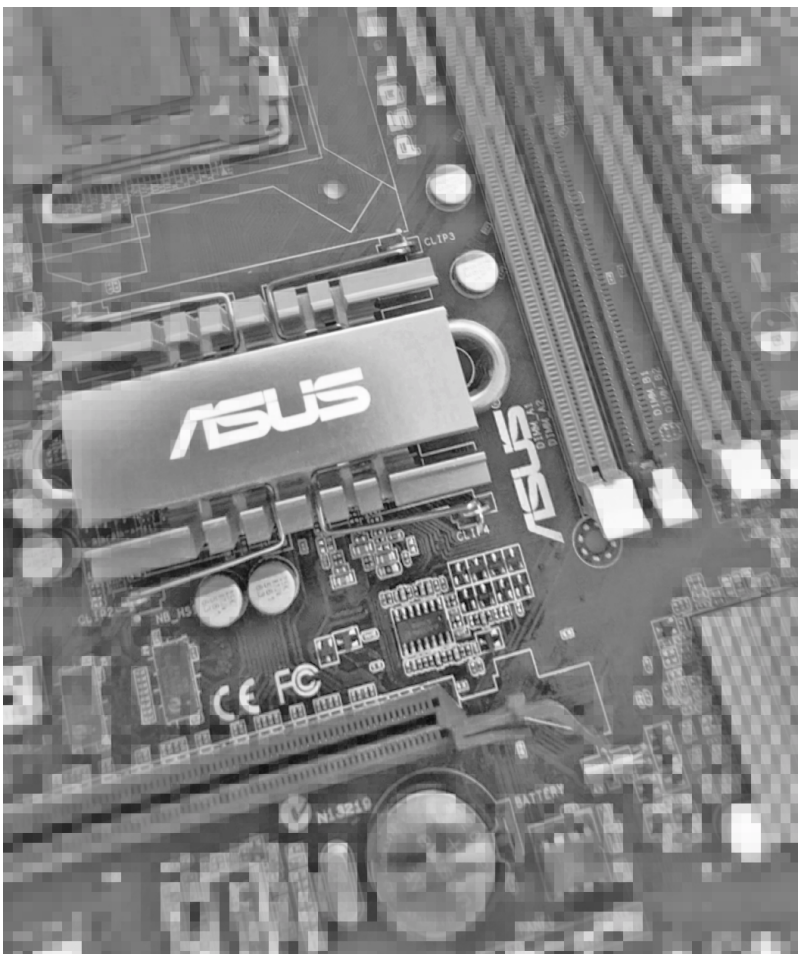
Building The System 6



Portion of an ASUS (AsusTek Computer Inc) Motherboard.

6.1 Putting it all together

We have already covered quite a large number of concepts in computer architecture, and introduced some key terminology. However, looking back on the past few chapters, we have taken a system level view of the way computer systems work. Block diagrams do not tell the whole story. On the other hand, this is not a digital electronics text in the engineering sense, and it is not aimed at advanced computer science or computer engineering students.

In order to understand how things connect, link, and build, into a complete computer system, we will now explore things in more detail but try to avoid too much electronics, and too much physics. The place to start with this task is perhaps most appropriately the **motherboard**.

In a modern computer system, certainly a general purpose one, there will likely be a self-contained circuit board: a **mainboard** or **motherboard** as it is sometimes known. This motherboard has all of the circuit connections, chip-sockets, busses, and miscellaneous additional chips[81] to allow a complete high performance computer system to be built, simply by plugging in and/or connecting the appropriate modules and chips to configure the system. Figure 6.1 shows a typical motherboard with some of the most important connections labelled.

[81] The term often used for various minor chips and components needed to tie things together is 'glue logic', in computer parlance.

These circuit boards are designed with some very demanding constraints, often relating to the way in which electronic signals behave in wires. On a circuit board, wires are formed by metal tracks built onto or into the material making up the board[82] . As we decided not to get too far into physics, it is enough to say that the length of these wires can cause high frequency signals to travel through them quite poorly. A very short wire will begin to lose signals only at very very high frequencies (let us say of the order of 10GHz for the sake of argument). As wires get longer, the frequencies that are **attenuated** get lower. Starting at perhaps 8GHz for extremely short wires, then 5GHz, and then when wires are just a bit longer than that we find that 3GHz or 4GHZ frequencies are affected. Now, this is where we start to encounter design issues, because many modern CPU's operate in this frequency range. So a processor running at 3.8GHz will find it very difficult to send signals at those speeds across a wire track on a motherboard, unless that wire is very short.

[82] These are known as printed circuit board tracks, and being metal, they have capacitive and resistive properties. When capacitance and resistance are combined, they create a signal attenuating effect.

But motherboards are actually quite big[83], so the location of the chips, the CPU, the memory DIMMs and SIMMS, and other devices must be carefully considered. Even so, it may prove impossible to make motherboard wires operate at frequencies of 5-10GHz.

[83] Early motherboards could easily be larger than 300 x 300mm, though in recent years these have shrunk significantly as far as desktop style PC's are concerned: A micro-ATX board may be about 240 x 240mm, and nano-ITX boards are about 120 x 120mm

