Faculty of Engineering, Computing & Creative Industries

Edinburgh Napier UNIVERSITY

Module : Systems and Services

Module Number : CSN08101

Session: 2014-15

CONTENTS

Unit 1 : Introduction to hardware

Introduction to computer hardware & the fetch execute cycle.

UNIT 1: INTRODUCTION TO HARDWARE

LEARNING OUTCOMES

On completion of this unit you should understand :

- The main buses (address, data & control) found in microprocessor systems
- The main constituents of a simple microprocessor core, such as the 8051
- The Fetch-Decode-Execute cycle
- The nature of Assembly language and machine code instructions
- Basic number systems: binary and hexadecimal

USEFUL URLS

Processor manufacturers:

http://www.intel.com/
http://www.amd.com/

PC manufacturer/Distributors:

http://www.dell.com/uk/
http://www.viglen.co.uk/
http://www.pcworld.co.uk/

Guides:

http://www.tomshardware.com/
http://arstechnica.com/

BOOKLIST

Paper books do tend to date quickly. You are probably better using online resources. Wikipedia sometimes gets some bad press in academic circles, but is a pretty good starting point. Here are some traditional books if you are interested:

'Structured computer organization', Tanenbaum, 0-13-020435-8.
'Computer organization and architecture: designing for performance', Stallings, 9th Ed, 0-13-081294-3.
'Computer architecture, design and performance', Wilkinson, 0-13-518200-X.

TUTORIAL1: REVISION OF NUMBER SYSTEMS AND HARDWARE BASICS

LEARNING OUTCOME

This module assumes that you have already studied an introductory computer systems module, such as CSN07101 or equivalent. This first tutorial is designed to ensure that you have understood some of these basic concepts, which are briefly revised in this unit. Note that there is section on number systems at the end of this unit.

1. Convert the following binary numbers to decimal and to hexadecimal:

   10010111 11011010 01010111

2. Convert the following decimal numbers to binary, then to hexadecimal:

   58 111 217

3. (optional) Convert 99 to binary, then find out how -99 is represented as an 8 bit binary number by finding the two's complement of the binary number.

4. (optional) Using the answers from the previous questions, show what the result of subtracting 99 from 111 is in binary.

5. Describe the function of the three main buses in a microprocessor system.

6. If the system clock frequency is 2.2 GHz, how long is one clock tick?

7. Some early versions of the PC used 24 bit addresses. How many address locations could be referred to?

8. The majority of software is stored on the hard disk. What must happen to it before the processor can execute it?

9. Sketch a block diagram of a simple microprocessor system. Describe the function of each feature in the diagram.

10. Using the diagram from the previous question, describe the sequence of operations that occurs when an instruction is fetched, then executed. Assume the instruction will store data from the accumulator into main memory.

11. A microcontroller needs to read in some data from an I/O chip. What are the two values that would be read in to the processor along the data bus?

12. A current PC might be able to use 32GB of DRAM. In principle, how many address lines must be implemented for this to happen? (note the actual pattern of address lines is complicated by the use of DRAMs; this will be covered in more detail in unit 3).

13. What would happen as a result of adding 100 to the value currently stored in the Program Counter/Instruction Pointer?

14. When instructions are fetched and executed in a processor, does the Fetch phase always take the same time? Does the execute phase take the same time?

PRACTICAL 1: CPU SIMULATION

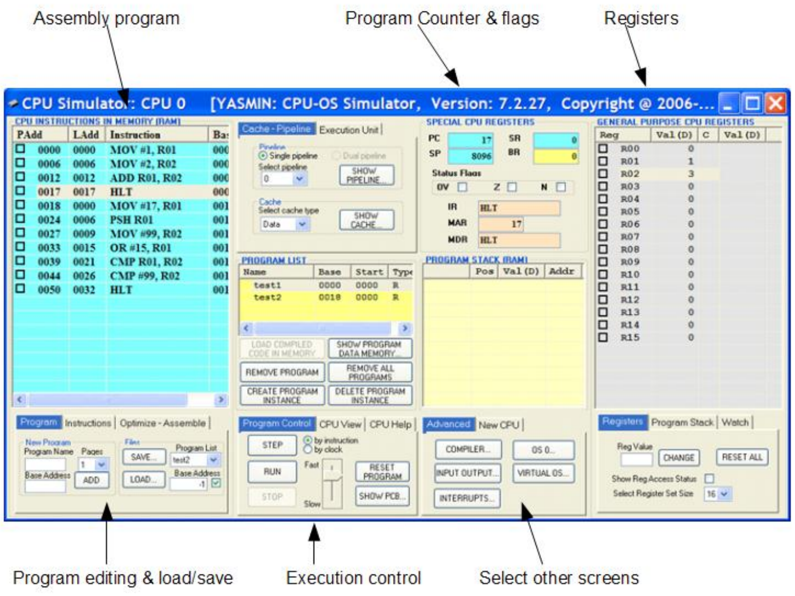
OBJECTIVE

This lab introduces the CPU-OS simulator and some simple assembly language programs. These programs are used to show the Fetch-Decode-Execute cycle and the use of RAM to store variables. It uses the programs **loopcount.sas** & **count5.sas**

INTRODUCTION

The CPU-OS simulator is an integrated simulator for a processor (Central Processing Unit: CPU) and operating system (OS) that allows you to experiment with a wide range of hardware & software features. The processor is not based on any specific processor. It has some features that are RISC like (multiple identical registers) and some features that are CISC like (such as variable length instructions). It comes complete with a high level language (HLL) compiler, so it is possible to write either high level or assembly level programs. The compiler converts programs to the assembly language level so that you can see how the compiler works. As with the processor, the HLL is not a particular language: it's a fairly generic C-like language. It has standard loops, subroutines and so on, as well as some support for OS type features.

The simulator was developed at Edge Hill University with the help of funding from the HEA. It is freely available under a creative commons licence. http://www.teach-sim.com/


Assembly program
Program Counter & flags
Registers
CPU Simulator: CPU 0 [YASMIN: CPU-OS Simulator, Version: 7.2.27, Copyright @ 2006-...
Program editing & load/save
Execution control
Select other screens

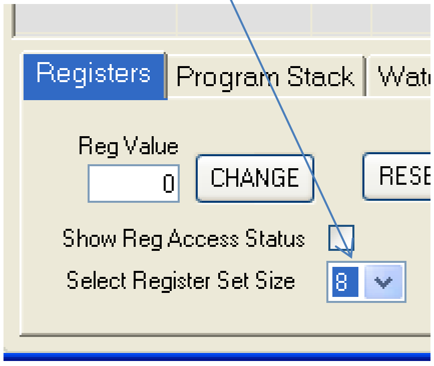
CPU INSTRUCTIONS IN MEMORY (RAM)

PAdd	LAdd	Instruction
0000	0000	MOV #1, R01
0006	0006	MOV #2, R02
0012	0012	ADD R01, R02
0017	0017	HLT
0018	0000	MOV #17, R01
0024	0006	PSH R01
0027	0009	MOV #99, R02
0033	0015	OR #15, R01
0039	0021	CMP R01, R02
0044	0026	CMP #99, R02
0050	0032	HLT

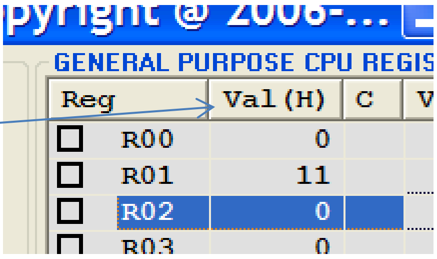
THE SIMULATOR SCREEN

The main screen has a number of sections. The left hand side shows the current program in assembly language. The general purpose registers are shown on the right. Special purpose registers, such as the Program Counter (PC), Stack Pointer (SP) and Status Flags (OVerflow, Zero & Negative) are shown in between. Most features are explained by hovering the cursor over them.

An interesting feature of the simulator is that you can select the number of general purpose registers there are: the 'Register Set Size' (from 8 to 64). I usually keep it at 8 or 16 to keep the screen from becoming too cluttered.

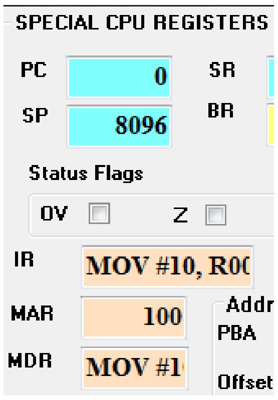


The registers hold 32 bits each. You can select whether the contents are displayed in decimal or hex by clicking on the top of the Val column.



SPECIAL REGISTERS

As with most processor, there are some special purpose registers. PC, the Program Counter, tracks where the processor has got to in a program. The Flags indicate status: has there been an overflow (OV)? Is the result zero (Z)? IR is the Instruction Register that holds instructions while they are being decoded. MAR holds addresses being sent out from the processor, MDR holds data coming into the processor. SP is the Stack Pointer: a register used to locate the stack, a temporary storage location in RAM.

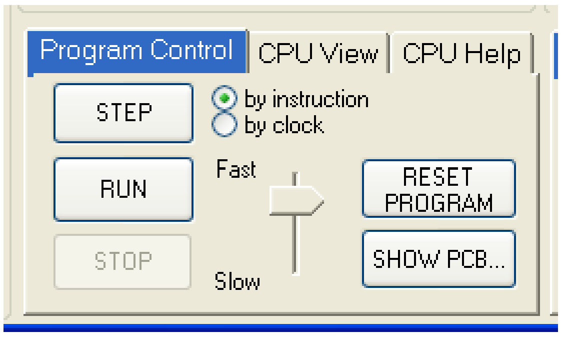


ENTERING PROGRAMS

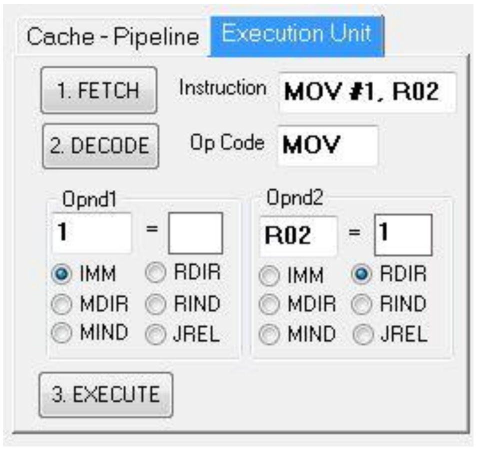
Assembly language and high level programs are entered in different ways. Assembly level programs are entered in a slightly idiosyncratic way: you select them from a menu one at a time. This is a bit tedious, but you will only ever be doing this for short programs. You can save programs and load them again. The high level language system is a bit more intuitive in that you type in programs with an editor which allows you to cut and paste from other documents. If you have done any programming in the past, this should seem fairly natural.

RUNNING PROGRAMS

The simulator can run in a variety of modes. The most obvious way to run it is to use the RUN or STEP buttons to execute the code normally one instruction at a time. If you RUN the code, then you can select the speed it runs at using the slider. This can be useful if you are visualising some aspects (such as the contents of registers) as the program executes. The line that is about to be executed is highlighted in grey.



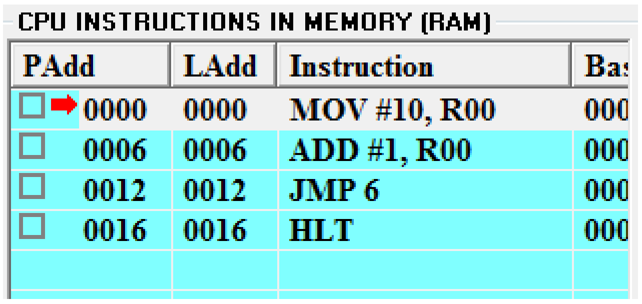
Step: [✓ by instruction] by clock



Instruction	MOV #1, R02
Op Code	MOV

	Opnd1	Opnd2
Value	1 =	R02 = 1
Mode	[✓ IMM] RDIR MDIR RIND MIND JREL	IMM [✓ RDIR] MDIR RIND MIND JREL

If you want to see the detailed sequence of operations, then you can also select the execution to go by clock rather than by instruction. This means that the instructions are broken down into the different stages (fetching the code, decoding and so on). This can be useful if you are looking at details of how instructions move through the instruction pipeline for instance. If you do that, the fetch.. decode.. execute sequence can be controlled directly by using specific buttons for these actions. These buttons are on the 'Execution unit' tab which is not shown by default. This bit of the simulator is a bit buggy: you don't always get the correct part of the sequence displayed.



CPU INSTRUCTIONS IN MEMORY (RAM)

PAdd	LAdd	Instruction	Ba
0000	0000	MOV #10, R00	000
0006	0006	ADD #1, R00	000
0012	0012	JMP 6	000
0016	0016	HLT	000

A SIMPLE PROGRAM.

Click the **LOAD…** button on the simulator near the bottom left. Load the program loopcount.sas. You should see this 4 line program. The Physical Address (**PAdd)** where each instruction is stored is shown on the left (**LAdd** is the Logical address; ignore this for now, it is used by the operating system.) You can see what each line of the program does by clicking on the **STEP** button. If you try **STEP**ping through the program you will see a number of things. The main section of the program is a simple loop that repeats for ever. The program starts with the number 10 being put into the general purpose register R0, then each time round the loop, one is added to the register. You can use the **RESET PROGRAM** button to go back to the beginning at any stage. Check you understand what is going on by answering the following questions:

What does this simple program do?

____

The next instruction to be executed is highlighted in grey. What register holds the address of this instruction?_______

What length is each instruction, i.e. what is the difference in the address of each instruction?__________________

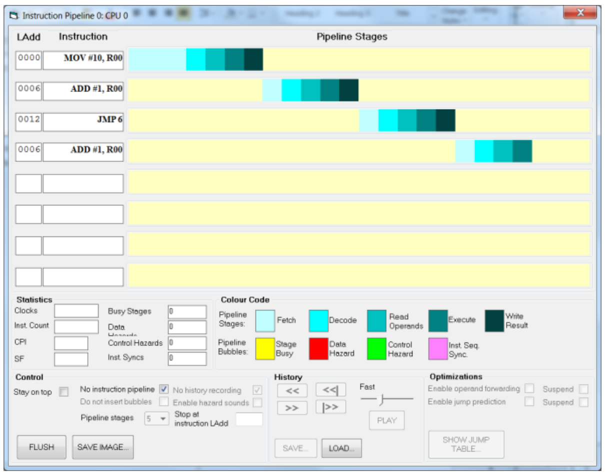
There is a 'Halt' instruction at the end of the program: **HLT**. Is it ever executed? ___________________________

Normally we will be able to see what is going on more clearly by using the **STEP** button, but you should also try using the **RUN** button to let the program execute normally. Experiment with the **Fast…Slow** slider to see the range of speeds that are available. If you leave the program running, where do you see the loop count incrementing?

____

THE INSTRUCTION PIPELINE: FETCH, DECODE, EXECUTE

Normally when you are running through a program, the instructions are executed with the **by instruction** box selected (next to the **STEP** button). In order to be able to see the workings of an instruction pipeline, you need to select the **by clock** box instead (some versions of the simulator label this **by single tick**). Now each press on the **STEP** button is equivalent to a single clock, rather than a whole instruction. If you try it out, you will see that a typical instruction on this processor takes about 4 or 5 clocks to execute.



Instruction Pipeline 0: CPU 0

LAdd	Instruction
0000	MOV #10, R00
0006	ADD #1, R00
0012	JMP 6
0006	ADD #1, R00

If you now select the **SHOW PIPELINE button** near the top middle you will get another window showing the instructions as they run.

Try running through a few instructions: you will see how they are colour coded to show the stages as they run: 1 Fetch, 2 Decode, 3 Read operands, 4 Execute, 5 Write result. If you tick the **Stay on top** box on the bottom left of the pipeline window, you will be able to see the pipeline while you are clicking at the main window to step. The default view, as seen here, is for there to be no instruction pipeline. Later on we will look at how adding an instruction pipeline affects things.

On the main simulator screen, press the **RESET PROGRAM** button to start again. Now we want to look at some of the special registers. Making sure that execution **by single tick** is still selected, step through the program. You will see the contents of MAR and MDR change.

What is the significance of the number that appears in MAR? ____

What is the significance of the contents of MDR? ____

On a real processor, the contents of MDR would not be text. What would it be? ________________________________

What happens if you clear the **No instruction pipeline** box?

____

DECISION MAKING

In high level languages, there are usually a number of control constructs: if, case, loops and so on. At the processor level there is a much more limited way of controlling the way a program executes, mostly using the flags. First clear the old program with the **REMOVE PROGRAM** button. Load the program count5.sas. This program initialises two registers, and then starts a loop where one is incremented each time round the loop. Step through the program, and you should find that lines 1 – 3 are fairly straightforward: initial values are put in R0 & R1, and then R0 is incremented:

```
1. MOV #0, R00
2. MOV #5, R01
3. INC R00
4. SUB R00, R01
5. JGT 6
6. HLT
```

The end is more complicated: arithmetic is done, then a decision is made based on the results of the arithmetic.

In line 4, is R0 being subtracted from R1 or is it the other way round? _____________________________________

In the subtraction, what happens to the original values of R0 & R1 ____

Line 5 is a conditional jump: it only jumps if a particular condition is met. The mnemonic JGT is short for Jump if Greater Than. The instruction does NOT mean Jump if > 6, so what is the meaning of the 6?

____

The jump goes back to line 2, putting the number 5 back into R1, why is this needed?

____

The JGT instruction actually jumps depending on the state of the Zero flag (Z in the special registers). When you are doing the subtraction of R0 & R1, when will this give a result of 0?

____

Click on the instruction in line 2 then select the **EDIT** button (in the **Instructions** tab). Change the loop count to 10 then reset and re-run the program to check your change has worked (i.e. the program loops 10 times).

VARIABLES AND RAM

The previous program used some numbers. The numbers (9 & 5) were embedded directly in the program, a method known as immediate memory addressing. This is not very flexible as the numbers are part of the program; useful for constants, but not much else. In practice we want to be able to work with numbers that can change on each running of the program: variables. There are a number of ways of implementing this, but usually they use RAM to store the temporary values. Here is a program in the High Level Language of the simulator, with the equivalent assembly language in the second column. You will see that each line of the HLL compiles to several lines of assembler:

```
var x integer                 Program variables
var y integer
                              MOV #1978, R02             ;Move 1978 into R2,
x = 1978                      STW R02, 6                 ; Store in RAM location 6
                              MOV #3468, R02             ;Move 3468 into R2,
y = 3468                      STW R02, 9                 ;store in RAM location 9
                              LDW 6, R02                 ;Get the variable x
x = x + y                     MOV R02, R01               ;  put in R1
                              LDW 9, R02                 ;Get variable y -> R2
end                           ADD R02, R01               ;Add R2 & R1
                              MOV R01, R02               ;
                              STW R02, 6                 ;Store result in X
                              HLT
```

Remove the old program then You can load the program variables.sas. Click on the **SHOW PROGRAM DATA MEMORY** button to see the contents of RAM. Step through the first two lines of the program: that will load the number into the variable X in RAM. How does the number appear in locations 6, 7 & 8?

____

The first byte is a code to say an integer is being stored. Then the number 1978 is stored in two halves. You can use calculator to check what 1978 is in hex. The number is 16 bits long, i.e. two bytes. Is the hex number stored in RAM with the most significant byte at the lowest or highest address?

____

Is this little- or big-endian? (check Wikipedia if you don’t know what this means)

____

There are two peculiarities to this simulator. The first is that the compiler uses R2 for reading & writing to memory. There is no particular need for this (you can try editing the program so different registers are used). This means that in a complied program, there are often extra steps: copy from one register to R2, then from R2 to RAM; it could be done in one. The second peculiarity is that when variables are stored in RAM, a code is also stored to show what sort of variables there are (e.g. 02 for an integer).

Can you think of an advantage of doing this?

____

and a disadvantage?

____

Most real processors just store the data directly in RAM, without this code. Run the program to the end and check that the result of the addition gets stored in variable x. Click on the first instruction so that it is highlighted, then click the **EDIT** button (on the **INSTRUCTIONS** tab). Change the number in line 1 from 1978 to 31000; click the **EDIT INSTRUCTION** button to confirm the change. Run the program to see what happens.

Why does the modified program do what it does?

____

Which flag has changed at the last instruction?

____

Finally, change the number in line 1 back to 1978 and then edit the ADD instruction so that it becomes a subtract (SUB R02, R01). Run the program again. What is the state of the N flag after the SUBtract?

____

What is the final answer in decimal and hex? ____

LECTURE 1: INTRODUCTION AND BASIC HARDWARE CONCEPTS

CONTENTS:

SYSTEMS & SERVICES MODULE

Two main topics:
- Computer architecture.
- Operating systems.

Computer Architecture is further divided:
- Units 1-3: PC architecture & the Pentium (Alistair Armitage)
- Units 4-6: microcontrollers & I/O (Frank Greig)

Learning Outcomes: Block A (Computer Architecture):
- "Analyse the architecture of a range of processor based system".
- "Compare and contrast the different approaches to system architecture".

PRACTICAL OUTCOME

1. To be able to understand hardware specifications.
2. To be able to evaluate alternative PC solutions.
3. To understand current trends in computer architecture.
4. To have an introductory knowledge of some non-PC computing systems (Microcontrollers, mobile systems).

AN INTRODUCTION TO COMPUTER HARDWARE (REVISION OF CS1)

Contents:

Processor

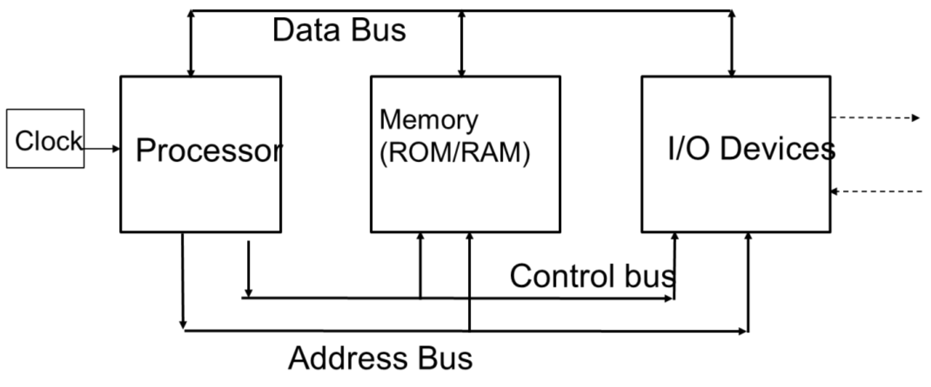
Memory

Busses

I/O

OVERVIEW

This section will briefly introduce some of the main features of computer hardware. The main idea is to identify the key considerations in computer systems. One problem is that there is a wide range of computer systems, from small embedded microprocessors to complicated PCs. We will look at two examples: the 8051 family of low cost microcontrollers and the Pentium family used in most PCs. Many text books start with a simplified view of the components at the heart of the computer system.

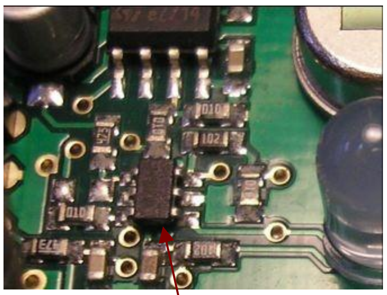


A SIMPLE TEXT BOOK MICROPROCESSOR

The preceding picture shows, in an idealised form, the processor chip, and how it communicates with memory chips and peripherals. The processor communicates directly with the semiconductor memory chips (ROM & RAM). A simple embedded microprocessor could look quite similar to this simple picture (except that the three components might be combined into one chip).

A £15 PIR security Detector

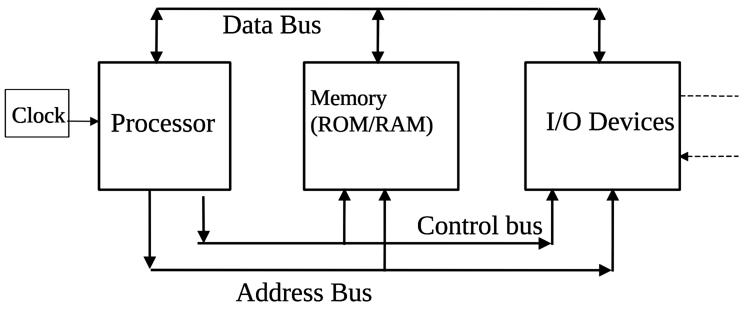
Microcontroller & IR sensor



A 30p Microcontroller

Processors & Busses

The processor is the heart of the system. In a simple embedded processor system the processor will control all the other circuits. The processor communicates with the rest of the system via buses. There are 3 main types: Address, Data & Control. In a small embedded processor, these buses may be all there is. In a desktop PC, the situation is more complicated, with the same main types of bus, but different levels operating at different speeds. We will be looking in more detail at how PC systems use a hierarchy of buses.



Basic System Architecture: Clock

The processor is driven by a fixed frequency clock. The frequency is determined by a crystal and is very accurate (as in wrist watches). What is the frequency of current PCs?

Would you know how to get the clock period (the length of a clock cycle) from the frequency?

The clock frequency gives a rough measure of the performance of a computer, but is not the only factor determining performance.

Buses

The Buses are the main means of communication within a processor system. Each bus is a collection of similar signals: usually conveyed by copper tracks and pins on the motherboard. The control bus is a collection of signals (mostly from the processor) by which the processor controls the rest of the system. For instance the processor may send signals to write to an Input/Output (I/O) device. Or, it may send signals to read from the memory chips.

Address bus

The address bus is used by the processor to select a particular chip, and the location within a chip that is to be used. Normally, it is only the processor that generates addresses, so it is shown as a unidirectional bus. The more address lines there are, the more locations can be addressed. With 2 lines there are 4 combinations (00, 01, 10, 11). Each additional line doubles the number of combinations (adding a third line would give me the previous four combinations with a 1 at the front, and the same four with a 0 at the front).

Powers of 2

Early processors had 16 address lines. This gave 2^{16} possible combinations of addresses i.e. 65536. This is usually referred to as 64 kbytes. When the PC first came out, it had 20 address lines. 2^{20} gives approximately 1 million combinations: 1 Mbyte. What is the current PC memory size?

Number of Address lines	Number of memory locations	Abbreviation
1	2	
8	256	
10	**1,024**	**1 K**
16	65,536	64 K
20	**1,048,576**	**1 M**
24	16,777,216	16 M
30	**1,073,741,824**	**1 G**
32	4,294,967,296	4 G

A Shortcut

There is a simple pattern:

10 lines gives 1 K	(roughly 1 thousand)
20 lines give 1 M	(roughly 1 million)
30 lines gives 1 G	(roughly 1 billion)
40 lines gives 1 T	(roughly 1 trillion)

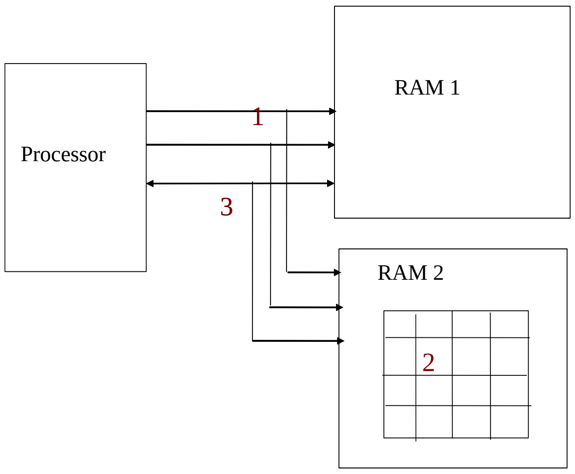
If you have a number nearby you can find the power of two from this. For instance, how many addresses do 22 lines give? (What is 2^{22}?)

THE DATA BUS

The data bus is used to transmit data from one point to another. Usually multiples of 8 bits are sent at a time. So, 16, 32 or as many as 128 bits of data may be sent together. This is a classic example of a parallel bus: data travels together in parallel. This mode of transfer only works well over short distances (say between chips on a circuit board). For longer distances there are problems with getting the data bits all to arrive at the same time. Instead, data is sent along a single line over long distances.

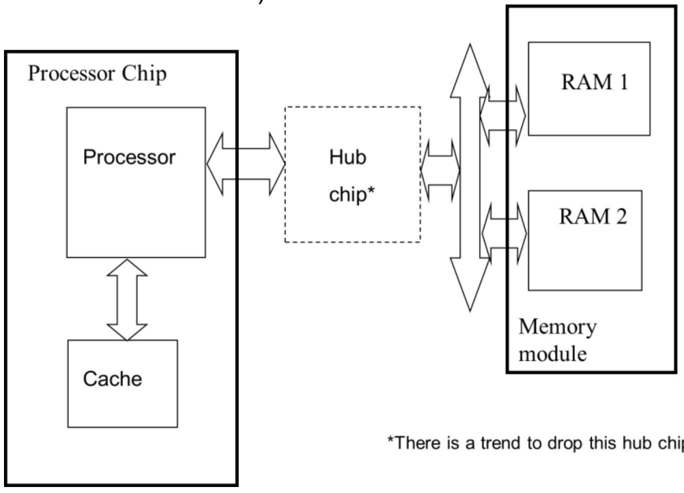
Example: reading from RAM

1. The Address and Control signals are set-up
2. The RAM chip fetches the data from the correct location.
3. The data is sent to the processor along the data buses.



READS AND WRITES

In a PC data may go from RAM to the processor or vice versa. This is controlled by the processor using control lines. As everything is viewed from the point of view of the processor, data going out of RAM to the processor is a 'Read', Data going to the RAM from the processor is a 'Write'. The processor has to read its instructions from memory before it can execute them: there are a lot of reads. Sometimes the instruction results in data being read from RAM, or stored in the RAM. This can result in traffic going either way. PCs use the classic 'Von Neumann' architecture: instructions and data are stored in one common block of RAM. What would be the implications of storing them in separate blocks (the 'Harvard' architecture)?



*There is a trend to drop this hub chip.

DATA TRANSFER IN A PC

Data transfers to and from RAM are slightly different in a modern PC: The processor comes in a module with local cache; many data transfers use this, and don't involve the RAM. RAM chips come in a module containing a number of chips. The processor transfers data via a 'hub' or 'bridge' chip that handles the memory (and graphics). (Note in the diagram that all the buses have been combined into one to avoid cluttering the picture).

RAM AND ROM

There are two key types of memory chip. RAM is the main one used in PCs. It has the disadvantage that it is volatile (it loses data when the power is switched off). However, it can be made cheaply and has the advantage that data can be written to it very rapidly (a few nanoseconds). Because of this, it is the main type of memory used in PCs; there will be more on this later. ROM has the advantage that it doesn't lose data when power is removed. Depending on the type it can be slow, or impossible, to change the data stored in it. In PCs it is used to hold some low-level I/O routines (the BIOS) and a small amount of software that runs as the PC starts up (before the main software can be loaded from disk). Flash disks also use this, but these act like disk drives, not main memory.

THE FETCH (DECODE) EXECUTE CYCLE

Driven by the clock, the processor continually **fetches** instructions from memory. A special register, called the Program Counter or Instruction Pointer holds the location in memory (i.e. the address) of the "next instruction". Once the instructions are inside the processor they are **decoded** to work out what needs to be done. They are then **executed**. This may involve different things: performing arithmetic, storing results, making decisions. This is referred to as the Fetch-Decode-Execute cycle (often abbreviated to Fetch-Execute).

THE EXECUTE OPERATION

The Instruction Decoder examines the contents of the Instruction Register and sends appropriate signals to other parts of the CPU to carry out the actions specified by the instruction. There will be different types of instruction, so exactly what happens depends on the instruction. For instance, the instruction may involve arithmetic, and is executed entirely inside the processor. It may involve moving data in or out of the processor, using the buses again. This cycle of Fetch and Execute repeats indefinitely.

Example Instructions

- Reading operands from registers in the Arithmetic Logic Unit. Or reading them from the memory, using the the buses.
- Causing the circuits of the Arithmetic Logic Unit to perform arithmetic or other computations.
- Storing data values from the ALU into registers. Or storing in memory using the buses.
- Changing the value of the Program Counter (Jump or Branch instructions).

Instruction type 1a: the 8051

Each instruction does a relatively simple thing, such as add a number to the contents of the A register. Typically it will have two sections: the 'opcode' and the operand. Opcodes are effectively verbs: MOV, ADD, INC (for MOVe, ADD, INCrement). The operand is the object(s) that are being worked on. As an example, there is an instruction to add the contents of the R6 register to the A register: **ADD A,R6**
The instructions may take 1,2 or 3 bytes to store.

Instruction Type 1b: the Pentium

A processor, such as the Pentium family, has many hundreds of possible instructions. Far more than the 8051. Some of these are very specialised (e.g. 3-dimensional graphics calculations). The instructions, when stored in memory, can take up different amounts of room (1-3 bytes). This type of processor. Like the 8051, is known as a Complex Instruction Set Computer (CISC). Every new generation of Pentium has to be able to run old software; this complicates new designs.

Questions

Why do most people never need to learn the 8051 or Pentium instructions?

Can an old compiler, written for an early version of the Pentium, still work for a new Pentium, with all its new instructions?

Instruction Type – 2

An alternative approach is to use a much smaller set of regular instructions. This is known as a Reduced Instruction Set Computer: RISC. Each is of a fixed length (usually 32 bits), making the fetch operation much simpler. The operations are simpler, so achieve less, but can run more quickly. Compilers have to be more sophisticated in order to do the same work with simpler instructions. This approach is used by the ARM processor in many mobile phones and tablets (actually many processors are hybrids).

Object code

The code that runs on the processor is referred to as object code (or machine code). As different processor use different instructions, the object code for different processors varies considerably. In one family, such as the Pentiums, the code will be the same. However, Pentium code will not run on an 8051 (and vice versa). In the processor, the object code is stored as binary. This is too difficult for humans to understand, so is normally displayed as Hex. A slightly friendlier version is the use of Assembly language mnemonics.

Assembly language

```
ADDRESS  OBJECT     ASSEMBLY LANGUAGE      MEANING
000F     7F05       MOV R7,#0X05           Move the number 5 into Register 7
0011     7E1A       MOV R6,#0X1A           Move the number 1A into Register 6
0013     EF         MOV A,R7               Copy the contents of R7 into A
0014     2E         ADD A,R6               Add the contents of R6 to A
```

This could have come from a line of C code: x = 5 + 26;

Remember also that the code will be stored in binary; hex is just used to show the object code more compactly:

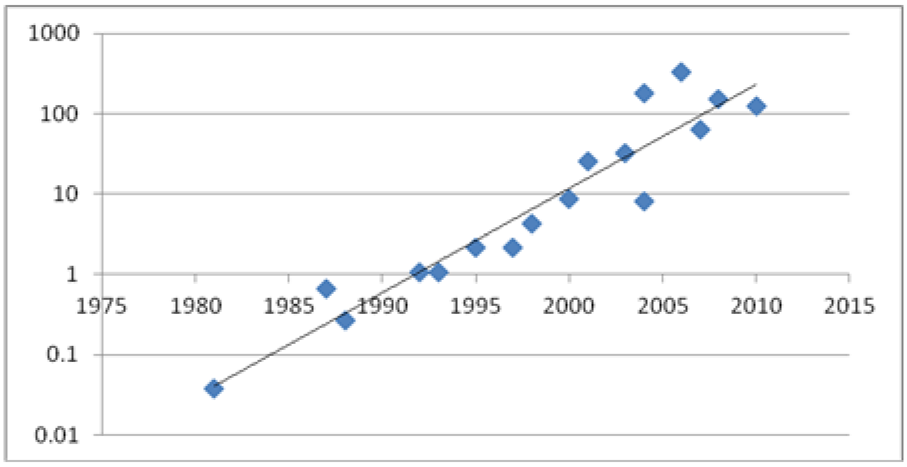
Code stored in memory as binary ->

Address		
0000000000001111	01111111	Instruction 1
0000000000010000	00000101	
0000000000010001	01111110	Instruction 2
0000000000010010	00000111	
0000000000010011	11101111	Instruction 3
0000000000010100	00101110	Instruction 4

Further Reading: Developments in bus systems

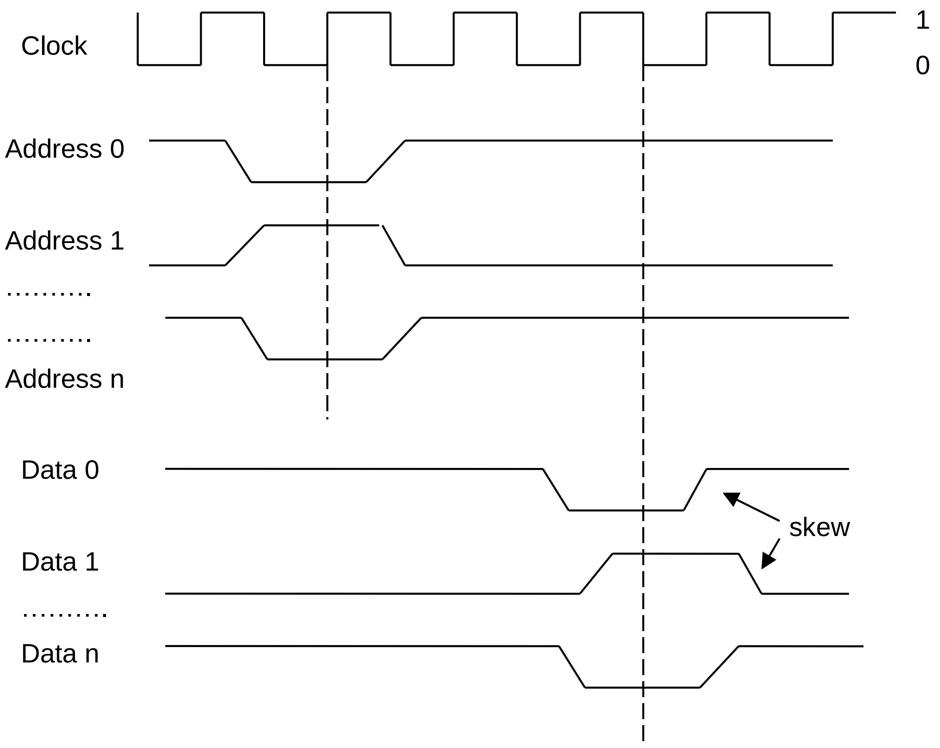
As the technology has improved, the data rates of buses that transfer data around a computer have increased. The table shows maximum data transfer rates for some of the common computer buses since the original PC with its ISA bus was introduced. If you plot the data on a graph, you can see that the rates increase by a factor of 10 every 8 years or so. This is equivalent to a speed-up of x5000 in 30 years or doubling every 1.4 years. This is not far off Moore's law, which generally refers to a doubling every 18 months (or 1.5 years). Strictly speaking, Moore's law refers to the number of transistors on a chip, but it seems to apply to other features of integrated circuits.

Bus Technology	Year	Rate (Gbit/s)
ISA	1981	0.04
MCA	1987	0.66
EISA	1988	0.27
VESA	1992	1.07
PCI (33 MHz)	1993	1.07
PCI (66 MHz)	1995	2.13
AGP 1 (1x)	1997	2.13
PCI - X	1998	4.27
AGP 2 (4x)	2000	8.53
Hypertransport 1.0	2001	25.6
PCI Express 1.0 (x16)	2003	32
DMI 1 (x4 link)	2004	8
Hypertransport 2.0	2004	179
Hypertransport 3.0	2006	333
PCI Express 2.0 (x16)	2007	64
QPI (2.4 GHZ)	2008	154
PCI Express 3.0 (x16)	2010	126



Increasing the speed

Early computer buses were almost exclusively Parallel: signals went down multiple lines in a bus together. So, the original PC (XT) had 20 address lines and 8 data lines. When the processor wanted to select, say, a specific memory location, the address lines would all be activated together. Different patterns of ones and zeroes on the 20 address lines would all appear together and select one chip and one location on the chip. After that, the data would be transferred by being putting on the data lines. With 8 data lines, a complete byte would be transferred at one time. Even though the clock rate was not very high (a few MHz) the fact that all lines were transmitting at the same time meant that reasonable amounts of data could be transferred. In a typical transfer, the address values appear first, with the data appearing after a delay. Note that the data values may not arrive at exactly the same time: usually there is a clock edge that is used to decide when to read the data. On a bus like this, data can generally be sent in either direction, but not at the same time.



There are obviously two ways to improve the data rates: increase the transfer rate or increase the number of lines. Data buses increased from 8 to 16 to 32 and eventually to 128 lines. However, these lines need physical connections, which add to the cost and complexity of chips, so 128 is about the limit. This is only an increase of x16 compared with the original PC, so most of the increase has to come from increasing the speed.

As integrated circuits gets faster, the clock rate for transfers has increased, but there are limits to this for traditional parallel buses. At current clock rates, signals only have a nanosecond or two to get from one chip to another. But, to be interpreted correctly, all the data bits have to arrive at the same time. If they don't arrive together (this is called skew) then they may be misread. In the diagram above, the data doesn't quite arrive at the same time, but there is a short period where the data lines are all valid. If the clock period was much shorter, this might not be the case. As it's not possible to route tracks across circuit boards so that each track is exactly the same, skew in length leads to skew in the data arrival times. Skew problems have limited the data rates for parallel buses, though recently systems such as QPI have got round this by using extra circuitry to detect and correct for skew.

