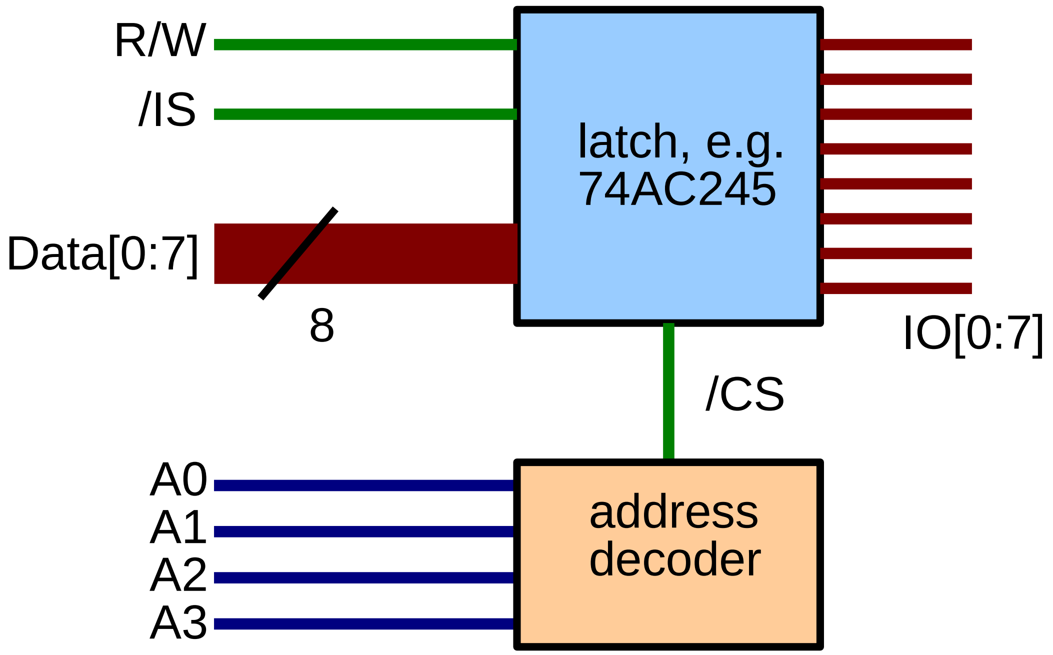
In workstations, PCI can be 64-bits wide and run at 66MHz, giving 8 bytes×66=528MBytes/second.

However these calculations assume no overheads for addressing, control information, and possible slow peripherals requiring wait states. They are theoretical maximums.

For an 8-bit multiplexed data/address bus running at 8MHz, bus bandwidth might be 1 byte×8MHz/2 = 4MBytes/second.

Bus interfacing

Two main approaches are taken to I/O: the first is to memory-map I/O ports. This means that an external address decoder on a shared address bus will drive I/O latches. Writes/ready to memory within the range encompassed by the address decoder will perform I/O reading and writing (an example TMS320C50 I/O port interface):



R/W is high or low depending if a read or write is wanted. When the address bus has the correct address, the /CS signal becomes active. Soon after, the /IS signal is driven low to indicate that an I/O read or write should occur.

This circuit displays partial address decoding...

Bus interfacing

DMA (Direct Memory Access):