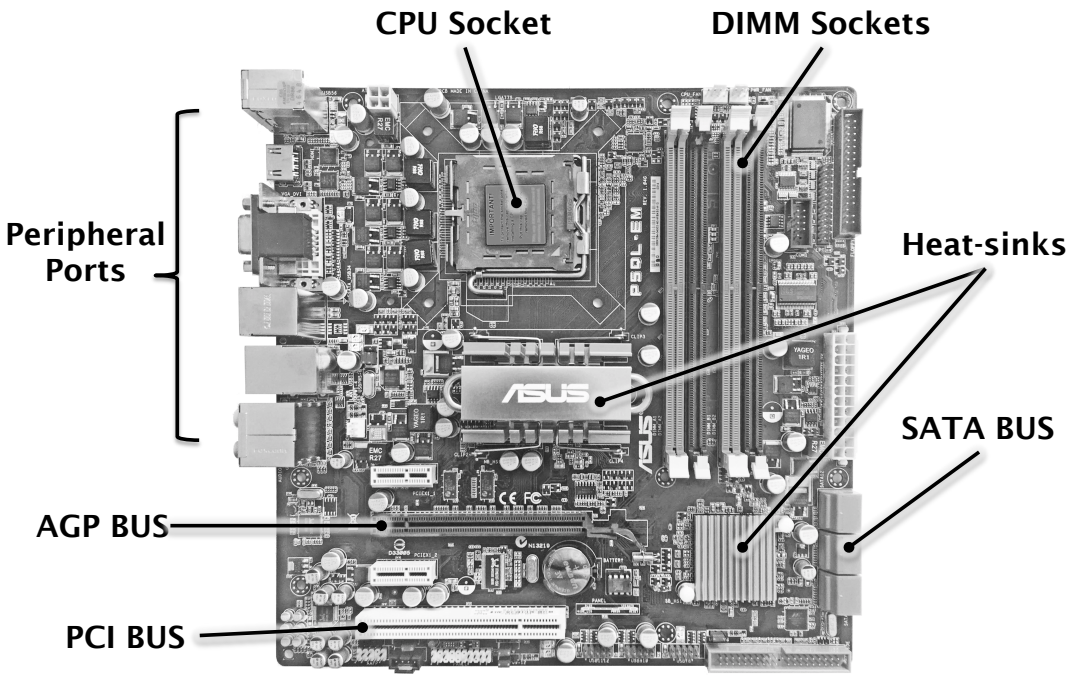


Figure 6.1: Motherboard Example. Showing peripheral ports (top left), DIMM memory sockets (top right), CPU socket (top centre), PCI BUS (bottom left), AGP BUS (centre left), SATA busses (bottom right), and examples of heat-sinks (centre right).

In order to overcome the limitations of circuit board design we have just mentioned, there is an answer of sorts. We can simply send signals at lower frequencies, and accept that this means data will be transferred more slowly. Whereas a system can certainly be designed to rely mainly upon a system bus, operating upon the same frequency as the processor, this would force the whole system to run at the lowest device speed in the system. Instead, many modern bus systems operate at their own local bus frequencies (that is, at a different speed to that of the processor), and many of these busses also operate according to individually selected protocols, known as **bus standards**. As a result of this fundamental design compromise, the idea of a bus has evolved far beyond a simple group of wires connecting parts together, with a single simple clock regime, as we first envisaged in our rudimentary system block diagram (Figure 2.1 of Chapter 2). We will now explore this idea further.

6.2 The system bus vs the dedicated bus

As we have already mentioned in earlier chapters, the system bus is, in a simplistic view, just a collection of signal wires capable of connecting various basic system components together and allowing them to exchange information. At the very least, we would expect CPU and memory to communicate via this bus, and also some form of Input/Output device interface to be visible on the same bus (the basic von Neumann Model). This system bus is also sometimes referred to as the **host bus** (where the CPU is the host), but also somewhat less intuitively the **local bus** and the **front-side bus**, a **memory bus**, or the **main bus**.

It is also useful to remind ourselves that the definition of a bus isn't just a group of wires connecting devices together, but connecting **multiple devices** together. A bus allows multiple devices to engage in data transfers. In the simplest mode, the data transfer is between any two devices connected to that bus, but there are scenarios where multiple devices can receive data from one or more other devices sending that data (known as a **broadcast or multicast**).

In both of these scenarios there is the concept of a **bus master** and a **bus slave** device (or devices). The master is in control of the data transfer, and decides when it begins[84] , what is transferred, and when to end the process. The start, middle, and end of the data transfer process may be referred to as a **bus transaction**.

[84] Although the master may initiate a transfer, it may sometimes be doing so in response to a request from a slave device. Therefore, it may control the transfer but not the whole chain of events.

Apart from the system bus, there may be **secondary busses** in our computer system. These can connect to the system bus via an interface chip (known as a **bus bridge** or **auxiliary bus controller**), or even connect directly to the CPU using a **dedicated bus** in its own right. In a very high performance system, this latter case has a number of advantages. We shall see examples later.

We might now envisage a block diagram of a system, evolved from our first computer architecture in Chapter 2, but advanced enough to represent the main features we have just discussed. One possible case is shown in Figure 6.2.

Figure 6.2 is only one possible combination of busses in a computer system or, in other words, one particular **bus hierarchy**. This is a term which reflects the fact that the bus architecture has a tiered structure, such that the CPU has a host bus or system bus at the top level, and there are auxiliary busses that are subordinate to that. We could have

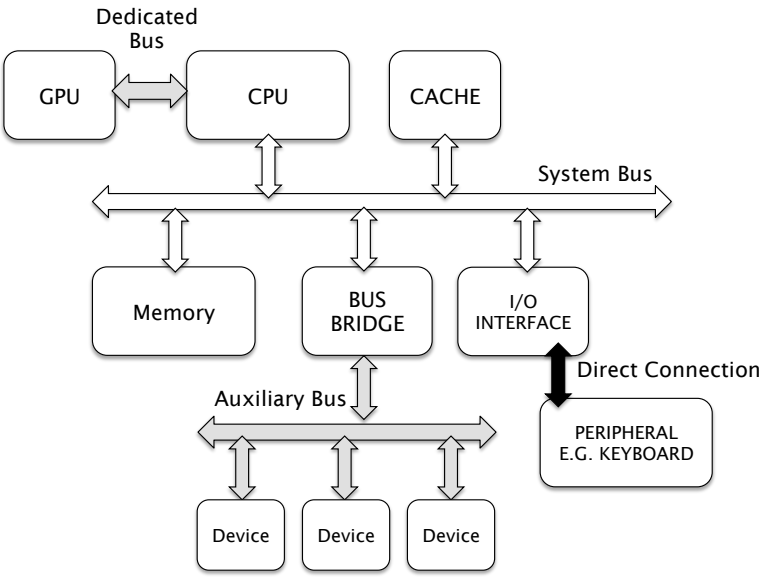


Figure 6.2: System diagram with more comprehensive bus architecture. System includes a bus bridge hosted on the main bus, extending an auxiliary bus to host three additional IO devices.

further levels of busses connected in even lower tiers if we wish to do so.

In this example we can see several important aspects of the system that need explanation:

First of all, the system bus permits several components to connect to the CPU, including a standard IO interface, which then connects to a keyboard by a direct connection in this example. If this direct connection can only connect **point to point**, then it is not a bus in the standard definition. We may well have a bus in which only two devices exist because that is all that we have added to the system, but that is not classically a direct point-to-point connection either.