PARALLEL VS SERIAL

If you send the bits down a serial link (one after the other) you can eliminate skew problems and increase the clock rate. For longer distances (say outside of a PC box) this has been the approach for some time, as it is almost impossible to have long parallel buses (Ethernet has effectively been a serial bus for decades). This is the approach that has been used for some time now on many internal PC buses. PCI-Express is a good example of this. At its simplest, it consists of two serial links: one transmitting data in one direction, the other transmitting data in the other direction: dual simplex. By running the links at a higher rate than is possible for a parallel bus, the total throughput can be higher.

In practice, the improvement in speed from the higher data rate is offset by the fact that only one bit is being sent at a time (or potentially with some systems one bit in each direction). The other factor that permits large data rates is to have multiple links. If you look at the table, I quoted the data rate for PCI Express 1.0 (x16). This means 16 individual links running together. But isn't this just back to a parallel bus? Not really, because the links are essentially independent. The links don't need to keep in perfect step, the way the lines on a true parallel bus do.

DIFFERENTIAL SIGNALLING

The older buses tended to use simple 'single-ended' signalling. This means that each bit is carried on a single line, where the value of the bit (one or zero) is given by the voltage level with respect to a common zero volts line. In the timing diagram earlier, I didn't bother to show this zero volts line: it is assumed to be there, and I only need to show one line per bit.

Recent systems tend to use differential signalling. Two lines are used to represent each bit. Here it is the voltage difference between the two lines that represents the logic level. This takes more lines but has advantages in noise immunity and in being able to use low voltages. Noise immunity comes from the fact that noise spikes tend to be received on both lines, but when the logic level is measured, it is the difference between the two lines that is measured. This effectively subtracts out the noise. In the older single-ended systems, this doesn't occur, and much larger voltages have to be used to ensure the logic levels stand out from the noise.

Incidentally, it is common now to connect flat panel displays using the Flat Panel Display Link (FPD Link). There are several versions of this standard, but they all use Low Voltage Differential Signalling: a standard for sending differential signals down pairs of lines. These links are often referred to as LVDS links, but that is not strictly correct, as LVDS is used for many things apart from sending data to displays.

NETWORK TECHNOLOGY

Many of the ideas that are now used on communications between chips, such as packet based transmission, come from the world of networking. Intel's Quick Path Interconnect (QPI), designed for high speed inter connections between processors (or processor and North Hub), is a typical example of this. It is based on a five-layer network model (physical, link, routing, transport, and protocol layers) and transmits packets, complete with checksum and a header containing a destination address. It can be used to connect multiple processors into a mesh, or just two chips (in which case some of the features, such as routing, are not used). Other features that have transferred from network to inter-chip communications include the use of redundant code schemes, such as 8b/10b, where 10 bits are used to transmit 8 bits of data, allowing for error checks, clock recovery and balancing of the transmission lines. The other big difference is that communications are going away from the idea of shared buses (where typically one device can talk at any time) to dedicated point-to-point links. Not sharing the bus bandwidth between multiple devices has speed advantages. Here is a summary of some of these features:

Bus	Connects	parallel or serial	Differential	packet based
PCI Express	CPU to graphics, Hubs to Graphics or I/O	s	y	y
DMI	North to South hub	s	y	y
QPI	CPU to CPU or North hub	p	y	y
HyperTransport	CPU to CPU or North hub	p	y	y

REVISION: NUMBER SYSTEMS

DECIMAL NUMBERS

The decimal number system uses base-10 arithmetic: this means that the basis is ten different digits. The power of this system (unlike roman numerals) is that the position of the numbers is also significant. So, a number like 486 is the same as 4 lots of one hundred, plus eight lots of ten, plus 6 lots of one. The columns could be labelled units, tens, hundreds and so on (from right to left):

… Thousands hundreds tens units

This type of system can be used with other bases. However, it helps if we can generalise the values of the columns:

… 10^3 10^2 10^1 10^0

(if you are not used to the maths, take my word that 10^0 is equal to 1)

BINARY NUMBERS

These work in the same way as decimal numbers, but now the columns are worth powers of 2, not 10 (i.e. we are using base 2 arithmetic).

… 2^3 2^2 2^1 2^0

Or 8 4 2 1

If we know what the columns are worth, we can work out what a binary number means in decimal. For instance, 1101 is the same as 1 eight, 1 four, 0 twos and 1 unit = 8 + 4 + 1 = 13.

This system can continue to larger numbers, though in these classes you are most likely to meet numbers up to 8 bits long (8 bits is known as a byte). Here is a bigger example:

10101011 = $(1 \times 2^7) + (1 \times 2^5) + (1 \times 2^3) + (1 \times 2^1) + (1 \times 2^0)$
OR
(1 x 128) + (1 x 32) + (1 x 8) + (1 x 2) + (1 x 1) = 128 + 32 + 8 + 2 + 1 = 171

You will often see groups of 8 bits. I/O devices generally work in multiples of 8, so you will see a port reading 8 switches for instance to provide a byte. In these cases, it is also helpful to be able to label individual bits, starting with bit 0 at the bottom:

		b7	b6	b5	b4	b3	b2	b1	b0		
MSB	$\longrightarrow$	1	0	0	0	1	1	1	0	LSB	$\longleftarrow$

The bit on the left is also called the Most Significant Bit (MSB), and the bit on the right is the Least Significant Bit (LSB). Incidentally, a group of 16 bits is usually referred to as a word (4 bits may be referred to as a nibble, 32 bits as a long word or double word).

If you work it out, the largest 8 bit number is the same as 255 (128 + 64 + 32 +…+ 1), so our 8 bit numbers can go from 0 to 255. 16 bit numbers go from 0 to 65535. These are usually referred to as unsigned integers (we will meet signed numbers later).

DECIMAL TO BINARY CONVERSION

This is not as easy for numbers of any size. For small numbers, it can be done by inspection. For instance, how could I get 11 into binary? By adding an 8 to a 2 and a 1 -> 1011. For bigger numbers, there is a method that is methodical: keep dividing by two and recording the remainder. The first remainder is the LSB, and the last remainder is the MSB. For example, to convert 242 to binary:

÷ 2	242			(2 goes into 242 121 times, with a remainder of 0)
÷ 2	121	r 0	LSB $\longleftarrow$	(2 goes into 121 60 times, with a remainder of 1)
÷ 2	60	r 1		
÷ 2	30	r 0		
÷ 2	15	r 0		
÷ 2	7	r 1		so **242 -> 11110010**
÷ 2	3	r 1		
÷ 2	1	r 1		
	0	r 1	MSB $\longleftarrow$	

Many calculators (including the Windows calculator) can also do the conversion between number systems. You should practise with the Windows calculator switched to scientific notation to make sure you know how to use it.

HEXADECIMAL

Binary is always used in computers. However, long strings of binary numbers are difficult for humans to handle. For instance are the numbers 1010111110011111 and 1010111111001111 the same? In fact they aren't, but you have to look carefully to see how they differ. Hexadecimal (base 16) is used simply as a short cut for representing binary numbers. It turns out that each hex digit represents exactly 4 bits, so it is relatively easy to group binary numbers into chunks of 4 bits and convert them to hex (and vice versa). Of course, we don't have 16 different digit symbols, so the letters A…F are co-opted

Converting between binary and hex should just be a matter of splitting the binary into groups of 4 bits, then looking up the table:

```
00111010 ->  0011  1010
               3   A

1001001001111111 ->
1001     0010     0111     1111 ->
   9     2     7     F
```

Converting between hex and binary is even easier: just look up the table for each hex digit:

```
7     C     B     5 ->
0111     1100     1011     1001
```

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

A couple of tips: Firstly, grouping bits by separating every clump of 4 bits with a comma can help. I got into the habit of writing a binary number as 0111,1100,1011,1001, and this makes it easy to convert groups into hex. Secondly, when going from binary to hex, it is important to group bits starting at the bottom (the LSB). Normally, numbers will be 8 or 16 bits, and it won't matter which end you start from. In some circumstances though, you might have a different number of bits. An Analogue to Digital Convertor might produce a 10 bit output. How do you get this to hex? All that needs to be done is to add some zeroes on the left:1100111000 is the same as 0011,0011,1000 -> 338 in hex.

This last number illustrates a possible source of confusion: 338 could be a decimal or a hex number (AFF0 is obviously hex). Usually a number on its own is assumed to be decimal unless we say otherwise. A mathematician would write 338_{16} to show it was to the base 16. This is not easy to write on a keyboard, so different conventions are used. You will mostly see a number proceeded by 0x to denote hex (i.e. 0x338); this is the convention used by C.

Hexadecimal to decimal conversion (and vice versa) is best done using binary as an intermediate: convert hex to binary, then the binary to decimal. Of course, Windows calculator does this easily.

ADDITION OF BINARY NUMBERS

In a computer, numbers are only added together two at a time. If a whole column of numbers have to be added, they are added two at a time. Adding two numbers in binary is relatively easy: the only complication is when there is a carry (which can never be more than one). The principle is to start at the LSB (on the right) and add digits. At any one time, we have two digits to add, plus possibly a carry of one from the previous column. So, when I add the numbers, I am effectively adding three numbers that each could be 0 or 1. The answer I get will be 0, 1, 2 or 3 (in decimal). Of course I am doing this in binary, so the answer could be 0, 1, 10 or 11. In the last two cases I have generated a carry to the next column. An example:

```
 110100
 110101
1101001
```

(note: the carries are shown in bold)

- Add the bottom two digits: 0 + 1 = 1. Write down the one; there is no carry.
- Add the next two digits, together with the carry from the previous stage: 0+0+**0**=0.
- Add the next digits and the carry: 1 + 1 + **0** = 10; write down the lowest digit and carry 1.
- Add the next digits and the carry: 0 + 0 + **1** = 1; no carry generated.
- Add the next digits and the carry: 1 + 1 + **0** = 10; write down the 0 and carry 1.
- Add the next digits and the carry: 1 + 1 + **1** = 11; write down the lowest digit, the carry becomes the next digit.

You will see that the answer is one bit longer than the starting numbers. Just as with adding two decimal numbers, the carry can never be more than one (9 + 9 = 18), so the answer will only ever be one bit more. However, this may cause problems: you might add two 8 bit numbers, and get a 9 bit result. The extra bit is not lost (the processor has a carry flag that is used to store the last carry). You do have to be careful though to make sure you don't get (or take care of) this sort of overflow. If you want to add 200 to 200, you can't do it with 8 bit numbers as 400 doesn't fit into 8 bits. You would be better to use 16 bit numbers that can easily hold the answer. The problem is even worse with multiplication, where the answer can be many bits more than the starting numbers, but this is beyond the scope of this course.

SIGNED NUMBERS

In normal arithmetic, we use a minus sign to represent negative numbers. In binary, we only have ones and zeroes, so we have to use a different system. The commonest is two's complement arithmetic. In this, we assume that the magnitude of the bits remains the same in each, but the MSB is worth a negative amount. For an 8 bit number, the columns are worth: -128, 64, 32, 16, 8, 4, 2, 1. So 1000,0110 would be (-128) + 4 + 2 -> -122
Then, any number with a 1 in the MSB will be negative, and if there is a 0 in the MSB it will be positive. Some examples:

Signed binary	decimal
0000,0000	0
0010,1000	40
0111,1111	127
1000,0000	-128
1111,1111	-1

If you work it out, the range of numbers that can be represented is -128 to +127. If you have 16 bits, then the MSB is worth -32,768. Numbers can go from -32,768 to +32,767. You can check all this with the Windows calculator, which is quite happy working with these negative numbers. When it is switched to binary, make sure you select 'byte' if you are working with 8 bit numbers. Turn on 'digit grouping' in the view menu to see the binary bits grouped into clumps of 4.
Arithmetic works fine with either signed or unsigned integers, but the interpretation of numbers differs depends on which type is being used. This is why in many high-level languages, integer variables have to be declared as being signed or unsigned so that the compiler knows what it is dealing with.

2'S COMPLEMENT

To get the negative version of a number, you just need to invert each bit (this gives you the one's complement) then add one to get the two's complement.

```
47    00101111
1's comp:   11010000
Add 1 00000001
2's comp:   11010001
```

To check 1101,0001 is -128 + 64 + 16 + 1 => -128 + 81 => -47

There is a short cut: start with the LSB then copy all the bits up to and including the first one, then invert the other bits. For instance 88 = 0101,1000 when turned into 2's complement becomes 1010,1000 (starting at the bottom, I left the zeroes and the first one unchanged, then complemented the rest).

One advantage of 2's complement is that subtraction can be done by adding negative numbers. Addition can be done much more easily in hardware than subtraction. Suppose I want to subtract 23 from 108. In binary, 108 is 0110,1100

23 is 0001,0111 (Note how I have extended this to 8 bits with leading zeroes; this only works if all numbers are 8 bits, or 16…). I can get the 2's complement of 23: 1110,1001. Now I add them together:

```
108      0110,1100
-23      1110,1001
      0101,0101   = 64 + 16 + 4 + 1 = 85
```

END OF UNIT - SUMMARY OF ACHIEVEMENT

On completion of this unit you should understand :

- The main buses (address, data & control) found in microprocessor systems
- The main constituents of a simple microprocessor system
- The Fetch-Decode-Execute cycle.
- The nature of Assembly language and machine code instructions
- How to investigate the main features of a PC based system

Unit 2 : PC architecture & the Pentium

PC architecture, motherboards, the Pentium family of processors and an introduction to Pentium assembly language.

Student Study Material

UNIT 2: PC ARCHITECTURE & THE PENTIUM

LEARNING OUTCOMES

On completion of this unit you should understand :

- Current developments in motherboard architectures for PCs
- The main functional units of a Pentium class processor
- The main registers and basic assembly language of a Pentium
- The way in which a high level language is converted to the machine code of a processor

PC SYSTEM ARCHITECTURE

Contents:

TUTORIAL 2 : PC MOTHERBOARDS

(This tutorial has two parts: the first is specifically about motherboards, the second part has more general questions taken from previous tests)

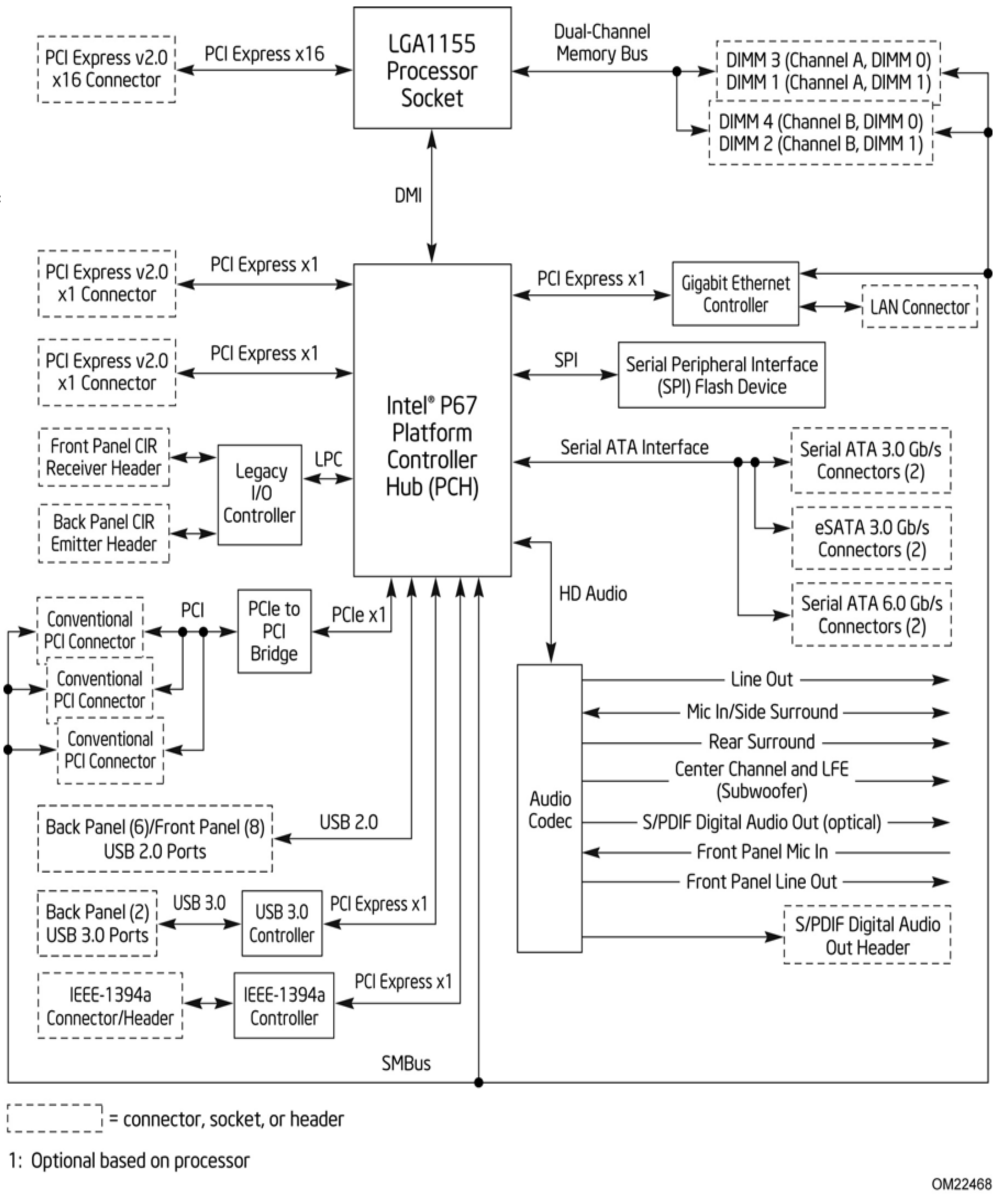
LEARNING OUTCOME

On completion of this tutorial you should understand:

- *The main components on a typical PC motherboard*
- *The way the possible options that allow the motherboard to be tailored to specific needs.*

INTRODUCTION

The Intel DP67BA is a recent motherboard for desktop PCs suitable for general purpose PCs. There are options that allow a system to be built up to suit particular specifications. These options include the type of processor, the number and type of disk drives, and the amount of main memory. The purpose of this tutorial is to familiarise you with the sort of facilities that are provided by the board, hopefully giving you a better idea of how the computer system is built up. On the next pages is a data sheet that originally came from Intel's web site. I have assembled the relevant sections into the accompanying handout. You should be able to find answers to the following questions in the data sheet, but ask the demonstrator if you need help.



PCI Express v2.0 x16 Connector
PCI Express x16
LGA1155 Processor Socket
Dual-Channel Memory Bus
DIMM 3 (Channel A, DIMM 0) DIMM 1 (Channel A, DIMM 1)
DIMM 4 (Channel B, DIMM 0) DIMM 2 (Channel B, DIMM 1)
DMI
PCI Express v2.0 x1 Connector
PCI Express x1
PCI Express v2.0 x1 Connector
PCI Express x1
Intel® P67 Platform Controller Hub (PCH)
PCI Express x1
Gigabit Ethernet Controller
LAN Connector
SPI
Serial Peripheral Interface (SPI) Flash Device
Front Panel CIR Receiver Header
Back Panel CIR Emitter Header
Legacy I/O Controller
LPC
Serial ATA Interface
Serial ATA 3.0 Gb/s Connectors (2)
eSATA 3.0 Gb/s Connectors (2)
Serial ATA 6.0 Gb/s Connectors (2)
Conventional PCI Connector
Conventional PCI Connector
Conventional PCI Connector
PCI
PCIe to PCI Bridge
PCIe x1
HD Audio
Audio Codec
Line Out
Mic In/Side Surround
Rear Surround
Center Channel and LFE (Subwoofer)
S/PDIF Digital Audio Out (optical)
Front Panel Mic In
Front Panel Line Out
S/PDIF Digital Audio Out Header
Back Panel (6)/Front Panel (8) USB 2.0 Ports
USB 2.0
Back Panel (2) USB 3.0 Ports
USB 3.0
USB 3.0 Controller
PCI Express x1
IEEE-1394a Connector/Header
IEEE-1394a Controller
PCI Express x1
SMBus
= connector, socket, or header
1: Optional based on processor
OM22468

OPTIONS

1. What processors can be used with this board, and what limits the choice of processors?
2. Where does the graphics get connected?
3. What is the minimum and maximum amount of cache and RAM that is available?
4. What determines the amount of cache?
5. The specification of memory chips for this board is very restricted. What exactly is specified? (i.e. what would you have to ask for if you were upgrading the RAM?)
6. Any idea what the 'Dual-Channel Configuration' for the DRAM is?
7. How many disk drives (hard disk or optical disks) could be plugged in before you would have to add an expansion card or use an external USB drive?
8. There is support for RAID. What is RAID?
9. Which of the following are built in, and which would need extra plug-in cards? (note, you may need to refer to the Block Diagram as well as the table summarising the features):
   - Local Area Network support
   - Graphics
   - Sound
   - Hard disk drives

OTHER FEATURES

10. SPI Flash memory is used to hold the BIOS*. This is a type of memory that doesn't lose data when the board is powered down, so it retains the BIOS. However, it can be re-programmed. Why would this facility be needed?
11. What can be done if more USB ports are needed than are actually provided?
12. Any idea what an S/PDIF connector is?
13. Legacy controllers for the old serial, parallel and PS-2 ports are no longer provided. What could you do if you had an old device that needed to be connected via one of these interfaces?
14. What is the difference between ‘conventional’ PCI and PCI Express?

***BIOS: The Basic I/O System that configures the PC on power up and provides the lowest layer of features used by the operating system. Amongst other things, the BIOS is responsible for loading the operating system from disk.**

Table 1. Feature Summary

Form Factor ATX (9.60 inches by 11.60 inches [243.84 millimeters by 294.64 millimeters])
Processor • Intel® Core™ i7, Intel® Core™ i5, and Intel Core™ i3 processors with up to 95W TDP in an LGA1155 socket
— One PCI Express* 2.0 x16 graphics interface
— Integrated memory controller with dual channel DDR3 memory support
Memory Four 240-pin DDR3 SDRAM Dual Inline Memory Module (DIMM) sockets
Support for DDR3 1333 MHz and DDR3 1066 MHz DIMMs
Support for 1 Gb, 2 Gb, and 4 Gb memory technology
Two independent memory channels with interleaved mode support
Unbuffered, single-sided or double-sided DIMMs (Double-sided DIMMs with x16 organization are not supported.)
non-ECC memory
Minimum recommended total system memory: 512 MB
Support for 1.35 V low voltage JEDEC memory
Chipset Intel® P67 Express Chipset consisting of the Intel® P67 Express Platform Controller Hub (PCH)
Graphics Discrete graphics support for PCI Express 2.0 x16 add-in graphics card
Audio 10-channel (7.1 + 2) Intel High Definition Audio via the Realtek ALC892 audio codec
Peripheral Interfaces
• Two USB 3.0 ports are implemented with stacked back panel connectors (blue)
• Fourteen USB 2.0 ports:
— Six ports are implemented with stacked back panel connectors (black)
— Eight front panel ports implemented through four internal headers
• Two Serial ATA (SATA) 6.0 Gb/s interfaces through the Intel P67 Express Chipset with Intel® Rapid Storage Technology RAID support (blue)
• Four SATA 3.0 Gb/s interfaces through the Intel P67 Express Chipset with Intel Rapid Storage Technology RAID support:
— Two internal SATA ports (black) — One internal eSATA port (red) — One back panel eSATA port (red)
• Two IEEE 1394a ports:
— One port via a back panel connector & one port via an internal header for front panel cabling
Expansion Capabilities
• One PCI Express 2.0 x16 add-in card connector
• Two PCI Express 2.0 x1 add-in card connectors
• Three Conventional PCI bus connector
BIOS • Intel® BIOS resident in the SPI Flash device
• Support for Advanced Configuration and Power Interface (ACPI), Plug and Play, and SMBIOS
LAN Support Gigabit (10/100/1000 Mbits/s) LAN subsystem using the Intel® 82579V Gigabit Ethernet Controller

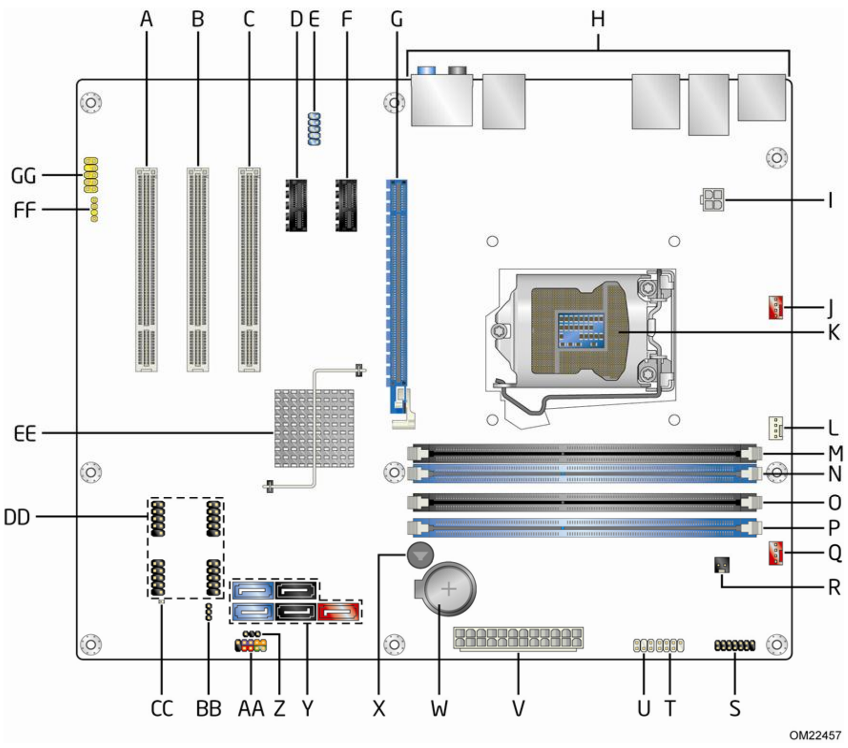
Supported Memory Configurations

DIMM Capacity	Configuration	SDRAM Density	SDRAM Organization	Number of SDRAM Devices
512 MB	SS	1 Gbit	64 M x16/empty	4
1024 MB	SS	1 Gbit	128 M x8/empty	8
1024 MB	SS	2 Gbit	128 M x16/empty	4
2048 MB	DS	1 Gbit	128 M x8/128 M x8	16
2048 MB	SS	2 Gbit	128 M x16/empty	8
4096 MB	DS	2 Gbit	256 M x8/256 M x8	16
4096 MB	SS	4 Gbit	512 M x8/empty	8
8192 MB	DS	4 Gbit	512 M x8/512 M x8	16

Note: “DS” refers to double-sided memory modules (containing two rows of SDRAM) and “SS” refers to single-sided memory modules (containing one row of SDRAM).

Processor	Processor Frequency	Intel HD Graphics	Cache
CORE 3			
I3-2120	3.30 GHz	Yes	3 MB
I3-2105	3.10 GHz	Yes	3 MB
I3-2100T	2.50 GHz	Yes	3 MB
CORE 5			
I5-2500K	3.30 GHz	Yes	6 MB
I5-2400	3.10 GHz	Yes	6 MB
I5-2300	2.80 GHz	Yes	6 MB
CORE 7			
I7-2600	3.40 GHz	Yes	8 MB
I7-2600K	3.40 GHz	Yes	8 MB
I7-2600S	2.80 GHz	Yes	8 MB
XEON			
E3-1260L	2.40 GHz	Yes	8 MB
E3-1220L	2.20 GHz	No	3 MB

A	PCI Conventional bus add-in card connector
B	PCI Conventional bus add-in card connector
C	PCI Conventional bus add-in card connector
D	PCI Express x1 bus add-in card connector
E	IEEE 1394a front panel header
F	PCI Express x1 bus add-in card connector
G	PCI Express x16 bus add-in card connector
H	Back panel connectors
I	Processor core power connector (2 x 2)
J	Rear chassis fan header
K	LGA1155 processor socket
L	Processor fan header
M	DIMM 3 (Channel A DIMM 0)
N	DIMM 1 (Channel A DIMM 1)
O	DIMM 4 (Channel B DIMM 0)
P	DIMM 2 (Channel B DIMM 1)
Q	Front chassis fan header
R	Chassis intrusion header
S	Low Pin Count (LPC) Debug header
T	Consumer IR emitter (output) header
U	Consumer IR receiver (input) header
V	Main power connector (2 x 12)
W	Battery
X	Piezoelectric speaker
Y	SATA connectors (5)
Z	Alternate front panel power LED header
AA	Front panel header
BB	BIOS Setup configuration jumper block
CC	Standby power LED
DD	Front panel USB 2.0 headers (4)
EE	Intel P67 Express Chipset
FF	S/PDIF out header
GG	Front panel audio header



TEST PAPER STYLE QUESTIONS

1. On a PC motherboard, the faster chips are nearest the processor. Either explain or use a diagram to show how the main chips and buses are connected, and what their relative speeds are.
2. In a typical PC, there is no direct connection between the processor and I/O peripherals. Explain, using a diagram if necessary, how this connection is done, and why it is not practical to directly connect, say, a disk drive to the processor.
3. Explain why the Graphics and Memory Controller Hub chip on a PC motherboard has to be physically close to the processor, while the I/O hub can be further away.
4. In modern PCs, many of the buses have changed from parallel to serial data connections. Explain why this has happened.
5. Briefly describe the function of the main hub chip(s) that make up the backbone of a PC motherboard.
6. Describe one way in which the speed of communications between the processor and memory is being improved on current PC motherboards.
7. Describe how the motherboard of a multi-core PC differs from the motherboard of a PC with multiple separate processors (such as Xeon or Opteron chips).
8. Many of the components that make up the motherboard are interchangeable, for instance if you want to upgrade a PC. Which are the main components that are fixed?
9. Describe the advantages and disadvantages of the big.LITTLE style of processor as used in the Galaxy S4 phone.
10. Describe two ways in which a typical processor chip on a mobile platform (phone, tablet etc.) differs from that of a desktop PC.

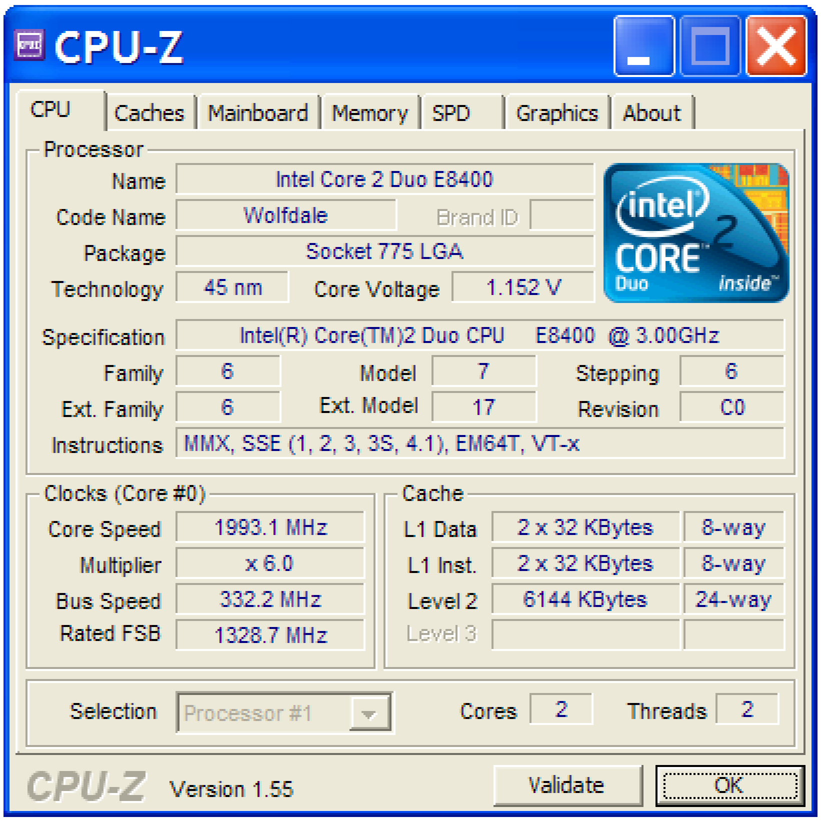
PRACTICAL 2: INVESTIGATING THE MAIN FEATURES OF THE PC SYSTEM

INTRODUCTION

The properties of the computer you are sitting at are set by a number of features. In the hardware, it depends on the type of processor the PC has, but also the chipset that controls most of the communication with the rest of the PC's facilities. The operating system too (Windows, Linux) will also determine a lot of the properties. The purpose of this lab is to investigate the hardware: the second half of this module will be concerned with the operating system. We will start with the processor, and then work out to the peripherals.

THE PROCESSOR

In order to qualify as a PC (and be able to run Windows) the processor must belong to the Intel family of processors that started with the 8086, then developed into the 80186, 80286 and eventually the Pentium processors we have now. There are many versions of this architecture, sometimes referred to as the x86 architecture, though Intel prefers to use IA32 or IA64 (for Intel Architecture 32 or 64 bit). To find out about the specific chip, we are going to use a free utility: CPU-Z This uses a specific assembly language instruction: CPUID to query the processor. Launch CPU-Z (the lecturer will explain how) and you should get a display similar to the following:



Processor
Name: Intel Core 2 Duo E8400
Code Name: Wolfdale
Brand ID:
Package: Socket 775 LGA
Technology: 45 nm
Core Voltage: 1.152 V
Specification: Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz
Family: 6
Model: 7
Stepping: 6
Ext. Family: 6
Ext. Model: 17
Revision: C0
Instructions: MMX, SSE (1, 2, 3, 3S, 4.1), EM64T, VT-x

Clocks (Core #0)
Core Speed: 1993.1 MHz
Multiplier: x 6.0
Bus Speed: 332.2 MHz
Rated FSB: 1328.7 MHz

Cache
L1 Data: 2 x 32 KBytes, 8-way
L1 Inst.: 2 x 32 KBytes, 8-way
Level 2: 6144 KBytes, 24-way
Level 3:

Selection: Processor #1
Cores: 2
Threads: 2

CPU-Z Version 1.55

PROCESSOR NAMES

There are so many versions of the Pentium chips that naming becomes quite complicated. In my desktop, the processor is called an 'Intel Core 2 Duo E8400' The term 'Core' refers to the broad division of architecture (the previous class was the Netburst). It's a terrible name, because it used to refer to the number of cores: Duo being two obviously. However, it has just become a name for the family: current newer members being Core i3, Core i5 & Corei7.
What is the Name of the processor on your PC?:

____

The 'technology' is the typical line size on the chip. What is this for your PC, and what voltage does the core run at?

____

You can use Wikipedia to see more details of the specific chip on your PC. Use it, or Intel's web pages, to find out what the Intel 'Tick-Tock' process is:

____

____

The precise version of the processor is also given by values in fields such as the family, model and so on.

INSTRUCTIONS SETS

Early processors had very small instruction sets. As they have expanded, new categories of instructions have been added. Obviously, software can only use the facilities of the processor that it is running on. The following categories of instructions are probably available on your processor; use the web to find out what they are for:

MMX____

SSE____

EM64T____

VT-x____

CLOCK

In the specification, what is the clock rate of your core?_______________________

Is the Core speed the same, and if not why not? ____________________________

CACHE

There is a summary of the cache information on the first tab, but if you go to the cache tab you will get more information on the cache memory. There will be more on the cache later in the course, but for the mean time, find out: How many levels are there on your processor, and how much is at each level?

____

Why is the top level typically listed as X2 (or perhaps X4)?

____

What is the difference between I- Cache and D-Cache?

____

MAINBOARD

The details of what memory and peripherals can be used with the PC depend largely on the chipset. If you go to the Mainboard tab and find the chipset, you should be able to use the web to get a block diagram of your chipset. What is the chipset and Southbridge (if any) on your PC?____

MEMORY

Again, this will be covered in more detail later. For the moment, find out:

How much DRAM is on your PC: ____

How many channels there are connecting to the DRAM: ____________________________________

What the DRAM frequency is: ____

How many times slower or faster the DRAM is than the processor: ____________________________

If the delay before accessing data is given approximately by the cycle time, t_{RAS} how many processor cycles happen in the time it takes to access data

____

SERIAL PRESENCE DETECT (SPD)

DRAM comes in broad families (DDR, DDR2, DDR3), but even within a family there can be variations in things like the voltage that the memory runs from, and the delay before data is ready to read (latency). Information about these features is held in a small ROM chip on the memory module so that the hardware settings can be configured to match the requirements of the memory module. During startup of the PC hardware, the BIOS performs a Power On Self Test (POST) sequence which includes reading this information from the Memory Module. It is read out via a serial link with two wires (clock and data) using a variation of the I^2C protocol known as the System Management Bus (SMB). This is a low data rate bus defined originally by Intel and used typically for communication between sections of the motherboard such as for power management.

What is the clock frequency for the memory (in MHz)? ______________________

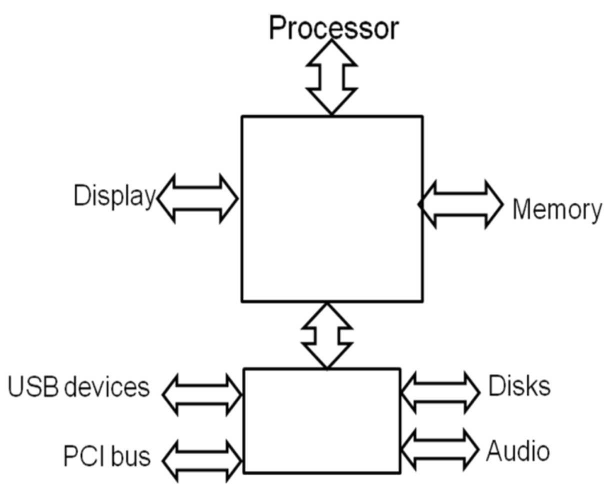
What is the maximum bandwidth (the PC2- figure)? ________________________

What is the ratio between these two? _____________________________________

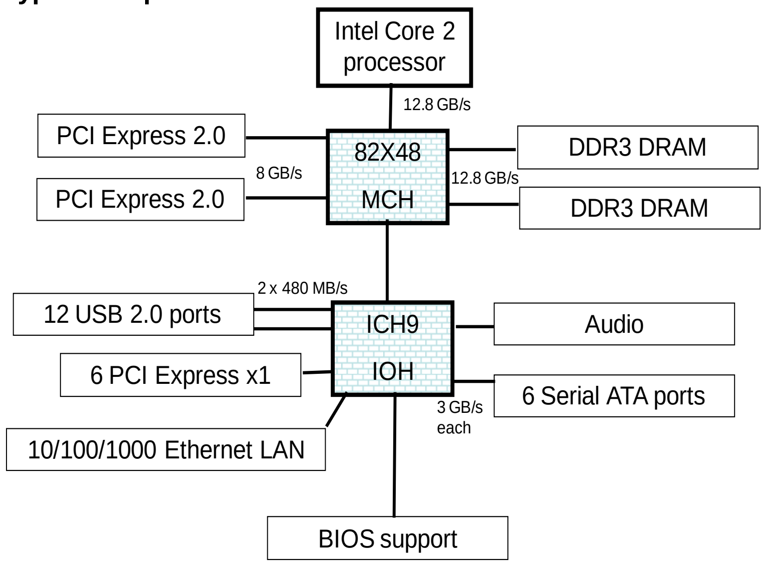
Why is there this ratio? ____

HUB-BASED ARCHITECTURES

The key part of the PC is now the chip-set that provides the connections to the buses. Initially, these were relatively simple bridges between layers (e.g. PCI to ISA). Now they have taken on more functions, become more central to the design, and come in closely coupled sets. These changes have led to what is best described as hub-based systems, such as the Intel G45, X48 & E8500 sets.



Typical Chipset



1/2/3-CHIP SETS

Chip sets vary, but usually have a Memory Controller Hub (MCH) nearest to the processor. It will connect to the main DRAM and perhaps the graphics system. The old name for this is the North Bridge. Next down is the I/O Controller Hub (ICH). This will provide the disk drive, network and USB/Firewire connections as well as the PCI (South Bridge). There may also be a Firmware hub. This may provide some slow connections, but is largely there to hold the Basic I/O System software (BIOS). Increasingly, the move is to have one hub, a 'platform/system controller hub'. This can be done either by incorporating the GMC Hub into the CPU, or by combining the two hubs into one.

QUESTIONS

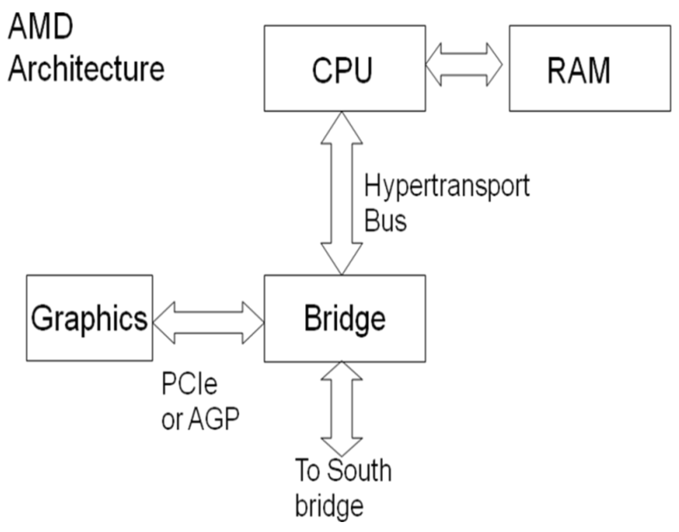
Increasingly, the USB 2.0 bus (or Firewire) is taking over from PCI. What are the advantages?