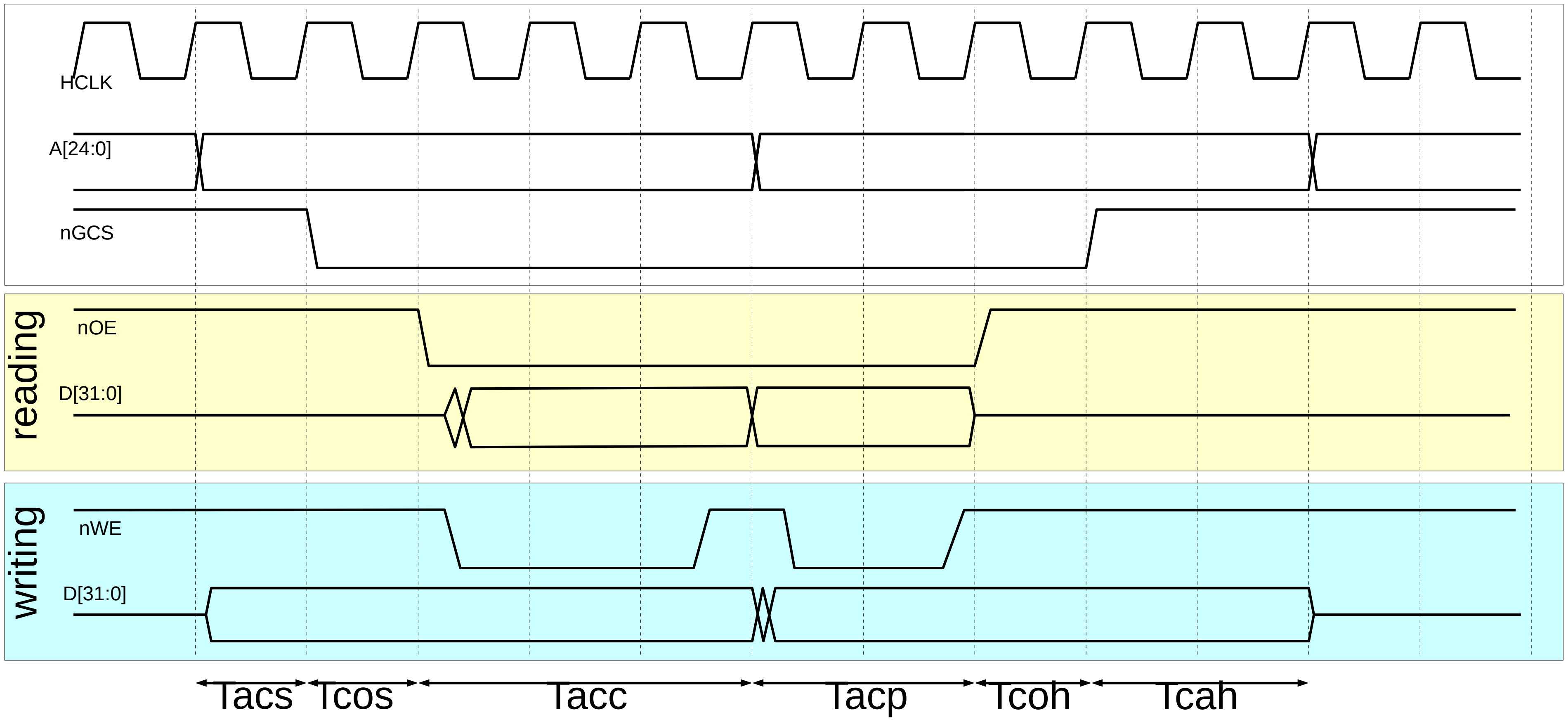
The bus controller (CPU) will **relinquish control** of a bus to allow peripheral devices to communicate directly without the overhead of CPU intervention, and also allow the CPU to continue other tasks during the DMA transfer.

A DMA device issues a **bus request**, and is given a **bus grant** signal by the CPU in order to begin DMA transfer.

The parameters of the transfer are usually set up in software by the CPU.

Bus interfacing

Consider the following memory bus diagram for the Samsung S3C2410 ARM:



Refer to book section 6.2, page 255 for a detailed explanation of the read and write transactions shown, plus the meanings and timing specif cations of each of the control signals.

Bus interfacing

Serial buses use fewer device pins than parallel buses,and can therefore reduce pin-count. They also consume less power, so better are for battery-powered systems.

Serial buses can also be synchronous or asynchronous; PC serial ports cater for asynchronous transmission. Serial bus standards for audio (such as I^{2}S and S/PDIF are synchronous).

USB and firewire are examples of modern serial buses...

Question to consider: Why are serial buses so popular and what further advantages do they have over parallel buses? Do they have any disadvantages?

Bus interfacing

Another connectivity approach usual on micro-controllers such as PIC, MSP, H8S and some DSP processors like the ADSP2181, is to provide a few general purpose I/O pins on the CPU package. These are addressed directly by specialised instructions. e.g.

```
SET    FL1            ;on ADSP2181, this sets pin FL1 high
or
BSF    PORTA,0        ;on PIC, this sets bit 0 of I/O port A high
```

Since these are controlled through on-chip registers, reading and writing to them is quick; there is no need to assert an address on the address bus, wait for it to settle etc.... They also reduce the chip count and reduce power consumption.

These GPIO pins also allow us (if we have enough of them) to write software to implement almost any bus standard. For example, if we don't have a built-in SPI controller on our CPU, we can write a small SPI program that controls a few pins to act as an SPI bus. For serial protocols, this is called bit-banging.

Bus interfacing

There is much more to the topic of bus interfacing – for something that sounds simple, it can become quite complex!