Also connected to the system bus is a component we have already mentioned, the bus bridge. We may be more specific about this component if we know which kind of bus it supports: a **PCI bridge** for example, supports a **PCI bus**. The bus bridge creates a connection that the CPU can recognise on its host bus, but which also allows the CPU to communicate with devices on the other side of the bridge (those on the auxiliary bus). This can be the same type of bus, but is possible even if they use a different interface standard and bus protocol. A bus bridge is often just a particular chip on the motherboard. The advantage of this is that a CPU need not provide every possible bus connectivity that might be

selected by a designer; instead the system designer simply supplies the appropriate bridge component.

A further observation is that devices on the auxiliary bus can communicate with each other without necessarily involving the CPU, but on occasions can also communicate with the CPU via the bridge. Therefore, individual busses can potentially operate independently of each other. A bridge permits both compartmentalisation and cooperation.

The CPU in this example also has a dedicated bus connecting to the GPU component. This is a highly specialised bus connection, going directly between the GPU and the CPU (not via the system bus). This allows it to operate continuously without interfering with the system bus bandwidth, and without system bus transactions interfering with its own behaviour. In this case, the device connected is a GPU and dedicated busses are often used to permit high performance GPU communication to the CPU (among a number of other scenarios).

6.2.1 Concurrency in bus architecture

Looking at our example system, there are clearly multiple buses, as well as other kinds of connections. This results in some quite important advantages, and in particular the concept of **concurrency** becomes apparent.

It should be possible to see that in our example system, the system bus can operate any one transaction at a time between a set of possibilities including CPU and memory, cache and memory, CPU and the IO interface, and so on. However, whilst this is happening, the auxiliary bus can also be performing transactions between its local devices too.

In other words, the host bus and the auxiliary bus can run **concurrently**[85] , provided they do not want to cooperate in a data transfer between their two buses. Equally, whilst these two busses are operating concurrently, the dedicated bus can also be operating. These three busses may not interfere with each other at all during these operations and therefore will not hinder each other's performance.

[85] The term 'concurrently' means several entities operating (e.g. transferring data) at the same time.

With concurrency, the system has effectively been subdivided, or partitioned into more or less independent parts that can work simultaneously on different things. The only exception to this is if the CPU decided to communicate with an auxiliary device (or vice versa), via the bus bridge, in which case both busses would be involved in the transaction and neither bus could be used by any other devices at that time.

This is an important observation, because being able to do three things at once means more work done in less time. It can also be measured in another way:

Suppose the system bus has a data transfer bandwidth of 200 Megabytes/sec, the auxiliary bus has a data transfer bandwidth of 500 Megabytes/sec, and the dedicated bus has a data transfer bandwidth of 1 Gigabyte/sec.

Then we can determine that the system has an effective transfer bandwidth of 1.7 Gigabytes/sec. This is because we can potentially add all of the independent bus capacities together (in the best case scenario), giving us $1000 + 200 + 500 = 1700$ Megabytes/sec.

If we wished to boost performance further, we could add multiple bus bridges to the host bus and have other independent groups of auxiliary bus devices, each adding a further 500 Megabytes/sec of system bandwidth.

But of course, this optimistic evaluation requires all available busses to be used in near optimal fashion simultaneously (perhaps a rare occurrence). Therefore, when a system has heavy work to do in particular categories of activity, it can be made more efficient by using a concurrent bus hierarchy, provided that the right kind of busses and devices exist, and their behaviours are compatible with efficient **bus utilisation**.

6.3 Bus standards

The development of **Bus standards** has several important purposes. First of all, a well defined standard, maintained by a recognised organisation, provides precise electrical and functional operating principles for that bus, allowing any manufacturer to develop a device that is **bus-compatible** with that standard. This ensures technology is interchangeable, and where possible, open to competition[86].

[86] The vast majority of standards are defined to allow open interchangeability of products. However, a few standards are designed specifically to provide protected intellectual property, via patents for example, or simply for the purposes of branding. These are known as proprietary standards and can be tightly controlled by the owner.

There are a large number of standards, some of which are less frequently used, and considered semi-obsolete. We refer to these semi-obsolete standards as **legacy bus standards**, and computer systems either do not support them any more, or provide support primarily for convenience to allow older technologies to continue to be used for a period of time. A good example of this is the **VESA** bus standard, which was largely replaced by the **PCI** bus in the 1990s.

We cannot cover all bus standards in detail. However, covering a few will give us enough understanding of how busses work such that given

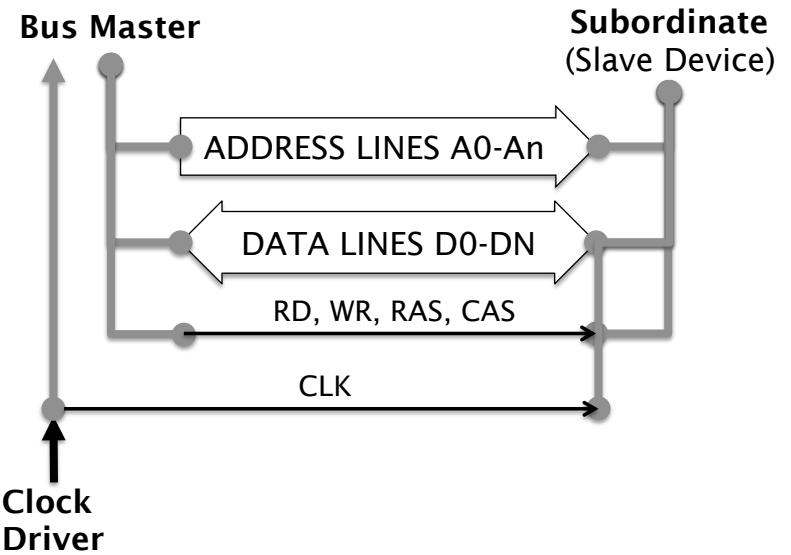


Figure 6.3: A simple hypothetical bus Example. Showing Address lines, data lines, and several control lines, plus the clock.

the documentation for another bus standard we have not met before, we should be able to grasp the basic concepts of the specification. Therefore, we will start with a generic case: the system bus as a point of comparison, and move on from that foundation.