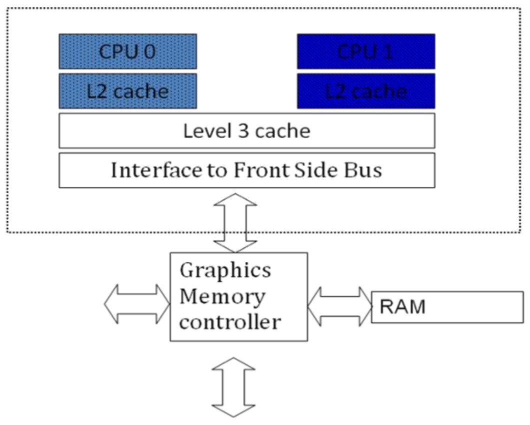
Why 'Dual-channel' RAM?

AMD MOTHERBOARDS

AMD pioneered a different approach: the memory controller is built into the CPU, so the North bridge is simpler. This has advantages in multiple processor systems: the single North bridge does not become a bottle-neck. However, with multiple core processors, the cores still share a memory controller.

By incorporating the whole of the GMC Hub into the processor, some recent Intel processors achieve a similar architecture to the AMD style: the memory is connected directly to the processor. In the old system, where the processor communicated to the GMC Hub (Northbridge), a wide variety of Front Side Buses (FSB) were used. The processor can use the DMI bus, developed for communication between the two hub chips.



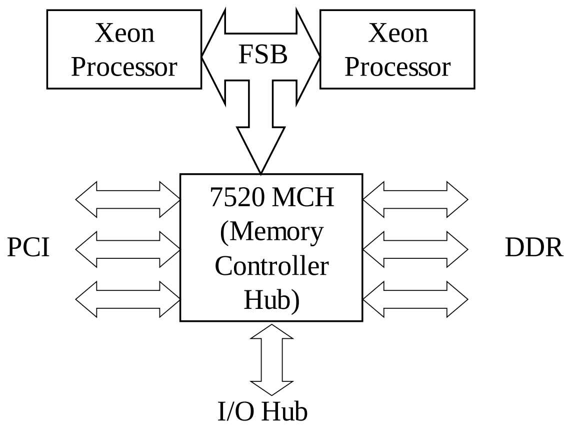


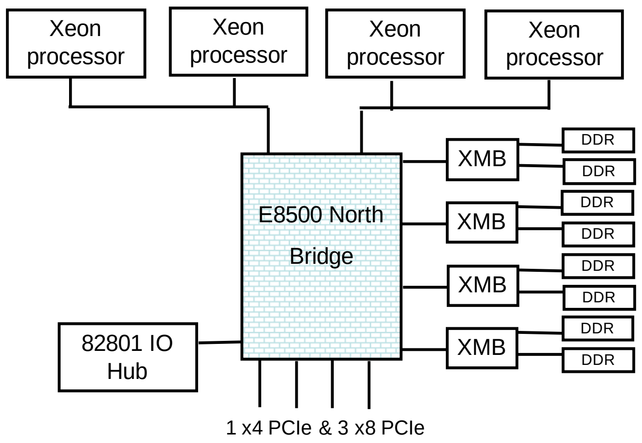
A single processor chip can now contain 2, 4 or 6 cores. Note that the connection off the chip can become the bottleneck. However, most of the time, the cores will communicate with cache, not the external RAM. A single processor chip can now contain 2, 4 or 6 cores.

MULTIPLE PROCESSOR SYSTEMS: INTEL

For better performance, it may be effective to use larger numbers of processors. In this case, the hub chips act like switches. They will connect one of a number of processors to one of a number of banks of RAM or I/O buses. The Pentium family can be connected together like this, but the number that can be connected depend on the model, and the Operating System used. The Itanium and Xeon families are designed to work well in multiple processor systems.

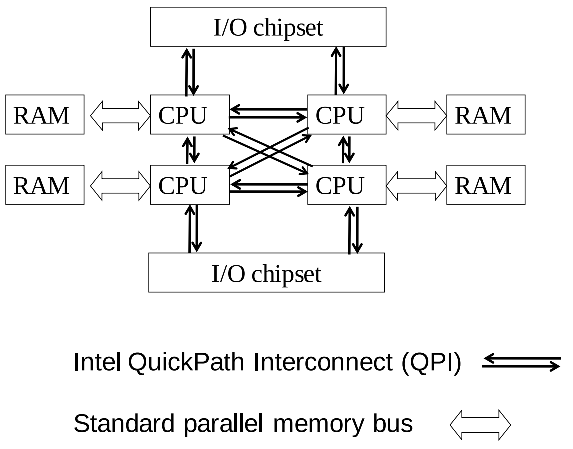
Can you spot a problem with this sort of shared bus?





E8500 4-WAY SERVER CHIPSET

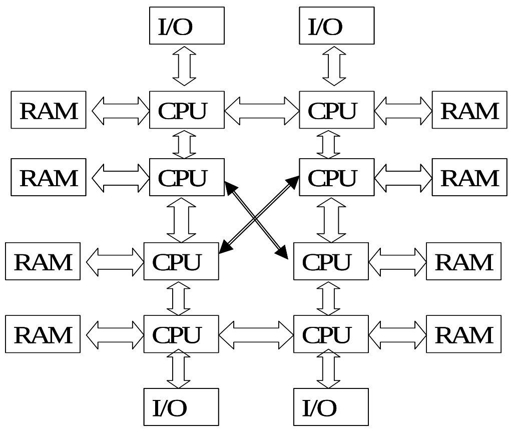
(using Dual Independent Buses)


I/O chipset
RAM
CPU
CPU
RAM
RAM
CPU
CPU
RAM
I/O chipset
Intel QuickPath Interconnect (QPI)
Standard parallel memory bus

INTEL QUICKPATH INTERCONNECT (QPI)

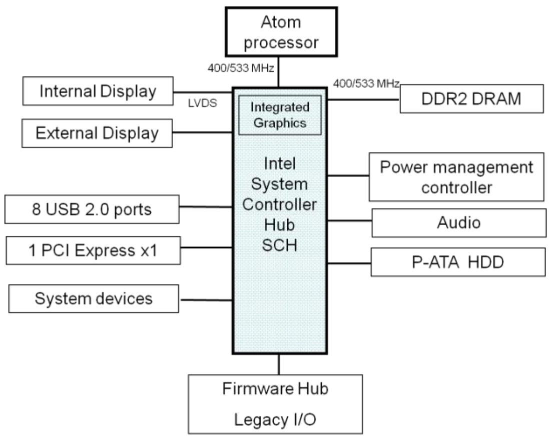
Intel have introduced a new system to replace the Front Side Bus: Quick Path Interconnect. There are three significant features to this:

- Each chip has its own memory controller, so the connection to the memory will be less of a bottle neck.
- The bus is a multiple-lane serial bus (a bit like PCIe). Almost anything going off chip at the highest data rates is shifting from parallel to serial.
- As well as connecting the core to the IO hub, it can be used to connect cores together.



MULTIPLE PROCESSOR SYSTEMS: AMD

Opteron 2xx series processors can be connected together in dual configurations. 8xx series processors can be connected in twos, fours or eights. In the diagram: ‘I/O’ is a bridge chip, typically connecting to PCI or PCIe.

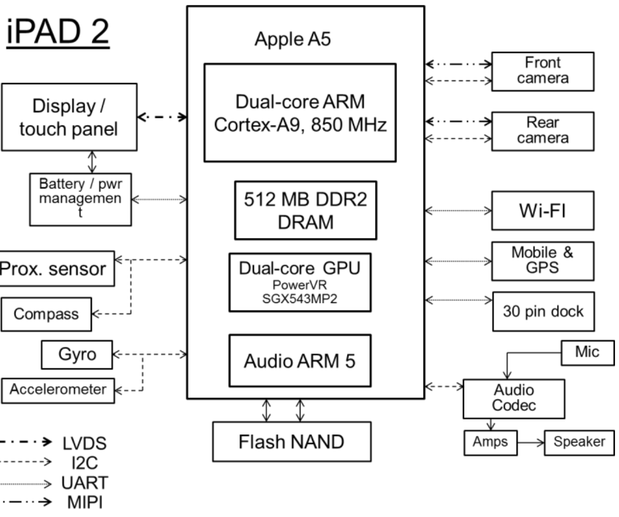


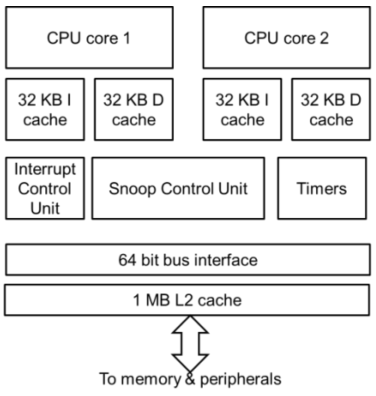
ATOM BASED SYSTEMS

The need for low-power systems for use in Netbooks with long battery life has meant a return to relatively simple single-core processors: Atoms. These run the same software as Pentiums, but only have a single core though they do Hyperthreading. The simpler systems use a System Controller Chip that combines the two chips of the bigger chipsets.

MOBILE SYSTEMS

In mobile processing (tablets, phones etc.) it is still important to have reasonable processing and graphics performance, but battery consumption becomes an important factor. ARM's processors are dominant in this area: better low-power performance than Intel's chips. ARM processors are actually designs for cores that manufacturers put into a system chip. There are numerous variants that have different levels of performance, features and power consumption. They all run the same standardised instruction set (i.e. the same software runs across the range).

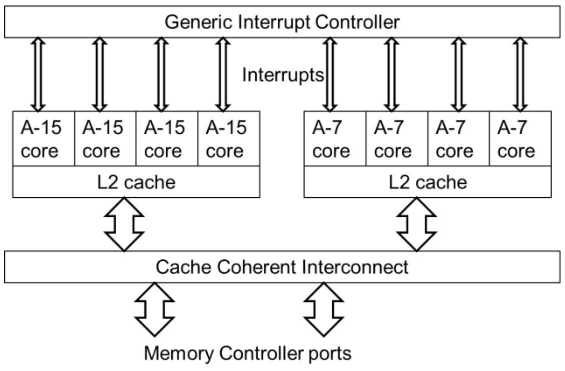




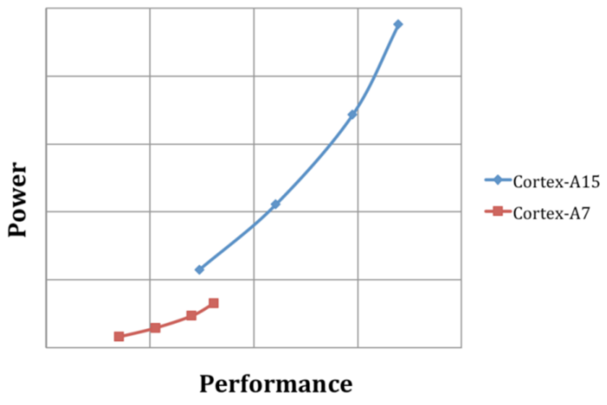
ARM A9 processor

'BIG LITTLE' PROCESSING

On a desktop PC, power consumption is not a big problem; many chips are optimised to give the highest computing performance. In small embedded systems, power consumption may be the deciding factor, and chips are optimised for low power. Many mobile applications sometimes need high performance (e.g. media decoding) but often need to conserve power. It is really hard to design a processor that is optimum for both performance and power. If a number of processors can be designed onto the same chip, it becomes practical to have processors optimised for high performance (that may not be needed all the time) and other processors that can run using much less power when demand is low. ARM are a pioneer in this area of 'big little' processing; they call it big.LITTLE processing: pairing 'big' A 15 with 'little' A 7 processors.



SAMSUNG GALAXY S4: EXYNOS OCTA-CORE 5410 CHIP



WHY DOES BIG.LITTLE WORK?

Although the ARM 7 & ARM 15 cores have quite different performance, they still run exactly the same instructions. The ARM 7 uses a relatively simple instruction decode pipeline and instructions have to be completed in strict sequence. The ARM 15 can start on more instructions every clock cycle, has a complicated decode sequence, and instructions can complete out of order. The Caches in the processors are designed differently too. However, these features only affect the speed at which instructions run; all the processors still run the same instructions. This means that tasks can be switched between high- and low- performance processors as needed: a very flexible strategy.

Are there disadvantages to this sort of processing?

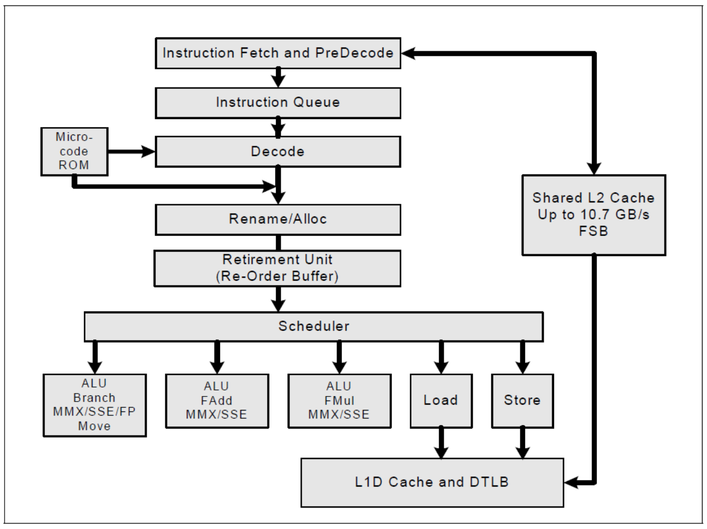
Is this significant?

which of these is this equivalent to? RAM latency Interrupt response Hard disk latency

THE PENTIUM PROCESSORS

Contents:

- The main Pentium functional units
- Pentium Registers
- Pipelining
- Instructions


Instruction Fetch and PreDecode
Instruction Queue
Micro-code ROM
Decode
Rename/Alloc
Retirement Unit (Re-Order Buffer)
Scheduler
Shared L2 Cache Up to 10.7 GB/s FSB
ALU Branch MMX/SSE/FP Move
ALU FAdd MMX/SSE
ALU FMul MMX/SSE
Load
Store
L1D Cache and DTLB

Intel Core pipeline

BACKGROUND

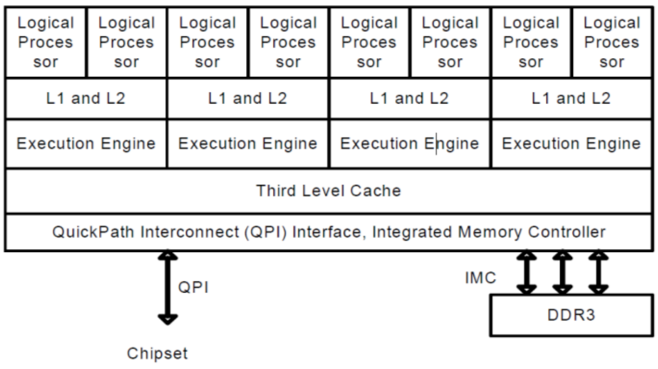
The Pentium family has its origins in the 8086/88 microprocessor that was used by IBM in the original PC of 1980. There have been many subsequent generations after the '86: '186, '286, '386, '486, and many versions of the Pentium. Each new processor has had to be able to run the software that was written for previous processors; many peculiarities have been inherited. Much of this is in Chapter 7 of Williams "Computer System Architecture".

EXECUTION CORE

The heart of the Intel Core architecture is a complex set of hardware that can work on multiple instructions at the same time. Part of this is made possible because each processor core has multiple Arithmetic & Logic Units (ALU). Some of these have specialised uses (e.g. Floating point Add or Multiply). An instruction queue can hold multiple instructions to be worked on. Several can be started at the same time: they don't need to finish in the same order. To keep the pipeline functioning, there has to be cache on the same chip.

MULTIPLE PROCESSORS ON ONE CHIP

A high-end processor now has multiple processors on one chip. The heart of the processor may well be replicated 2, 4 or more times on a single chip. This gives better performance when executing multitasking software. This needs support from the Operating System, but Windows and Linux have had this feature for many years. An example is the core i7 processor: quad core, but actually more like 8 'logical processors'.



At the heart of the system is the 'logical processor'. Each one of these is roughly equivalent to an updated version of the original 8086 processor that was in the first PCs. The logical processor contains the key features that make up the state of the processor:

- Registers to hold data while it is being worked on
- The Program Counter
- Flags to show the state

This is the part of the processor that a programmer interacts with, and is the part we will be focussing on next. Intel refers to this as the Basic Execution Environment.

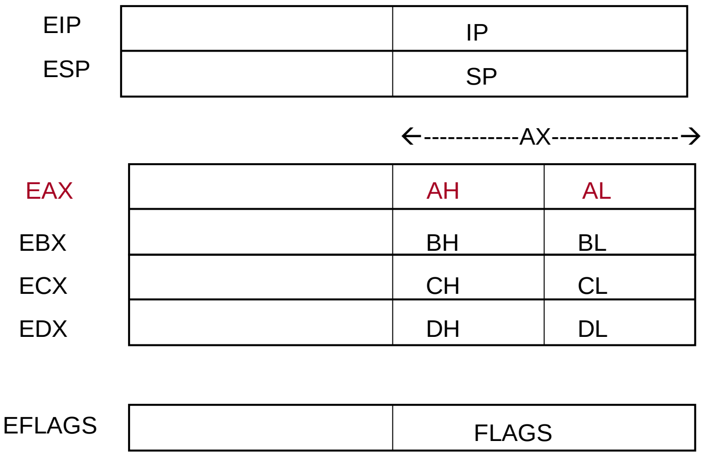
THE REGISTERS: INTRODUCTION

This is the bit that the programmer sees. To understand at a fundamental level how to program a microprocessor, it is necessary to become familiar with them. In a Complex Instruction Set Computer, such as the Pentium, many have special features:

- ECX is used as a counter.
- CS & EIP are used to point to the next instruction.

The names are often (but not always) abbreviations (EIP Extended Instruction Pointer).

When the '386 was introduced, the original 16 bit registers were extended to 32 bits, the extended registers are preceded by 'E'). There is some duplication & overlap: AX is just another name for the bottom 16 bits of the EAX register.



Some of the Pentium Registers

DATA REGISTERS

There are four general purpose registers used to hold data and integers while they are being manipulated. They can hold different amounts of data:

8 bits (AL, BL CL, DL, AH, BH, CH, DH) .
16 bits (AX, BX, CX, DX).
32 bits (EAX, EBX, ECX, EDX).

MOV AX, 1234H ; Move (hex) 1234 into AX.

ADD EBX,EAX ;Add the contents of EAX to the contents of EBX & put result in EBX.

Destination Source

POINTER REGISTERS

These are used to hold addresses so that items in memory can be located. EIP, the Extended Instruction Pointer, is used to hold the address of the next piece of code to be fetched. ESP points to the stack, an area of RAM used as a temporary store. DS is used to point to a table that defines the current data segment.

Question What would happen if you changed the contents of the instruction pointer?

STATUS WORD

This is also referred to as the Flags register. It contains a collection of individual bits, the status of each bit giving information/control:

There is a flag to show if the result of an operation was 0.
There is a flag to show if the result was +ve or –ve.

These are used to control execution:

```
SUB EAX,EBX  ;Compare two registers.
JZ next_bit  ;Jump to 'next_bit' if result is 0
…..          ;Otherwise continue at this line.
```

OTHER REGISTERS

There are a variety of special purpose registers. Some have only one task: CR3 points to the start of page tables. There are a group of 64 bit floating point registers; these are used in the manipulation of numbers: 123.456, -0.0034. In the PII, and later processors, there are also a group of 64 bit registers used in multimedia applications (the MMX registers). These are useful if the same operation has to be done to a number of data items (e.g. all the pixels in an image). (Single Instruction Multiple Data).

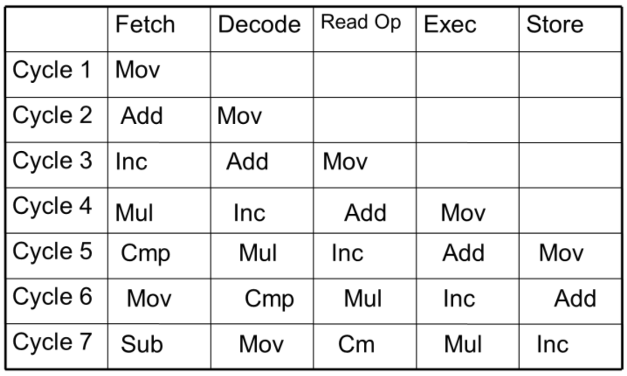
DECODE UNITS

The complex instructions have to be broken into simple operations, e.g.: ADD AX, 1234h
In English: Fetch the number in the AX register, add 1234 to it, store the result back in AX. In Pentium Microcode:

- Fetch instruction into decode unit.
- Decode it (work out what it will do).
- Read the Operand (number being operated on).
- Do the arithmetic (execute the instruction).
- Store the result in the Destination.

PIPELINING

The processor is driven by a clock that controls the internal logic; typically about 3 GHz on a new PC (3,000,000,000 ticks per second). One instruction is too complex to do in a single cycle, but they can be organised into a pipeline. In normal circumstances, an instruction finishes every clock, even though each instruction has taken 5 cycles. Needs Parallel hardware, so that one bit can be fetching an instruction, while another is decoding the previous one, and so on.



	Fetch	Decode	Read Op	Exec	Store
Cycle 1	Mov				
Cycle 2	Add	Mov			
Cycle 3	Inc	Add	Mov		
Cycle 4	Mul	Inc	Add	Mov	
Cycle 5	Cmp	Mul	Inc	Add	Mov
Cycle 6	Mov	Cmp	Mul	Inc	Add
Cycle 7	Sub	Mov	Cm	Mul	Inc

PIPELINE PROBLEMS

When code is executed in normal order, the pipeline speeds up the process. But, there are problems with branches:

```
CMP EAX, EBX
JE Next_bit
```

Depending on the numbers in EAX and EBX, execution after these two instruction could go one of two ways. Until the instruction is executed, it is impossible to say which way. The pipeline may contain half decoded instructions that are not needed.

THE STRUCTURE OF INSTRUCTIONS

I've used Assembler Mnemonics (ADD AX, BX). These are relatively easy to understand. The computer actually stores, and uses, binary numbers (machine code). The instruction ADD AX, BX is actually 03C3 in hexadecimal:

0 0 0 0 0 0, 1, 1, 1 1, 0 0 0, 0 1 1

- 000000 Is the code for an ADD instruction.
- 1 to use a register as the destination.
- 1 to use a word register (i.e. a 16 bit register).
- 11 to get both starting numbers from registers.
- 000 & 011 are the codes for the AX and BX registers.

ANOTHER EXAMPLE

MOV EAX, 12345678h

Move the hex number 12345678 into EAX. This instruction would be 48 bits long, too long to write in binary. In hex it appears as:

66 B8 78 56 34 12

66 is a prefix that means use 32 bit numbers.
B8 = 10111000; 10111 for MOV, 000 for EAX.
Note that the number itself appears in the instruction, but lowest byte first.This is called 'Little Endian'; other systems (Motorola, Sun) use Big Endian, a source of confusion.

ADDRESSING MODES

- **MOV AX, 1234h**, is an example of Immediate addressing: the number is part of the instruction.
- Transfer between registers (direct): **MOV AL,BL**
- Moving data directly to and from memory:
  - **MOV AX, [1000]** ;Move to AX the data in RAM at location 1000.
  - **MOV Total, BH** ;Move contents of BH into the variable Total.

The Pentium has many other modes used to access data structures, such as arrays. These often use a register as an index: MOV EAX, table[ESI]

QUESTIONS

What happens to the original numbers in the instruction **ADD EAX, EBX**?

What would the instruction **ADD EAX, [2000]** mean?

HIGH LEVEL LANGUAGES

Fortunately, High Level Languages (C++, Java..) shield us from the difficulties of low level code. The compiler translates each HLL instruction into appropriate Assembler/Machine Code.

```
NewTotal  =  OldTotal  +  Sum;        MOV EAX, Oldtotal
                                      MOV EBX, Sum
                                      ADD EAX, EBX
                                      MOV NewTotal, EAX
```

64 BIT PROCESSING

AMD was first to add 64 bit support to their processors. The AMD64 processors enable larger amounts of data to be manipulated at any one time, and also larger amounts of memory to be addressed. Intel followed with their EM64T. This is compatible with the AMD system. Both systems only work properly if the operating system supports it: Linux and more recent versions of windows. Any compiler used also has to support 64 bit processing.

AX = 16 bits, EAX = 32 bits, RAX = 64 bits.

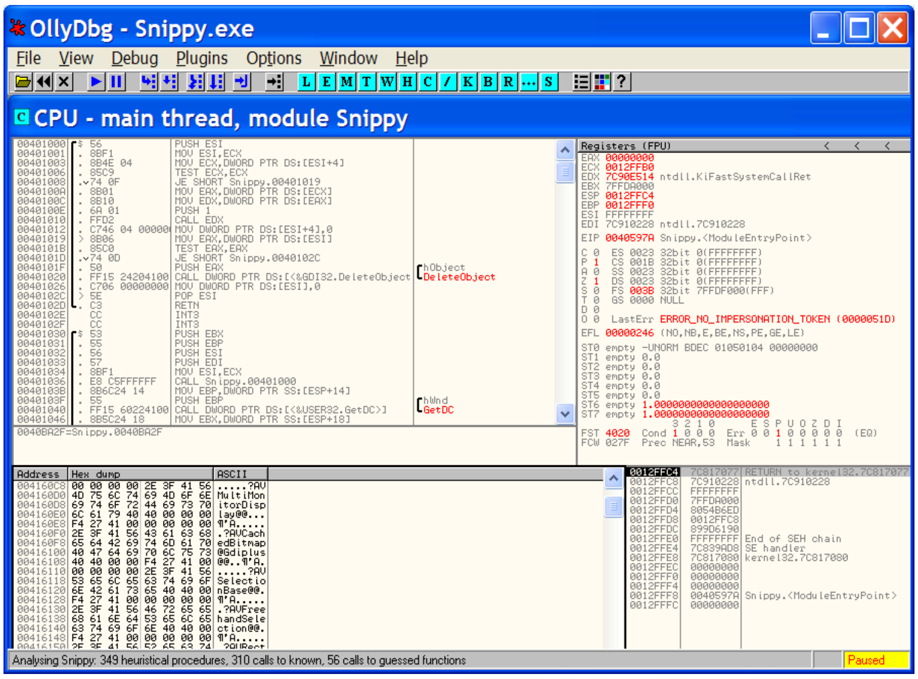
The 64 bit Instruction pointer, RIP, gives access in principle to programs of 264 bytes (roughly 16 million Terabytes).

ASSEMBLER AND SECURITY/FORENSICS

One area where assembler is still used is in security and forensics. When analysing a virus or rootkit, the source code isn't available. It may well have been written in C++, but all the analyst has available is the actual code of the virus (perhaps running on a machine or in an .EXE file). Tools like OllyDbg can capture running code or code on a disk. The display will show the assembler mnemonics.

FURTHER READING: ASSEMBLY LANGUAGE AND SECURITY/FORENSICS

One area where assembler is still used is in security and forensics. When analysing a virus or rootkit, the source code isn't available. It may well have been written in C++, but all the analyst has available is the actual code of the virus (perhaps running on a machine or in an .EXE file). Tools like OllyDbg can capture running code or code on a disk. The display will show the assembler mnemonics. Here are some examples, just to give an idea of what goes on.


OllyDbg - Snippy.exe
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S ?
CPU - main thread, module Snippy
Paused

```
00401000 ┌$ 56            PUSH ESI
00401001 │. 8BF1          MOV ESI,ECX
00401003 │. 8B4E 04       MOV ECX,DWORD PTR DS:[ESI+4]
00401006 │. 85C9          TEST ECX,ECX
00401008 │.v74 0F         JE SHORT Snippy.00401019
0040100A │. 8B01          MOV EAX,DWORD PTR DS:[ECX]
0040100C │. 8B10          MOV EDX,DWORD PTR DS:[EAX]
0040100E │. 6A 01         PUSH 1
00401010 │. FFD2          CALL EDX
00401012 │. C746 04 00000 MOV DWORD PTR DS:[ESI+4],0
00401019 │> 8B06          MOV EAX,DWORD PTR DS:[ESI]
0040101B │. 85C0          TEST EAX,EAX
0040101D │.v74 0D         JE SHORT Snippy.0040102C
0040101F │. 50            PUSH EAX                                   ┌hObject
00401020 │. FF15 24204100 CALL DWORD PTR DS:[<&GDI32.DeleteObject   └DeleteObject
00401026 │. C706 00000000 MOV DWORD PTR DS:[ESI],0
0040102C │> 5E            POP ESI
0040102D └. C3            RETN
0040102E    CC            INT3
0040102F    CC            INT3
00401030 ┌$ 53            PUSH EBX
00401031 │. 55            PUSH EBP
00401032 │. 56            PUSH ESI
00401033 │. 57            PUSH EDI
00401034 │. 8BF1          MOV ESI,ECX
00401036 │. E8 C5FFFFFF   CALL Snippy.00401000
0040103B │. 8B6C24 14     MOV EBP,DWORD PTR SS:[ESP+14]
0040103F │. 55            PUSH EBP                                   ┌hWnd
00401040 │. FF15 60224100 CALL DWORD PTR DS:[<&USER32.GetDC>]        └GetDC
00401046 │. 8B5C24 18     MOV EBX,DWORD PTR SS:[ESP+18]
0040BA2F=Snippy.0040BA2F
```

```
Registers (FPU)
EAX 00000000
ECX 0012FFB0
EDX 7C90E514 ntdll.KiFastSystemCallRet
EBX 7FFDA000
ESP 0012FFC4
EBP 0012FFF0
ESI FFFFFFFF
EDI 7C910228 ntdll.7C910228

EIP 0040597A Snippy.<ModuleEntryPoint>

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_NO_IMPERSONATION_TOKEN (0000051D)

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty -UNORM BDEC 01050104 00000000
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 1.0000000000000000000
ST7 empty 1.0000000000000000000
               3 2 1 0      E S P U O Z D I
FST 4020  Cond 1 0 0 0  Err 0 0 1 0 0 0 0 0  (EQ)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

```
Address  Hex dump                 ASCII
004160C8 00 00 00 00 2E 3F 41 56 .....?AV
004160D0 4D 75 6C 74 69 4D 6F 6E MultiMon
004160D8 69 74 6F 72 44 69 73 70 itorDisp
004160E0 6C 61 79 40 40 00 00 00 lay@@...
004160E8 F4 27 41 00 00 00 00 00 ¶'A.....
004160F0 2E 3F 41 56 43 61 63 68 .?AVCach
004160F8 65 64 42 69 74 6D 61 70 edBitmap
00416100 40 47 64 69 70 6C 75 73 @Gdiplus
00416108 40 40 00 00 F4 27 41 00 @@..¶'A.
00416110 00 00 00 00 2E 3F 41 56 .....?AV
00416118 53 65 6C 65 63 74 69 6F Selectio
00416120 6E 42 61 73 65 40 40 00 nBase@@.
00416128 F4 27 41 00 00 00 00 00 ¶'A.....
00416130 2E 3F 41 56 46 72 65 65 .?AVFree
00416138 68 61 6E 64 53 65 6C 65 handSele
00416140 63 74 69 6F 6E 40 40 00 ction@@.
00416148 F4 27 41 00 00 00 00 00 ¶'A.....
00416150 2E 3F 41 56 52 65 63 74 .?AVRect
```

```
0012FFC4  7C817077 RETURN to kernel32.7C817077
0012FFC8  7C910228 ntdll.7C910228
0012FFCC  FFFFFFFF
0012FFD0  7FFDA000
0012FFD4  8054B6ED
0012FFD8  0012FFC8
0012FFDC  899D6190
0012FFE0  FFFFFFFF End of SEH chain
0012FFE4  7C839AD8 SE handler
0012FFE8  7C817080 kernel32.7C817080
0012FFEC  00000000
0012FFF0  00000000
0012FFF4  00000000
0012FFF8  0040597A Snippy.<ModuleEntryPoint>
0012FFFC  00000000
```

```
Analysing Snippy: 349 heuristical procedures, 310 calls to known, 56 calls to guessed functions
```

Early versions of DOS & Windows used software interrupts to get access to operating system functions. A value was put into the ah register to say which operation was required. A virus might call the "write file" function, like this:

```
        mov     ah,40h
        int     21h
```

However, it is fairly easy to check for, and prevent, this sort of access by looking at the value moved into the ah register, so the next thing is to disguise which particular function is being called.

```
        mov     ah,30h      (the code for "Get DOS version")
        add     ah,10h      (add 10,   30 + 10 = 40)
        int     21h
```

(More recent versions of Windows use a different mechanism to access system calls.)

Here is a simple example just to show the flow of control in a virus that spreads by infecting up to two files:

```
        mov     counter,2       ; Set initial counter value
        ….                      ; some more code
        dec     counter         ; Decrement counter
        ….                      ; some more code to infect file
        cmp     counter,0       ; Two files already infected?
        je      stop            ; Stop if so
        mov     ah,4F           ; Otherwise proceed with the next file
        jmp     f_next1
```

Finally, here is an example (part of the Conficker worm) that patches the Windows code for the vulnerable function NetpwPathCanonicalize(). Five bytes are replaced with a jump to the worm code. At the end of the worm code, execution returns to the later stages of the unaltered code.

```
8BFF              MOV EDI,EDI                     E9 FE52AF01       JMP 01AF52FE
55                PUSH EBP
8BEC              MOV EBP,ESP
53                PUSH EBX                        53                PUSH EBX
3BDF              CMP EBX,EDI                     3BDF              CMP EBX,EDI
0F85 8EDE0000 JNZ NETAPI32. 5B8780FC              0F85 8EDE0000 JNZ NETAPI32.5B8780FC
```

Incidentally, one thing this example shows is how the length of the instruction can vary. The Jump instruction (which has a number of forms) is E9 for a jump with a 32 bit relative offset. So the whole instruction takes 5 bytes: one for the Jump, four for the amount to jump forward or back.

END OF UNIT - SUMMARY OF ACHIEVEMENTS

On completion of this unit you should understand :

- The main features of typical PC motherboards, and the options that are available to customise a PC
- Some current trends in mobile computing
- The main functional units, and assembly language, of a Pentium processor
- The way in which high level language programs are translated to the machine code instructions of a processor

Edinburgh Napier UNIVERSITY

Unit 3: Memory & RISC processors

DRAM, cache, RISC processor and the ARM family.

Student Study Material

UNIT 3: MEMORY & RISC PROCESSORS

LEARNING OUTCOMES

On completion of this unit you should understand :

- The differences between types of processor architecture (CISC, RISC)
- The main features of a typical RISC processor: the ARM family
- The different types of DRAM memory, and the main characteristics used in specifying DRAM
- Current developments in cache memory, and how cache is organised in a typical processor

RESOURCES REQUIRED

Project Software

A short assembly language program encapsulated in a C++ program (PentiumProg.cpp) to be run under a debugger, such as Visual Studio Express

Hardware

An ordinary PC for accessing pdf files

TUTORIAL 3: PENTIUM ASSEMBLY LANGUAGE

INTRODUCTION

This tutorial complements the practical lab exercise on the same subject area. It is intended to introduce assembly language programming at a level that will help you understand the way in which a processor works. The tutorial is divided into two sections. Due to the vagaries of lab scheduling, I do not know if you will have done the lab by the time you do this tutorial. So, section A is introductory, and repeats much of the lab material. If you have not yet done the lab, you will need to work through this section A. If you have done the lab, and are happy that you understand that, then you can skip to section B.

SECTION A: ELEMENTS OF ASSEMBLY LANGUAGE

Assembly Language Instructions

These are written in short form using 'mnemonics': abbreviations that are supposed to remind you of the instruction's purpose. Some (MOV, MUL, ADD) are fairly obvious in meaning. Some are less obvious, but are usually an abbreviation for something (e.g. JGE means **J**ump if **G**reater than or **E**qual to). They vary for different processor families, and I am going to refer to the instructions for the Pentium family (essentially these are the same for every member of the family from the 80386 through all '486, P1,,P4).

Registers

These are used to store data values that are currently being worked on. The Pentium family has a particularly complicated register structure due mostly to inherited 80x86 features. For instance, the main accumulator can be referred to as AL (the bottom 8 bits), AX (the bottom 16) or EAX, (all 32). The full EAX register can hold 32 bits, usually written as 8 hex digits (preceded by 0x to denote hexadecimal). What would the contents of the Al register be after the following?

```
        MOV    AL,0X80
        ADD    AL,0X1F
```

What would the contents of the EAX register be after the following?

```
        MOV    EAX,0X1234567
        MOV    BL,0X10
        ADD    AL,BL
```

Source & Destination

An instruction, such as ADD AL,BL takes numbers from two source registers, manipulates them, then stores the result in the destination register. The Pentium always has the destination as the first register after the instruction (e.g. MOV dest, src). Note that the original contents of the first register are lost. Other processors may list the destination second. A recent trend (e.g. Itanium IA-64, Power PC) is to specify three registers: 2 source, and one destination. This leaves the contents of the source register unchanged. ADD AX,0X1234 is a valid instruction, so what would be wrong with this instruction?

```
                        ADD 0X1234,AX
```

Addressing Modes

For the Pentium family, data moves always involve at least one register. However, the source or destination of the data could be from different places , accessed in a variety of ways called addressing modes. You have already seen examples of addressing modes. In an instruction such as MOV AX, 0XFFFF, the data that is to be moved into the register is contained in the instruction itself (this is known as immediate addressing). It can only be used for data sources. A second mode of addressing is to use register-to-register addressing: MOV AL,BL. The simplest way of getting data from variables in RAM is to give their address in square brackets: MOV AX,[0X9988] means go to location 9988, fetch the contents, and put them in AX. Subtle point: AX is the name of a 16 bit register, so this instruction fetches 16 bits by getting 8 bits from location 9988, then another 8 bits from 9989. Likewise, MOV [0X9988],AL would take the 8 bits from the AL register and store them in RAM at location [0X9988]. The addresses in RAM can be large (8 hex digits). I will often use simple examples where I have omitted leading zeroes. The previous address would actually come out of the processor as 0X00009988, or (in binary) 00000000000000001001100110001000. Data locations in RAM can also be given names; this is essentially what a variable is. So, a valid instruction could be: MOV EBX,total where 'total' is presumably some location in RAM.

What would be the effect of the instruction: MOV [0XABCDEF00],EAX

Conditional branches

A section of code can always jump to another bit using the JMP instruction (e.g. JMP End, where End is a label given to another bit of code). However, a processor has to be able to make decisions, and this is usually done using conditional branches or jumps. This is done in two stages. First something is done that will affect the flags (a collection of bits that are set or cleared under various conditions). Then a conditional branch is inserted. This tests the value of a particular flag, for instance:

```
    CMP   EAX,EBX        ;Compare the contents of the two registers
    JE  Same             ;Jump if Equal to the code labelled Same
        ….               ;Continue here if not the same
```

Note that if the contents of EAX and EBX were not the same, execution would continue with the instruction following the JE instruction. The Pentium has a number of conditional jumps, examples are: JE, JNE, JG, JLE (Jump if Equal, Jump if Not Equal, Jump if Greater than, Jump if Less than or Equal). There are several flags. A couple of important ones are the Zero flag (set to 1 if the result is zero), and the Carry flag (set to one if the result overflows the register, meaning that there is a bit to be carried). For instance:

```
    SUB   EAX,0X10       ;Subtract 10 from EAX
    JZ  Next             ;Jump to Next if the result is zero
```

Looping

On many processors, this is done by doing a conditional jump back in the code. The Pentium is unusual in having a specific instruction, LOOP, that uses the Count register (ECX) as a loop counter:

```
        MOV AL, 0X00     ;Clear AL
        MOV ECX, 0X08    ;Set the counter to 8
Here:   ADD AL, 0X02     ;Add 2 to AL
        LOOP Here        ;Loop back
```

Each time the LOOP instruction is encountered, the contents of ECX are decremented. If the result is not zero, the LOOP goes back to the label. Once ECX reaches zero, the LOOP is ignored, and execution continues on the following line. This is very similar to a 'for' loop in a High Level Language. What is the contents of AL after the above instructions are executed?

SECTION B

1. What will be the contents of the al and bl registers after the following code has executed:

```
        mov   al,0x10
        mov   bl,0x08
        Sub   al,bl
```

A. al = 0x02;   bl = 0x08;
B. al = 010x;   bl = 0x02;
C. al = 0x10;   bl = 0x08;
D. al = 0x08;   bl = 0x10;
E. al = 0x08;   bl = 0x08

2. What will be the contents of the al and bl registers, and the Zero Flag (ZR) after the following code has executed.

```
        mov   al,0xff
        mov   bl,0xff
        sub   al,bl
```

A. al = 0x00;   bl = 0xff;   ZR = 1;
B. al = 0x00;   bl = 0x00;   ZR = 0;
C. al = 0xff;   bl = 0x00;   ZR = 0;
D. al = 0xff;   bl = 0x00;   ZR = 1;
E. al = 0x00;   bl = 0xff;   ZR = 0;

3. What will happen after the following code has executed.