Refer to chapter 6 in the book for details of many specif c system buses that can be commonly found in today's embedded devices.

Real-time issues

Many embedded systems are also real-time systems.... but what is real-time?

"any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified period"

Burns & Wellings *Real-Time Systems & Programming Languages*

Basically, systems that need to meet some **deadlines.** These can be classif ed as either hard or soft:

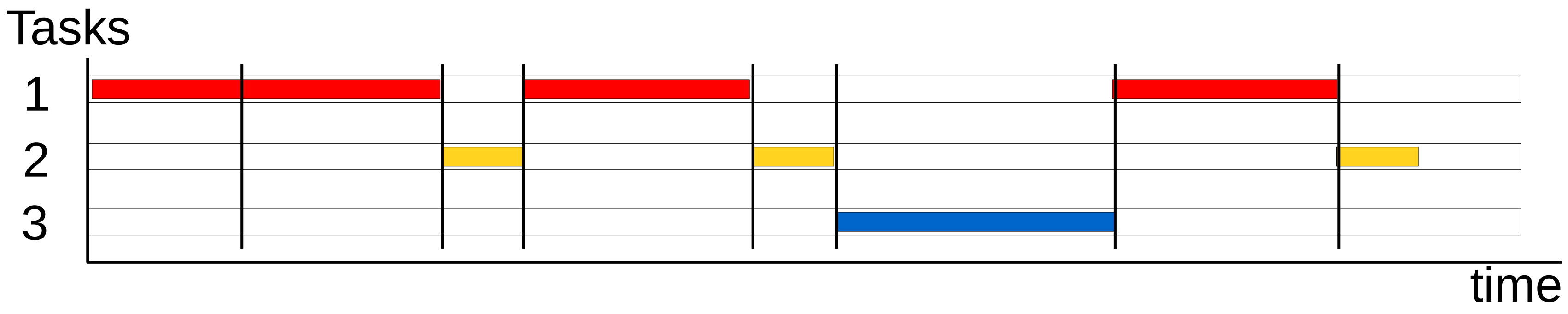
hard	these deadlines **must** be met otherwise the system will **fail**
soft	these deadlines **should** be met but the occasional miss can be tolerated

Some deadlines have both soft and hard real-time limits *(eg. response should be < 10ms but must be < 500ms)...*

Real-time issues

Real-time systems are often multitasking systems – these switch between processing tasks as required. Tasks also respond to external events (which notify the CPU through interrupts), or switch based on hardware timers or various software 'events'.

The hardware needs to be able to support the different tasks – this can have implications on hardware stack, memory management, and some advanced features such as hyperthreading support.



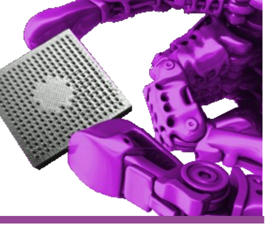
Interrupts

Interrupts are the main means by which external devices inform the CPU of a change in state. The alternative is to use device polling, where the CPU repeatedly and periodically interrogates each external device.

Polling wastes CPU bandwidth because it usually returns a negative result, and also increases the maximum response time of the CPU (which would be equal to the polling period). It also makes software more complicated.