6.4 A generic system bus

By generic, we mean in this case, a very standard example of a system bus system without any specialisations. The features of this example will be found in almost any system bus in a modern computer system. An illustration of this case is given in Figure 6.3.

This diagram represents the origin, destination, and the flow of information in the bus, which you may remember is simply a group of wires linking multiple devices together.

Looking at Figure 6.3, we can see that there is a special component, the **clock driver**, which we have not yet mentioned. This is the circuit that generates the regular on/off signal pulse that represents the **system clock**[87].

[87] The accuracy of a system clock is important, and the precision of the clock signal is guaranteed by the use of a crystal-oscillator component precisely shaped and trimmed to meet the specification that permits it to oscillate at a precise frequency.

All devices in the system use this same clock, even the master device. The clock signal acts as a synchronisation mechanism and determines

when each event begins and ends on the bus.

Next we have **address lines**. These are a group of wires that represent the unique numerical address of a memory location with which the master wishes to interact. In a system with 10 address lines, these would normally be named **A0** through to **A9**. The number of address lines is related to the addressable memory of the system.

Because addresses are binary, the relationship is determined by the base-2 number system, such that given a number of address lines, we can determine the address range and number of locations as follows:

Definition 6.4.1 ***Available address range***

Where a = number of address lines:
then there are
$$2^a \text{ addressable locations}$$
For example, if a = 10, then there are $2^{10} = 1024$ ***locations***.

We can easily see that to address 1 Megabyte of memory, where each byte is in its own location, then we have to have 20 address lines. Computer systems with large address spaces can easily have 32 address lines, and billions of addressable locations.

Similarly, the data lines, **D0-Dn**, represent the size of data values that can be transferred across the bus. This might be a byte, in which case we would expect to see data lines **D0** to **D7** being represented. However, many computer systems have 16, 32, or perhaps even 64 data lines.

We can see already that a computer system with 32 address lines and 64 data lines has accumulated a requirement for 96 bus signal lines, and these are implemented as metal tracks on a circuit board. This reinforces what we said earlier, relating to the difficulties of designing motherboards to run at high speeds. There are alternatives that can reduce this problem however.

There are a few additional signals present in our diagram. **RAS** and **CAS** relate to **Row-Address-Select**, and **Column-Address-Select** signals. These are required by memory chips to allow them to be given the row and column information required to access individual memory locations. Some processors generate these signals directly, whilst others make use of an additional chip called a **memory controller**.

Because memory can be addressed in rows and columns, we can potentially reduce the number of address lines. If, for example, memory is

arranged into a 16-bit row, and a 16-bit column address scheme, then we can send the two parts one after another (i.e. row then column) and then the address lines require only 16 lines in total. This is known as **multiplexed addressing**, because we multiplex (i.e. switch) between row and column address content on those lines. Of course this also means that it now takes two clock cycles to send a complete address, instead of only one cycle without multiplexing, so this reduction in address lines does not come for free.

Two further important signals are also needed: **RD** (**read**) and **WR** (**write**). These signals are used to control which direction the data is transferred: if the master sends data to the slave it is a **write transaction** and the WR signal is enabled. On the other hand, if the master receives data from the slave, it is performing a **read transaction** and RD is enabled instead. So RD and WR indicate if the slave device is being read from or written to.

It follows that any device wishing to act as a bus master must be capable of generating all of the required signals on the system bus (except for the clock). Devices which cannot do this can only operate as slave devices (most memory chips fall into this category as they never act as bus master).

6.5 PCI: a very successful Bus Standard

We will now look at a more complex bus standard, an example of an auxiliary bus architecture, and one in very common use in modern computer systems: The **PCI** Bus Standard. Some of the key features of the original PCI bus standard are as follows:

- 33 MHz Bus Frequency,
- 32 bit Data Width,
- 32 bit Address Range,
- Bus Arbitration requires 5 clock cycles,
- Bus Turnaround requires 1 clock cycle,

The PCI bus employs bus multiplexing, but not in the same way as we encountered earlier for DRAM addressing. Instead of subdividing the address into two parts, the PCI bus uses a single bus to transfer address and data values. This is known as the **address-data bus**. This way, 32 address lines and 32 data lines can actually be the same wires, just at different times. Due to the way the PCI bus works, address and data do

not need to be transmitted at the same time. This is therefore another form of multiplexing (data/address multiplexing).

We now have quite a lot of information, and thus might attempt to estimate some kind of initial performance measures for this bus standard:

Bus bandwidth estimate

Suppose that every single clock cycle allows a data transfer, then we transfer :

33 million x 32 bits, or 33 million x 4 bytes

$=$ **132 Million bytes/sec**

So bus bandwidth appears to be about **126 Megabytes/sec**.

This is the correct answer in the absence of any other information, but is also a somewhat simplistic assumption. What this calculation tells us is the **raw bandwidth** of the bus, but not the whole picture. Remember the specification: there are 5 clock cycles for arbitration, and once clock cycle for turnaround. These are **protocol** requirements of the functionality of the PCI bus that must be respected.

Bus arbitration is a process whereby a controller (in this case a PCI controller chip), decides which one among the devices in the system is allowed to be **granted ownership** of the bus at the next opportunity. It then spends some clock cycles setting up the data transfer - a total of 5 clock cycles.

Once a device has finished with the bus, it must relinquish or **release ownership** of the bus, so that another device can use it. This requires 1 clock cycle under normal conditions (the **turnaround cycle**).

So we see that actually, to transfer one 32-bit data word on the bus requires seven clock cycles (five for arbitration, one for the actual data, and finally one for turnaround).

This **protocol overhead** impacts performance significantly. If it takes seven clock cycles to transmit one data word, then our 132 Million bytes/sec is never going to be achieved, Indeed, we can only achieve 1/7th of this peak bandwidth, resulting in data transfer rates of around 18 to 19 Million bytes/sec, a very poor substitute for our original expectations.

In order to avoid the worst impact of bus overheads, the data transfers we perform on busses can often be performed as block transfers, similar

Transaction Length	1	4	8	16	32	64
Clock cycles	7	10	14	22	38	70
Efficiency	14%	40%	57%	73%	84%	91%
Transfer rate MB/s	18	50	72	92	106	115

Figure 6.4: Data transfer efficiency on PCI bus with various block sizes. It can be observed that as transaction length is increased, transfer efficiency increases too.

to the burst-mode of the DRAM we looked at in Chapter 5. But for bus transactions, these blocks are often much longer.