```
        mov   al,0xff
        mov   bl,0xff
        sub   al,bl
        jz    Next_bit
```

A. bl and al will contain 0xff, the conditional jump will take place
B. The zero flag will be cleared to 0, the conditional jump will take place
C. The zero flag will be set to 1, the conditional jump will not take place
D. The zero flag will be cleared to 0, the conditional jump not will take place
E. al will contain 0x00, the conditional jump will occur

4. What will al contain after the following code has executed.

```
            mov   al,0x10
            mov     ecx,0x0003
    Here:   add     al,0x05
            loop    Here
```

A. 0x15
B. 0x25
C. 0x0f
D. 0x1f
E. 0x10

5. What will the contents of ax be after the following code has executed:

```
mov ax,[2000]
mov bx,[1000]
add bx,ax
```

A. 2000
B. 3000
C. The same as the contents of memory location 2000
D. Twice the contents of memory location 2000

6. Explain what is happening in each of the following lines of code:

```
mov     ax,0xf1aa
mov     bx,0x2b03
sub     ax,bx
jz      finish
```

7. In the following lines of code, explain what size of numbers are being manipulated, and what happens at the conditional jump instruction.

```
Line1:  sub     rax,rbx
        jz Line1
Line2:  jmp last_section
```

SECTION C. ADVANCED

The next section shows some examples where I have taken the output of the C++ compiler to show the assembly language that is produced by the compiler. The lines of C++ code are shown in bold, the assembly language in ordinary font. This is here purely for interest. I have added comments: see if you can follow what is going on.

1. Initialising the variables at the start of the program

```
     signed int Var1 = 11, Var2 = 18, Result = 0;
00401009  mov        dword ptr [Var1],0Bh         /* Put value into 32 bit (dword) location pointed to by Var1*/
00401010  mov        dword ptr [Var2],12h         /* Note the instructions are quite long: 7 bytes */
00401017  mov        dword ptr [Result],0
```

2. Using a system call

```
   cookie ^= GetTickCount();                                           /* Call a System function  */
00401402  call       dword ptr [__imp__GetTickCount@0 (43A010h)]       /* Result will be returned in EAX*/
00401408  xor         eax,dword ptr [cookie]                           /* XOR the value with the value in cookie*/
0040140B  mov         dword ptr [cookie],eax                           /* Store the result in the variable cookie*/
```

3. building up the two outcomes of an if statement using jumps:

```
      if ( initret != 0 )
004014B7  cmp        dword ptr [initret],0                   /* Compare the value in the variable with 0 */
004014BB  je         _cinit+52h (4014C2h)                    /* If the variable was 0, skip to 4014C2h*/
          return initret;                                    /* If the value was not 0, do the next bit*/
004014BD  mov         eax,dword ptr [initret]                /*Put the value into EAX so it will be returned*/
004014C0  jmp        _cinit+99h (401509h)                    /*Jump to the end of the routine*/
004014C2  push        offset _RTC_Terminate (402B20h)        /*Code to do if we don't return yet*/
…………
00401509  ret                                                /*End of routine, return to caller*/
```

PRACTICAL 3: PENTIUM ASSEMBLY LANGUAGE

LEARNING OUTCOME

On completion of this exercise you should have gained an understanding of the key features of a Pentium processor by observing the execution of an assembly language program, and the effect this has on the registers. This will also help you understand how a processor can make decisions using the flags and conditional instructions.

INTRODUCTION

The purpose of this lab is to introduce the way in which instructions are executed inside a Pentium microprocessor. The internal operation of the processor will be demonstrated by stepping through a low-level program, while observing the contents of the registers. This will give an understanding of the basic operation of the microprocessor, and its abilities to do arithmetic operations, make decisions, and control the flow of program execution.

BACKGROUND

In order to be able to see what happens when a program executes inside a processor, it is necessary to have software that will display the contents of registers, and allow you to step through a program one line at a time. These features are provided by a debugger, a program designed to help debug programs as they are being developed. This handout has been written assuming that you will be using Visual C++, which has a suitable debugger built in. There are many other possible debuggers; for instance, Borland produce a stand-alone debugger (Turbo debugger) that could have been used instead of the Visual C++ system.

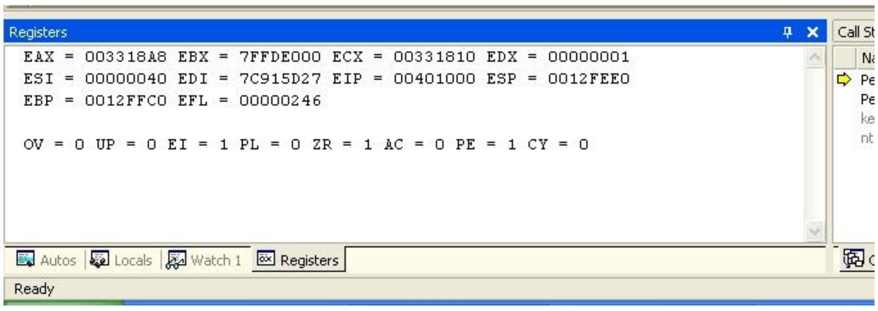
PROGRAM

I have written a single program for you to step through in order to see what is happening inside the microprocessor. In fact, the program consists of a number of separate sections, which will be explained as you work through the exercise. I have combined them into a single program, just to keep things simple (the program is listed at the back of this handout).

PROCEDURE

In principle the procedure is straightforward:

- Start Visual C++
- Load the program and build it
- Start the debugger
- Step through the program a line at a time, following the description in this worksheet.



```
EAX = 003318A8 EBX = 7FFDE000 ECX = 00331810 EDX = 00000001
ESI = 00000040 EDI = 7C915D27 EIP = 00401000 ESP = 0012FEE0
EBP = 0012FFC0 EFL = 00000246

OV = 0 UP = 0 EI = 1 PL = 0 ZR = 1 AC = 0 PE = 1 CY = 0
```

In practice there is a bit more to it than that. The main thing to remember is that it is quite possible to step through the program quickly. You will only learn what is going on if you work through it slowly, checking carefully to see what happens to the registers as you go.

REGISTERS

The debugger shows the registers in a window like the one shown. If it does not, you may have to add the register window to the view once you have started debugging. Any value that has just changed is highlighted in red. The 4 general purpose registers (EAX, EBX, ECX & EDX) are shown first. These will be the main ones to watch in this lab. Note also the Extended Instruction Pointer (EIP). This increases in value as a program is stepped through. The Extended Flags register (EFL) is also shown. For increased visibility, individual flags are 'broken out' from this: for instance, ZR is the Zero flag, CY is the Carry flag.

RUNNING THE PROGRAM

To run the program, you need to start Visual C++ (The lab demonstrator will give you specific instructions: these depend on the lab you are in). You don't need to know anything about C++, it is just being used as a way to get a debugger running. The program has been saved as PentiumProg.cpp. There are a few files that are needed, so copy across the whole of the PentiumProg folder to your local C: drive if it is not already on your PC. Again, the demonstrator will give specific instructions for this.

Having done this, it is now possible to compile the program. The quickest way is to use the **Debug..Step Into** option on the menu. This compiles the program, and executes one line of code. There is a short-cut key: **F11**. Pressing this will execute one line of code, specifically, the line that the cursor is at the start of. The only thing you might have to do, is to get the registers visible using **View..Debug Window > Registers. I**f you do not see the individual flags, you may have to right-click on the registers window and select **flags** from the optional registers that can be viewed.
Now, all you need to do is to go through the program pressing F11 to execute a line of code. Use the notes below to guide you through what is happening. If you want to go back to the start of the program, use **Debug..Restart** from the

main menu. If you want to quickly jump ahead through a section, insert the cursor where you want to go (move the cursor to the insertion point, then left-click it), then use **Debug..Run to Cursor**. You can run through the program as many times as you want.

SECTIONS OF THE PROGRAM

The program is organised into a number of sections, separated by NOPs. NOP is short for No Operation, and just means that the processor idles for a cycle. I have just used these as spacers to delimit the sections of the program.

SECTION 1: REGISTERS.

The first section shows how the registers work; in particular how 8, 16 or 32 bit quantities can be manipulated. All the instructions use the ‘accumulator’, or ‘a’ register

```
1    mov eax, 0x0             //1: registers
2    mov al,0x11
3    mov ah,0x22
4    mov ax, 0x3333
5    mov eax, 0x1234abcd
```

The first line puts a 32 bit value into eax. In hexadecimal, a 32 bit number contains 8 digits, so I could have written mov eax, 0x00000000, but the compiler is smart enough to allow leading zeroes to be omitted. Notice the prefix 0x that denotes a hexadecimal number. You should see the contents of eax change to 00000000. The second line puts an 8-bit value in al, the lower of the two 8-bit sections of the accumulator. You should see the bottom section of eax change. Line 3 does a similar thing with ah, the higher of the two 8-bit sections. The 4th line use ax, which is just the two 8-bit registers grouped into one. The strange naming scheme is a relic of the early days of this family: al, ah and ax were the three registers that were available. All the processors from the 80386 onwards also contain eax, which extends the accumulator to 32 bits. Line 5 shows a number being put into this extended 32-bit register. Once you get to the end of section 1, there are a couple of nops to execute before getting to the next section. As you execute these, the only thing that will change is eip, the extended instruction pointer.

SECTION 2: ARITHMETIC

This section sows how a few simple arithmetic instructions work. There are more complicated ones for multiplication, division, and working with floating point numbers, but these are beyond the scope of this course.

```
1    mov eax, 0x90            //2: arithmetic
2    mov ebx, 0x10100088
3    add eax, ebx
4    mov ecx, 0x20000000
o    sub ecx, eax
```

Lines one and two put 32 bit numbers into eax and ebx. Line 3 takes the contents of the two registers and adds them together. The answer goes into the first register specified (i.e. eax). You should see that after line 3, the contents of eax has changed, but not the contents of ebx. (Aside: 90 in hex is the same as 144 in decimal. Adding this to 0x10100088, which is 269484168 in decimal, should give 0x10100118, i.e. 269484312 in decimal).

Line 4 moves another number into ecx, then line 5 does a subtraction. Note that for an instruction of the form: *sub m,n* the number in *n* is subtracted from the number in *m* and the result is placed in *m*. The result is of course given in hexadecimal. (If you want to convert this number to decimal, the easiest way is to use the scientific version of the Windows calculator). Inside the microprocessor, the number is held as a 32-bit binary number. This is almost impossible to display meaningfully, so the display is given in hex (four bits make up each hex digit).

SECTION 3: VARIABLES IN MEMORY

In the previous section, all the operands were contained either directly in the instructions, or in the registers (what is called immediate or register addressing). However, for most programs, it is necessary to get numbers from variables (i.e. sections of main RAM) into and out of the processor. In the example program, I declared three variables: Var1, Var2 and Result at the beginning. The section of code actually implements the C++ statement: **Result = Var1 – Var2**;

```
1    mov eax, Var1            //2: variables in memory
2    mov ebx, Var2
3    sub eax, ebx
4    mov Result, eax
```

In the first two lines, the contents of the variables are copied from RAM memory into the registers of the processor. If you were observant you might have spotted that the numbers appear differently in the registers. This is because I put starting values in the variables right at the beginning of the program. The values I used, 11 and 18, were in decimal. Inside the processor, they are displayed in hex: 0B & 12. In line 3, ebx is subtracted from eax, and the result is put in eax. Finally, the answer is moved out of the processor’s accumulator register and written to the variable in RAM. There is also a variable window that shows the contents of the variables directly (you may have to use **View..Debug Windows > Variables** to see this). The subtraction, 18 – 11, gives the correct result: -7. Do you understand why this is represented as FFFFFFF9 within the processor? If you are unsure on this, the CS1 notes covered the appropriate number systems.

SECTION 4: FLAGS

This section does a simple bit of arithmetic in order to show how the flags give information about results of operations.

```
1   mov eax, 0x90000000        //3: flags
2   mov ebx, 0xa0000000
3   add eax, ebx
4   sub eax, 0x30000000
```

Lines 1 and two load some starting numbers into registers eax and ebx. Line 3 adds the two numbers together and puts the result into eax. However, the result of adding the two numbers should be 0x130000000, which is 33 bits long, too big to fit in the register. The bottom 32 bits are held in eax, and the Carry flag is set to one to show that there is a bit to carry. This bit is quite difficult to see, and you may have missed it. If you are not sure what happened, go through it again. While you are executing some of the instructions, (such as the simple *mov* instructions) the flags do not change. When an arithmetic instruction is executed, the flags may change depending on the result of the operation. They can then be used by later sections of the program (for instance to allow for the carry).

SECTION 5: DECISIONS

I have split this part into two. The idea here is to see how decisions can be made by the microprocessor. The format is always the same: do something that will affect the flags, then branch one way or another depending on the contents of the flags. The difference between the two parts is that in one, the branch is taken, in the other, it is not.

```
1           mov al, 0x11        //5a: decisions
2           mov bl, 0x11
3           sub al, bl
4           jz _EQ
5           mov cl, 0x00
○           nop
7   _EQ:    mov cl, 0xff
```

The first two lines move numbers into registers. Line three does a subtraction. As the numbers were the same, the result (in al) is zero, so the Zero flag is set (you should see ZR =1). Line 4 is the one that makes the decision. The instruction, jz, is short for ‘Jump if Zero’. As the zero flag is set, the jump is made to the label _EQ. In other words, lines 5 and 6 are skipped, and line 7 is executed after line 4. If the numbers had been different, the answer would not have been zero, and lines 5 & 6 would have been executed.

```
1   _EQ:     mov cl, 0xff
1.           nop
2.           mov al, 0x11            //5b: decisions
3.           mov bl, 0x22
4.           sub al, bl
5.           jz _EQ
6.           mov cl, 0xaa
```

The next section really starts at line 3 above. Two numbers are loaded into registers and subtracted. As the numbers were different, the result is not zero, so the zero flag is cleared. Then, in line 6, the jump is not made, and execution continues normally at line 7. If the jump had been made, the processor would have gone back to line 1 (where I inserted the label _EQ). This should give an idea of how loops can be set up. Note that there are more conditional jumps than the one I have shown (jump if not zero, jump if equal, jump if greater than, jump if equal, jump if less than..).

SECTION 6: LOOPING

The final piece of code demonstrates the use of a special instruction (loop) to perform a repeated loop, rather like a ‘for’ loop in a high level language.

```
1           mov eax, 0x0        //6: looping
2           mov ecx,  0x04
3   _AA:    add eax, 0x03
4           loop _AA
```

Line 1 clears the eax register. Line 2 puts a count value in ecx (note: this is the only register that can be used in this way). Line 3 is the body of the loop: every time round the loop, 3 is added to the contents of eax. Line 4 is the decision part of the loop. The special instruction ‘loop’ decrements ecx, then loops back if ecx hasn’t yet reached zero (you should be able to see ecx decreasing as you go round the loop). After four times round, the value of ecx drops to zero, the loop terminates, and execution continues normally.

CONCLUSIONS AND FURTHER WORK

Firstly, if you have reached this far, well done (assuming of course that you have actually followed roughly what is going on!). What I have tried to do is give an idea of how a microprocessor works at a fairly low level. This has not made you an expert assembly language programmer, but that was not the intention. Very rarely do people have to write assembly language nowadays. The commonest exceptions to this are when talking to hardware, or when out to get the fastest performance (such as in games programming). Obviously, nobody would write in assembly language if they could use a high-level language. What you should have is an idea of some of the features and limitations of a modern microprocessor. For instance the fundamental limitations of register size meaning that everything has to be in 8, 16 or 32 bits, and the fact that only ecx can be used as a loop counter.

If you are interested, there is a lot you can do to adapt the program to see the effects of changing the code. If you stop the debugging, then you can easily alter the code. Once you have made a change, pressing F11 will bring up a box telling you that files are out of date, and asking if you want them re-built. Select yes, and the debugger will re-compile and start the new program. Try a simple change first, such as altering the value stored in a register, or the actual register that is used. For instance, in the last example, you can change the number of times that the loop is executed by changing the value that is stored in ecx (section 6, line2). For a more advanced challenge, try altering the decision examples so that the value from two variables is subtracted, rather than the immediate values in the code (section 5). Finally, you could see what happens if the 'jz' instruction is replaced with a 'jnz' instruction.

One last thing to think about. When stepping through the program, I asked you to make sure the registers were visible. In fact, we only looked at a small selection of the registers: the main data registers and flags. Other registers could be selected in the register window by right-clicking. Here is a brief description of some of them. Depending on the version of Pentium on your PC you may seem a slightly different selection:

Segment: These enable segmented virtual memory systems. They were used in IBM's OS-2 operating system, but are not really used in Windows or Linux, which both used paged virtual memory systems.

Floating point: These registers can be used for floating point operations.

MMX: 'MultiMedia eXtension' registers. This is actually another name for some of the floating point registers that can be used for multimedia instructions. They are large registers and can hold, for instance, several pixel values in one register.

SSE, SSE-2 etc. 'Streaming SIMD Extensions'. Here SIMD stands for: Single Instruction, Multiple Data. Similar to MMX, the idea is that a register can hold several values (pixels, sound samples…). The same instruction can then be applied to all the values. Very useful for image processing, sound processing and other multimedia applications.

EM64T: This is the Extended 64 bit register set that was first introduced by AMD and is now found on Intel processors. Registers that we have seen, such as EAX, are extended to 64 bits: RAX.

The Assembly Language Program

```
// PentiumProg.cpp

int main(int argc, char* argv[])
{
    int Var1 = 11, Var2 = 18, Result = 0;

_asm
{
        mov eax, 0x0        //1: registers
        mov al,  0x11
        mov ah,  0x22
        mov ax,  0x3333
        mov eax, 0x1234abcd
        nop
        nop
        mov eax, 0x90       //2: arithmetic
        mov ebx, 0x10100088
        add eax, ebx
        mov ecx, 0x20000000
        sub ecx, eax
        nop
        nop
        mov eax, Var1       //3: variables in
memory
        mov ebx, Var2
        sub eax, ebx
        mov Result, eax
        nop
        nop
        mov eax, 0x90000000    //4: flags
        mov ebx, 0xa0000000
        add eax, ebx
        sub eax, 0x30000000
        nop
        nop
        mov al, 0x11        //5a: decisions
        mov bl, 0x11
        sub al, bl
        jz _EQ
        mov cl, 0x00
        nop
_EQ:    mov cl, 0xff
        nop
        mov al, 0x11        //5b: decisions
        mov bl, 0x22
        sub al, bl
        jz _EQ
        mov cl, 0xaa
        nop
        nop
        mov eax, 0x0        //6: looping
        mov ecx,  0x04
_AA:    add eax, 0x03
        loop _AA

}
    return 0;
}
```

RISC: REDUCED INSTRUCTION SET COMPUTERS

Contents:

THE END OF CISC?

Complex Instruction Set Computers (e.g. Pentium) contain large processors running hundreds of possible instructions using dozens of addressing modes. Problems include the difficulty of handling variable length instructions, the fact that 80% of instructions are rarely used, and that processors were increasingly limited by external memory speeds (since the original PC, processors have speeded up by X200, memory by X20). (Most of this in Williams, chapter 21)

THE BIRTH OF RISC

During the 80s, an alternative approach arose, and proved to be successful, being widely used in machines such as Sun's SPARC based Unix workstations. Many of the best ideas have been incorporated into other processors (e.g. Pentium). The key idea is to keep the instructions simple, but run them fast. This has to be coupled with other factors, such as more sophisticated compilers that can convert high level languages into the simple RISC instructions.

SIMPLE INSTRUCTIONS

All the same length (usually 4 bytes) and do relatively simple operations. Fancy addressing modes are not supported. They don't need to be microcoded. This frees up chip space. This is used for a number of things, in particular lots of identical registers. Register files may be 256 registers long (c.f. the Pentium's 4 general registers). They are identical. Numbers can be kept in the processor, rather than continually being replaced in memory.

PIPELINING

This is actually a technique from the RISC world. Although I described it in the Pentium, the early members of the family didn't use it. Instructions typically took 4 or more clock cycles to execute. The simplicity of RISC instructions make pipelines easy to implement. RISC processors adopted pipelining before CISC processors. The SPARC processors were early adopters of multiple pipelines: 'superscalar'.

A RISC EXAMPLE: ARM

According to ARM[1], about 75% of all embedded 32 bit microcontrollers are based on ARM cores. They are used in PDAs (e.g. Blackberry, Palm), mobile phones (Ericsson, Nokia, iPhone), hand-held games machines, iPods, and many other products. Typically they are licensed as a core, then incorporated into a manufacturer's design. Their popularity is at least partly due to being very power efficient, so they are ideal for battery powered applications.

ARM FAMILIES

There have been a number of families of ARM processors since the line started in the mid 1980s. Numbering used to be simple (ARM1 ARM2…). Starting with the Cortex cores, the processors have split into three families:

- Application series: Cortex-A (Mobile phones & media devices)
- Microcontroller series: Cortex-M (Simple control applications needing I/O & interrupts)
- Real-Time series: Cortex-R (Signal processing, floating point & high performance embedded)

ARM FEATURES

Much smaller, simpler chips than typical Pentium. Fast interrupt response for handling hardware. Simple RISC instructions (about 50 machine code instructions compared with hundreds for a Pentium). All the instructions are the same size and execute (mostly) in one clock.

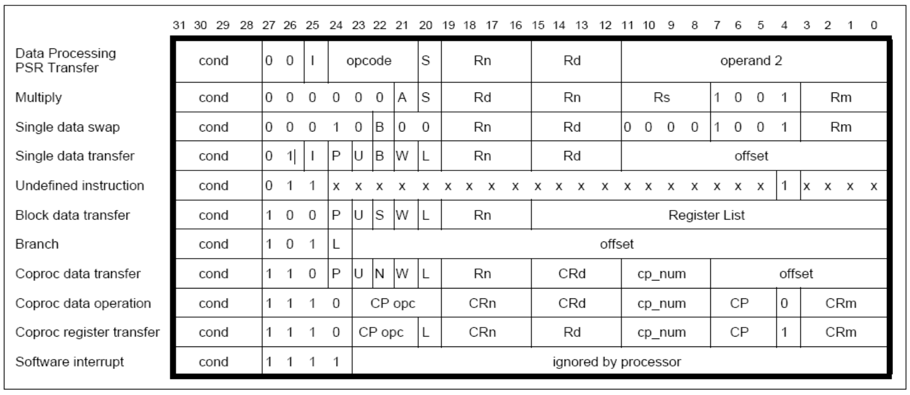
LOAD/STORE ARCHITECTURE

A complex processor like the Pentium has some compound instructions that will do things like getting a variable from memory, performing arithmetic on it and storing the result in a register. The ARM processors follow RISC principles: each instruction does one relatively simple instruction. So, for instance, it takes one instruction to get the number into a register, a second to do the arithmetic, and a third to store the result back into memory.

Example code