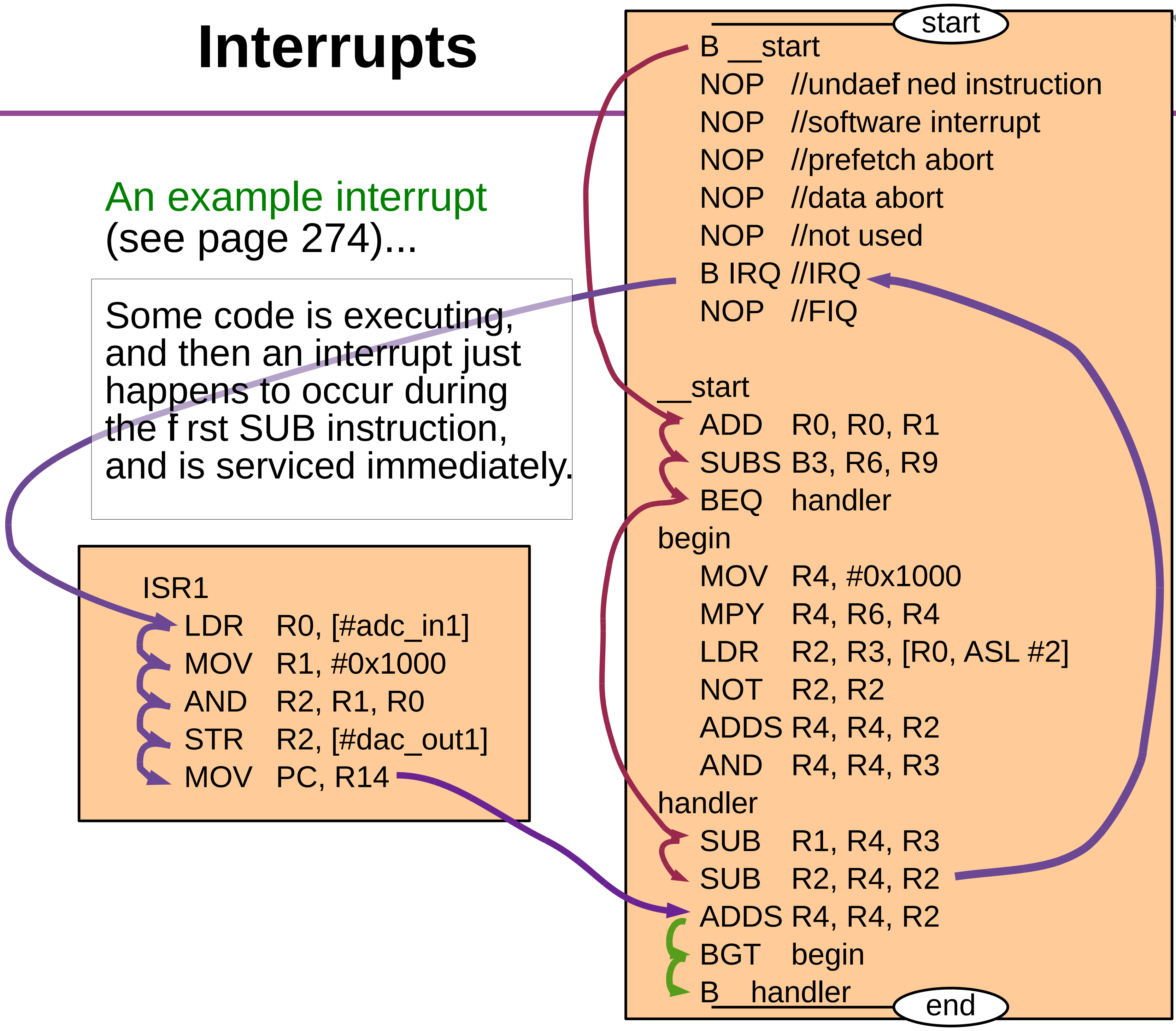
Interrupts are ideal for devices that require fast CPU intervention, or devices that change state only very infrequently. Devices that issue interrupts can reside anywhere, on any bus, serial or parallel, synchronous or asynchronous, because the interrupt is an <u>asynchronous</u> input.

Interrupts



An example interrupt
(see page 274)...

Some code is executing, and then an interrupt just happens to occur during the f rst SUB instruction, and is serviced immediately.



```
                start
        B __start
        NOP   //undaef ned instruction
        NOP   //software interrupt
        NOP   //prefetch abort
        NOP   //data abort
        NOP   //not used
        B IRQ //IRQ
        NOP   //FIQ

__start
        ADD   R0, R0, R1
        SUBS  B3, R6, R9
        BEQ   handler
begin
        MOV   R4, #0x1000
        MPY   R4, R6, R4
        LDR   R2, R3, [R0, ASL #2]
        NOT   R2, R2
        ADDS  R4, R4, R2
        AND   R4, R4, R3
handler
        SUB   R1, R4, R3
        SUB   R2, R4, R2
        ADDS  R4, R4, R2
        BGT   begin
        B     handler
                end
```

```
ISR1
    LDR   R0, [#adc_in1]
    MOV   R1, #0x1000
    AND   R2, R1, R0
    STR   R2, [#dac_out1]
    MOV   PC, R14
```

Interrupts

Although interrupts are asynchronous, most CPUs respond in a synchronous fashion by;

1. finishing the current instruction
2. saving the PC
3. setting the PC to equal a known memory address (a vector)

These locations in memory are known as interrupt vectors, and there will usually be one special address for every interrupt that can occur, such as the following for the ADSP2181;

0x0000	RESET, or startup from powerdown
0x0004	IRQ2 pin activated
0x0008	host write interrupt
0x000C	host read interrupt
0x0010	serial port 0 transmit interrupt
... and every 4 locations up to	
0x002C	powerdown condition

Interrupts

Since there are only 4 address locations reserved for each interrupt vector, this is not usually enough space for a complete interrupt service routine (ISR), so these locations normally contain a CALL to the appropriate ISR which is located elsewhere in memory.