As an example of how this improves performance, if we transfer 10 data words in a single transaction, rather than one, then the cost is 16 cycles (5 arbitration, 10 data, 1 turnaround). This equates to a data transfer efficiency of around 62% (10/16), and a data transfer rate of over 78 Megabytes/sec (62% x 126 Megabytes/sec).

Interestingly, if we calculate the efficiency of the bus at a range of different data transfer block sizes, we see a trend emerge, as illustrated in Table 6.4.

From Table 6.4 we can see that as the number of words transferred per transaction increases, the efficiency of the data transfer also increases. This is because we only ever have have one arbitration and turnaround cost but many data transfers. So it would seem that the answer to our problems is to increase data transfer block lengths to large numbers, and enjoy data transfer rates close to the maximum possible. This is indeed part of the solution. However, there are problems with this idea:

- We do not always want to transfer data in large blocks, so there will always be cases where short and low efficiency transactions will occur.
- The longer the block size, the longer other devices have to wait until their next turn. This impacts upon fairness, and ability to respond quickly to events.

In conclusion, we can see that with this very shallow view of the PCI interface, there are limitations on performance due to the bus protocol. However, all bus protocols have some kind of requirements and overheads, so there is nothing particularly inferior about PCI; indeed, it is one of the most widely used bus standards for internal module connectivity in general purpose computer systems.

6.6 Data rate matching and buffers

One aspect that might be apparent from studying the concept of data transfer on a bus, is that the capabilities of two devices may differ, even though they may share the same bus connection. This can also happen at a bridge, where a device on a fast bus is trying to send a block of data to a device on a slower bus on the other side of the bridge. In this situation we consider the idea of a **producer** and a **consumer**, which are in effect the sender and the receiver during a bus transaction, or succession of transactions between the two entities.

When a producer creates data faster than a receiver can consume that data, the receiver must **buffer** that data locally, until it can work through the backlog. This is something like an in-tray piling up in an office. As long as the buffer never gets completely full, the producer can continue to send data as fast as the bus will permit.

Meanwhile, if a receiver consumes data faster than it is being sent, then the receiver may spend time idle when it would be better doing useful work. This producer-consumer dependency reduces efficiency if the two entries are not well matched. We will see later that operating systems attempt to deal with programs wasting time waiting for an IO event, by using various resource management techniques.

Meanwhile, in the case of a bus bridge, incoming ports of the bridge can have local buffering, to allow block transfers to complete on the producer side of the bridge, and then for that data to be forwarded to the consumer bus device at a slower rate a little later. This means that the producer does not have to stall its data transfers.

6.7 USB: Universal Serial Bus

We will now look at an example of another kind of bus technology, with different applications.

It is more than likely that you have heard of **USB**, or more specifically the **Universal Serial Bus** (USB). Whereas PCI is ideal for **internal** devices such as modules plugged into a computer system motherboard, the USB system is optimised for convenient **external** peripheral connectivity (in other words, USB is a **peripheral bus standard**).

These days you will find that, most often, peripherals use a USB connector, or at least provide the option, and keyboard, mouse, printer,

auxiliary storage, flash memory sticks, and many other devices will use this standard by default as the interface of choice.

One of the reasons why USB has become so widely used is the capability to provide power to devices, albeit relatively small amounts[88], to the devices plugged into it. This is very useful for devices such as a computer mouse, touchpad, etc, which have low power consumption, but also allows other devices to eliminate the need for a separate power cable. Consequently, USB has allowed many power supplies and wires to be eliminated from the computer desk: undoubtedly a key advantage given the menagerie of external peripherals we find plugged into our modern computer systems. We will examine USB further, assuming for now that we are referring to the original USB 1.0 specification (there have been several revisions since this was first launched in 1996).

[88] This is certainly true for the original USB standards, where power in the range of a few hundred milli-amps is usual. However, there are many variations, and some USB standards support up to 3 Amps, which is quite a considerable amount of power.

USB1.0 has some key features that are worth highlighting:

USB1.0 Key Features

- The USB bus can supply limited power to a connected device. Data is transmitted in packets, along with a header, and an error check code.
- Data rates of up to 12 Mbits/sec are possible for USB 1.0.
- Packets can contain anywhere from 0 to 8192 **payload** data bits[89].

[89] Sending zero bytes may seem to defeat the purpose of having a packet to send data. Zero length packets are however used for certain special purposes within the USB system.

As we might expect, having investigated the limitations of the PCI bus, the true data rate possible for a USB bus is limited by the need to perform its specific protocol related signal activities, as well as to moderate the length of data transfers. There are particular points to note:

- Every USB transmission begins with a SYNC signal sequence to indicate the start of a new packet. This a fairly short pattern (roughly equivalent to about 8 bits of data), but it uses up bus transaction time for every packet sent.
- Every packet also includes a packet identifier (PID), which is also equivalent to 8 bits.
- Every packet also ends with an end of packet (EOP) bit sequence, though this is only equivalent to three bits.
- Every packet also includes an error check (a 16 bit CRC code).

We can see from this that every packet contains the equivalent of at least 35 bits before we even send a single data bit. This is the overhead due to the bus protocol.

There is a further complication with USB protocol, and this relates to an idea known as **bit-stuffing**. Because of the way USB operates, it has no bus clock. Instead, the data bits themselves allow a clock to be recovered, using a special circuit which detects the zero-one and one-zero transitions in the bit stream being received.

However, we have no control over what data the application is sending; as far as USB is concerned it is random. If too many consecutive bits are the same (all zeros or all ones), then the clock recovery circuit will eventually lose synchronisation. To avoid this, the USB hardware has to insert an extra bit to create an alternate transition when this situation is approached. It does this after observing 6 consecutive unchanged bits in the data sequence. Consequently, a data transfer may take up to 1/6th longer than expected, based purely on the number of bits inserted into the payload, in a worst-case[90].