```
ldr R1,[R5];   Get the number from the memory address pointed to by R5 and load into R1
ldr R2,[R6];   Load R2 from mem[R6]
Add R0,R1,R2;  Add R1+R2, store result in R0
Str R0,[R7];   Store the number in R0 into mem[R7] (Note the reversed order)
```

ARM Instruction set summary



	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data Processing PSR Transfer	cond				0	0	I	opcode				S	Rn				Rd				operand 2											
Multiply	cond				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm			
Single data swap	cond				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm			
Single data transfer	cond				0	1	I	P	U	B	W	L	Rn				Rd				offset											
Undefined instruction	cond				0	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	x	x	x	x
Block data transfer	cond				1	0	0	P	U	S	W	L	Rn				Register List															
Branch	cond				1	0	1	L	offset																							
Coproc data transfer	cond				1	1	0	P	U	N	W	L	Rn				CRd				cp_num				offset							
Coproc data operation	cond				1	1	1	0	CP opc				CRn				CRd				cp_num				CP			0	CRm			
Coproc register transfer	cond				1	1	1	0	CP opc			L	CRn				Rd				cp_num				CP			1	CRm			
Software interrupt	cond				1	1	1	1	ignored by processor																							

INSTRUCTION TYPES:- THUMB

Having fixed-length instructions makes memory fetches much simpler to implement. However, the standard ARM 32-bit instruction is a bit wasteful of storage. Many ARM processors can now use the Thumb instructions. These are the commonest instructions compressed down to 16 bits. They can be used in small systems with a reduced 16-bit data bus for low cost.

INSTRUCTION TYPES: JAZELLE

Some recent ARM processors use this technology to execute Java bytecode (the intermediate code produced by Java compilers). Most bytecode instructions can be executed directly in hardware (some need some software). This is used in mobile phones for running Java applications and games.

R0	R1
R2	R3
R4	R5
R6	R7
R8	R9
R10	R11
R12	
R14/lr	

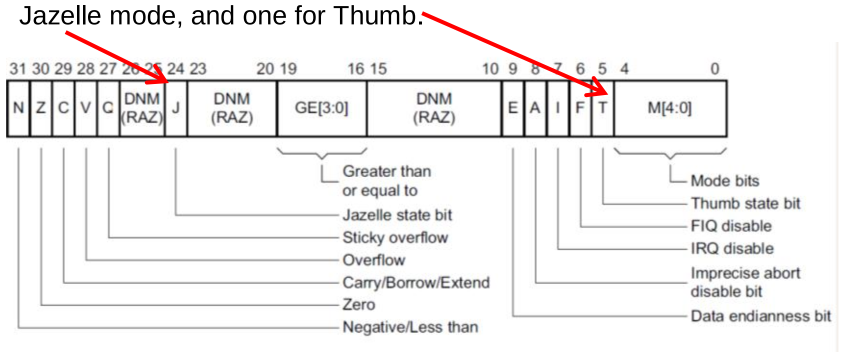
Question: How is Java executed in a traditional Pentium-based PC?

REGISTERS

There are 16 main registers (actually not many for a RISC design). All are 32 bits. R0 – R12 are available for general use. R13- R15 have special purposes. For instance R13 is the Stack Pointer and R15 is the program counter.

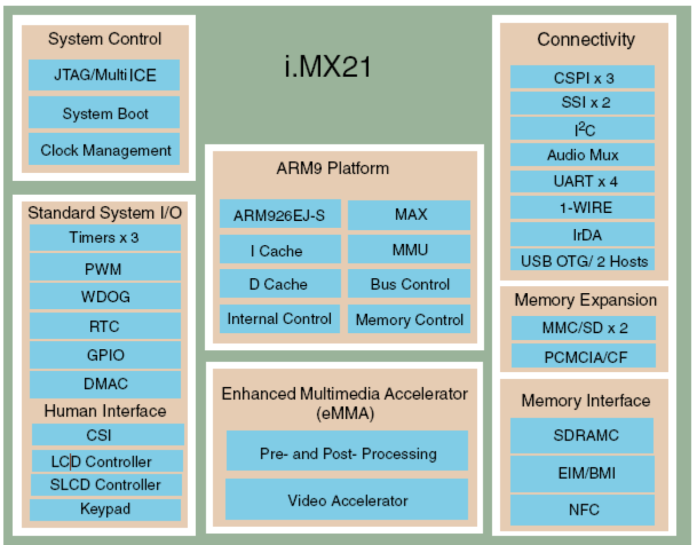
FLAGS REGISTER

On the ARM, this is called the Current Program Status Register (CPSR). It is a 32 bit register, but not all the bits are used. Some of the bits are used for traditional flags (zero, overflow, negative, interrupt enabled). Some control the mode of working. So, there is one to control if the processor is working in Jazelle mode, and one for Thumb.



Freescale Semiconductor's MC9328MX21

Note how the ARM is only the core, with application specific I/O added to the chip.



MAIN (DRAM) MEMORY

CONTENTS:

INTRODUCTION

The main memory in a PC consists of DRAM (Dynamic Ram). This is memory that has to be refreshed at regular intervals. A number of chips are typically assembled onto a module (e.g. a DIMM: Dual In-line Memory Module). 8 chips, each providing 8 bits, will give the 64 bits needed by a Pentium. A memory controller (which probably also talks to the graphics system) interfaces to the memory.

QUESTIONS

In the Apple II computer, the DRAM was clocked faster than the processor. Since then, DRAM clock rates have increased by about 9% per year, while processor clocks increased by about 50%. Why the difference?

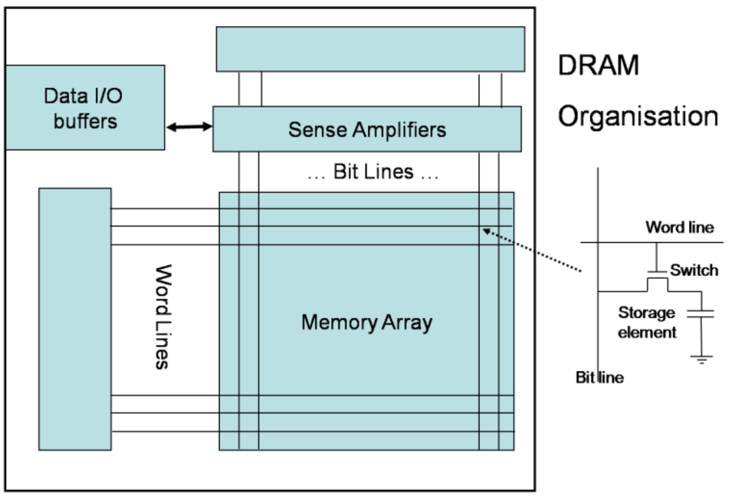
What is the purpose of DRAM (where does its data go?)

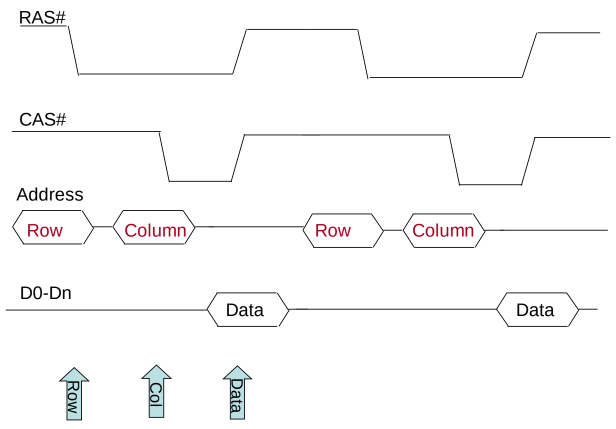
DRAM TYPES

The RAM in a PC is now a major bottleneck, working at a much slower rate than the processor. A lot of different types of RAM have been introduced to try and improve this situation. Cache is made from fast SRAM, but is too expensive for the PC's main memory, so DRAM is still used. It is organised so that the address is split into two halves: row and column.

ROW & COLUMN

A 64 M chip needs a 26 bit address. To save on pins, this is presented as two halves: a row, then a column address. Internally this is gated to the Row Decoder, then the Column Decoder.
In a basic DRAM this cycle of Row, then Column address is repeated for every cycle.

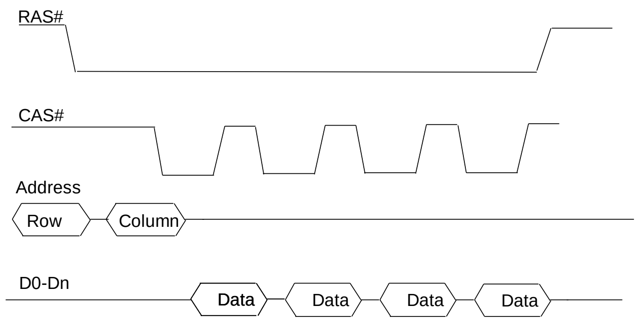




Read timing for conventional DRAM

EXTENDED DATA OUT

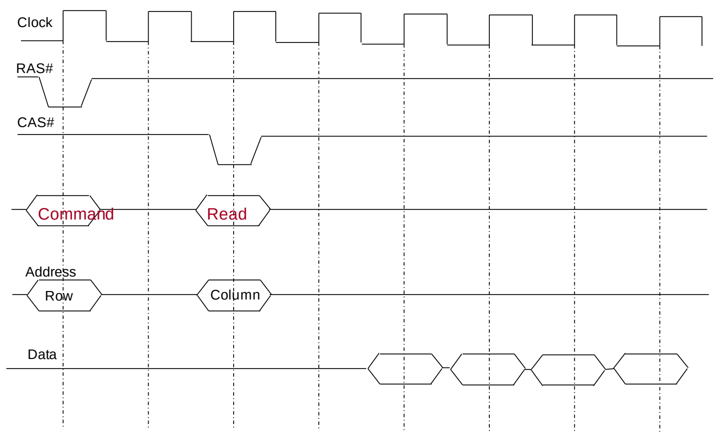
Originally, EDO DRAMs were developed to speed things up by providing one row address, then a series of column addresses, one for each block of data. This was further developed for burst, then pipelined, EDO. This uses one column address, then pulses the Column Address Strobe (CAS#) line low to get subsequent blocks of data. These burst modes assume that data is located at consecutive addresses. This works fine for filling cache lines.



Pipeline Burst EDO timing

SDRAM

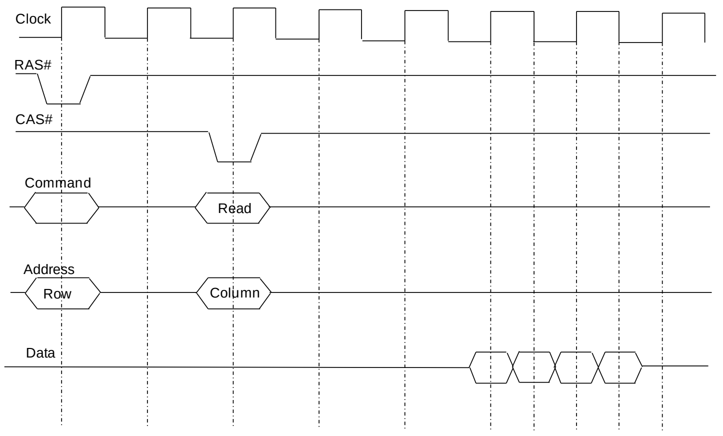
The next major shift was to synchronise the transfer to a clock: Synchronous DRAM. The SDRAM chip is normally classified by the clock rate it can operate at in MHz (e.g. 800MHZ). So, a typical memory module might be a “2GB DDR2 800” (£43 from Silicon Group, July 2010).



Synchronous DRAM read timing

DDR: DOUBLE DATA RATE

Current versions of SDRAM can send data to the processor twice per clock cycle: ‘Double Data Rate’ or DDR SDRAM. Both edges of the clock cycle are used. So, ‘DDR667’ would be used with a 333 MHz basic clock. The naming scheme is different. Now it refers to the bandwidth: PC6400 means 6400 MB/sec (800 M transfers/sec * 8 bytes). You need to remember the 64 bit bus width, i.e. a maximum of 8 bytes per transfer.



DDR DRAM read timing

LATENCY

It is not quite that case that the clock rate equals the rate that data can be read out. There is always a latency caused by the need to provide the row, then column, addresses. It may be quoted as ‘CAS latency’ (but actually there are several latencies to take into account). In a normal burst access, the first access will take perhaps 4 clocks. After that data is read on every clock. A typical burst might be 4-1-1-1, meaning the first block of data took 4 clocks, the next three each take one clock. (for the PC100, 4 blocks of data in 7 clocks = 70 nS). The CAS latency is not the only delay, but it is usually the most significant. There is also a precharge delay and a RAS to CAS delay (Trp & Trcd). Note that on the newest DDR2 memories, there may actually be more clock cycles in the latency, but the time is usually less because of the higher clock. A good memory now has a latency of about 20 nS.

DDR2

DDR2 produces data at double the clock rate, just as with DDR memory. The modules have a different number of pins, can run at faster clock rates and use less power (they run from 1.8V, DDR runs from 2.5V). DDR goes up to about DDR400, DDR2 starts at DDR2-400 and goes up to about DDR2-1300.

DDR3

This is the new version of DDR. It still uses Double Data Rate, but this is clocked at higher rates and runs using less power. It covers data clocking at up to 1600 MHz (from a clock running at 800 MHz). This compares with DDR2’s maximum of 1066 MHz. Latencies are high: can be up to 15 clocks for the CAS Latency. The voltage used (and therefore power requirement) has reduced to 1.5 Volts in DDR3.

CLOCK RATES/BANDWIDTH

The memory is now referred to, either by the DDR clock rate (DDR800), or by the maximum throughput, (PC6400). As 8 bytes are transferred at a time, the value is 8 times the ‘DDR’ clock rate. Note that, because of latency, the actual throughput will be different. Remember also that the ‘DDR’ clock rate is double the actual clock rate. For instance, DDR1600 uses an 800 MHz clock. As an example, a memory module could be classed as DDR3-1600. This means the peak transfer rate is 1600MHz, running from a 800 MHz clock. This might also be called PC3-12800 (8x the DDR figure, as 8 bytes are transferred at a time. If you want to work out the latency, (say 9 cycles) remember to use the 800MHz clock

TYPICAL SPECIFICATIONS

CT2KIT25664AC667 • DDR2 PC2-5300 • CL=5 • Unbuffered • NON-ECC • DDR2-667 • 1.8V • 256Meg x 64
CT2KIT51272AB80E • DDR2 PC2-6400 • CL=5 • Registered • ECC • DDR2-800 • 1.8V • 512Meg x 72
CT2KIT12864BA1339 • DDR3 PC3-10600 • CL=9 • Unbuffered • NON-ECC • DDR3-1333 • 1.5V • 128Meg x 64

(NB the number of clock delays are the real clock cycles. So, for the middle memory, CL = 5 refers to 5 cycles of the 400MHz clock)

RAMBUS (RDRAM)

Rambus is a very high-performance, but expensive technology. Intel and others (e.g. HP) have used it for high end servers. It has also been used in the PS2. Intel have recently dropped it from their plans. It will continue to be used in specialised applications. The PS3 uses a successor the Rambus: XDR DRAM (eXtreme Data Rate DRAM).

GDDR

Graphics cards often use specialised DRAM chips: GDDR2, GDDR3 or GDDR4. These are similar to DDR2 and DDR3. They are not identical. For instance, they aim to run faster, and may use a higher voltage to achieve this. This can lead to higher power consumption. Also they are generally more expensive.

A LATENCY EXAMPLE

A memory module is rated as being suitable for PC5333. The sheet specifies a CL (CAS Latency) of 9. What access time does this equate to?

PC5333 is the data rate in MB/sec. This is the same as DDR666.

This uses a 333MHz clock.

T= 1/f or 1/333M = 3nS.

There are 9 clocks in CL, so 27 nS for Tcl.

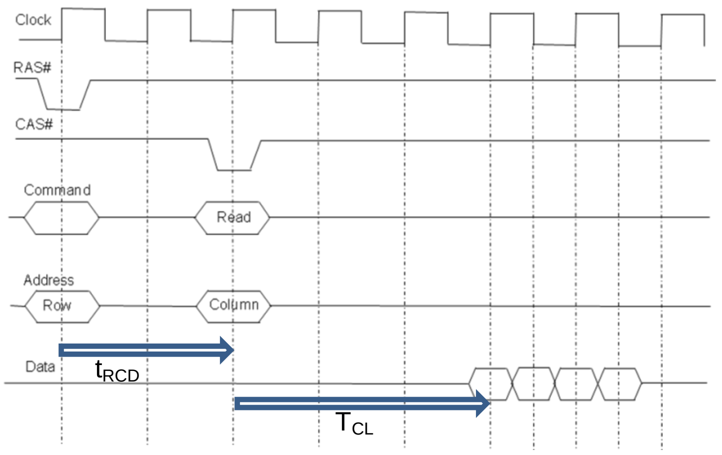
FURTHER READING: DRAM TIMINGS

DRAM timings are complicated. Generally 4 of the most important timings are given, though there are more times that you may come across. A standards body, JEDEC, has laid down how memory modules must work in order to be referred to as ‘JEDEC standard’. Typically four numbers are given, e.g. DDR3-800E memory might be referred to as 6-6-6-15. Each of these numbers gives the numbers of clock ticks for particular delays. The four figures are for t_{CL} – t_{RCD} – t_{RP} – t_{RAS}. These are abbreviations for:

• CAS latency or t_{CL}	the delay between sending a column address and the data appearing.
• RAS to CAS delay or t_{RCD}	the delay between the RAS & CAS signals
• Row Precharge time or t_{RP}	How fast access can move from one row to the next
• RAS active time or t_{RAS}	the time between a row being activated then deactivated

The clock ticks refer to the basic memory clock rate (not the double data rate that data is read from the DRAM). In the DDR3-800E memory referred to above, the actual clock rate would be 400MHz, i.e. 2.5 nS per clock tick. So, a t_{CL} of 6 means 6 x 2.5 = 15 nS

Here is what the most important of these look like on the earlier diagram:



The exact timing will depend on what is going on. A lot of the time we are at the correct row and column and the data comes out every half clock (1.25 nS in our example). We May be at the right row, but have only just specified the column and there will be a delay of t_{CL}. In the worst case we may shift to another row and column (a Random access) and we have to wait for t_{RCD} & t_{CL}. In the above example (6-6-6-15), this would be (6+6) x 2.5 = 30 nS.

On a memory module, the possible timing configurations are stored on a small permanent memory and read out via a few serial lines. This is done at boot time so the motherboard can determine what settings to use. The system is referred to as SPD (Serial Presence Detect).

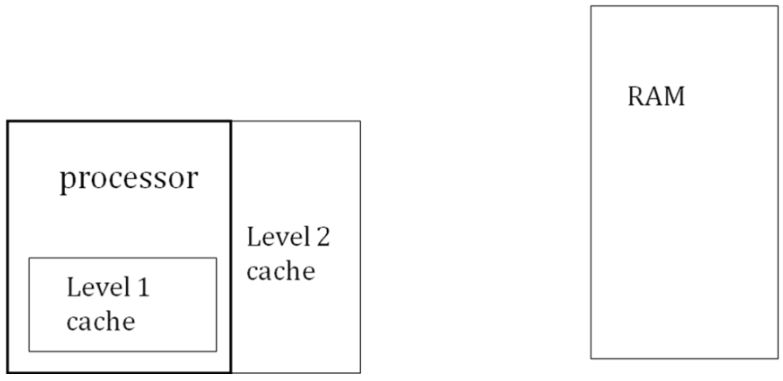
CACHE MEMORIES

CONTENTS:

- Cache organization
- Cache read and write strategies
- Cache placement strategies (associativity)

INTRODUCTION TO CACHE

Cache memory sits between the processor and Ram. With current processors, RAM is often too slow to work at the full clock speed of the processor. Many of the principles of cache memory are similar to those of virtual memory (principle of locality, replacement strategies). The key difference is that cache has to work in something like 5-10 nanoseconds. There is no time to run software; everything must be done in hardware. So, unlike virtual memory, the operating system is not involved.



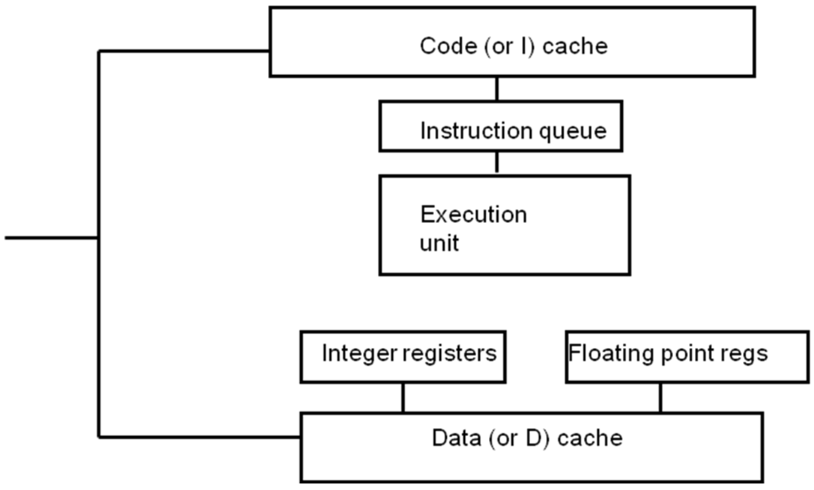
PC cache is usually organised in 2 layers:
L1 Cache is inside the processor core; it is searched first
L2 cache is at the side of the processor; it is searched if the data can't be found in the L1 cache.

Cache memory is relatively expensive SRAM. Typically the cache is one or two percent of the size of the main RAM. The principle of locality ensures that this is sufficient for at least 80-90% of the memory accesses to go successfully to cache. On a high-end processor, there may be a small amount (perhaps 64 Kbytes) in the processor chip, with a larger amount (1 Mbytes or so) as a second-level cache. This may be built into a module with the processor, or on the same chip as the processor. It used to be on the motherboard beside the processor, but the delays of sending signals down copper tracks have become relatively too large.

How fast does an electrical signal get along a pcb track in 1 nS? (approx speed about 1/3rd of the speed of light).

The cache built into a processor is often split into separate sections for data and code; the processor can then fetch from both caches at the same time (Harvard architecture). Second level cache is usually organised as one block that will store code and data.

CACHE INSIDE A PENTIUM



CACHE READS AND WRITES

When the processor needs to read data, it first tries the cache. If it is there (a hit), the data can be read in quickly. If the data is not there (a miss) the processor has to get the data from the main RAM instead (obviously this slows the processor down). The data will be copied into the cache so that it has an up to date copy.

CACHE WRITES

When data is written to the cache, there are several ways of handling the transfer to the main RAM. The simplest to implement is to "write-through" the data, i.e. write the data to the RAM every time the data is written to the cache. This means that the RAM is kept up-to-date with the cache (handy if there are other processors using the bus that might need to see the contents of RAM). The disadvantage of this is that the RAM may be written to more than is strictly necessary. For instance, there may be a whole sequence of changes to a variable or flag that the RAM doesn't need to know about if they are all happening to the variable/flag when it is in the cache.

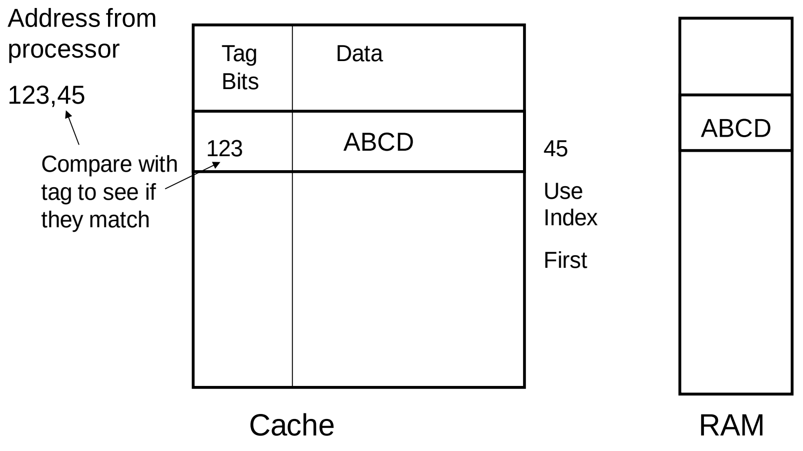
The other technique, "write-back" or "copy-back", only writes data to RAM if the cache is being emptied to make room for new data. This is more complicated to implement, but gives better performance. It does lead to possible problems if there are other processors (e.g. a DMA controller) that use the memory. They will need to know if the contents of RAM are up-to-date. In practice, the problem is not too great, since there are always more reads than writes. Question: why?

CACHE ORGANISATION

Cache is usually organised into lines/blocks, typically containing 8, 16 or 32 bytes. A line is written to/read from memory as one unit. Since cache is smaller than main memory, only some of the contents of the main memory are present in cache at any one time. Tag bits are used to identify what section of main memory is present in cache at any time. There are actually several distinct ways of organising the cache.

DIRECT-MAPPED CACHE

This is the simplest way of organising cache: use some of the address to define exactly where the line goes. Each block of main memory can be mapped into only one line of the cache. The bottom address bits will be the same in main memory and the cache.

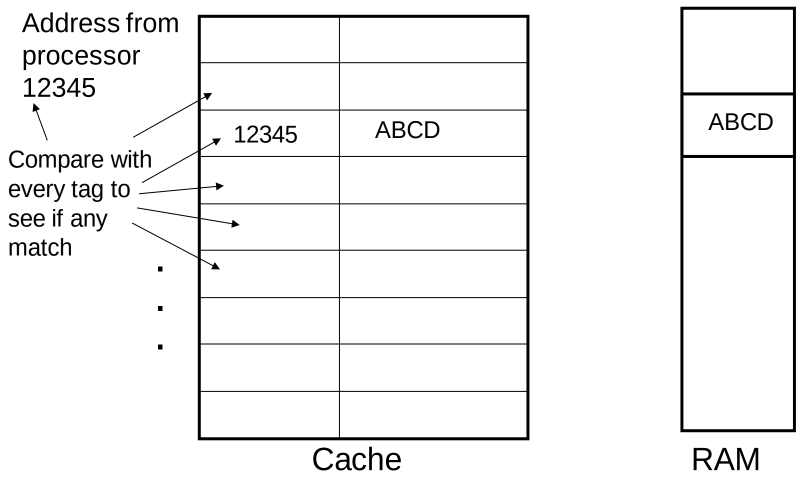


Suppose an address of 12345 is generated. This will cause line 45 to be checked. This line of the cache could contain data from a number of locations (e.g. 11145, 22245, 32145 etc.) so the tag bits for line 45 are checked. If there is a match, then the data is available in the cache. If the tag bits do not match, then the data must be fetched from RAM .

Although this technique is simple, it does have an obvious disadvantage. If data from address 12345 is in the cache, then there is no where to store data from any other address that ends in 45. With there only being one possible place to store any block of data, the system is inflexible and the hit rate can be relatively low since there is a greater chance that the block that is about to be used will have been replaced with a different one.

FULLY-ASSOCIATIVE CACHE

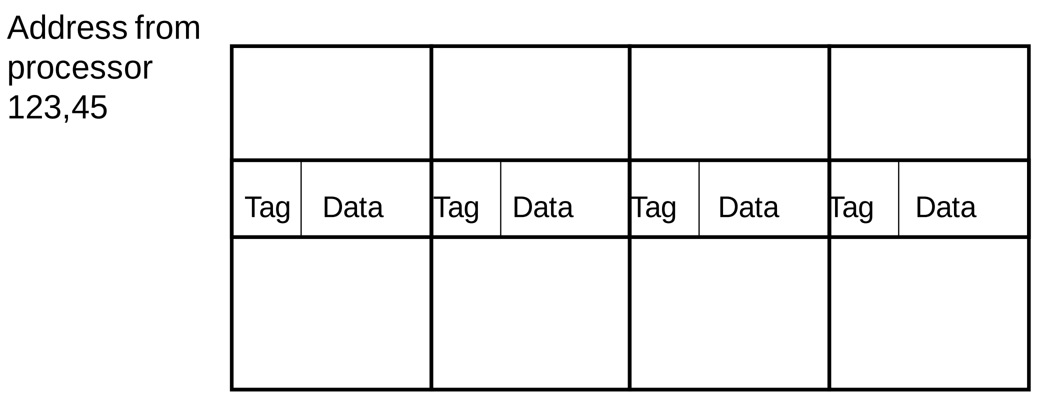
If data could be stored anywhere in cache, some of these problems would disappear. However, if data could be anywhere in cache, then the whole of cache must be searched simultaneously in order to find it. This is known as fully-associative mapping, and is a form of Content Addressable Memory


Address from processor 12345
Compare with every tag to see if any match
12345
ABCD
ABCD
Cache
RAM

In effect, the whole address is used for tag bits. The need for parallel hardware to search the whole cache simultaneously restricts this to small cache. Some microprocessors of a few years ago (Z8000, Z280) used this when cache sizes were smaller, typically 16 blocks of 16 bytes. A disadvantage is that there needs to be some way of implementing a replacement algorithm, since data can go anywhere. With Direct-mapped caches, data can only go in one location, so no replacement algorithm is needed.

SET-ASSOCIATIVE CACHE

This is a useful compromise that is now commonly used. There is more than one possible location for each block of data, but searching is kept simple by having a limit on the number of places data can be stored.



This example has four possible locations for each block of data. The index (e.g. 45 in the address 12345) is used to locate a 'set'. This set has four blocks associated with it, so it could hold the data at 12345 as well as at AAA45 and 11145 and 32145. These four blocks have to be checked simultaneously to see if any of the tag bits match (i.e. to see if 123 can be found).

Set-associative mapping is simpler to implement than fully associative mapping (only a few locations need to be searched, and a simpler replacement algorithm is needed). As there are more than one possible locations for data, there are fewer clashes than with direct-mapping. The more 'ways' there are, the fewer clashes there should be. In practice having a large number of ways adds little to the performance, and four/eight/sixteen ways are often used. The TLB in the Intel family uses four-way mapping.

REPLACEMENT STRATEGIES

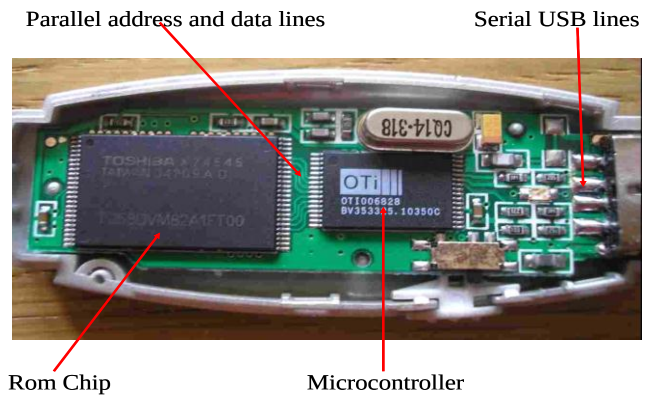
Cache replacement is very similar to virtual memory replacement. When a read miss occurs, data has to be fetched from RAM. This has to be copied into cache, as the Principle of Locality suggests it is likely to be needed again soon. But, if cache is already full, some data will have to be over-written. If the replacement strategy is chosen well, then the data that is overwritten is the data that is least likely to be needed. Of course it is impossible to predict this exactly, so an algorithm based on age or usage is used.

There is one difference between cache and virtual memory replacement. Cache replacement has to happen in nanoseconds (not milliseconds). This means that only simple hardware can be used. In practice, this restricts replacement algorithms to simple ones, such as FIFO or approximations to LRU. As these have been discussed in relation to virtual memory systems, they are not further dealt with here. There are many similarities between cache and virtual memory. The basic principle is: look in the smaller, faster memory first, then go to the slower, larger memory only when you have to.

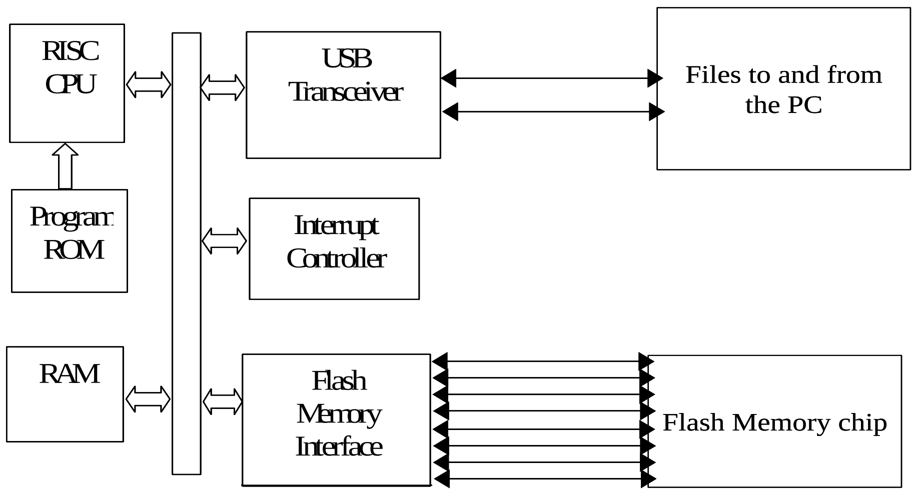
FURTHER READING: USB DRIVES ANDTHE MESI PROTOCOL

1. FLASH DRIVES

USB memory sticks are an interesting use of Flash technology: memory chips that don't lose data when the power is removed. They do need complete microcontrollers in them to convert from the files expected by the PC to the raw data stored on the chips. The USB interface is serial: it only uses a few lines. ROM chips need address and data lines, and data arriving in parallel.



A typical usb memory stick



Although the flash chips are semiconductor chips, they do not have anything like the speed of access of DRAM chips. This, combined with the fact that the access typically goes via the USB route, means that the speed of a flash drive is comparable with the millisecond access time of hard disks, not the tens of nanoseconds of DRAM. As far as the PC goes, the memory looks like another disk drive, not an extension to the main DRAM.

2. MESI INTRODUCTION: MULTIPLE CACHES

There are potential problems with any system that has multiple caches and levels of memory. This could be the L1 & L2 caches, or multiple processors or cores, each with their own on-chip cache. The problem is that the data must be consistent in the various caches. For instance, a processor in a two core system that communicates via shared memory must inform the other processors when it has modified the contents of memory (fig 1.). Otherwise, a core could read a variable into a register from Level 2 cache and make a change to it. Another core could read the old value of the variable from the L2 cache and not realise it had an old incorrect value. In practice, it is often necessary to use 'bus snooping' techniques in order for processors or cache controllers to find out the state of cache lines.

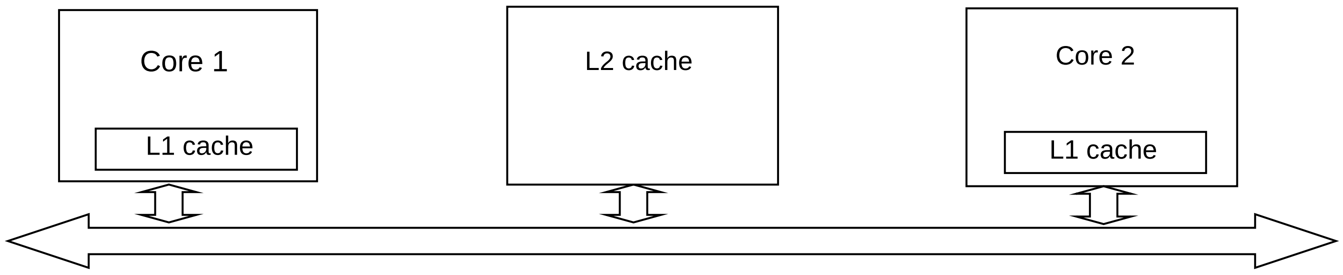


Figure 1. Two-Core System

THE MESI STATES

A commonly used technique is to use 2 hardware bits to define 4 states, abbreviated to MESI

MODIFIED

This marks a line of cache that has been modified. The data is therefore valid in this cache, but is not present elsewhere in other caches or RAM. A processor reading this line of cache would be successful (a hit). This state is sometimes referred to as Modified-Exclusive, because the up-to-date data is only in this cache.

EXCLUSIVE

This marks a line of cache that contains data that is available here, but not in other caches. A read to this would give a hit. It is sometimes referred to as Exclusive-Unmodified, as it is used to denote data that has not changed, so it will be the same as the corresponding data in RAM.

SHARED

A shared cache line has data that could be present in other caches. It contains the most up-to-data data, so a read access can be handled directly. If data is written to the cache, it is also switched to the external bus. This allows other caches to mark their contents as invalid, and in need of updating.

INVALID

If a line is marked as invalid, then it either contains no data, or the data is invalid. Any attempt to access this line (for a read or write) would result in a miss. Normally the cache controller would fill this line after a miss.

CONCLUSION

Real systems can be very complex. In practice, the difference between the Modified and Exclusive states is small. It is possible to illustrate most of the main features of this sort of system by combining the Modified & Exclusive States. In the Pentium the MESI protocol is used except for the I cache. The I cache is simpler, since this only ever holds code which does not change, so there is no need for the Modified or Exclusive states.

END OF UNIT - SUMMARY OF ACHIEVEMENTS

On completion of this unit you should have achieved the following :

- Studied a range of architecture types
- Experimented with Pentium assembly language and seen the effect of the instructions on the registers and on the flags.

On completion of this unit you should understand :

- A range of processor types and understand the key differences between CISC & RISC processors
- Current developments in DRAM and cache memories in current PC systems.

• Alistair Armitage • School of Computing • Napier University • Edinburgh •

Faculty of Engineering, Computing & Creative Industries

Edinburgh Napier UNIVERSITY

Module : Systems and Services

Module Number : CSN08101

Unit 4 : I/O Methods

Flag Polling, Interrupts
&
Direct Memory Access

Student Study Material

UNIT 4 : I/O METHODS

OVERVIEW

This unit reviews the basic concepts of Input-Output (I/O) hardware and data transfers before introducing the principles of Flag Polled, Interrupt and Direct Memory Access (DMA) driven I/O systems.

Polled and Interrupt operations are described in terms of how they integrate with program operations before the system hardware requirements that support interrupt operations are developed. Support for multiple interrupts, the interrupt controller, vector table and stack are discussed in some detail in order to give a deeper insight into the lower-level operations of the I/O system.

The requirements for DMA systems are established and methods of block and cycle-stealing transfers are discussed. The requirements for buffer memory in I/O systems are also discussed.

The relationships between the underlying system hardware and the Operating System (OS) are established, including the function of I/O Manager and system level device drivers. The mechanisms used to translate low-level, interrupt and DMA operations into OS events are also briefly introduced.

LEARNING OUTCOMES

On completion of this unit you should understand :

- Understand the basic types of I/O hardware and the I/O basic read and write data transfer operations.
- Understand the principles of flag polling and simple interrupt systems.
- Understand the system requirements for processing interrupt requests.
- Understand the Interrupt Controller and the mechanisms required to handle multiple interrupts.
- Understand the principles of Direct Memory Access (DMA) systems.
- Understand the DMA Controller and the mechanisms required to handle DMA data transfers.
- Understand the relationships between the OS and the underlying systems hardware.

RESOURCES REQUIRED :

PC microcomputer equipped with Internet Access

ADDITIONAL MATERIAL

There are many excellent websites with application notes and papers on topic of bus systems technology. Also Wikipedia provides a wealth of information and links on these topics:

TUTORIAL : I/O DATA TRANSFERS AND METHODS

LEARNING OUTCOMES

On completion of this tutorial you should understand : the principles of I/O data transfers; flag polling; simple interrupt systems; the system hardware requirements for processing interrupt requests; the detailed sequence of operations required to process interrupt requests; multiple interrupts and how they are handled; direct memory access techniques.

Use Figure 1 in the Unit 4 coursenotes as the reference model when answering the following tutorial questions.

1. For the generic I/O interface shown in Figure 2 :

   i. Describe the typical functions performed by Control, Status, I/O Port, ADC and DAC interfaces.
   ii. Describe, as a step-by-step sequence, how the processor writes a byte of data to the I/O Port 1 register.
   iii. Describe, as a step-by-step sequence, how the processor reads a byte of data from the Status register.

2. (a) Identify and briefly describe two serial interfaces commonly found in PC computer systems.
   (b) Compare parallel and serial methods of data transmission.
3. (a) Describe the method of controlling I/O data transfers using Flag polling.
   (b) Describe the typical response sequence when a processor unit receives a hardware interrupt request (/IRQ).
   (c) Compare flag polling and interrupt driven methods for controlling I/O data transfers.

4. (a) Describe the function and main operating principles of an Interrupt Controller (IC).
   (b) Using Figure 1describe the sequence of actions that processor takes when the Interrupt Controller receives an Interrupt request (IRQ1) from the DIGITAL I/O device.
   (c) Compare the mechanisms provided by an IC with those of a flag polled interrupt system for identifying and prioritising interrupt requests.

5. i. Assuming that an interrupt request from the Digital I/O device is currently being serviced. How does the system react when an interrupt request is received from the Serial I/O device.

   ii. Assuming that an interrupt request from the Digital I/O device is currently being serviced. How does the system react when an interrupt request is received from the Analogue I/O device.

6. Using Figure 1, describe how the Interrupt Controller deals with a situation where two interrupts arrive almost simultaneously, for example, say an interrupt request is made on IRQ1 very shortly after an interrupt request has been made on IRQ2.

7. (a) Describe the function and main operating principles of a Direct Memory Access Controller (DMAC).
   (b) Using Figure 1 as a reference describe the sequence of actions that processor takes when it receives a Direct Memory Access request REQ1 on the SERIAL I/O Channel.
   (c) Assuming that a DMA request from the Digital I/O device is currently being serviced. How does the system react when a DMA request is received from the Analogue I/O device.

8. (a) Under what circumstances is DMA the appropriate choice of interface.
   (b) Compare 'Block Mode' and 'Cycle Steal' methods of Direct Memory Access data transfer.
   (c) Discuss, with the aid of a system diagram, how a memory buffer might be used to synchronise the I/O processing speed requirements in a high-speed DMA based application.

9. (a) Identify the major functional blocks within the I/O subsystem and briefly describe their role during I/O operations.

   (b) What is the function of the Hardware Abstraction Layer (HAL) in the Windows OS.

FURTHER READING AND PRACTICAL WORK : PC I/O RESOURCE IDENTIFICATION

LEARNING OUTCOME

These topics are intended to broaden your reading and improve you approach to academic research and investigation. On completion of these exercises you should be able identify the I/O resources allocated to the I/O devices found on a typical PC microcomputer.

PRACTICAL

Navigate from the Windows Control Panel > System > Device Manager and :

1. Generate a list of I/O addresses and Interrupt numbers allocated to the following devices:

   Mouse
   Network Adapter Comport
   Parallel Port
   USB controllers

2. Select the System Devices option and repeat for the following devices :

   System Timer
   Real-time clock
   Numeric Processor

3. Specify the I/O address range allocated to the Interrupt controller, the DMA controller and the System Timer.

I/O SUBSYSTEMS AND DATA TRANSFERS

INTRODUCTION

The computer system architecture shown in Figure 1 presents a detailed view of a generic processor based system with the MEMORY and INPUT/OUTPUT blocks clearly separated into their respective subsystems. This diagram provides sufficient detail to support an in-depth study of I/O systems and data transfer techniques. **It will be used as the reference diagram throughout the notes and tutorials.**

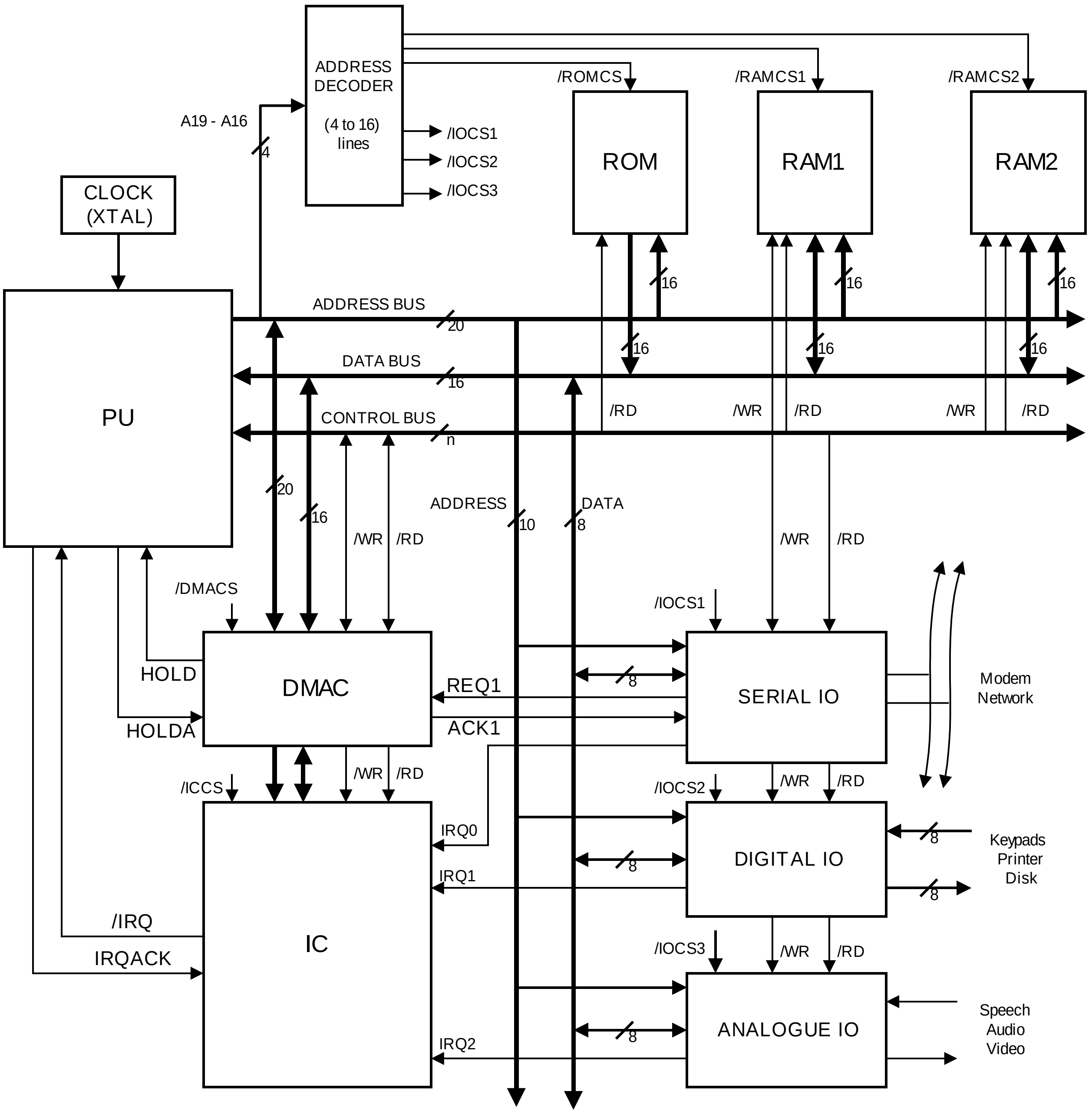


Figure 1. Processor System Reference Diagram - showing Memory and I/O Subsystems

The memory sub-system comprises a mixture of ROM and RAM chips and the Input-Output subsystem shows some common Digital and Analogue I/O interfaces. Also shown in Figure 1 is the detailed connection of the device Chip Select (/CS) control signals that, when used in conjunction with the ADDRESS DECODER, ensures that the individual memory and I/O devices are uniquely addressed. This important mechanism avoids any possibility of data bus contention - this issue is discussed further in the next section. To avoid cluttering the diagram the /IOCSx signal connections are broken but consider them as being physically connected in the same manner as the ROM and RAM chip select signals.

Figure 1 also shows two I/O Controller devices, an Interrupt Controller (IC) and a Direct Memory Access Controller (DMAC).These devices provide efficient mechanisms for implementing I/O systems and are discussed more fully in later sections.

DEVICE ADDRESSING - ADDRESS DECODING AND CHIP SELECTS

Memory systems in PCs and Embedded Systems are now required to support large amounts of memory and numerous I/O interfaces. Typically, memory sizes 0.1- 8GB and there is an ever increasing variety of peripheral devices to be supported, for example, USB, mouse, keyboard, disk drives, networks audio and video devices. As Figure 1 shows all these devices share the same address, data and control pins and if this arrangement is to work then certain operating conditions must be observed :

- **Only one device should be able to drive the data bus at any time.** All devices in a bus connected system share a common data bus and if more than one device attempts to write data to the bus at the same time a collision of 1's and 0's occurs on the data pins. This situation is known as a **'bus contention'** and it results in the corruption of data on the bus. Bus contentions may also result in damage to the devices that are attempting to simultaneously drive the bus. Contention issues are avoided by ensuring that when a device is not addressed it is effectively disconnected from the data bus. This is the function of the device Chip-Select (/CS) control signal which connects and disconnects devices from the data bus using TRI-STATE operation. Simply put this means that all non-selected devices are effectively powered-down and disconnected from the data bus. The function of the ADDDRESS DECODER block, in Figure 1, to decode the address bus signals so that only one chip select signal is active at any time, ensuring that bus contentions are avoided.

- **Every device connected to the bus system must be uniquely addressed.** The purpose of the address decoding circuit in Figure 1 is to ensure that only one device is active at any time. For any combination of address inputs only one device chip select output becomes active and only one memory or I/O device is enabled. The address decoding circuit effectively divides the available address space into blocks so that each device occupies a specific range of addresses.

Device addressing in Figure 1 has now become a little more complicated but it does demonstrate more clearly how the addressing system actually works. When the processor puts an address on the address bus the higher order address signals (A19 – A16) are used to enable a single device, the lower-order address (A15 – A0) signals are used to select the specific memory location or I/O register within the device. Note that addresses issued by the processor can address memory location or I/O devices the address decoder selects which block of memory or I/O is selected for data transfer.