Location 0x0000, the startup vector, normally contains a jump to main(), or to __start (a typical starting label in most assemblers).

Some of the vectors are caused by a signal on an interrupt pin (e.g. the IRQ2 pin going low), others by internal signals (e.g. the internal serial port transmit buffer emptying).

Some interrupts can be turned off or masked under software control and some can not: the latter are non-maskable interrupts (NMI).

Interrupts

Interrupts are usually prioritised, and on some CPUs, a higher priority interrupt can occur during the ISR of a lower priority interrupt (but usually on smaller CPUs all other interrupts are blocked during ISR execution).

The ADSP2181 interrupt priorities are (highest at the top);

RESET or startup from powerdown
powerdown
IRQ2 pin
host write
host read
serial port 0 transmit
serial port 0 receive
software interrupt 1
software interrupt 2
serial port 1 transmit, or IRQ1
serial port 1 receive, or IRQ0
timer interrupt

Interrupts

Interrupt timing: For fast peripherals, or real-time systems, it is often necessary to know how long it will take an ISR to respond to an interrupt condition. For this there is a *best case* and a *worst case*.

Worst case will probably be the occurrence of a lower-priority interrupt that first has to wait for a multi-cycle instruction to complete and then has to wait for a higher priority interrupt to occur.

Worked example:
A 20MHz CPU has a CISC instruction set: the slowest instruction is a DIVide that takes 100 cycles to complete. It has two interrupt pins called IRQ1 and IRQ2, with IRQ1 being highest priority. The ISR for interrupt 1 can take up to 30 cycles to complete. There is a shadow register set for interrupts. How long is *worst case* for the CPU to respond to IRQ2 (assume no overhead for interrupt vectors)?

Interrupts

Before any ISR can be called, the current instruction must finish.
In the worst case, the current instruction is the DIVide (just started).
So time for divide to finish is 50ns×100=5 μs.

In addition, the worst case would be that IRQ1 and IRQ2 occur together, so we would have to wait for this higher priority interrupt (IRQ1) to finish first. This would take 50ns×30=1.5 μs.

This gives a total response time of 6.5 μs. So the fastest peripheral that could be connected is one that interrupts less frequently that that, i.e. less than about 150kHz.

Interrupts

Some smaller CPUs and microcontrollers tend to share interrupt vectors between a number of different sources. In this case, the ISR first has to decide which interrupt actually occurred, by reading an interrupt status register, further lengthening the response time.

Interrupts can be edge-triggered or level-triggered (some CPUs such as the ADSP2181 allow both options).

Edge triggered signals are latched internally as soon as they occur. Level triggered signals usually have to be active for a number of clock cycles before they are latched internally.

This is a type of debounce on the interrupt input, which also means further delay before the interrupt is serviced.

Interrupts

Worked example 2: The ARM has 2 external interrupt sources; the standard interrupt (IRQ) and the fast interrupt (FIQ), with the FIQ having higher priority. The shadow register sets provide 6 usable shadow registers for the FIQ and only 1 for the IRQ (*assume we need to use 4 registers, and each register load to/from memory takes 2 cycles)*

The IRQ interrupt vector is midway in the interrupt vector table, whereas the FIQ vector is at the end (*this means that no jump is needed for FIQ from the vector table: the interrupt code can simply follow from this location*).

The longest instruction on the ARM7 is a multiple load of 16 registers from sequential memory locations, taking 20 clock cycles. It can take up to 3 cycles to latch an interrupt. Assume that two cycles are needed for every branch. There is one interrupt with higher priority than both FIQ and IRQ. Assume that this takes 25 cycles to complete.

Interrupts

We want to choose between FIQ and IRQ. How many cycles would each interrupt take (assuming there is only one external interrupt)?

IRQ worst case response time:

1. No. cycles taken to respond to interrupt:	
to latch external signal	3
to finish the slowest instruction	20
to first service higher priority interrupt	25
2. No. cycles need to reach start of ISR:	
to jump to the IRQ interrupt vector	2
to jump from there to the ISR	2
3. No. cycles at start of ISR before a response can be made:	
to context save 3 registers (3×2)	6
Total required before ISR can start:	**58**