[90] How often will this happen? Bit-stuffing events will happen fairly often, but it is rare for an entire packet to require continuous bit-stuffing. Therefore, the 1/6th figure is the worst possible case, not a typical one. Nonetheless, it is worst-cases that dictate minimum service guarantees in system design.

One of the consequences of bit-stuffing is that, even if every packet transmitted has the same payload length, the length of the stuffed packet, and it's transmission time, can vary, or **'jitter'**, by a certain degree[91]. This can impact upon the predictability of system behaviour.

A USB 1.0 packet can contain anywhere from zero to 1024 bytes (0 to 8192 bits). If we assume the least data we want to send is 1 byte, or 8 bits, then it could take up to 43 bit periods to transmit 8 data bits due to the overhead (a data transfer efficiency of only 18%). On the other hand, if we use the longest packet size, 8192 bits can be transmitted in 8227 bit periods, an efficiency of 99.5%.

[91] Jitter is conceptually a variation in some characteristic, and has many causes. This particular effect might be referred to as jitter in packet transmission time, or more loosely as **payload jitter** or **packet jitter**.

So, as in the case of PCI bus, it seems that the bus is more efficient if we send larger blocks, but the same negatives also apply: we don't always want to send large blocks of data in one operation, and the longer the data packets are, the longer other devices have to wait for their turn at using the USB[92] .

[92] Achieving good responsiveness and high efficiency is a balance, and this is often referred to as Quality Of Service (QOS), especially when it becomes apparent at a user level (e.g. jerky versus smooth video playback).

The data rate of USB 1.0 is rather slow, certainly by modern standards. Fortunately, USB 2.0 and USB 3.0 have been defined and released as updated bus standards, and also incorporate new features. USB 2.0 has a raw data rate of up to 480 Mbits/sec, and USB 3.1 operates at up to 10 Gigabits/sec.

When plugging a USB1 device into a USB2 bus system, things still work. This is called **legacy compatibility**. This is because USB releases are backward compatible with older bus versions, but they are then forced to run at the slower speed of the older bus. This is another example of legacy bus support, as mentioned earlier.

6.7.1 Plug-and-Play

Another important aspect of USB, is the concept of **plug-and-play**. It would not be a good idea to open up your desktop PC and start unplugging modules from the PCI bus with the power switched on. At the very least your system will probably crash, you may corrupt data, but at worst you may damage parts of your computer. These modules are not designed to be removed and replaced during system operation. On the other hand, a system where this is allowed is known as a **hot-swappable** system architecture, and sometimes as a **plug and play** system.

The USB standard is designed to facilitate fast and convenient connection and disconnection of peripherals, and sometimes these are plugged in and unplugged whilst the system is running. Hence the plug-and-play label. This does however require hardware and operating system functionality working together to achieve seamless operation. Nonetheless it allows the user to plug in a flash drive, unplug a printer, plug in a network adapter, unplug the memory stick, and so on, and all with the system happily carrying on with no problems at all. This, plus the ability to send power to peripherals over the same connection, is what has made USB so successful.

6.7.2 Hubs and ports

On a practical aside, each USB connector on the computer is known as a USB port. Typically, a general purpose computer will have 4 ports, though it depends (laptops often have two), some desktops may be configured to have more. However, any USB port can connect to a USB hub, rather than a peripheral, and this provides expansion capability.

For example, one USB port might plug into a 4-way USB hub, providing 4 new ports. Hubs can be cascaded, up to the point where the total number of devices in the system is a maximum of 127 devices. This is far more than the vast majority of users will need.

6.8 More common bus standards

A few other common bus and interconnect standards are of interest to computer systems design. Some of these have arisen from particular manufacturer initiatives, and others are widely used standards in particular applications. Table 6.5 shows a few of these.

Bus	Max Data Rates	Typical Uses
Firewire400	49 MBytes/sec	Firewire is a brand-named version of IEEE-1394 serial bus standard. Permits up to 63 devices, plug-and-play, used for video, storage, and high bandwidth peripherals.
Thunderbolt	5 GBytes/Sec (Thunderbolt 4)	Branded bus standard, up to 6 devices per connector port, high speed devices.
IDE/PATA	100 MBytes/sec	Integrated Drive Electronics, an early Disk storage interface bus.
SATA	600 MBytes/sec	Serial 'AT' Attachment, frequently used for disk interfacing in modern general purpose systems.
SCSI	2400 MBytes/sec	Small Computer System Interface, primarily for high performance and high-resilience disk systems (in conjunction with **RAID** configurations).

Figure 6.5: Some additional commonly used bus standards.

Thunderbolt in particular has become very widespread due to the prevalence of certain touchpads and smartphones, whilst **Firewire** is also widely used in APPLE products such as laptops and desktop machines. For disk storage systems, there are a whole class of bus standards, including **SCSI**, **IDE**, **SATA** (but also Firewire to some extent, particularly for external drives), and these provide various kinds of high speed data channels for disk storage units.

Often these disk storage interfaces use a concept known as **daisy-chaining**, whereby a series of individual connectors loop from one device to the next, forming a bus in the process. This reduces the need for multiple wires from devices to the central device. We will explore disk storage in more detail in the next chapter so we will not say more about this now.

6.9 Industrial and embedded standards

Whilst general purpose bus systems are designed primarily for high data rates, and convenience, there are other application domains where bus systems are used, and where requirements are quite different. A good example of this is in the **embedded systems**, and **industrial computing** domains.

In industrial settings, computer systems have to have resilience, fault-tolerance, immunity to electrical noise caused by heavy machinery, and

operate on power levels that are unnecessary or undesirable in office or domestic scenarios. As in the previous section, there are just too many systems to cover them all in detail. However, the following subsections will give an insight with a few particular examples.

6.9.1 I2C (Inter-Integrated Chip) bus standard.

A typical application example of the **I2C bus** system is shown in Figure 6.6 where our familiar simple computer system has been augmented by the addition of an I2C interface, typically a dedicated I2C interface chip, and a number of devices connected to the I2C bus which that chip supports. In some systems the I2C interfacing might be built directly into the CPU chip. In this case, the I2C interface would still be observed as a device with the same kind of functionality, but by incorporating it with the processor core it reduces the number of chips required to build a system.