The key point is: at the instant when the data transfer takes place only the processor and the selected device address (memory location or I/O register) is connected to the data bus – all other devices are effectively switched-off.

MEMORY AND I/O DATA TRANSFERS

The read and write data transfers previously described in relation to fetch-execute cycles are somewhat simplified, so they are re-visited here this time with the inclusion of the system clock, device control signals and address decoding logic taken into consideration. The intention is to generate a deeper understanding of processor system hardware operations and the data transfer process.

READING DATA

To read the data from a device the processor outputs an address value on the address bus and asserts the memory read signal(/RD). The address decoding circuit responds to this by activating one of its chip select outputs (/CSx) and a short time later, allowing for the memory or I/O device access time, the selected device drives a data onto the data bus. The processor then clocks the data from the data bus into an internal register to complete the read cycle. Consider, as an example, reading data from the Digital I/O port in Figure 1. The sequence of events would be: the processor outputs the address of the digital I/O port and asserts the memory read signal(/RD); the address decoding circuit responds by activating /IOCS2, and a short time later, allowing for the device access time, the digital I/O device drives a data on to the data bus; finally the processor clocks the data from the data bus into an internal register to complete the read cycle.

WRITING DATA

Writing data from the processor to a memory or I/O device follows a similar pattern. The processor outputs an address value on the address bus and at the same time also outputs the data on the data bus. The address decoding circuit responds to this by activating one of its chip select outputs (/CSx) so that a device and a unique location are now addressed. The processor then asserts the write signal (/WR) to begin writing the data into the device. A short time later, allowing time for the address, data and control signal to stabilise and the data to be written into the selected device, the processor removes the write signal to complete the write cycle. The system then continues with the next read or write operation.

In summary, the device chip select /CS signals are derived from the high-order address bus signals and are used to uniquely identify the device that is to be used for data transfer. The low-order address signals are used to identify the single memory location or I/O device register that is to be used in the data transfer. The read (/RD) and write (/WR) bus control signals are used establish the direction of data transfer between the processor, memory and I/O devices.

Note again that all instruction *processing activity and data transfer operations are synchronised to the system clock.*

INPUT – OUTPUT (I/O) INTERFACES AND PORTS

Communication between the processor system and external/peripheral devices takes place through a hardware interface. The primary role of the interface is to match the digital signal requirements of processor system to the external signal requirements of the peripheral device. Data transfers are implemented by reading from and writing to a series of internal registers within the interface device and the format and synchronisation of data transfers is controlled via **control** and **status** registers. In order to distinguish memory transfers from I/O transfers the term PORT **(**as in a ships 'porthole') is used to indicated that and I/O transfer links the internal bus system with external devices.

The generic I/O interface in Figure 2 shows that although the processor bus interface is purely digital, the peripheral side can be digital or analogue. Obviously, in the case of an analogue input and output the interface must do some signal conversion – discussed below. In addition to signal conversion, interfaces often manage I/O transfer operation by providing timing and buffering functions.

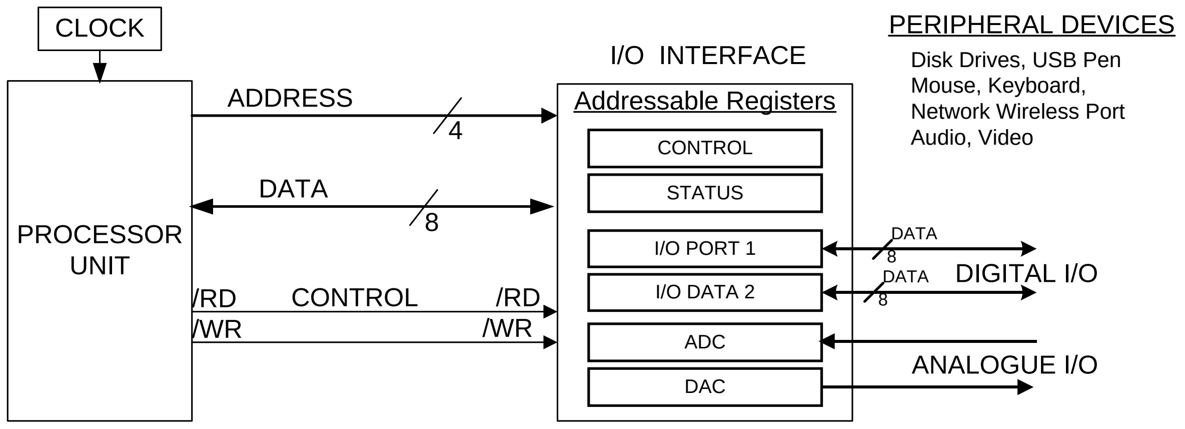


Figure 2 - Processor – A generic I/O Interface.

I/O READ and WRITE CYCLES

The I/O interface links the processor system to its peripheral devices. Figure 2 shows that the same bus address, bus data, and read and write signals are present and the only notable difference is that there are far fewer address signals, only 4 in Figure 2, and these are used to address internal registers and ports within the I/O device.

The peripheral side of the interface is quite different since computer systems now support a vast range of peripheral devices from common devices, such as, disks, networks, USB, mouse, keyboard, audio and video interfaces to more exotic devices such as game controller, touch and proximity sensors.

TERMINOLOGY - REGISTERS, FLAGS AND PORTS

These terms describe storage elements within the processor and I/O interface devices. A key feature of registers and flags is that they are addressable and as such are directly accessible to systems level programmes. This means that programmers have access to and can control low-level system hardware operations through registers, flags and ports

REGISTER

A register is a named placeholder within a processor or I/O interface device. For example, the Control Register in Figure 2, is the register that sets the operating characteristics of the interface. Usually parameters such as data rate, data format, signal direction and clock speed can be controlled by writing (or reading) flag bits within the Control Register.

FLAG

A flag is a named single bit with a register. For example, the Status Register in an I/O interface device will often have a READY flag that indicates when the device is ready to send or receive data. This is used by flag polling software to determine when the device is ready to transfer data.

PORT

The term PORT* is used to distinguish an I/O address location from normal memory address location. Operationally an I/O interface can be viewed as enhanced memory device. Binary data written to an OUTPUT PORT is available at the output terminals as digital or analogue signal levels. Similarly, binary data read from an INPUT PORT represents the signal value at interface input terminals. Ports provide a structured mechanism for interfacing peripheral devices to processor systems, making it relatively simple to connect, via the system bus, and communicate with external devices. I/O Interface devices for keyboards, printers, disks, mouse, modems, networks, graphics and audio systems are readily available from chip manufactures.

I/O Interfaces and Devices

As stated previously, there is a vast array of peripheral devices that can be hooked-up to processor systems. Common PC system examples are : disk drives, sound cards and USB ports, however, in smaller embedded systems, for example, phones and MP3 players, the peripheral devices are more likely to be touch based keypads, basic LCD Graphics displays, audio sounders or LEDs. Similarly, in measurement and instrumentation systems input sensors for measuring physical parameters such as temperature, proximity, flow, pressure, acceleration, direction and position are common peripherals.

Although the range of possible external devices is enormous their signal characteristics fall into just two categories: **Analogue or Digital**. A switch is obviously a digital device that produces either ON or OFF (1 or 0) signal, whereas a microphone is an analogue device that produces a signal voltage that varies as audio signals are detected. Consequently, the switch requires a simple digital I/O interface and the microphone requires a more complex interface that converts the microphone's analogue voltage into a stream of binary values - a digitised signal. These type of interface are introduced in the next section and more details case study examples are presented in Unit 6.

Digital I/O

Digital interfaces are relatively simple to implement because the signals on both sides of the interface have the same binary form. The processor side of the interface always uses parallel data transfers so that, in the simple case, where the required data is also in parallel format, the main role of the interface is to match the signal voltage levels. Many devices provide data in a serial stream of bits and so this type of interface has a secondary function – it must translate the parallel data into serial data and vice versa. These two basic forms of digital data transmission are termed - PARALLEL and SERIAL.

Parallel Interface

Parallel I/O data transfer is illustrated in Figure 3. It is fast, requiring 1 clock pulse to transfer N-bits of data. On the downside however, it requires 1 signal line (copper cable, connector pin, pcb track) per signal to be transferred. This makes parallel data transfer an expensive option, so parallel data transfer is restricted to applications where fast data is required over short distances. Common PC microcomputer parallel interfaces are, the disk drives and the printer port – but all this is currently changing, see Unit 5.

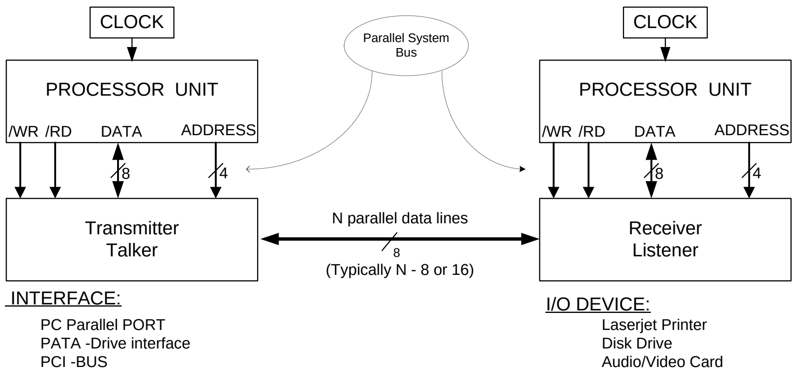


Figure 3. Parallel Data Transfer

Serial Interface

Many applications require data to be transferred over distances greater than a few metres and this situation is ideal for serial data transfer. Communications via the telephone or over a computer network use serial data transfer. Serial data is commonly transmitted over a single or pair of wires, as shown Figure 4 . The connection between the two communicating devices is simple and furthermore, the connectors (plugs and sockets) required for the serial interface are simpler and less expensive than their parallel counterparts.

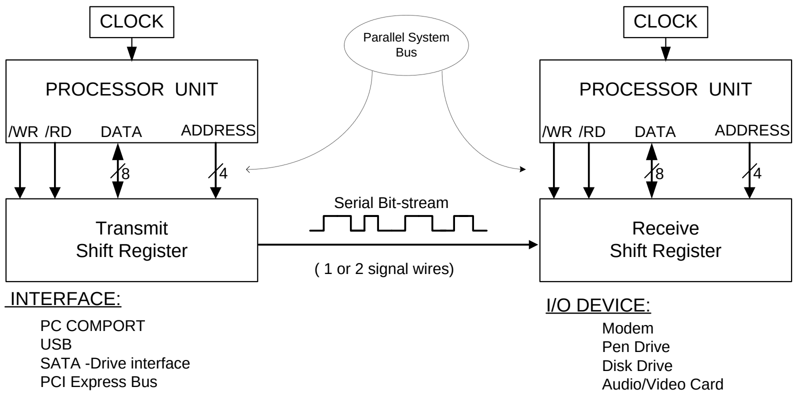


Figure 4. Serial Data Transfer

Since the processor system transfers data over the system bus, in parallel binary format, serial data transfers require a mechanism for of converting parallel data into a serial bit-stream. This is accomplished using a simple electronic circuit called a **shift register,** Figure 4. When transmitting the shift register loads data from the parallel bus and clocks it out as a serial bit-stream over a single wire. At the receiving end of the serial link the serial bit-stream is clocked into a similar receive shift register which then makes the data available in parallel format to the receiving processor system data bus. Common PC microcomputer serial interfaces are, USB, SATA for disk drives, PCI-e and

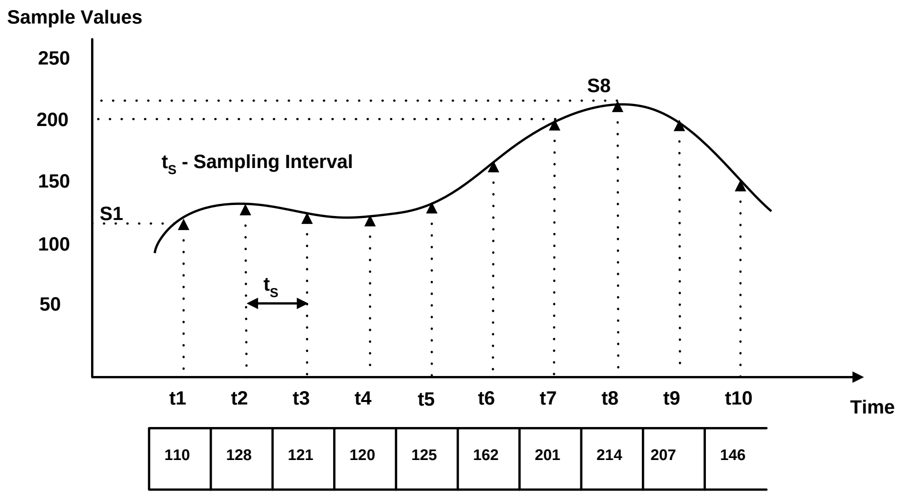
The obvious points of comparison between parallel and serial data transfer schemes are : **serial data transfer is slower but costs less in terms of cabling (copper) and connector costs, whereas parallel data transfer is fast but requires more copper and more complex connectors is therefore more expensive.** In summary, if the distances involved are greater than a few metres serial data transfer is the appropriate choice and for short distance high-speed data transfers a parallel data transfer scheme is appropriate.

The general trend in bus systems is to replace parallel systems with serial bus systems, for example, in the PC environment Universal Serial Bus (USB) is now used to replace LPT ports; Serial –ATA is now becoming the standard for disk drive, replacing PATA/IDE parallel interfaces; and PCI-Express, a high performance scalable serial bus is the replacement for the older parallel PCI-X bus. The usual commercial motivating factors apply here – serial bus technologies offer increased performance and reduced system costs. These bus technologies are discussed in Unit 5.

The speed of serial data transfers range from a modest 480Mbps for USB 2.0 to >100Gbps for Ethernet. At chip level, high-speed serial interface technologies are now used to connect are wide range of devices, for example, ADC, DAC, audio CODEC, Timer and Real Time Clock Calendar interfaces. Serial Peripheral Interface (SPI) and the Inter-Integrated Circuit (I^2C bus) – also known as the System Management Bus (SMB) and two common examples of high-speed serial buses that connect chips together.

ANALOGUE I/O

Many sensors (and transducers) generate analogue signals which must be converted into binary data so that a processor system can digitally process the signal information. Figure 5 shows a typical analogue signal. Its notable characteristic is that it has a value at all instants in time and is said to be time continuous. To digitised the analogue signal it is sampled at regular instants as shown by the arrows at t1,t2,t3 in Figure 5. A numeric data value is assigned to each sample based on the height of the signal at the sampling point and the value is then stored in memory. The table, or more correctly ARRAY, shown in Figure 5 represents the digitised signal.



t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
110	128	121	120	125	162	201	214	207	146

Figure 5. Analogue Signal Digitisation

The conversion between the analogue and digitised signals results in a loss of signal information – the digitised signal, as seen by the processor, has no knowledge of the signal between sample instants. In practice this presents no problem provided the analogue signal is sampled fast enough to record all signal details of interest. So what then is the optimum sampling rate?

The rule of thumb for estimating a suitable sampling rate is known as the Nyquist frequency and it states that :

The sampling frequency should be at least x2 the maximum frequency present in the analogue signal input.

In terms of audio sampling the input frequency range is 20Hz-20kHz, which means that the sampling rate should be at least 40kHz (44kHz is often quoted in CD quality sampling). For video applications, the signal frequency content extends from 0-5MHz therefore sampling rates in excess of 10MHz are required. Note that sampling at 10MSamples/s requires a lot of memory, if fact 10Mbytes to store 1 second of raw video signal – this bring into direct focus the requirement for data compression schemes such a MPG.

ANALOGUE INPUT - ANALOGUE-TO- DIGITAL CONVERSION (ADC)

The ADC interface converts analogue signals to a stream of binary sample values. It operates by comparing the input signal voltage to a internal reference voltage to produce a proportional binary weighted output value. In Figure 6 an input voltage of 0 volts generates a binary output value of 00000000; a value of 2.55 volts produces a value of 11111111; in-between a input value of 1.25 volts produces a binary value of 01111111. This particular ADC interface increases its output value by 1-bit for every 1millivolt change in input voltage.

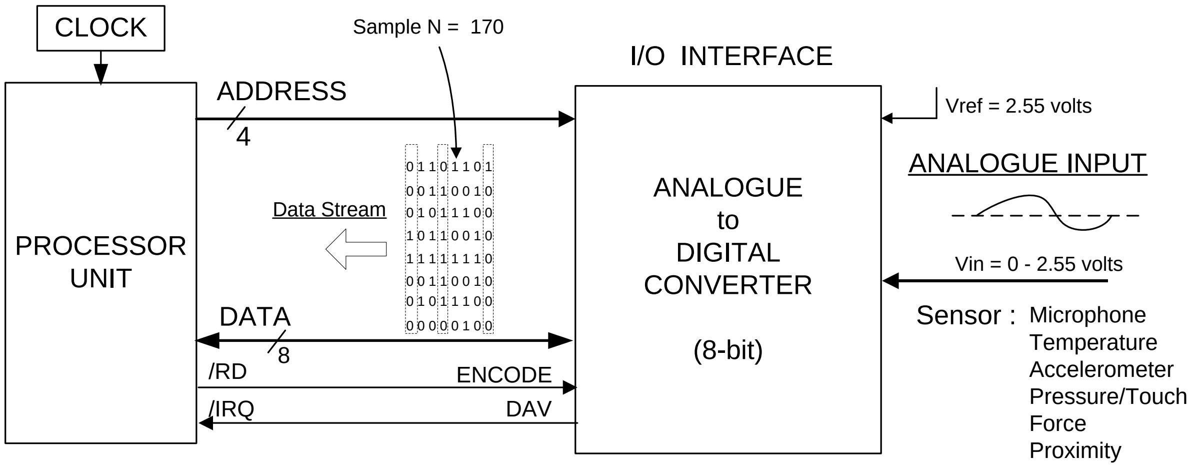


Figure 6. Analogue-to-Digital Conversion

The ENCODE signal in Figure 6 is used to initiate sampling events and is usually controlled by the processor. The Data Valid (DAV) signal is to signal the processor to indicate that a sample value is available. Very often the DAV signal is connected to a processor interrupt pin.

Signals derived from temperature, pressure, flow, position and acceleration sensors are common examples of analogue signal inputs that can be digitised by the ADC.

ANALOGUE OUTPUT - DIGITAL-TO-ANALOGUE-CONVERSION (DAC)

Many output devices require analogue signals to drive them and this requires an interface that converts the digital data generated by the processor system into an analogue signal voltage or current. Video displays, speech synthesisers and motors are typical examples of output devices that are controlled by analogue signal outputs.

A DAC performs the reverse process to the ADC. It converts binary sample values into a proportional analogue signal voltage or current.

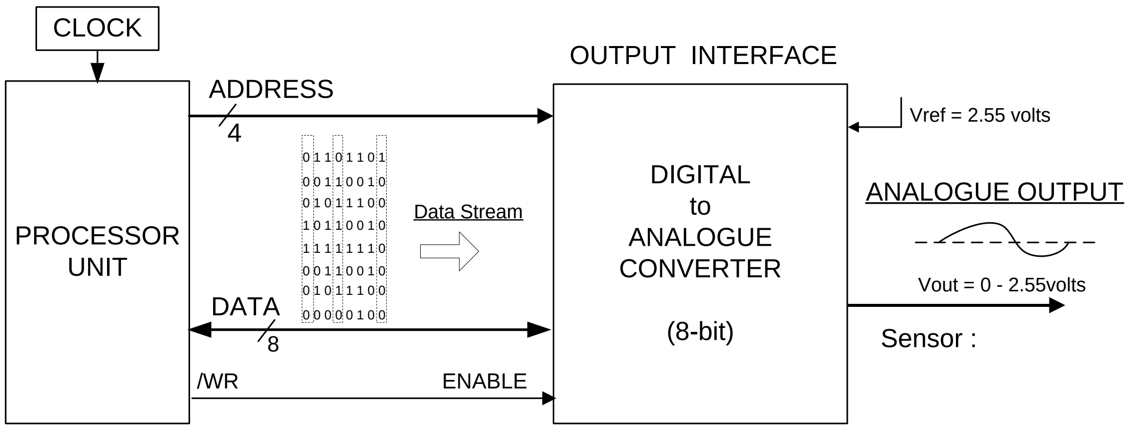


Figure 7. Digital-to-Analogue Conversion

The ENABLE signal is used by the processor system to 'tell' the DAC when to update the output signal.

It should be noted that the DAC output voltage is held fixed between sample updates. It the output sample rate is too low the output response can appear to have a stepped output profile and if this is not properly filtered, particularly in audio applications, the resulting sound contains high frequency 'clicks and pops' that seriously degrading the sound quality. Referring back to the ARRAY of sample points in Figure 5 it should be noted that the original signal can be reconstructed by replaying the original sample values through a DAC interface at the same sampling interval (ts).

The application of these interfaces is discussed in more detail in the Unit 6 - Peripheral Interface Devices.

I/O TECHNIQUES – SOFTWARE FLAG POLLING, INTERRUPTS AND DMA

The timely response to external events is an extremely desirable performance parameter in any computer system. In the context of I/O, an event might be an ADC signalling that it has valid data, or a serial interface signalling that a character has been received, or a timer indicating that it has reach a timeout state (often the overflow condition 0xFFFF -> 0x0000).. Processor systems handle I/O events using either – SOFTWARE POLLING, INTERRUPTS and DIRECT MEMORY ACCESS and these methods are described here.

I/O TECHNIQUES 1 – SOFTWARE POLLING

Flag polling is a basic I/O method that is applied in simple systems. The technique relies on the ***system software*** to periodically test the status of flags associated with a particular I/O interface. A typical example is shown in Figure 8, where the status register for device B is polled. If the flag is TRUE then Device B is serviced by calling a subroutine (method) to handle the I/O data transfer; if the flag is FALSE the polling routine simply moves on to test the next I/O status flag.

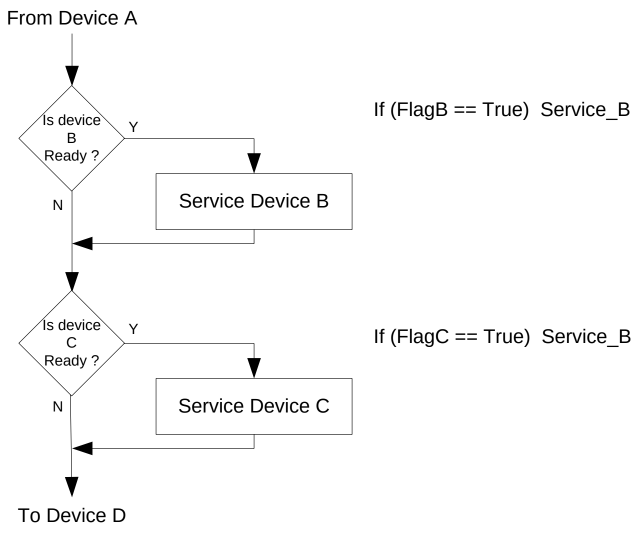


Figure 8. A typical Software Flag Polling Sequence

In a system with multiple devices the flag polling software sequentially reads the status flag bits from each device in turn, if the tested bit returns true, an event has occurred and the system software handles a I/O transfer for that device. This polling routine is implemented in software by simple function (subroutine) call.

Analysis of flag polling systems shows that external devices must be polled at a rate that is high enough to ensure that no data transfer requests are missed. This is highly inefficient because the processor operates much faster than the majority of external devices and the polling routine is frequently called with no data transfer requests pending. Obviously this wastes valuable processing time and reduces system throughput. Flag Polling is therefore applicable in situations where data transfer efficiency and system throughput are not important system parameters.

I/O TECHNIQUES 2 – INTERRUPT SYSTEMS

An interrupt system uses a combination of hardware and software to implement an efficient system for handling I/O events. The basic principle is that the processor continuously executes tasks and only responds to I/O device events when they request attention. In Figure 9 the flow of instruction execution is shown as a main program flow loop that executes continuously. When a device is ready to transfer data it generates a hardware 'Interrupt Request (IRQ)' which causes the main program operation to be interrupted and suspended. The processor then executes a jump instruction to a new address in order to begin executing a new instruction sequence that handles the event. This instruction sequence is called an Interrupt Service Routine (ISR) or Interrupt Handler and when the ISR completes the data transfer task it returns to the previously suspended task and the processor resumes from the point at which it was interrupted. Obvious sources of the interrupt events are ADC, Timers, Serial and Parallel interfaces but less obvious events such as numerical overflows, or a user programme attempting to execute a privileged instruction can also cause similar interrupt events (these types of event are also referred to as exceptions)

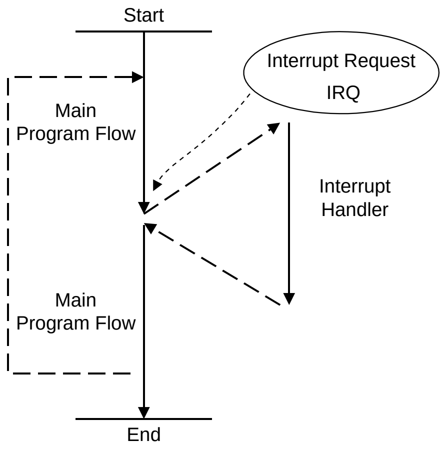


Figure 9. Program flow and the Hardware Interrupt Response

Analysis of the interrupt based system shows that it is the I/O device that initiates the request for a data transfer. If the I/O device does require data transfers it does not generate interrupt requests, the processor ignores it and continues with other processing tasks. Since the processor only responds when interrupt requests are received, the processor is released from the inefficient and wasteful process of repeatedly polling device I/O flags and returning with a negative response. Clearly this is a much more efficient I/O data transfer technique than the flag polling method.

MULTIPLE INTERRUPT SYSTEMS

Processor systems are generally interfaced to a number of I/O devices, for example, keypad, parallel and serial ports, timer, ADC, DAC, so the interrupt system must be capable of handling interrupt requests from several devices simultaneously. It must also deal with interrupt requests occurring at any time and in any order and it has to resolve situations where a second interrupt request is made while a previous request is being processed. In order to handle this level of complexity, processor systems incorporate a hardware device controller called an Interrupt Controller (IC).

Figure 10 is a repeat of Figure 1 which shows an interrupt controller interfaced to three common interface devices. In normal operation, the serial interface would generate an interrupt when it receives a message from the serial data link; the digital interface would generate an interrupt when a key is pressed; and the analogue interface would generate an interrupt when a valid data sample is available.

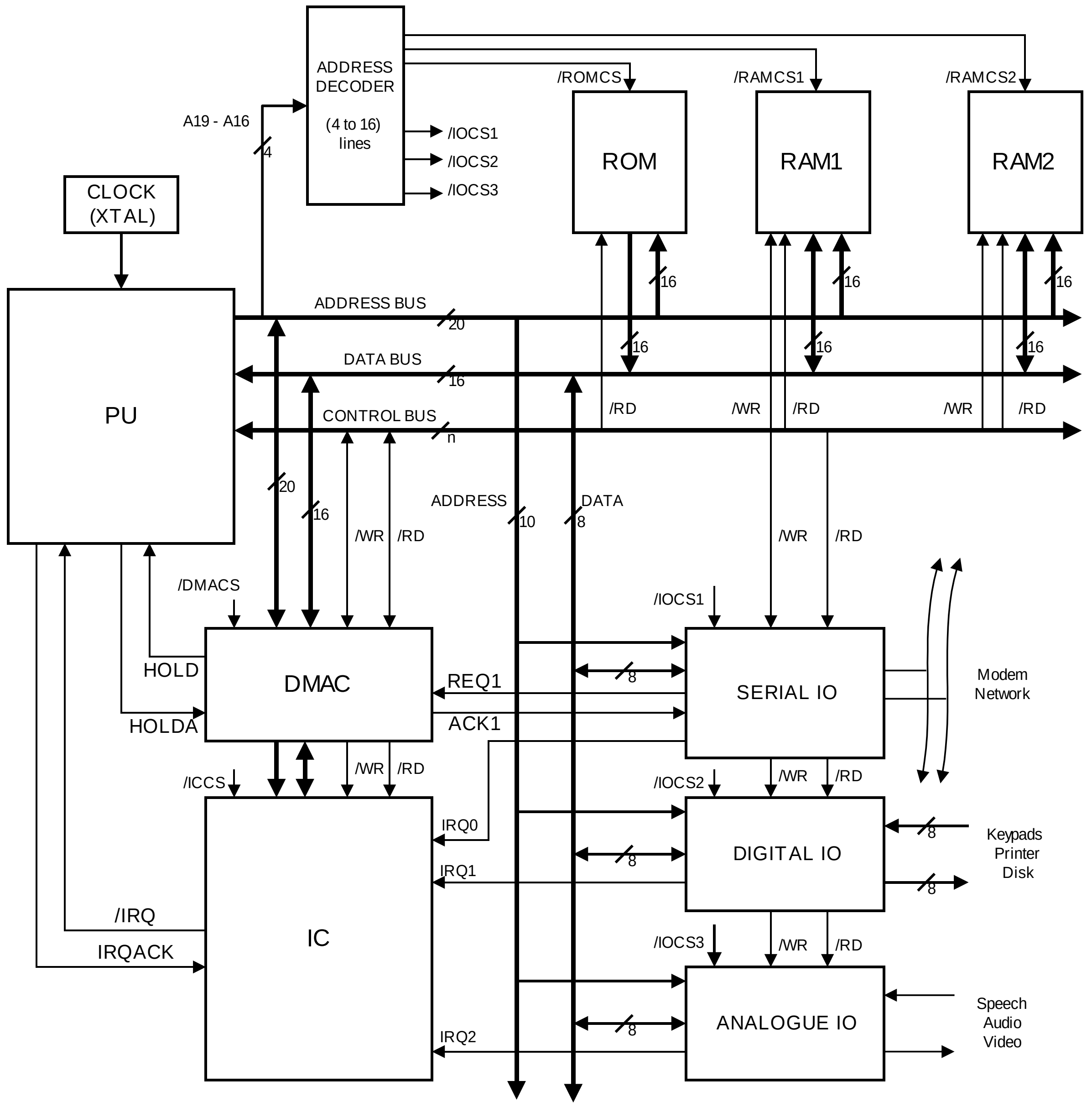


Figure 10. A Processor Based System showing the Interrupt and Direct Memory Access Controllers

In order to explain how the Interrupt Controller handles interrupt requests consider the situation where the processor is busy executing some task and a key is pressed causing the Digital IO interface to generate an interrupt request on IRQ1. A typical interrupt handling sequence would be as follows :

- The Digital IO interface sends the interrupt request IRQ1 to the interrupt controller which, for simplicity at this stage, passes the request directly to the processor by sending an /IRQ signal – imagine that IRQ1 and IRQ are directly connected, so that the interrupt request is channelled directly to the processor.
- The processor responds to the interrupt request by suspending the current processing task, acknowledging the interrupt request, via the IRQACK signal before jumping (vectoring) directly to an Interrupt Service Routine (ISR)) that processes the interrupt request - it this case the ISR reads a keycode from the digital IO port and stores it memory.
- When the ISR has completed this task the processor returns to its original task and resumes processing from the point at which the original task was suspended.

SYSTEM REQUIREMENTS FOR HANDLING INTERRUPTS

A closer inspection of the interrupt handling sequence described reveals several unresolved issues:

- If the processor automatically jumps to the start of the ISR, how does the processor know where to find its start address. To perform the JUMP, the start address of the address of the interrupt service routine must be fetched from somewhere and loaded into the program counter register. Where does this address come from?
- When the ISR completes how does the system return to the original pre-interrupted task?
- Assuming that the interrupted task was using processor resources, notably registers, how can the ISR return to the original task - surely the ISR overwrites the original register values as it executes. During the execution of the ISR some registers will have their values changed and the status flags be altered as a result of instruction execution. If the interrupted program is to resume after the interrupt has been serviced then the processor needs to be returned to its original ***status.*** For a successful ISR return, all the pre-interrupt register values must be restored and the processor should continue as if the interrupt never occurred (this requirement is often called,' Interrupt Transparency'). The real questions are : How does the processor preserve its original status during the interrupt service routine? And how does the processor system restore the original status when the ISR completes its execution?
- What happens if a second, or third , interrupt is received while the processor is currently executing an ISR. How does the processor cope with multiple interrupt sources?

To resolve these issues processor system hardware enlists some additional resources in the form of an INTERRUPT VECTOR TABLE, STACK MEMORY and INTERRUPT CONTROLLLER. These support elements are shown in Figure 11 and their role is described as follows:

INTERRUPT VECTOR TABLE

The interrupt vector table is nothing more than a memory look-up table that is automatically accessed in response to an interrupt request. Each interrupt source, for example the timer or serial port, is allocated a number of memory locations in the vector table (typically 4 locations) and in response to an interrupt request the processor automatically addresses the table, using the interrupt number supplied by the interrupt controller, and fetches the start address of the ISR into the Program Counter (PC) register. The address generated by the processor is called an **address vector** and the system programmer must write the start address of each ISR into the appropriate address vector during system initialisation. In the ADC example, that follows, the interrupt vector table would be initialised with the start address of the ADC_ISR so that the ISR code starts executing as soon as possible after the ADC interrupt is received.

The location of the Interrupt Vector Table is usually at either the top or bottom of RAM. Some processor place the table starting at location 0 while others use addresses at the top of memory, say around 0xffff. As an example, the PC interrupt vector table is located at the lowest 1K memory of RAM and there are 256 interrupt vectors whereas a simple microcontroller (MCU) device has a vector table with only 20 entries, located in Internal RAM (IRAM) at the address space 0x0000 – 0x00AB.

STACK MEMORY

A stack is an area of RAM memory that is used to store dynamically generated data. For example, local variables, subroutine parameters and subroutine return addresses automatically use the stack during the execution of subroutines. The stack operates as a Last-in-First Out memory and is also used to preserve the processor **status** during ISR processing. Register values, notably the PC and Status Flags (PSW), are **'pushed'** to the stack just prior to the start of ISR code execution and then they are **'popped'** from the stack at the completion of the ISR. This mechanism allows the interrupted program to be restored to the point at which it was interrupted. It should also be noted that any registers used during the ISR must also have their values pushed and popped - this operation is the responsibility of the system programmer and is often a source of subtle system bugs.

INTERRUPT CONTROLLER

An interrupt controller is a programmable interface device that gives the system's programmer total control over interrupt sources and interrupt events. The IC is initialised at power-up or during a system reset and when it is setup it provides almost automatic control of interrupt events. The IC provides the following functions :

Interrupt Masking – Enable/Disable

There are times when some, or all, interrupts should be ignored. The IC has a control register, identified as something like the IMASK register, that allows individual interrupts to be enabled and disabled. It should also be noted that the processor has a global interrupt enable status bit, found in the flags/processor status register, that is used to enable and disable all interrupt requests.

Interrupt Identification

Each interrupt requests causes an ISR to be executed and the starting address of the appropriate ISR must be supplied to the processor. At the start of the interrupt handling sequence, the interrupt controller automatically generates an interrupt identification byte that processor uses to generate a pointer/vector to the start address of the appropriate ISR. This pointer, called an Interrupt Vector, is used by the processor to locate, in the Interrupt Vector Table, the start address of the ISR which is then automatically reads into the PC register. Instruction execution now starts from the new address. This mechanism gives rapid access to the ISR when an interrupt is received and facilitates a rapid response to interrupt requests.

Interrupt Priorities

Some interrupts are more important than others. Some can be kept waiting whilst others must receive an almost instantaneous response. These prioritisation issues are also handled by the interrupt controller. The most common priority policy is 'RANK ORDER', where IRQ0 has the highest priority and IRQn has the lowest priority. Operationally, if a lower order interrupt request is received during the execution of an ISR it is ignored until the current ISR has completed its execution: In contrast, if a higher order interrupt request is received during the execution of an ISR it is serviced immediately, the current ISR is suspended until the higher priority ISR has completed its execution, then it is resumed*. An alternative strategy is 'ROUND ROBIN', where the highest priority is awarded on a rotational basis. If IRQn is currently executing, the highest priority is awarded to IRQ_{n+1}. This approach avoids starvation, where low priority interrupts are rarely, or never, serviced because higher priority interrupts fully occupy the processor.

* The mechanics of handling a second interrupt request are exactly the same as those for the original interrupt, the processor status gets pushed to the stacked, the vector address loads the ISR start address into the PC, the ISR is executed and finally, status is restored by popping the original register contents. This does mean, however, that a larger stack memory is required to deal with multiple interrupt requests. This operation is referred to as a **nested interrupt** sequence.

In order to illustrate the processes involved during an interrupt handling sequence consider
Figure 11. The diagram shows the system memory, a processor and an Analogue-to-Digital Converter (ADC) interface that generates an interrupt request when it has a valid sample value. System memory is divided areas for code and data and it is assumed that the processor is running a task, identified as the 'Main Program Code', in
Figure 11. Additionally, it is assumed that the processor global Interrupt Enable (IE) status flag is enabled so that the system is ready to respond to interrupt requests.

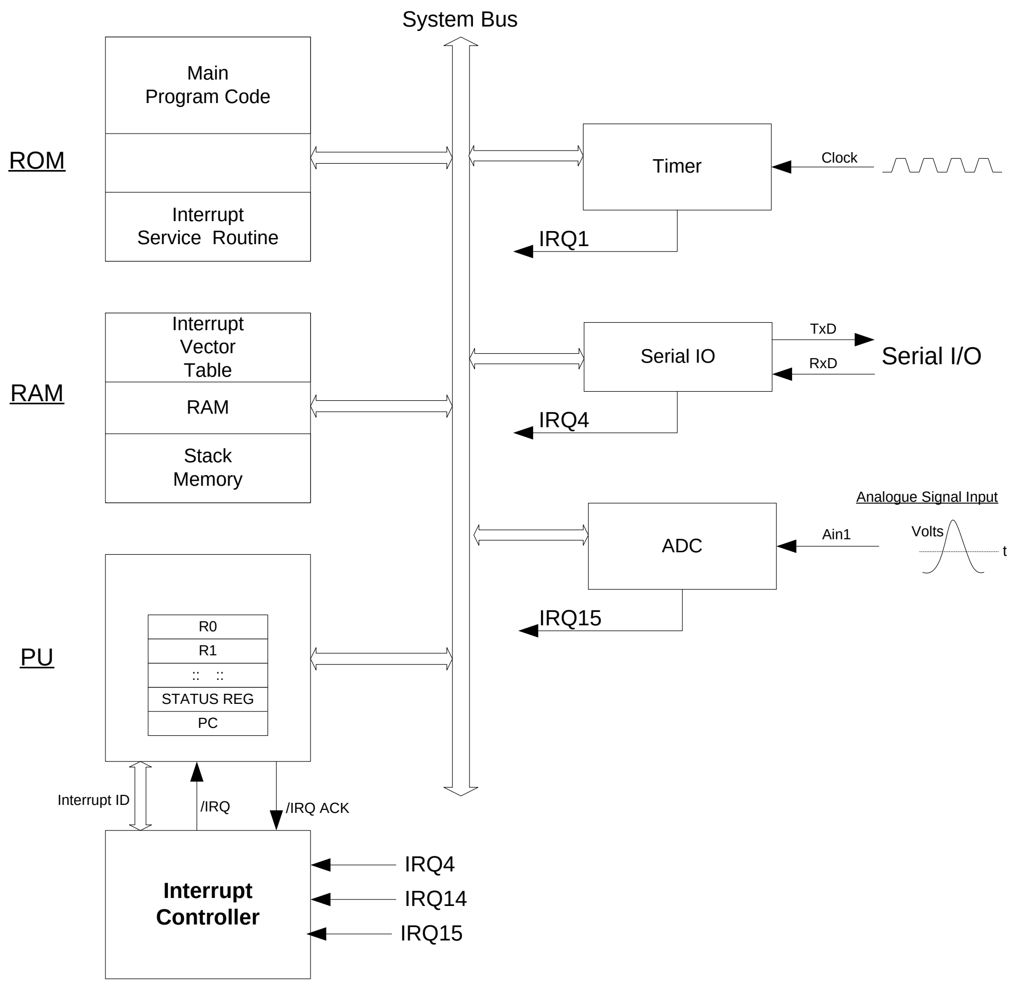


Figure 11 Multiple Interrupts – system components.

To demonstrate how the system manages interrupts via Interrupt Controller (IC) consider the following scenarios :

Scenario1 : The complete interrupt handling sequence for the ADC interface (IRQ15) can be described as follows:

1. When the ADC has valid data available it activates IRQ15 to signal an interrupt request to the IC. The IC then sends an interrupt request (/IRQ) to the processor.
2. The processor tests its interrupt request input at the end of every instruction cycle, therefore, it completes the execution of the current instruction before testing for the interrupt request.
3. Assuming that interrupt requests are enabled, the processor responds by initiating the following sequence :

   i. The processor saves its current status by **pushing** its current register values to the memory STACK[*].
   ii. Further interrupt requests are disabled by setting the global interrupt mask, for example EA=0.
   iii. The start address of the ADC interrupt service routine is loaded into the Program Counter. This start address is automatically fetched from the interrupt vector table.
   iv. The ISR is executed. In this case an ADC sample value is read from the ADC and stored in memory then another conversion cycle is initiated.
   v. When the ISR has completed this task it executes an return from interrupt instruction (RETI) which causes the processor to restore the original processor status by **popping** the pre-interrupt register values from the stack.
   vi. Finally the processor resumes execution of the previously interrupted task at the point where original interrupt occurred.

* Various alternative mechanisms exist to preserve processor status but the use of a stack is the most common and is therefore discussed here.

Although interrupt driven systems are more efficient and more responsive than flag polling systems they do result in more complex systems. Apart from the additional hardware resources required (the Interrupt Controller and RAM for the Stack and Vector Table) a thorough understanding of processor hardware and instruction set is required for the successful implementation of an interrupt driven system software. One further important point is that debugging interrupt driven systems presents much more of a challenge. The asynchronous nature of events and the reliance on dynamic (STACK) memory dramatically raises sophistication of bugs and consequently the level of debugging skills required.

Scenario 2 : The Serial interrupt handling sequence is currently executing and the Timer interface requests and interrupt.

Assuming rank order prioritisation:

1. The processor status prior to IRQ4 is stored on the stack; interrupt vector 4 has been accessed and its content, the Serial ISR start address, has been loaded into the program counter. The processor is currently executing the Serial ISR.

2. An IRQ1 is received. This is a higher priority interrupt so it is serviced immediately. The sequence described in Scenario 1 is repeated for the Timer ISR until it has completed. The STACK grows so that it now holds the register values representing the processor status of the originally interrupt program and the Serial ISR.