In the example, the I2C bus has multiple devices connected, an LCD display driver chip, an EEPROM for non-volatile data storage, a temperature sensor, and a keypad interface chip. This system could easily be the starting point for a central heating controller. It has all of the requirements, except for a way to communicate with the heating boiler, but this could also be accomplished by an I2C link if desired. Note that only two wires are required for an I2C bus: a very small hardware cost, making system assembly easier, and also meaning that the numerous interfaced devices need only two pins to connect to a system, allowing their electronics to be kept modest.

The I2C system uses a very straightforward protocol, the **SCL** (**Serial Clock Line**) provides synchronisation, and is always generated by the bus master, so that the devices on the bus which are receiving data will know when data bits are present. The **SDA** line (**Serial Data**) provides the means to transmit data bits, one bit per clock pulse.

Any device on the I2C bus can be master; however, in the example given, only the CPU is capable of doing so. There is no overall controller for the bus. Any device may attempt to become master, and as a result, several devices might try to do this at the same time. However, the protocol is able to deal with this using a conflict resolution algorithm. The following is an example of how the system ensures only one master is active at a time:

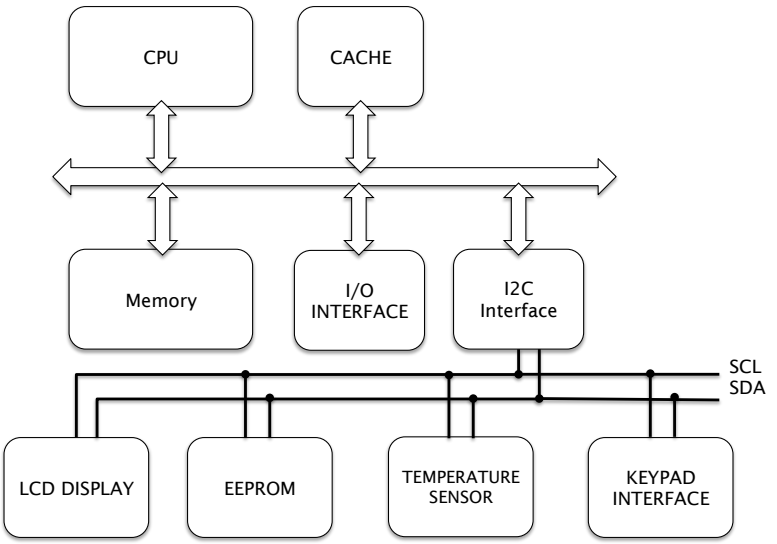


Figure 6.6: I2C bus added to our standard von Neumann architecture. Showing a system scenario in which an LCD display, a keypad, a temperature sensor, and a non-volatile memory are linked to the I2C bus.

Bus Master resolution protocol

- Potential Master waits if SDA appears to be active already (it cannot be master if another device is already busy).
- When SDA is inactive, the master attempts to start a transmission.
- The would-be master monitors the state of SDA, and if it is not as expected, then this means another device is attempting to be master too. The device then backs off and stops trying to be master.
- Otherwise, if no conflict is observed, the would-be device assumes master status, and proceeds.

This is an example of a **back-off-and-retry** arbitration mechanism. These are often used where there is no central controller for a bus system. In most situations, even if two or more masters attempt to gain possession of the bus, only one will succeed. This is based on the assumption that two masters will not communicate with the same slave device simultaneously (and by simultaneously, we mean exactly at the same time down to the first bit of transmission happening in the same clock cycle for both masters). In the rare case where this actually happens, problems can occur. Therefore, I2C systems are normally designed such that two masters never access the same slave, or never without some form of mutual agreement[93].

[93] The concept of a semaphore as a software gate-keeper between two competing entities is one potential solution; another is some form of token exchange, or a third option is a time-division protocol.

The full specification of I2C transmission capabilities is beyond the scope of this text, but it is well defined in additional reading sources. There are also variations of this standard such as INTEL's **SMBus**, which implement subsets of the full I2C standard, and have become popular in certain systems applications.

6.9.2 CAN bus

CAN Bus (**Controller Area Network**) was originally designed for the automotive industry, to allow many electronic devices distributed around the chassis and bodywork of a vehicle to link together via a two-wire interface. The alternative was for every device to have a bundle of wires leading back to the central control module of the vehicle, and these 'wiring looms' as they are referred to, were complex and difficult to maintain. Having a single two-wire bus that can run around the vehicle from one device to another, is a much more streamlined solution in terms of wiring.

An important feature of CAN bus data transfer is the idea of **differential signalling**. Rather than having a single wire transmitting zeros and ones, the CAN bus system uses two wires, transmitting mirror image data patterns, such that if the first line transmits 0-1-0 as zeros and positive voltages, the second line would transmit zeros and negative voltages.

This duplication may seem like a waste of effort, but when the problems of electrical noise and interference are added to an ideal signal, the result is degraded reliability. However, the differential (mirror image) signal levels allow this noise to be cancelled out, thus making the bus very resilient to noisy situations that occur in factories, near heavy machinery, and of course in automotive products. In industrial situations, this can be an essential requirement.

The concept of differential signalling is shown in Figure 6.7 where the top case shows a standard one-wire signal level, affected by noise. This can cause data bits to be misread in severe cases: something that could be catastrophic in an industrial control system for example. The second panel shows differential signalling, where **D+** represents the normal signal, including noise, and **D-** represents the inverted signal, but with the same noise (not the inverse noise)[94] . When these two signals are combined in the right way, the difference between them preserves the binary transitions, but cancels out almost all of the noise. This allows systems to work very reliably in electrically noisy environments.

[94] Because this noise is coming from an external source and affects both lines in virtually the same way.

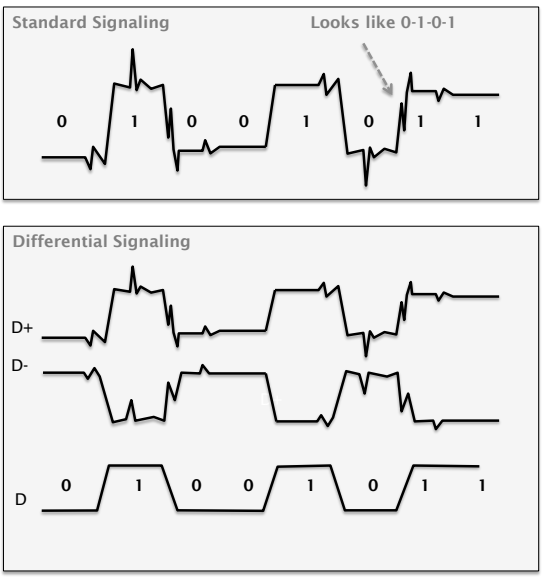


Figure 6.7: Differential signalling concept. Showing standard signal (top panel), and differential signalling (bottom panel).