3. The Timer ISR executes to completion and then returns to the point in the Serial ISR where it was interrupted. It resumes from that point and if no further higher priority interrupts are received it will completed and return to the main program execution.

Scenario 3 : A Serial interrupt handling sequence is currently executing and the ADC interface requests and interrupt.

Assuming rank order prioritisation:

1. The processor status prior to IRQ4 is stored on the stack; interrupt vector 4 has been accessed and its content, the Serial ISR start address, has been loaded into the program counter. The processor is currently executing the Serial ISR machine instructions.

2. An IRQ15 is received. This is a lower priority interrupt so it is ignored until the Serial ISR has completed.

3. When the Serial ISR has returned, the IC raises another processor IRQ signal and identifies the interrupt as IRQ 15. The IC has held IRQ15 pending until the Serial ISR has completed.

4. The sequence described in Scenario 1 is then repeated.

I/O TECHNIQUES 3 – DIRECT MEMORY ACCESS (DMA)

Many data transfer applications require high-speed data transfers, say > 100MB/s. Typical examples being applications such as high-speed data acquisition, reading and writing to disk drives, updating graphic displays and the streaming of network packets. A notable characteristic of this type of application is that the data tends to be formatted in blocks or bursts, for example, when a disk locates the correct track and sector it reads the data as a serial stream of bits. Another example would be that of a data packet when it is received from a network interface it is received as a bit-stream. Programmed I/O is unsuitable here because the time taken to process the I/O transfer using machine instructions severely limits the achievable data transfer rate (bandwidth). Applying interrupts to the task does not improve the situation either, in fact, it actually makes it worse because there is time overhead associated with processing the interrupt request and performing the data transfer using program I/O instructions. Direct Memory Access (DMA) resolves these issues by delegating the complete data transfer operation to hardware with minimal intervention by the system processor.

If the task is simply to transfer data, at high-speed, between a peripheral and a processor system (or vice-versa) without any intermediate processing then DMA is the appropriate I/O transfer method. Expanding on this slightly, if the task is simply to transfer data at high-speed between two devices then DMA is the appropriate I/O transfer method. This caveat allows for the possibility of memory-to-memory transfers using DMA - a technique often used to speed-up video applications.

DMAC ARCHITECTURE

A DMA system interface employs a Direct Memory Access Controller (DMAC) to take control of data transfers. The system processor (PU) is effectively ' put to sleep' while the data transfer takes place and then 'reawakened' when the transfer has completed. Figure 12 shows a basic DMA interface with the DMAC set to manage data transfers between the external I/O device, for example an ADC or a network interface, and the system RAM.

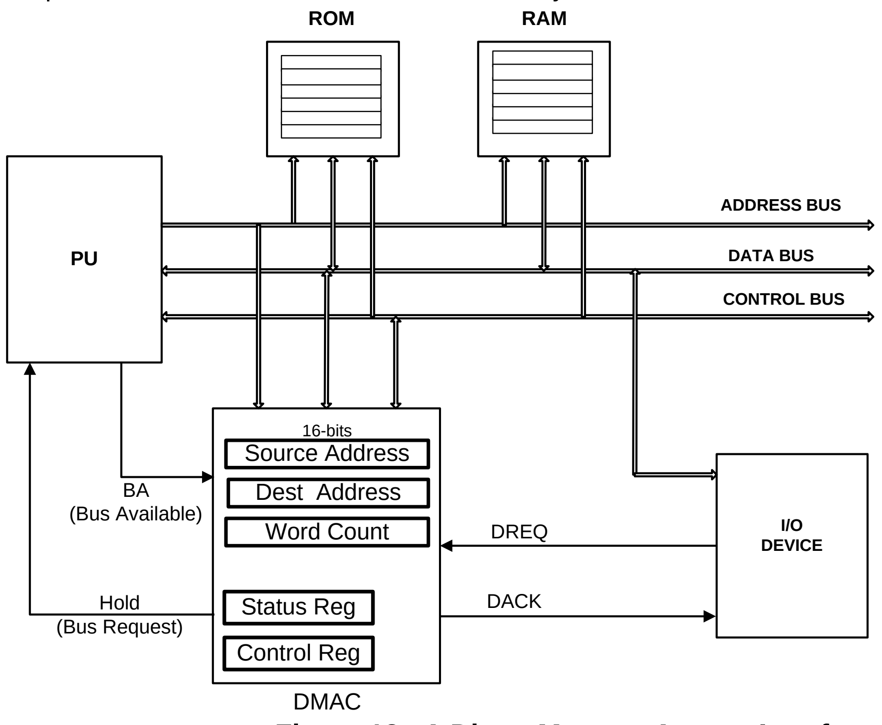


Figure 12. A Direct Memory Access Interface.

A DMAC typically supports 4 channels and is controlled like other programmable interface devices via internal control, status and data registers. Each channel has three main registers :

- **Source Address Register** - that stores the current memory address for data transfer.
- **Destination Address Register** - that stores the current memory address for data transfer.
- **Word Counter** - this is initially set equal to the block size of the transferred data and it is decremented after each word is transferred. When the word counter reaches 0 a Terminal Count (TC) Flag within the Control Register is set to indicate that the transfer is complete. Normally a hardware interrupt is generated at this point to signal the processor that the DMA transfer has completed.

At system power-up, the system initialisation code sets-up values for the address registers and word counters. These specify the source and destination of the data and the number of bytes to be transferred.

The DMAC **Control Register** is used to setup transfer parameter such as the transfer mode, for example, block mode or single cycle data transfers, the direction of the data transfer and the priority of DMA requests. Similarly, the DMAC **Status Register** indicates the current condition each DMA channel, for example, the ***terminal count*** flag (one for each channel) that indicates when a DMA transfer is in progress or is complete.

DMA TRANSFER MODES - BLOCK/BURST AND CYCLE STEALING

BLOCK/BURST MODE

The most basic form of DMA transfer, the one described above, is a '**Block Mode'** data transfer. It is appropriate when characteristic of the data to be transferred is bursts, for example, data from a disk drive, or data packets arriving over a network, or high-speed data ADC sampling. With reference to Figure 12 a typical DMA data transfer sequence would be implemented as follows:

- The I/O device requests a DMA data transfer by generating a DMA request - it asserts DREQ signal.
- The DMAC signals this request to the processor by raising the HOLD signal and in response the processor completes its current bus cycle (note that this is not an instruction fetch-execute cycle as is the case for interrupt requests) and then it puts the address and data buses into their high-impedance (tri-state) condition. Effectively, the processor is disconnected from the system bus at this point –' it is sleeping'. The processor then sends an acknowledgement Bus Available (BA) to the DMAC to signal that it can now take control of the system bus.
- When the DMAC has control of the system bus, including the read and write control signals, it sends an acknowledge signal DACK to the I/O device to signal that it is ready to begin the transfer. The DMAC has complete control of the system at this time.
- The DMA transfer now takes place - a block of data is transferred from the peripheral to memory, from memory to the peripheral, or possibly, from memory to memory.
- When the data transfer is complete the DMAC removes the HOLD signal which 'wakens' the processor so that it can resume control of the system bus and begin processing normal instruction fetch-execute cycles. In 'sleep' mode the PU retains all its original register values so the resumption of normal instruction fetch-execute cycles is almost instantaneous.

An extension to the basic I/O-memory transfers, as described above, is the Memory-to Memory data transfer. This is often performed during graphics and video processing where images are moved around the screen. Since moving screen pixels is actually achieved by moving data in memory, moving whole screens of data is equivalent to moving blocks of data in memory, hence burst DMA is an appropriate solution.

The BLOCK or BURST mode DMA transfer method has one major drawback - the processor is not active for the duration of the block data transfer. If the block size is large, many machine cycles are required to complete the transfer. Any I/O requests, for example, interrupt requests, that occur during the DMA transfer are held pending, or possibly missed, until the DMA transfer has completed. This may not be an issue when booting data from a disk but it may have a catastrophic impact on the response time of a real-time system. For this reason an alternative DMA technique, CYCLE STEALING, is often used.

CYCLE STEALING/SINGLE CYCLE MODE

In cycle stealing DMA transfer, the same hardware setup as in Figure 12 is used, but the data transfers are implemented as frequent single cycle transfers which are interspersed with normal processor bus read and write cycles. Typically, single word transfers are interspersed with instruction fetch execute cycles. The DMAC grabs a clock cycle to implement a single word transfer and then returns control to the processor - in effect the DMAC steals a cycle from the processor.

Cycle stealing has the advantage that the processor is not 'sleeping ' for long periods. The system can, therefore, respond quickly to other external I/O events, such as interrupts, and real-time performance is maintained.

A REFLECTION ON THE BASIC FETCH-EXECUTE CYCLE

Notice that the basic fetch-execute cycle as described earlier in the course has now become somewhat more complex: It now reads :

>> Instruction Fetch >> Instruction Execute >> Test for Interrupt request?

DMA requests can be expected at anytime and are tested for at the end of every bus cycle. Explicitly this means that DMA requests are tested at the end of every bus read or write data transfer.

BUFFERED DMA INTERFACES

As discussed previously, typical applications for DMA handle data streams arriving in blocks or packets. In general, the processor system can generate or consume data very much faster than peripheral devices can generate it and this leads to the situation where the processor is starved of data and has to wait while the peripheral catches-up. Conversely, when data is sent from the processer, the peripheral is overwhelmed with data because the processor is sending data much faster than the peripheral can consume it. In such circumstances an intermediate memory buffer is required and its purpose is to even-out the speed mismatch between the processor and the I/O device. A Buffered DMA interface is shown in Figure 13.

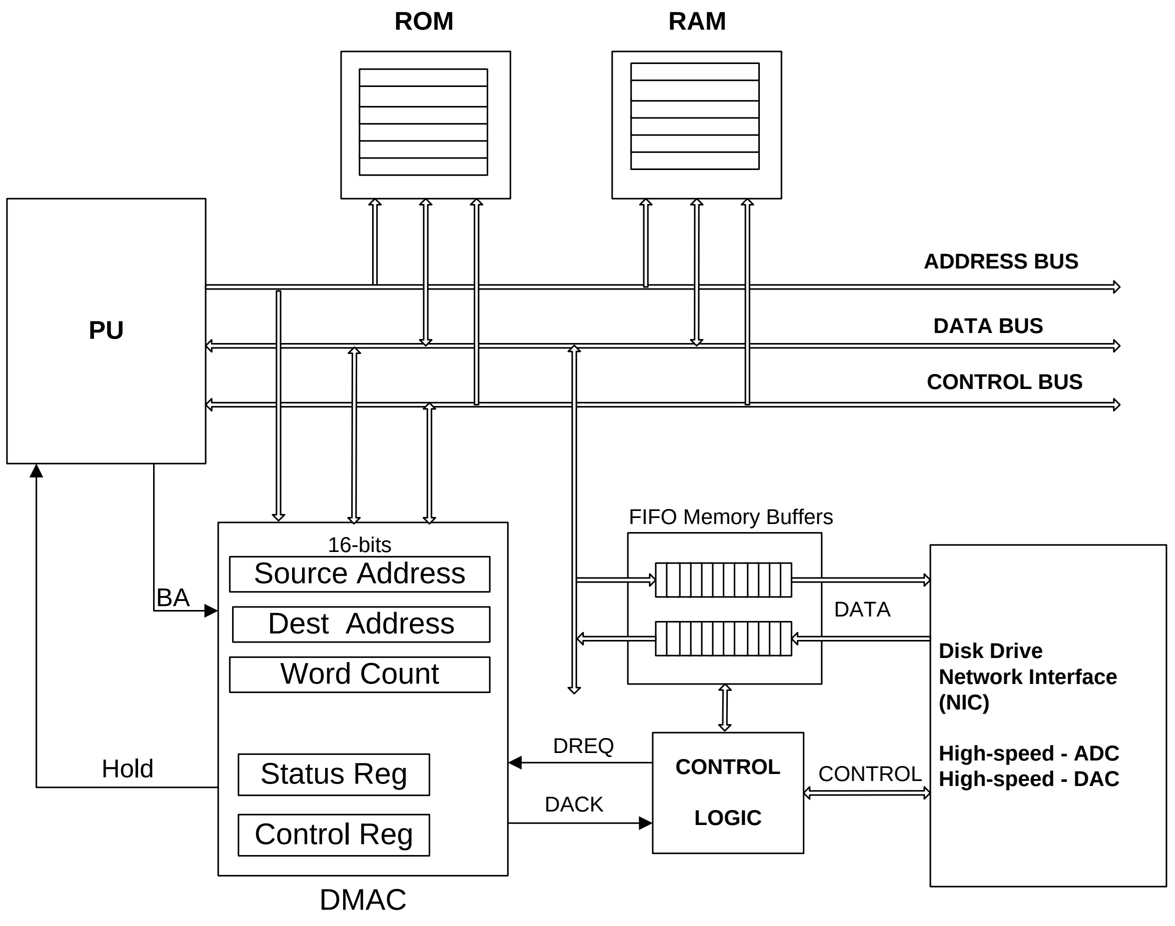


Figure 13. A Buffered DMA Interface

The addition of the First-In-First-Out (FIFO) memory buffer allows data to be stored temporarily before it is transferred to its destination. The FIFO operates like a pipe where data is written into one end and is extracted (read) from the other. The key point is that the FIFO can perform read and write operations simultaneously – the FIFO can be filling at one end whilst simultaneously emptying at the other. Typical FIFO lengths are 2-512k (Bytes or Words).

In addition to the standard READ(RD) and WRITE(WR) control signals, FIFO memory devices have several control signals that indicate their status, for example, FULL , HALF-FULL and EMPTY and these are often used to generate interrupt or DMA requests. Note that in Figure 13 there are actually two FIFO's, one in the output direction for write data and one in the input direction for read data.

The following sequences how the FIFO control signals are used to perform buffered data transfers:

A typical buffered read transfer, say from RAM to an output device, would use a DMA transfer to write a block of samples into the FIFO and when the FIFO becomes, say HALF-FULL, the control logic generates a signal to the I/O device which then starts to empty the FIFO. The processor system continues to write data to the FIFO and only stops if the FIFO becomes full. Additionally the I/O device continues to read data from the FIFO and when the FIFO empties to say, HALF-FULL or EMPTY, a new DMA request is generated.

A typical buffered write transfer from an input device into system RAM begins with the I/O device writing a block of samples into the FIFO. When the FIFO becomes, say, HALF-FULL, the control logic generates a DMA request which the processor system responds to by reading data from the FIFO until it becomes empty, at which point the DMA transfer stops until the buffer fills again. Note that, while the DMA transfer is actually taking place the IO device can still be simultaneously writing data into the other end of the FIFO.

DMA IN MICROSOFT WINDOWS

Windows supports two types of DMA :

- System DMA, which is similar to the architecture described in Figure 12, uses a separate DMA controller to centralise the management and control of DMA transfers. This technique has fallen out-of-favour in PC systems but is still widely used in many embedded system architectures

- Bus-Master DMA is effectively a distributed DMA architecture and is the most common approach in current PC systems. In this type of architecture individual devices are equipped with DMAC hardware and any device is capable of becoming the bus master in order to initiate DMA transfers with the host system memory.

MANAGING I/O THROUGH THE OPERATING SYSTEM

A fundamental role of any Operating System (OS) is to manage computer system resources. Within the OS it is the task of the I/O Manager to manage I/O hardware resources and I/O data transfers. Figure 14 identifies the main the components the makeup a typical I/O subsystem.

The OS manages I/O devices as if they were files and user mode applications gain access to I//O devices using the standard file operations - OPEN, READ, WRITE and CLOSE. When an application requires access to I/O hardware it firstly creates a file, which becomes is a link (called a 'handle') to the *Logical Device File* created by the device driver when it is loaded. The I/O Manager redirects any instructions made to the file to the device driver and the device driver executes the associated operation. Communication is made by the driver to the actual registers and ports on the hardware device using a set of well-defined HAL routines.

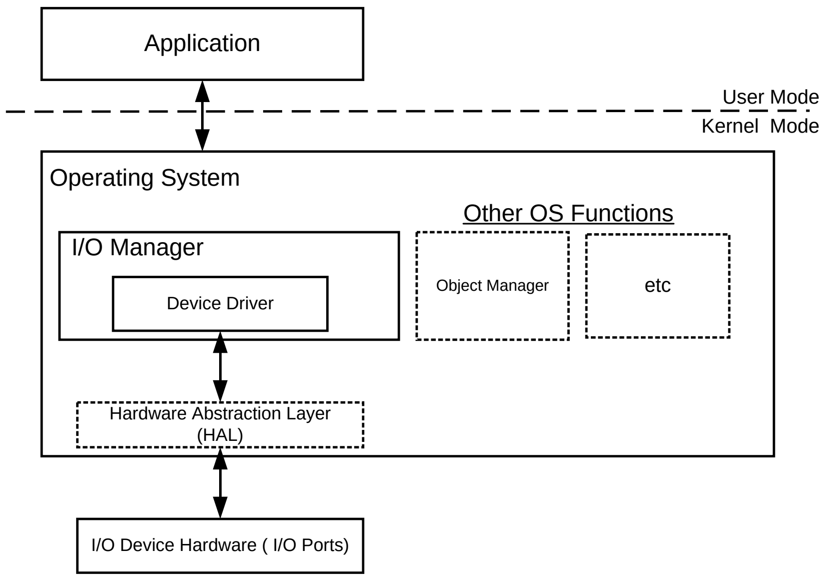


Figure 14 The I/O Subsystem

The separation of USER and KERNEL MODES improves the system reliability and security. User applications cannot use certain ' privileged ' processor instructions, for example IN and OUT instructions, only the OS has access to these instructions since it is only the OS that can perform I/O. Any user mode application that attempts to use a privileged instruction is trapped by the OS and the user is alerted via and a suitable warning message, typically something like 'privilege mode exception error'. Users have no direct access to the system hardware and communication with all hardware is performed safely by the OS using a combination of the I/O manager, the device driver and the HAL.

It should be noted that many simpler embedded operating systems do not offer such protection and users can directly access hardware resources. This is perfectly acceptable in the majority of cases because the OS is operating autonomously (no users) and the design criteria is for highest performance and fastest response time to external events.

I/O MANAGER

The I/O Manager controls access to all File Objects on the operating system, and provides the typical create, read, write, open and close commands as the available interface. The important feature regarding the I/O Manager is that it provides a packet based asynchronous communication I/O subsystem. The data packets are called I/O Request Packets (IRPs) and they provide the main mechanism behind device driver communication.

Note that the I/O Manager is responsible for the dynamic loading and unloading of the device driver. Drivers are loaded and unloaded into the I/O manager as the physical devices are added and removed from the system. Drivers can be loaded when the OS boots, or automatically when the device is added to the system (plug n play), or manually by the system user.

DEVICE DRIVER

The device driver forms the link between the OS and the actual physical devices. File I/O requests submitted by to the I/O manager are translated into sequences of I/O operations during the actual transfer of data. It should be noted from Figure 14 that there is not a direct link from the device driver to the actual I/O ports and registers, although this can be the case in simpler embedded systems. HAL, is discussed below, but in essence it's goal is to allow the same device driver to operate on multiple hardware platforms - PC's for example use a range of processor types and motherboard manufacturers. The device driver issues I/O requests in a standard manner and the HAL translates them into machine specific operations, such a I/O port and register accesses.

HARDWARE ABSTRACTION LAYER (HAL)

A HAL does not exist on all OS's but it is used extensively in Windows and Linux. HAL provides an elegant solution to the problem of running the same operating system on different system hardware architectures - for example, Windows runs on Intel and AMD processors which have different instruction mixes. HAL is a layer of software that isolates the OS functionality from the low-level, processor specific operations. It achieves this by providing a standard set of functions, for example :

Readport_UChar(); Writeport_Uchar(); Read_Register_UChar(); Write_Register_Uchar();

When the OS is installed HAL links to a set of machine specific macro routines that do the actual machine specific register and port data transfers. The HAL routines provide a consistent interface that allows the same device driver to run on machines with different architectures. Machine specific hardware operations, such as, I/O port management and data transfers, interrupt and DMA management are hidden all from the device driver. HAL makes the translation and system builders, notably BIOS developers in a PC environment, must link their machine specific BIOS routines to the routines made available by the OS via the HAL.

I/O DEVICE HARDWARE

As discussed in previously, and shown in Figure 2, programmable I/O interfaces contain a number of internal registers and ports. Some interfaces, such as, parallel printer ports, are simple, consisting of only a few registers and ports, however, more complex devices, for example, an audio soundport interface is much more complex and may contain between 200-300 registers and ports.

The previous discussion highlights some of the underlying complexity associated with performing I/O data transfers. The presence of an OS, such as Windows or Linux, provides a consistent well-structured environment that allows multiple users to perform I/O through a variety of device interfaces both reliably and securely. It should be noted in closing, however, that many embedded computer systems do not have the resources, notably memory and processor power, to support a complex operating system. In such circumstances direct I/O transfers are possible and although this results in faster data transfers, reliability is significantly reduced.

END OF UNIT - SUMMARY OF ACHIEVEMENTS

On completion of this unit you should have achieved the following :

- Understanding of the principles of programmable interfaces and I/O data transfers.
- Understanding of the principles of flag polling and simple interrupt systems.
- Understand the system hardware requirements for processing interrupt requests.
- Understand the Interrupt Controller and the mechanisms required to handle multiple interrupts. system.
- Understand the principles of Direct Memory Access (DMA) systems.
- Understand the relationships between the Operating System and the underlying systems hardware.

• Frank Greig • School of Computing • Edinburgh Napier University • Edinburgh •

Faculty of Engineering, Computing & Creative Industries

Edinburgh Napier UNIVERSITY

Module : Systems and Services

Module Number : CSN08101

Unit 5 : I/O Bus Systems

PCI,SATA and USB

Student Study Material

UNIT 5: I/O BUS SYSTEMS

OVERVIEW

This unit introduces of Input-Output (I/O) bus systems and the underlying technologies that data around computer systems. The basics of both serial and parallel data transfers methods were covered in Unit 2 and this Unit builds on that work to take a more in depth look at the bus technologies found in PC systems, in particular, PCI, SATA, USB.

A common theme that runs through the discussion is that many of the current developments in bus technologies are firmly based on developments in network technologies. The shift from parallel to high-speed serial transfer technologies and the use of layered architectural models are two of the most striking similarities. The well known layered model architecture used in networks now has its equivalents in PCI,SATA and USB bus systems and a layered model is introduced for each of these bus systems.

LEARNING OUTCOMES

On completion of this unit you should :

- Understand the principles of parallel and serial data transfers systems and be able to compare their inherent strengths and weaknesses.
- Understand the operating principles of the common bus systems - PCI, SATA and USB.
- Know where to locate and apply Web based resources that inform system developers about current and future developments on a variety of bus technologies.

RESOURCES REQUIRED :

PC microcomputer equipped with Internet Access

ADDITIONAL MATERIAL

There are many excellent websites with application notes and papers on topic of bus systems technology. Also Wikipedia provides a wealth of information and links on these topics:

PCI TECHNOLOGY AVAILABLE ON THE WEB

Source	Comment
www.pcisig.com/specifications/pciexpress/	All things PCI
www.intel.com/technology/pciexpress/devnet/	Whitepaper – pdf format + tutorials
http://www.youtube.com/watch?v=7HEbXQItb6E&feature=related	Excellent Video tutorial PCI evolution
www.hardwaresecrets.com/article/190	Useful tutorial
zone.ni.com/devzone/cda/tut/p/id/3540	Useful tutorial
http://www.inno-logic.com/resourcesPCIE.html	Useful tutorial

SATA TECHNOLOGY AVAILABLE ON THE WEB

Source	Comment
www.sata-io.org	Official SATA site
www.sata-io.org/technology/6Gbdetails.asp	General Information on all things SATA
www.addonics.com/emerging_technologies/sata_tutorial.asp	Useful SATA information – protocol analysers
http://www.serialtek.com/sata_protocol_overview.asp	Very good with videos

USB TECHNOLOGY AVAILABLE ON THE WEB

Source	Comment
www.USB.org	Official USB site
http://www.inno-logic.com/resourcesUSB3.html	General Information on all things USB
http://www.inno-logic.com/resourcesUSB3.html	General Information on USB3.0
www.beyondlogic.org	Useful USB information – protocol analysers
www.nxp.com/acrobat_download/literature/9397/75009316.pdf	USB-OTG tutorial (pdf)
http://www.everythingusb.com/superspeed-usb.html#3	USB 3.0 information.
http://www.inno-logic.com/resourcesWUSB.html	Wireless USB Introduction

.

TUTORIAL : BUS SYSTEMS

LEARNING OUTCOME

On completion of this tutorial you should understand the principles of serial and parallel data transfers and be able to describe the operating principles of the common bus systems - PCI, SATA and USB.

1. i. PCI Express (PCIe) is the latest step in the evolution of the PCI Bus. Describe the main features of the PCIe bus interface.
   ii. PCI Express is seen as a replacement for PCI-x (parallel bus) and AGP bus interfaces in graphics systems. Discuss the limitations of the existing buses and identify the improvements that PCI Express will bring.

2. i. PC microcomputers have adopted the Universal Serial Bus (USB) as the standard for serial peripheral interface devices. Summarise the main features of the USB interface paying particular attention to data transfers speeds, bus topology, transfer types and device enumeration.

   ii. USB 2.0 has now evolved to USB 3.0. Discuss the main improvements offered in the latest version of the USB protocol.

3. i. Serial Advanced Technology Attachment (SATA) is the latest step in the evolution of disk drive interfaces. Describe the main features of the SATA interface paying particular attention to data transfer speed and connection technology.
   ii. External SATA (eSATA) interfaces are becoming readily available in PC systems. Discuss the basic operational details of eSATA interfaces.
   iii. Will eSATA put an end to USB connected disk drives?
   iv. What are the benefits of using SATA in RAID systems?

FURTHER READING AND PRACTICAL WORK : BUS SYSTEMS

LEARNING OUTCOME

These topics are intended to broaden your reading and improve you approach to academic research and investigation. On completion of this exercise you should be able locate and use Web based support resources for broadening your knowledge on the subject of PC bus system technologies.

PRACTICAL

Using the resources found at the Websites listed above your starting point to answer the following :

1. Locate the specification for a new motherboard and list all the available bus systems – PCI, USB, SATA etc. Against each bus note the number of interfaces provided and their speed of operation.

2. For PCI :
   i. What speed of operation is proposed for the next generation of PCI Express systems?
   ii. The serial nature of PCI EXPRESS could lead to the development of new distributed PC Architecture, where, for example, the processor and memory unit is completely separate for the graphics and disk units but are linked using PCI connections. Conduct a search to ascertain if there is any developments in this area and discuss what you believe would be the advantages of such an architecture.

3. For USB :
   Universal Serial Bus (USB) has evolved into much more than a PC-centric high-speed wired peripheral interface bus. Two notable examples of this are USB –OTG and USB-UWB. Describe the main operating principles behind these derivative USB technologies and evaluate their likely impact on existing peripheral interface methods.

4. For SATA :
   i. What are the current and future data rates?
   ii. What is eSATA ?
   iii. What is SATA express ?

Bus Systems 1 - Peripheral Component Interface Bus (PCI)

To compete with the advances in processor technology, bus systems must get ever faster. In PC systems, for example, the **Dual Independent Bus** (DIB) architecture was introduced in order to keep pace with increases in processor speeds. DIB replaced the original single system bus with a **Frontside Bus** and a **Backside Bus**. The architecture illustrated in Figure 15 shows that the backside bus provides a direct, fast channel, between the processor and the Level 2 cache while the frontside bus links directly to the Northbridge device and houses the memory controller which interfaces to the system memory. Additionally, the Northbridge chip has an AGP interface and a BUS BRIDGE that links to the Southbridge chip and provides a PCI bus.

In Figure 15 the devices, identified as Northbridge and Southbridge act as signal routers that transfer data to and from the processor - they are complex devices that are required to support a wide-range of devices and signal protocols.

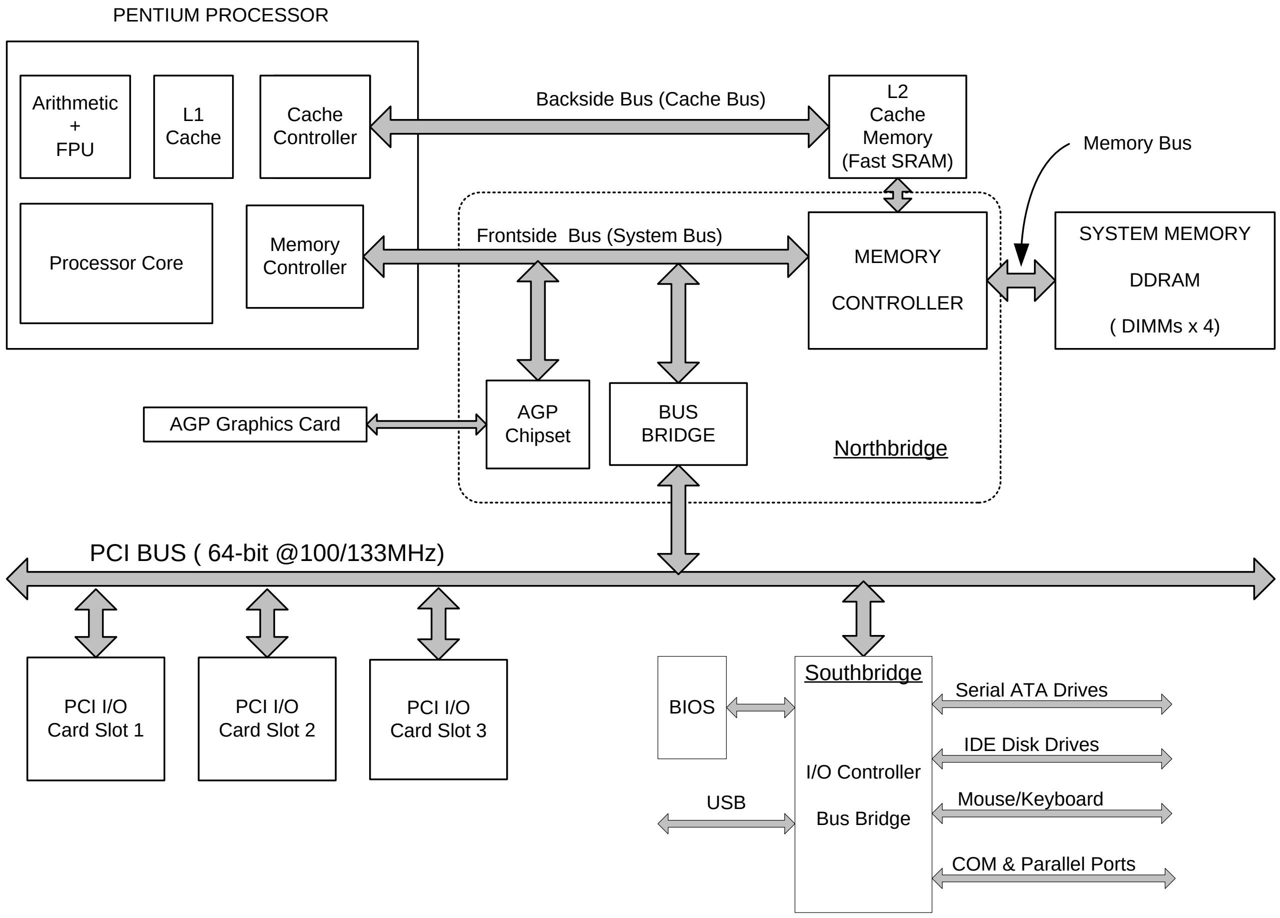


Figure 15. A legacy PC Bus System

At the highest level, support for external devices is provided by the **Peripheral Component Interface Bus** (PCI) which allows plug-in peripheral cards, for example, as sound, graphic and network controller cards.

In addition to the PCI bus, a PC provides bus systems that are dedicated to specific system functions such as Mouse, Keyboard, Disk Drive, etc. These functions are provided by the Southbridge device which acts as an I/O Controller for wide range of buses and peripherals, including USB, SATA, Mouse and Keyboard. Figure 15 also shows that the Southbridge chip also provides an interface to the BIOS chip and legacy interfaces, such as Comports, IDE and Parallel/Printer Ports.

PCI – Limitations

In the form represented in Figure 15 the PCI bus is a parallel system and it suffers from several performance limitations.

- PCI is a parallel bus requiring expensive multi-pin connector with around 100 contacts.
- PCI is a shared bus meaning that all devices share the same data leading to bus contention and arbitration issues.
- Being a shared, multi-drop, bus means that the electrical characteristics of the bus are difficult to control at high-speeds. Clock Skew, signal reflections and crosstalk all degrade the signal integrity and cause bus errors as the speed increases. These are the real factor that limit the speed of parallel multi-drop bus systems

It is safe to say the PCI in its parallel form has reached the end-of-the-road. It can no longer meet the demanding speed requirements of Graphics cards and Gigabit network connections, and this has led system designers to develop new bus systems that will meet the demands of high –speed peripherals.

PCI EXPRESS

PCI Express (PCIe) bus technology has now replaced older - PCI, PCI-X and AGP bus systems . The reason is simple, these bus systems can no longer meets the speed requirements of high-performance graphics/multimedia and server systems. Speed improvements in processor and memory technology have not been matched by speed improvements in PCI and AGP systems and the multi-drop nature of PCI bus means that the technology has reached its limit in terms of bus clock speed. Speed-up techniques like quad edged clocking at 133MHz gave an effective 533MHz clock frequency and assuming a 64-bit parallel bus gives a limited bus bandwidth of 8 x 266 = 4.2GBps for PCI-X and this does not nearly meet the requirements of high-end graphic systems and network servers (8-16GBps). Since PCI has reached its limits a new approach is essential.

In terms of options for a solution, the obvious candidates are: increase the clock speed and/or make the bus wider (say 128-bits instead of 64) but these are quickly ruled out for the following reasons:

- It takes a finite amount of time for electrical pulses to travel down a wire (1ns/10cm approximately) and there comes a point where the smallest variation in the physical length between adjacent bus lines causes the signals to skew sufficiently to cause bit errors. The fact that PCI-X can deliver at 266MHz is something of a miracle and anything in excess of this is impossible for a parallel multi-drop bus system. Additionally, the interaction between high-speed pulses travelling on adjacent pins generates high levels crosstalk that also degrades the signal integrity and leads to increasing bit-errors.

- Increasing the bus width means much larger, more complex connectors that are also more expensive to manufacture. Also a larger amount of signals radiates more electrical noise and interference and again the cures, screened connectors and wired connections, would increase costs. Also larger connectors occupy significantly more board area and increase the potential for failure – essentially more pins means that there is more to go wrong.

Future systems need bus bandwidths in excess of 8GBps and PCIe is the current solution. It borrows on heavily from advances in network technology, in particular Gigabit Ethernet and switching technology. Current Network technologies, for example ,packet switching and gigabit/second cable drivers are used to good advantage in specifying the ***scalable*** high-speed ***serial*** bus that is PCIe. This approach is in complete contrast to PCI and PCI-X parallel bus systems where read and write operations are performed using parallel address and data buses. PCIe adopts a completely different architecture, one that can be described as **point-to-point packet-switched system based on a layered network model**.

PCI Express has a layered architecture and the similarities to the ISO Reference and TCP/IP network models in Figure 16 are obvious. For the PCIe layered model this ensures the reliable delivery of data in packets over a serial point-to-point link. For the current PCIe specification the link speed is 500MB/s and given that physical layer allows for multiple serial links (the scalable aspect of the specification) then increases in data rate are easily achieved.

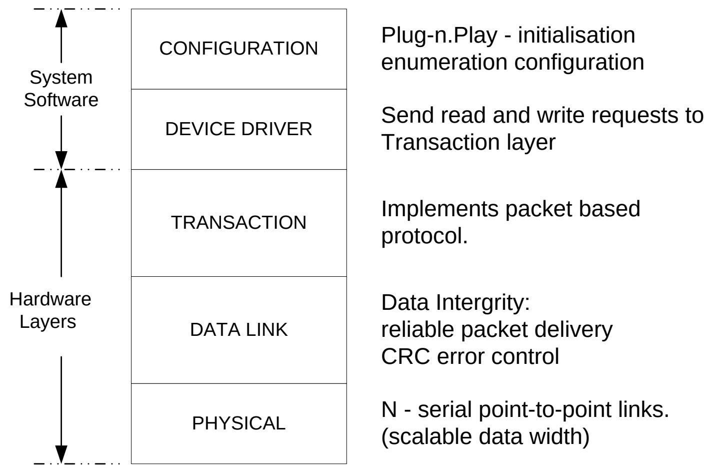

System Software
Hardware Layers
CONFIGURATION
DEVICE DRIVER
TRANSACTION
DATA LINK
PHYSICAL
Plug-n.Play - initialisation enumeration configuration
Send read and write requests to Transaction layer
Implements packet based protocol.
Data Intergrity: reliable packet delivery CRC error control
N - serial point-to-point links. (scalable data width)

Figure 16 PCI Express Layered Architecture (PCI - Protocol Stack).

The layered model approach offer the following advantages:

- It facilitates future changes. By separating the physical layer from the link layer, independent development of faster physical link speeds is possible. There is a 10Gbps upper limit on copper but optical connections are an obvious way forward.

- It ensures software compatibility between PCI Express and existing PCI systems. All the data transformations take place in the lower layers and are implemented in hardware. The device driver layer has no knowledge of how the layers below achieve the exchange of data.

PCI EXPRESS – PHYSICAL LAYER

Link-speeds are maximised using low-voltage differential signalling. As shown in Figure 17 two pairs of wires are used to give a full-duplex connection, however, the bus supports dual-simplex operation which means that both links can operate in the same direction and the available bandwidth is doubled. Reliable data synchronisation and data recovery is ensured by encoding the transmitted data using the 8b/10b encoding scheme. 8b/10b allows the originating clock signal to be embedded in the data stream but leads to a 20% overhead. Given a 0.5Gbps link speed then the corresponding data bandwidth is 5 x 0.8 /8 = 500MBps. Note that these figures apply to PCIe 2.0: the PCIe 1.0 link speed was 250MBps.

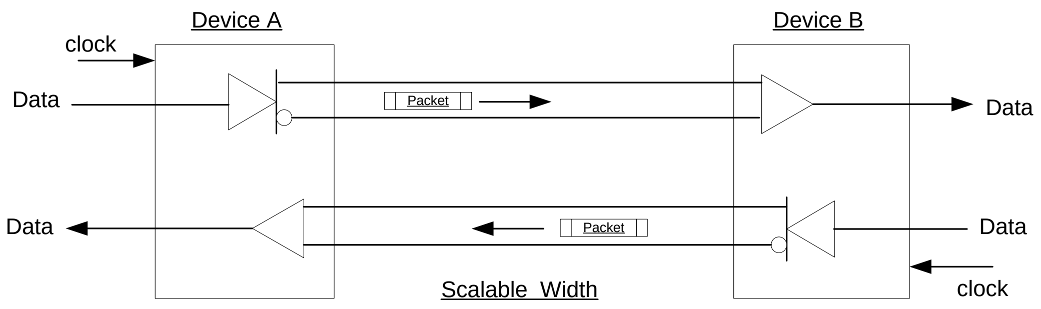


Figure 17 - Differential transmit and receive communication links.

A signal pair as shown in Figure 17 is referred to as a 'lane' and the scalable nature of physical layer allows multiple lanes to be added to increase the available bandwidth. PCI Express x1 operates at 250MBps in either direction. The figure given in brackets if the aggregated speed for dual-simplex operation. Table 1 compares the data rates for various bus technologies.

BUS	MAXIMUM TRANSFER RATE GB/S	COMMENTS
PCI	1.06	8Bytes x 133MHz (64-bit bus)
PCI – X 64 /266	2.128 GBps	
PCI – X 64 /533	4.256 GBps	
AGP 8X	2.1	4Bytes x 66MHz x 8 (32-bit bus)
PCIe 1x (ver: 1.0)	0.25 [0.5]	0.5 Gbps dual simplex operation
PCIe 1x (ver: 2.0)	0.5 [1.0]	1.0 Gbps dual simplex operation
PCIe 1x (ver: 3.0)	1.0 [2.0]	1GBs = 8Gbps
PCIe 2x	2.0 [4.0]	
PCIe 4x	4.0 [8.0]	
PCIe 8x	8.0 [16.0]	
PCIe 16x	16.0 [32.0]	
PCIe 32x	32.0 [64.0]	
SATA 2.0	0.30 Gbps (8b/10b coding)	300MB/s $300 \times 10^9 \times 8/10$
SATA 3.0	0.60 Gbps	600MB/s
Gigabit Ethernet	1.0 Gbps	8b/10b gives 0.8 x 1250 = 1000
USB 2.0	0.06 Gbps	480/8

Table 1 A Comparison of Maximum Bus transfer rates.

The scalable nature of PCI Express means that additional channels can be added (aggregated) to create faster bus speeds, typically 16GBps at present but moving to 32GBps as future systems demand higher bandwidths. At these speeds PCI Express is more than capable of meeting the demands of high performance systems such as graphics interfaces, servers and RAID arrays.

PCI EXPRESS 3.0 & 4.0