6.10 Rack-mount, hot-swap, and servers

Whilst a typical desktop or laptop computer has a motherboard, and is built primarily for one user, the concept of a mainframe system has persisted into modern computing practice, if somewhat reborn in new guise.

The server platform is the modern equivalent of a mainframe, providing computing resources to many users as if simultaneously. To achieve this, servers are built with particular specifications, often based on racks full of motherboards (a **rack-mount** system), each with processors housing multiple processing cores, large disk arrays, and large amounts of memory. Single server racks could quite easily support several hundred processor cores. In some systems the compute resources are racked in one section and disk storage arrayed together in another section. This provides a more convenient way to organise and maintain modules.

Sometimes these systems are designed to be fault tolerant, and with automated backup systems, to ensure a users' data can be recovered in the event of a major system breakdown, and can even switch in a spare processor card to service a customer's program load in the event of a particular CPU failing. This often takes only a matter of seconds, so that services appear to be unaffected.

Such systems also rely upon the ability for engineers to connect and disconnect modules such as hard-disk units which may have failed, without having to turn off the computer system. This is known as **hot-swapping**, and is one of the design requirements of many server systems. As we have noted, some bus systems, such as USB and SATA permit this, whilst other bus systems do not.

6.11 IO device mapping and IO servicing

In Section 5.8 we learned about the idea of memory maps as representations of how devices, primarily memory modules, are mapped into the physical address range of the computer system. We also noted that a special case of memory mapping is the idea of an IO device appearing to be part of the address space.

Consequently, any suitable IO device or interface chip can appear as a group of memory addresses. This is known as **device mapping**. The CPU, or indeed any other device connected to the bus, can see these IO addresses, and potentially read from or write to them.

An important concept in **IO servicing** (the process of ensuring IO devices get the information they need when they need it), is the idea that a device can gain a suitable response from the system when a particular IO event occurs.

In terms of IO devices notifying the CPU that something of interest has occurred, there are two options. **Polling** requires the CPU to regularly read the status register of the IO device (via its IO address space), and check its status. **Interrupts** provide an alternative where the CPU never checks, but instead receives a special trigger signal (an interrupt) when an event arises. Using interrupts means that the CPU does not waste time checking lots of devices frequently.

Once a device has gained the attention of the CPU, the CPU will typically then control a data transfer, from the device to an area of memory, or vice-versa. This is known as a **CPU driven data transfer**. Again, this involves a lot of CPU effort, but this can be avoided if a device uses a technique known as **DMA**.

Direct memory access (**DMA**) allows an IO device to take over control of the system bus (become bus master) and transfer data to the chosen destination without CPU involvement. This could conceivably be

a device-to-memory transfer, a memory-to-device transfer, or a device-to-device transfer.

DMA is typically highly efficient as it often uses large block transfers, and often the CPU will continue to run its programs simultaneously, by relying upon its own on-chip cache rather than memory accessed via the system bus. Some interfaceable devices and chipsets support highly automated DMA. For example, an audio chipset may be able to fetch blocks of data from memory continuously by way of repeated DMA block transfers, without CPU involvement. This allows capabilities such as audio streaming to be smoothly executed without placing demand on the CPU.

One aspect that must be considered carefully with DMA is the impact this has on other devices in the system. If multiple devices want to perform DMA, then there needs to be some care taken to ensure that devices that need to maintain a minimum data-rate (such as audio playback) are getting enough bus bandwidth to fulfil their service requirements, even though other devices may want to perform DMA at the same time.

Therefore, there is a trade-off between having very long DMA transfers, causing other devices to wait a long time to get their turn at using the bus (service latency), versus much shorter transfers that allow all devices to get frequent use of the bus in turns, but with the penalty of lower data transfer efficiency.

6.12 Summary

In this chapter we have explored the nature of computer systems from the point of view of connecting components into working cooperative systems. We have explored the basic ideas of a motherboard, and the mechanisms that such a circuit board embodies as a hierarchical system of busses and ports.

The role of bus standards, providing clear definitions of bus protocols and operating principles helps us to understand the exact requirements for a bus performing a data transfer transaction, and the cost of having such specifications. We discovered that this protocol overhead has a performance impact, and that this can be estimated and evaluated, whilst the configurations and choices relating to issues such as data payload size will impact upon both the efficiency of the bus system and the latency - that is, the time delay between successive bus transactions, and the subsequent effect this has on user service levels.

Ultimately, with suitable design choices, we can design systems that utilise busses as mechanisms to divide and conquer data bandwidth and deliver superior performance.

6.13 Terminology introduced in this chapter

Address line	Address-data bus
Addressable locations	Attenuation
Auxiliary bus	Available address range
Back-off-and-retry	Bit-stuffing
Broadcast	Bus arbitration
Bus bridge	Bus hierachy
Bus master	Bus protocol
Bus slave	Bus standard
Bus transaction	Bus utilisation
CAN Bus	CAS
Clock driver	Column-Address-Select
Concurrency	CPU socket
Daisy-chaining	Data line
Dedicated bus	Device mapping
Differential signalling	DIMM socket
Direct Memory Access	DMA
Embedded system	Firewire
Front-side-bus	Host bus
I2C bus	IDE
Interrupt	IO servicing
Legacy bus standard	Local bus
Main bus	Mainboard
Memory bus	Memory controller
Motherboard	Multicast
Multiplexed addressing	PCI bridge
PCI bus	Plug-and-play
Point-to-point	Polling
Printed circuit board	Peripheral bus standard
Producer-consumer	Rack-mount
RAS	Raw bandwidth
Read-transaction	Row-Address-Select
SATA	SCL
SCSI	SDA
Secondary bus	Serial clock line (I2C)
Serial Data line (I2C)	Turnaround cycle
Universal Serial Bus	USB
Write-transaction	

These terms are defined in the glossary, Appendix A.1.