In keeping with the normal trend for increased performance specifications PCIe 3.0 provides a basic link speed of 1GB/s. This x2 improvement over PCIe 2.0 @ 500MB/s has been achieved through improvements to the signalling methods, clock recovery method and signal encoding, for example 8b/10b encoding has been replaced with a scrambling encoding scheme.

PCI Express 4.0 which has been announced promises 16 GT/second (16 billion discrete data packet transfers/second) but it won't be available until around 2014/15.

PCI EXPRESS – LINK LAYER

The link layer is responsible for reliable packet delivery. This layer adds a packet sequence number header and a trailer in the form CRC error check. The sequence number ensures that packets are re-assembled correctly if retransmission is necessary. The link layer protocol implements flow control to ensure that packets are only sent when the receiver has buffer space available to receive them. This avoids buffer overflows and makes better utilisation of the available bandwidth.

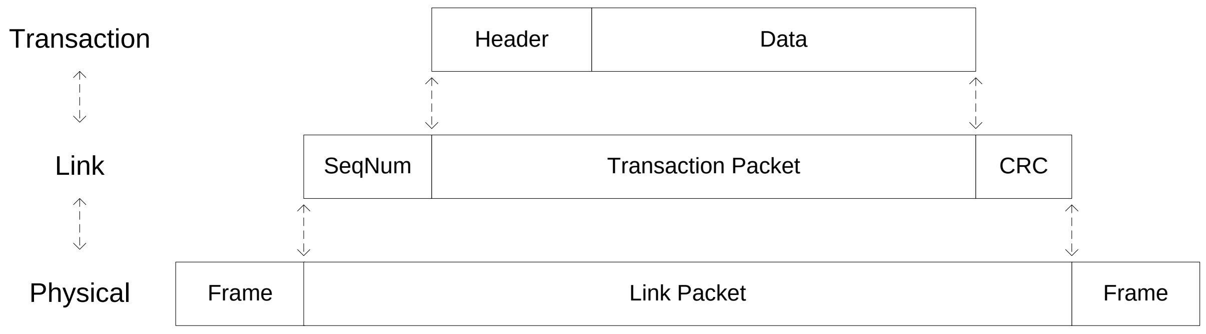


Figure 18 PCI Express packet encapsulation.

Note the encapsulation of data as it passes down the layers: Physical frames encapsulate link and transaction layer packets in the transmit direction. Similarly, packets are stripped of their encapsulation in the receive direction. Virtual communications links are established with equivalent layers in sending and receiving devices.

PCI EXPRESS – TRANSACTION LAYER

The transaction layer processes requests received from the software/driver layer. Typically, these could be for read or write transfers but equally they can be for bus transactions, such as, interrupt, direct memory access and power management. Transaction packets can request 32 or 64-bit transfers and they include memory, I/O addresses and data within their payload.

The transaction layer communicates with the upper driver layers through address spaces, namely, memory address, I/O address, configuration and message. Memory and I/O are obvious; the configuration and message address spaces need some elaboration:

- The configuration address space is used by the operating system, via the driver layer, firstly to establish the hardware I/O that is present in the system. It then uses plug-n-play to allocate resources such as memory, I/O, interrupt numbers and DMA channels.

- The message address is the mechanism that translates parallel bus activities, for example, real signals such as interrupt, direct memory access requests and bus errors into a packet stream. Parallel bus transactions are simply encoded into packets and forwarded across the serial bus. The transaction layer identifies and processes these activities so that they are completely invisible the software based driver layer.

PCI EXPRESS – SOFTWARE LAYERS

Backward compatibility is a key concept for the successful introduction of PCI express. Simply put – software that run on PCI and PCI-X technologies should run, without modification, on PCI Express platforms. To achieve this the lower layers of the PCI Express platform must look like the flat - address, data and control - model because that is what the older PCI device drivers saw. With backward compatibility assured, existing software runs unaltered and forward development can ensure that new devices take advantage of the increased bandwidths available with PCI Express.

PCI EXPRESS BUS INTERFACE

The application of PCI Express in a PC environment is shown in Figure 19.

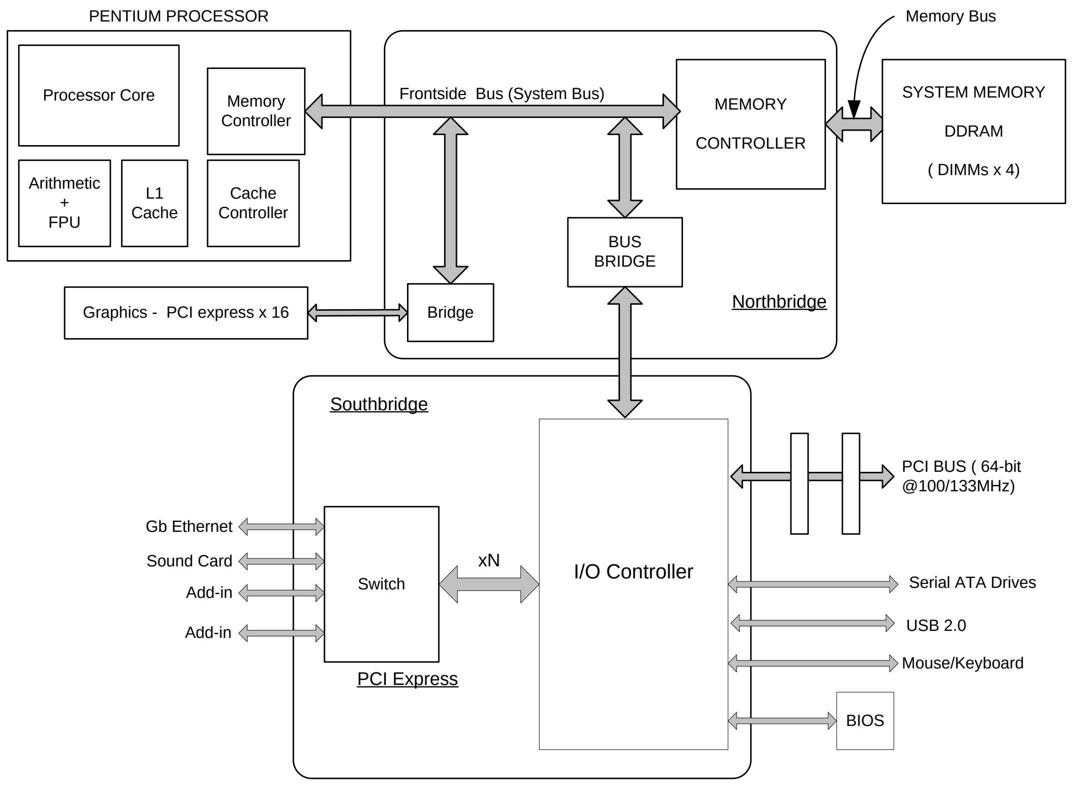


Figure 19 A PC system using PCI Express bus interfaces.

The main features are :

- The Northbridge and Southbridge chips now include bridges that provide support peripheral devices. In the Northbridge chip, AGP has been replaced by a dedicated x16 PCI Express link. The graphics interface, can achieved a 16GBps data transfer rate, by aggregating 16 x 1.0GBps lanes.
- The Southbridge chip provides a xN PCI Express switch that allows add-in external devices. In Figure 19 four I/O PCI express lanes are provided for standard add-in peripheral devices, NICs, sound cards etc.
- The Southbridge chip provides an I/O controller to support for, USB, SATA etc and also provides legacy support PCI-X 2.0 bus systems.

PCI EXPRESS CONNECTOR

The PCI Express connector in Figure 20 looks similar to the PCI- X 2.0 types but they are very different. This graphics card has 16-serial lanes, occupying (16x 4 = 64) pins, whereas PCI-X 2.0 uses 64 parallel address and data pins.

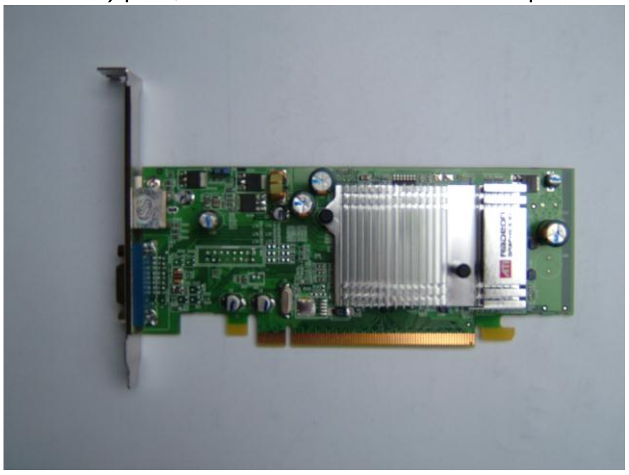


Figure 20 A Graphics Card showing PCI Express connector.

ADDITIONAL FEATURES OF PCI-EXPRESS

- PCI Express boards conform to the ‘ Hot plug and play’ standard. Boards can be inserted and removed with the power switched on.
- PCI Express is software backward compatible with existing PCI systems. The existing software device drivers and operating system support does not need to be changed.

BUS SYSTEMS 2 - SERIAL ADVANCED TECHNOLOGY ATTACHMENT (SATA)

SATA is the replacement for Parallel ATA (PATA) better known as the IDE (Integrated Drive Electronics) disk drive interface. The need-for-speed in PCI bus technologies applies equally to disk drive interfaces where the familiar 40-pin IDE ribbon cable (or 80-pin for Ultra/ATA) connector has reached the end-of-the-line. Higher data rates and lower costs are again the common themes and unsurprisingly, the solution is the same – move from parallel to serial bus implementation. The similarities between SATA and PCI Express should not go unnoticed, notably, both are serial, both use point-to-point connections and both use packet based data transfers. The major difference is that SATA is a dedicated link, it is for disk drives only, whereas PCI express is a general purpose peripheral interface. Table 1 give a comparison of SATA versions and their technical a operational improvements over the older PATA/IDE connections .

SATA 2.0 /SATA 3.0	COMMENTS – SATA IMPROVEMENTS
300MBps/ 600MBps >> 3Gbps/6Gbps (using 8b/10b)	Improved speed with each revision
First-party DMA	Improved performance
Command Queuing	Intelligent data handling
CRC	Enhanced error detection
Point-to-Point	Simpler connection.
7-wire connector	Smaller lighter cables and lower cost connectors 2x2 serial + 3 ground connections.
1m	Longer cables
Hot plug-n-play	Ease of use

Table 2 Some basic SATA operating characteristics.

The most obvious improvements in SATA are speed and ease of use. SATA is designed for future speed improvements with the benefits of simpler connections offered by adopting a serial interface.

The architecture of SATA is similar to PCI Express with the important exception that SATA is not intended to be scalable - there is a single point-to-point link between a host and a drive. SATA again borrows heavily from network technology, using packet switching, gigabit/second cable drivers and a layered communications model.

Similarities to the ISO Reference and TCP/IP network models in Figure 21 are again obvious. Overall the SATA layered model ensures the reliable delivery of data in packets over a serial point-to-point link operating at 3.0Gbps in a single direction.

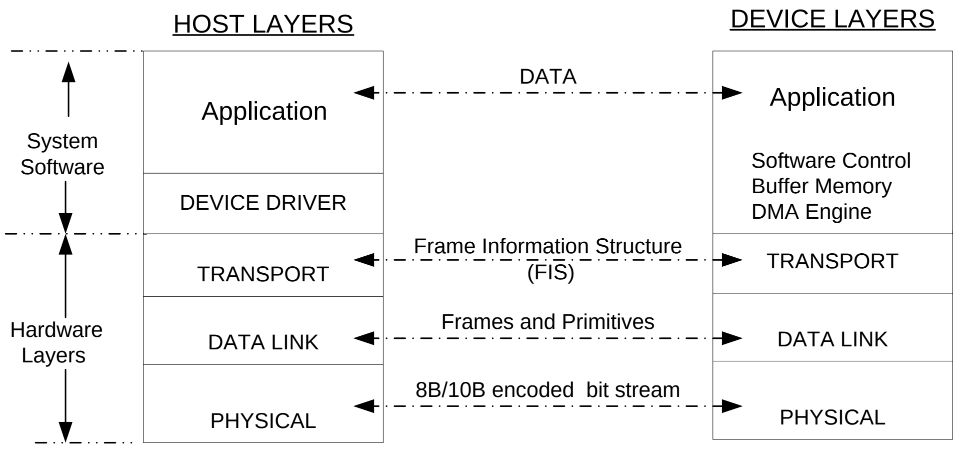


Figure 21 SATA Layered Architecture (SATA - Protocol Stack)..

The layered model approach offers the following advantages:

- It facilitates future changes, for example, separating the physical layer from the link layer allows independent development of faster physical link speeds - 10Gbps is the upper limit copper so optical connections are an obvious way forward.
- It ensures compatibility between PATA interface and SATA. All the data transformations take place in the lower layers and are implemented in hardware. The existing device driver layer has no knowledge of how the layers below achieve the exchange of data and can therefore be made compatible.

SATA – Physical Layer

Figure 22 shows a SATA interface with two pairs of wires used to give a full-duplex connection. Bit rate is maximised using low-voltage differential (LVDT) signalling. Reliable data synchronisation and data recovery are ensured by encoding the transmitted data using the 8b/10b encoding scheme. 8b/10b allows the originating clock signal to be embedded in the data stream but leads to a 20% overhead. Given a 3.0Gbps link speed then the corresponding data bandwidth is 3.0 x 0.8 /8 = 300MBps.

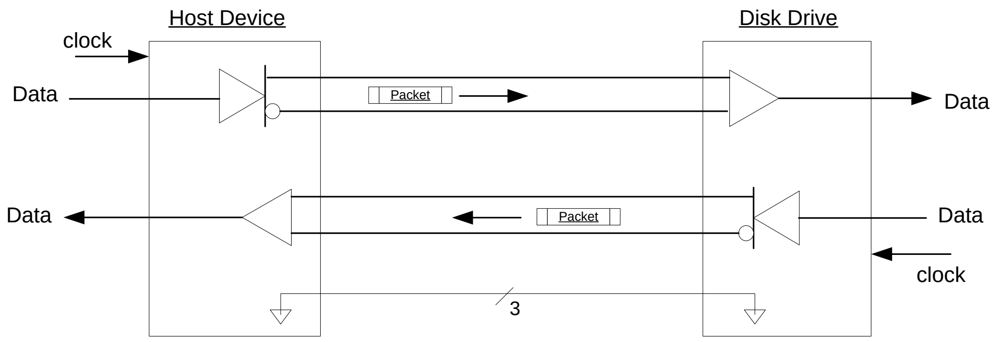


Figure 22 - SATA differential transmit and receive and power links.

Figure 22 shows the 7-pin connections. Four lines implement two differential pairs operating at around 250mV. The three ground signals provide a solid reference 0V reference and are placed between the transmit and receive signal pairs in order to minimise signal interference from crosstalk. SATA disk drives also require a 15-pin power connector

SATA – Link Layer

The link layer deals with two type of data structures, **primitives** and **frames**. A primitive is a single DWORD (32-bit word) that controls the transfer of data between a host and a drive device. Typically a primitive can issue a command or indicate status, for example:

X-RDY – indicates that the host or drive has a data payload ready for transmission.
R_RDY – indicates that the host or drive is ready to receive a data payload.
R_OK - indicates that the host or drive has received a data payload without error.
HOLD/HOLDA - used to control the flow of data payloads (not in the DMA sense).
SOF/ EOF – start of frame and end-of-frame delimiters.

Frames are the mechanism that communicates information between a host and a drive device and a frame comprises multiple dwords and, as shown in Figure 23. The Frame Information Structure (FIS) encapsulates data and primitives and are assembled in the transport layer and further encapsulated with a SOF primitive and trailed by a CRC and EOF primitive in the link layer.

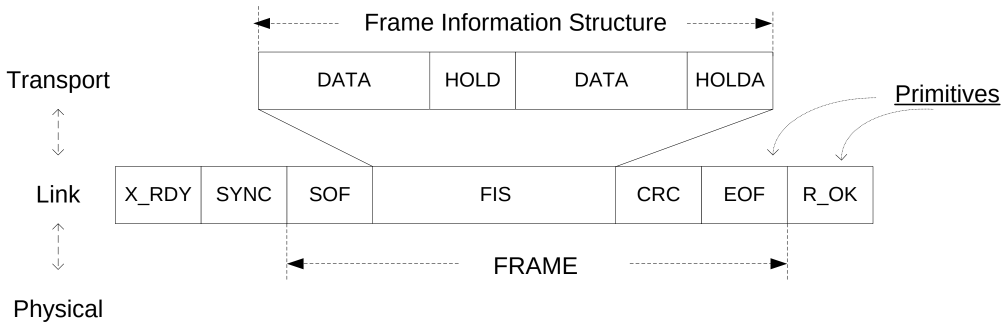


Figure 23 SATA frame and primitive transmission.

This link layer also encodes the data into a 8b/10b bit-stream, calculates and adds a CRC error check, provides frame flow control and reports errors in reception and transmission to the transmission layer. Figure 28 shows the relationship between the transport and link layers. The link layer sends streams of primitives and frames.

SATA – Transport Layer

The transport layer processes Frame Information Structures (FIS). In the forward direction a FIS is constructed in response to a request received from the application software/driver layer. When a FIS is received from the link layer the transport layer determines its type and distributes the payload to the correct stream. First Party DMA is a notable FIS that allows a drive device to implement a DMA transfer. Typically, a drive sets up the transfer and informs the host before initiating the transfer without any further host intervention. This layer also manages buffer FIFOs, flow control, acknowledgements and error reporting to the application layer.

SATA – Application Layers

Backward compatibility is a key concept for the successful introduction of SATA. To achieve this the lower layers of the SATA platform must look like the flat, address, data and control model of the original PATA interface. Backward compatibility is assured, existing software runs, and forward development can add functionality and speed.

BUS SYSTEMS 3 - UNIVERSAL SERIAL BUS (USB)

INTRODUCTION

The Universal Serial Bus (USB) now appears in a vast range of peripheral devices, for example, PDA's, Camcorders, Mobile-phones, Digital Cameras, MP3 players and Printers. From a user perspective, USB makes connecting new devices very easy using plug-and-play connectivity and support for multiple devices. USB 2.0 operates a 480Mbps (60MBps)which can easily handle multimedia data streams, putting USB firmly into the video and storage interface markets, for example, cameras, personal video recorder and external disk storage. In practice USB achievable data rates of 40MBps is typical. The latest version, USB3.0, increases the speed still further to 5Gbps (640MBps) allowing multiple high-speed devices to be handled simultaneously.

USB expands the PC' s interfacing capabilities via external ports, typically 4, eliminating the need for users or system builders to open the system chassis for the installation peripheral interface cards. Since USB supports upto 127 peripheral devices simultaneously, it allows users to attach numerous devices such as printers, scanners, digital cameras and speakers from a single PC. USB also allows for automatic device detection and installation, giving true plug-and-play for users and the wide acceptance of USB is such that every PC motherboard now comes equipped with a minimum of 4 USB ports.

USB BUS TOPOLOGY

USB uses a tiered star topology and is PC-centric. As is shown in Figure 24 there is a host hub, usually inside the PC, that initiates all bus transactions that initiates all bus transactions. In the existing specification there can only be one host per bus - the PC - and the specification does not support any form of multi-master arrangement. However, the development of USB-On-The-Go changes that – this is discussed later.

Figure 24 shows that multiple devices can be added using external low-cost hubs which are usually 4,6 or 8 port devices. This can add to system costs and clutter with more boxes and cables on the desktop but many devices now have integrated hubs, for example, keyboards contain a hub that allows a mouse and other devices such as a digital camera to attached via a USB connector located at the back of the keyboard. Also many monitors now have built-in hubs.

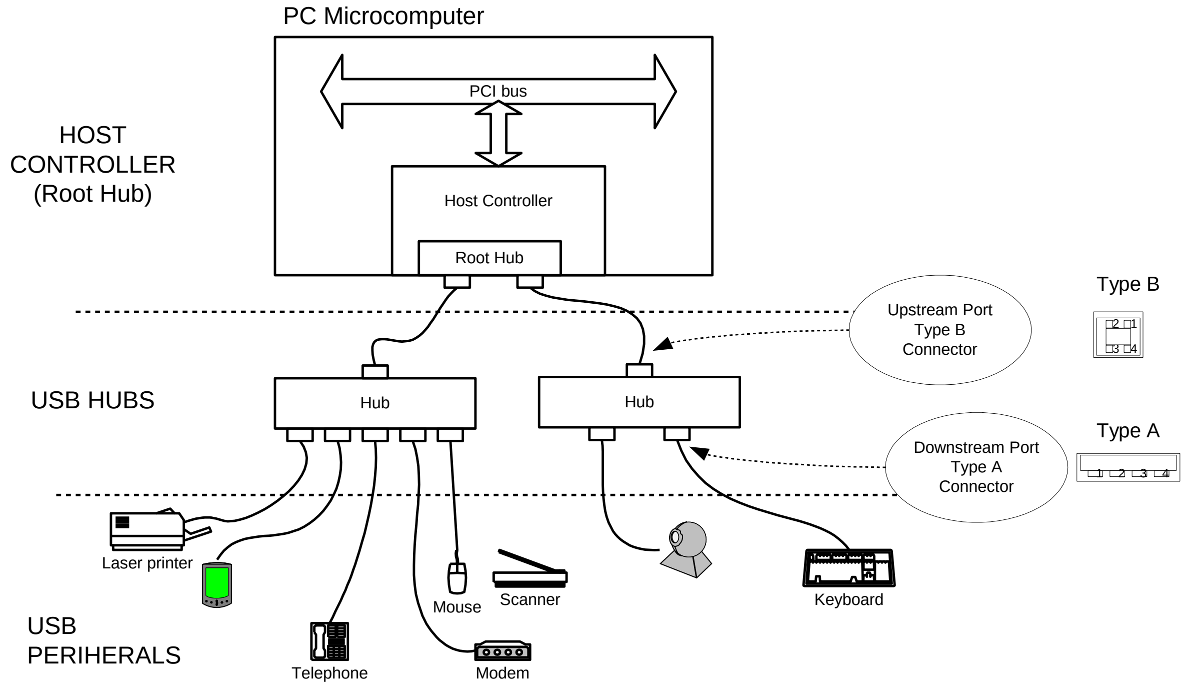


Figure 24 USB Tiered Star Topology.

PLUG-N-PLAY OPERATION

USB supports plug-n-play operation with dynamically loadable and unloadable drivers. The user simply plugs the device into the bus and the host detects this addition. The USB protocol then interrogates the newly inserted device, a process called **enumeration,** and then it loads the appropriate device driver and application software. The loading of the appropriate driver is done using a PID/VID (Product ID/Vendor ID) combination which is supplied by a newly discovered device during the enumeration process. Device enumeration is possible because the USB host is constantly checking for connected devices.

Device Interface Architecture

USB peripherals tended to use a standard processor cores, for example, the Intel8051, on to which USB functions are grafted. The resulting structure is shown in Figure 25 where the Serial Interface Engine (SIE) is responsible for handling the low-level USB operations and protocols. USB data is exchanged between the host controller and a set of endpoint memory buffers (FIFO's). The processor core communicates with the SIE and uses the endpoint data to execute device operations. The separations of these tasks optimises the performance of the peripheral and, importantly, removes the need for system implementers to become involved with the lower-level details of the USB protocols. This is a significant attraction in terms of time-to-market and overall development costs.

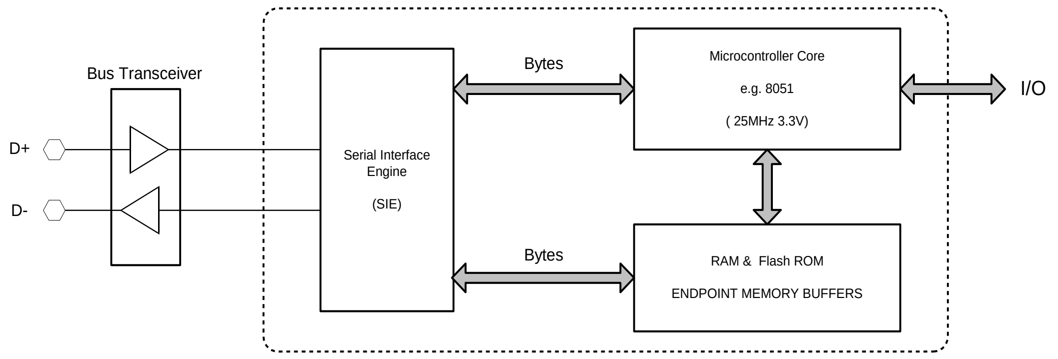


Figure 25 USB Peripheral Interface

USB Communication Model

USB simplifies hardware connection and ease of use but, on the downside, complexity is added by the need to implement a communications protocol that supports a variety of peripheral devices connected to a common bus. Although the physical bus topology has multiple tiers of communications through hubs, the communication model is logically, a point-to-point connection, from the USB host to any peripheral. Figure 26 shows the basic communications model which based on a layered architecture.

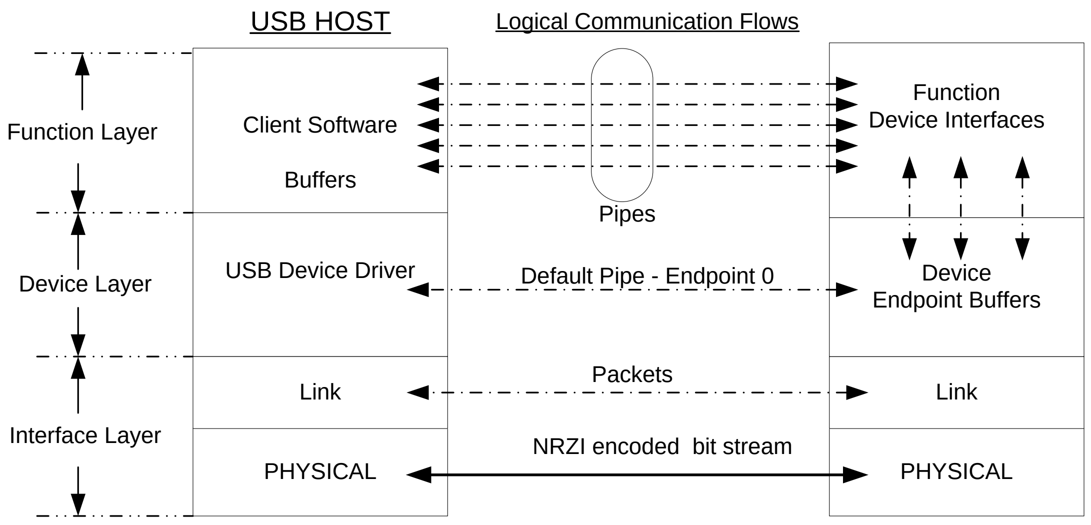


Figure 26 USB Communications Model (USB - Protocol Stack).

At the center of the communication model is the USB host, which acts as the bus master, controlling and scheduling all activities associated with the attached devices.

This configuration is referred to as a host-based communication model and, in the standard USB model, there can only be one host in the system. The **host initiates all bus transactions** and is the only device that uses system resources, requiring memory space, I/O address space, and interrupt request lines from the host operating system. USB peripherals, formally referred to as 'functions', are not mapped into memory or I/O address space, nor do they use IRQ lines or DMA channels.

Central to the USB communication model are the abstractions - **buffers, pipes and endpoints.** These are used to model the logical communications paths between the host and functions (devices). At the upper, function layer, data communication is seen as the transfer of data from a memory buffer in the host to an endpoint in a function.

- Data flow is logically connected using the abstraction of a **pipe**. Parameters associated with a pipe are the bandwidth allocation, the type of data transfer (Control, Interrupt, Bulk or Isochronous), the direction of data transfer and the packet and **buffer** memory allocations. Each pipe carries a unique data type between a host and peripheral or vice versa.

- An **endpoint** is in reality a, first-in-first-out (FIFO) memory buffer which the host and function use to exchange data. An important design parameter for USB devices is the number of endpoints that they support - 8 or 16 is typical. With this number of endpoints, system programmers have the flexibility to assign different buffers for individual data streams, for example, in a multimedia USB device, different endpoints would be allocated to voice (isochronous), data (bulk), and control information. Since these data types must be handled differently they are connected through the different endpoints.

USB – Physical Layer

The physical layer provides the actual communications between the host and its connected functions and it is responsible for physical connection, signalling and packet transfers. Figure 17 shows a USB interface with a single differential pair of signal wires to give a half-duplex connection. Reliable bit synchronisation and data recovery is ensured by encoding the transmitted data using the NRZI encoding scheme which allows the originating clock signal to be embedded in the data stream. The USB 2.0 maximum link speed is 480Mbps with a corresponding data bandwidth of 60MBps.

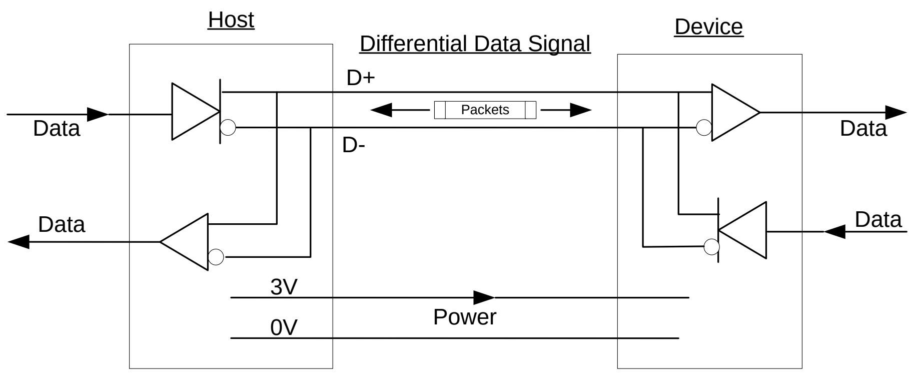


Figure 27 - USB Half-duplex differential connection and power links.

USB simplifies the actual wire connection between devices, using use only four wires, two are for the power supply (power and ground), and two for differential signaling (D+ and D-). Standard USB connectors come in two styles and it is impossible for the user to mix-up the connections because the connectors are physically incompatible. The connector that is nearest the host controller hub (the upstream side) is called a Series A connector and has a rectangular shape, see Figure 24. The connector nearest the peripheral, on downstream side is called the Series B connector and is a square shape with upper corners shaved to give an orientation key. With the tiered star topology, the Series A connection is always at the upstream portion of the star while the Series B connection is at the downstream connection, see Figure 24. High-speed connections support cable lengths up to 5 meters using high quality shielded cables that provide higher performance and consequently higher cost.

USB has numerous power conservation modes. Suspend is an option that reduces power consumption through software control and it has two modes of operation - global and selective. All devices must support both types of suspend modes. Global suspend places all devices in a suspended state. Selective suspend places only devices which have been inactive in a suspended state. Devices enter a suspend state after 3 ms of bus inactivity and must consume no more than 500 μA total when suspended.

USB – PACKETS

Communication between a host and a function is transacted as an exchange of packets, typically short sequences of token, data, status and start-of-frame packets. Figure 28 identifies the structure of these packets.

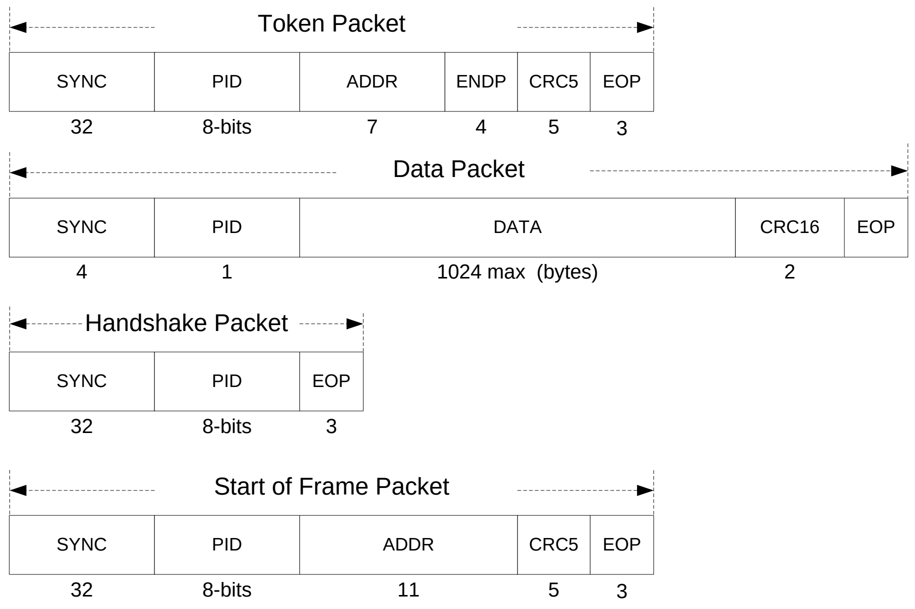


Figure 28 Common USB packet types.

Packets are assembled by grouping several fields together with some fields being ever present and others being optional, for example, the SYNC, PID EOF fields are always present but the address and data fields are optional. Note also in Figure 28 that smaller packets are presented with bit-fields, and the larger, data packet, is presented with byte-fields. The contribution made by each field can be summarised as follows :

PACKET FIELDS

- **SYNC** - in USB 2.0 is 32-bits long and is used by the receiving circuitry to lock to the incoming bit-stream. The SYNC value is chosen to maximise the number of transitions, thus giving the receiving clock the best opportunity to synchronise. In NRZI coding a sequence of consecutive 0's gives and alternating signal so the SYNC value is all 0's with the last two bits reserved to act as a delimiters for the PID field that is about to start.

- **PID -** the **Packet Identifier Field** is a byte-wide field that identifies the type of packet :

  - A token packet has four possible PIDs – SOF, IN, OUT, SETUP
  - A data packet transfers has four possible PIDs – DATA0, DATA1, DATA2 and MDATA.
  - A handshake packet has four possible PIDs - ACK, NACK, STALL. and NYET.
  - Special PIDs include – PRE (preamble), ERR, SPLIT, and PING.

  The format if the PID field is worthy of note. It uses the upper 4-bits to give 16 PIDs and it pads the lower 4-bits with the complement of the PID value (/PID). This simple mechanism give a simple and efficient error detection mechanism.

- **ADDR** – the address field is 7-bits, allowing 127 devices to be addressed by the host.

- **ENDP** – the endpoint field is 4-bits and it identifies the destination of a packet within a device. 4-bits allows a maximum of 16 possible endpoints within a device and **endpoint 0** is reserved as the device control pipe

  Note that the combination of the ADDR and ENDP fields uniquely identifies the destination of a packet within a device.

- **CRC - Cyclic Redundancy Check** is performed over the packet data payload only. For a token packet a the CRC field is 5-bits and for a data packet the CRC field is 16-bits.

- **EOP - End-of-Packet** is obvious. It is a 3-bit field that, strangely, pulls both data lines low (called a single-ended-zero) for 2-bit times and then terminates with a single 1 bit.This nullifies the benefits of using differential signalling.

USB – Packet Types

Packet types are defined by the PID field can be split into four types as shown in Figure 28.

- **TOKEN** packets can be IN – the host wants to read data, OUT- the host wants to send data, SETUP - the host wants to begin a control transfer.
- **DATA** packets have a maximum data payload length of 1024 bytes and can be DATA0 – even data, **DATA1** – odd data (alternate data packets toggle their data bits as a means of maintaining receiver synchronisation), **DATA2** - is isochronous data, MDATA - is split isochronous data.
- **HANDSHAKE** packets report the status of transactions, indicating the success or failure of data and control transactions. Handshake packets can be ACK – a packet has been received successfully, NAK – a device cannot send or receive a packet at this time, STALL – the endpoint is halted, NYET – currently there is response from the receiver.
- **START-of-FRAME** packets are issued every 125 microseconds. An SOF packet indicates that transactions between the host and peripherals can occur within the next 125µs time period. These time intervals are often referred to as 'microframes'. SOF frames are token only frames and they include an 11-bit frame number that is automatically incremented by the host – at 0x7FF it simply rolls-over. All devices hear SOF frames but they do not generate a response, this means that delivery of SOF frames is not guaranteed.

USB – Device Link Layer

The device layer processes transactions that are initiated by the host controller. The host may schedule a single data transfer within a microframe or as a block over several consecutive frames. The actual scheduling depends on a number of factors including: transaction traffic, type of transaction, and bandwidth requested by the peripheral. A typical transaction, as shown in Figure 29, consists of a three phases during which a token packet, a data packet and a handshake packet are exchanged.

Control transfers are bidirectional and are used for exchanging configuration, command or status information. A control transfer, as shown in Figure 29, consist of 2 or 3 stages; a setup stage, data stage (optional), and a status stage. CRC is performed on control packets since accuracy is important. Control transfers are given a guaranteed 10% bus bandwidth allocation. If the host rejects the request, then a Not Acknowledgement (NAK) is sent back to the peripheral. The peripheral then repeats the request, waiting for an acceptance from the host for the transfer request.

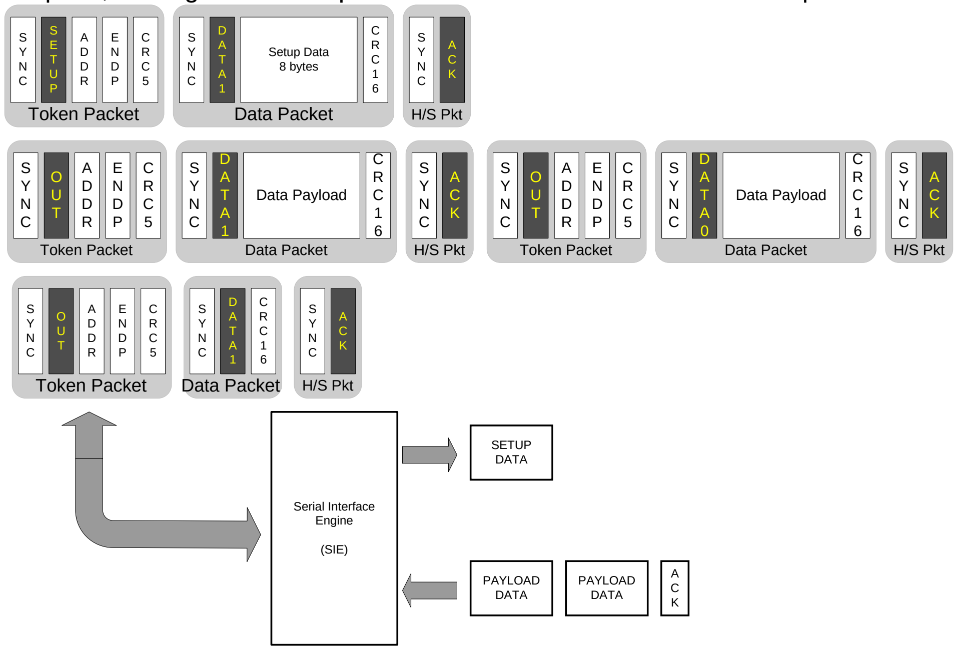


Figure 29 A Control Transaction

Bulk transfers are ideal for large amounts of data whose integrity must be guaranteed, but whose delivery is not time critical. Printer data or scanner data are natural candidates for transmission via a bulk transfer. A bulk transfer is designed to claim unused bus bandwidth when other transfer requirements have been met. The maximum data payload for bulk transfers is 64 bytes.

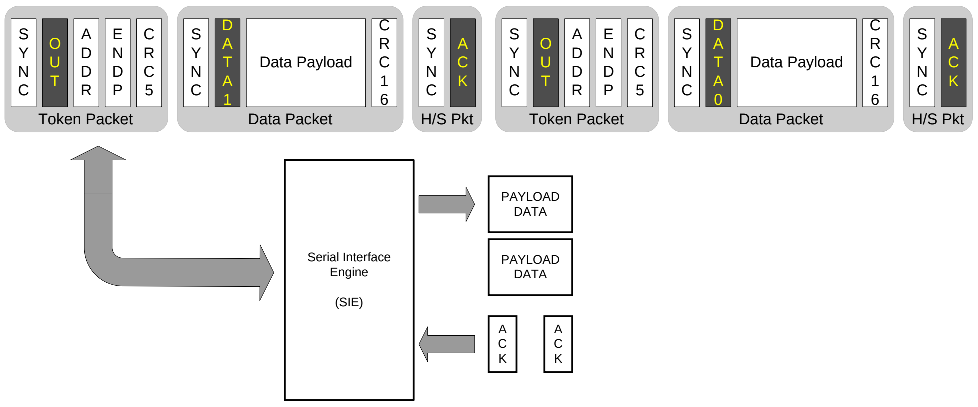


Figure 30 A Bulk Transaction

Isochronous transfers are specifically targeted for streaming data such as audio or video. Since isochronous data is time sensitive, it is guaranteed access to the bus bandwidth. A maximum 90% of bus bandwidth can be reserved for isochronous transfers. Errors occurring during an isochronous transfer are ignored; a constant data rate is the more important that accuracy. Streaming data is more tolerant of errors, for example, a byte error in an audio data stream will cause a blip in the sound quality but it is not a fatal event. The maximum packet size for isochronous transfer is 1024 bytes.

Interrupt transfers are not interrupts in the conventional sense of interrupts generated on PC platforms. Instead they are used to poll devices to determine if they have data that needs to be transferred to the host. Hence, the direction of interrupt transfers are always from the function to the host (IN only). If a device does not have data to send, then the device returns a NAK, indicating that no data is available. Maximum packet size for interrupt transfers is 64 bytes. Typical candidates for interrupt transfers would be a mouse or a keyboard, peripherals that have a small data payload but require a fast response.

USB – Function Layer

The function layer connects the host application to the endpoint device interfaces within the peripheral device. File based I/O read and write operations are used at this level to exchange data or control information, for example, a mouse x and y coordinate data value can be read or display (LCD) values can written.

USB 3.0

In essence USB 3.0, or SuperSpeed USB as it is also know, increases the bus speed of USB 2.0 from 480Mbps (60MBps) to a massive 5Gbps (640MBps). To achieve this, and to maintain backward compatibility, the standard USB 2.0 4-pin connection - power (x2) + differential signal (x2) connections, has been increased to include two additional Shielded Twisted signal Pairs (STP) as shown in Figure 31.

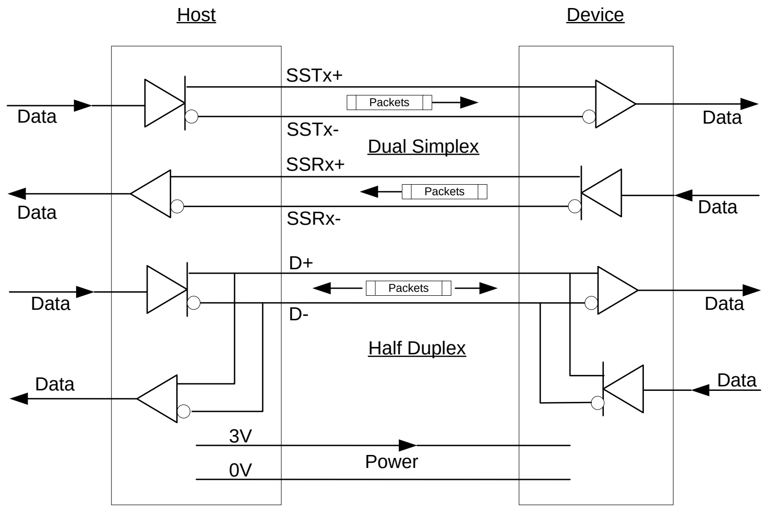


Figure 31 USB 3.0/SuperSpeed Connections

USB 3.0, therefore, uses 8 connections to provide - power (2) + 3 differential signal pairs (6 –connections). This is comparable to PCIe where hi-speed differential links are aggregated to provide very high bus bandwidths. In fact at the physical and link layers, USB 3.0 has more in common with PCIe than it has with USB 2.0. The STP serial links operate in dual-simplex mode giving a highly flexible data transfer protocol, for example, both links can be aggregated in the same direction or data can be sent simultaneously in opposite directions.

In addition to the increased bus speed, USB 3.0 also offers more efficient power management which results in efficiency gains of around 60% over comparable USB 2.0 operations. The two main improvements over USB2.0 are :

- The host or device can initiate power management of the link (in USB2.0 the host only initiates link power management).

- At the protocol level devices can enter low-power states between service interval, for example during video streaming (isochronous transfers) the link can be powered-down to conserve energy. Additionally, the USB3.0 protocol lets devices inform the host of their latency (by including Latency Tolerance Messaging in the protocol) so that the host can be powered down in the intervals slow devices.

With these enhancements USB 3.0 looks well set to compete in the battleground that is high-speed serial interface protocols.

The USB Evolution

As state previously, USB is the most widely used external peripheral interface with currently around 2 billion installations. This success has led to the introduction of enhancements that widen the scope for the adoption of USB into devices. Standalone operation through USB-OTG and wireless connectivity using WUSB are two of the evolutionary strands that are briefly introduced here.

USB On-The-Go

The previous discussion identified that USB is a PC-centric system - in the original specification there can only be one host per bus and all peripherals are connected in a tiered tree topology from the host. The specification does not support any form of multi-master arrangement where devices other than a PC could master the bus. USB On-The-Go (OTG) changes that. OTG specifies that there can be more than one bus master, an enhancement that allows USB to be used in point-to-point connections that do not require the presence of a PC host. Direct USB connections can be made between devices such as PDA's, Video Cameras, Mobile-phones, Digital Cameras, MP32 players and Printers Figure 32 illustrates some possible applications, in particular, a digital camera connected to a printer so that images can be printed directly.

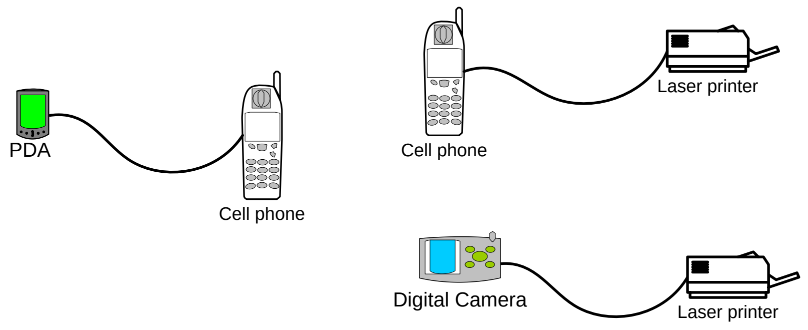


Figure 32 Possible USB-OTG applications.

OTG devices operate with both host and peripheral capabilities and can dynamically switch between roles. Such devices are often referred to as Dual Role Devices (DRD).

Wireless USB (WUSB)

Why not? The massive installed base of USB devices makes it the most successful PC interface ever, so why not build on this success. Wireless USB does exactly this, and it makes use of Ultrawideband (UWB) technology to provide a high speed low power solution. At 480 Mbps (@ 2m) and 110 Mbps (@ 10m) WUSB is fast enough to facilitate multiple video streaming applications. Like its wired counterpart a wireless USB hub can host up to 127 devices and hosts use point-to-point connectivity – devices talk only to the hub and device-to-device communication takes place via the hub. Ultrawideband technology is a new technology and WUSB will be its first commercial deployment.

WUSB is an emerging technology that is, perhaps, an obvious step in the evolution of USB. The demand for small compact mobile devices mandates a wireless based connectivity solution. WUSB provides a comparable bandwidth to USB2.0 (480Mbps) and operates within a range of 10m (wired USB maximum cable length is 5m). It is anticipated that future provision will extend the bandwidth to 1Gbps.

COMPARING HIGH SPEED SERIAL CONNECTIVITY

It is obvious from the discussion of PCI Express, SATA, USB that modern bus systems now have many characteristic in common and a few that make them unique. Table 1 provides a simple comparison of USB2.0, SATA and PCIe 2.0 operating parameters.

PARAMETER	USB 2.0/3.0	SATA 2.0/3.0	PCI EXPRESS 2.0/3.0
Topology	Tiered Star	Point-to-point	Point-to-point Scalable
Connection	Master-Slave	Master-Slave	Master-Slave
Serial/Parallel	Serial	Serial	Serial
Operation	Half-Duplex	Full-Duplex	Full-Duplex
Bandwidth (Mbps)	480/5000	150, 300, 600	500/1000 x N
Cable Length	5	1	1
Max Hosts	127	xN	xN
Signal Encoding	NRZI/8b/10b	8b/10b	(8b/10b)/ (128b/130b)
Data Transfer Types	Asynchronous Isochronous	Asynchronous	Asynchronous
Plug-n-Play	Yes	Yes	Yes
Hot Swapping	Yes	Yes	Yes
Bus Power	Yes	No	No
Bus Current (A)	0.5	-	-
Media	Twisted Pairs	Twisted Pair	Twisted Pair
Target Area	PC Peripherals	Disk Drives	PC Interfaces

Table 3 Comparing High Speed Serial Connectivity.

END OF UNIT - SUMMARY OF ACHIEVEMENTS

On completion of this unit you should:

- Understand the principles of parallel and serial bus systems and know the advantages and disadvantages of each.
- Understand the principles of parallel and serial data transfers systems and be able to compare their inherent strengths and weaknesses.
- Understand the operating principles of the common bus systems - PCI, USB, Firewire and SATA.
- Know where to locate and apply Web based resources that inform system developers about current and future developments on a variety of bus technologies.

Faculty of Engineering, Computing & Creative Industries

Edinburgh Napier UNIVERSITY

Module : Systems and Services

Module Number : CSN08101

Unit 6 : Peripheral Interface Devices

Analogue Data Acquisition, Sound, Video, Graphics and Network Interface Architectures

Student Study Material

UNIT 6: PERIPHERAL INTERFACE DEVICES

OVERVIEW

This unit provides an introduction to common peripheral interfaces, for example, Sound, Video, Graphics and Network Interfaces. The topic is initially introduced using a generic PC style Data Acquisition System (DAS) interface which is used to identify the individual interface functions and explain their role. The basics of analogue signal digitisation, and reconstruction, multiplexing, memory buffering are discussed in order to provide the platform for the subsequent discussion which provides an introduction to more dedicated architectures for sound, video and graphics interfaces. The Network Interface is included simply because it is now a standard interface in the majority of computer systems and it also provides an excellent example of a programmable digital interface device.

LEARNING OUTCOMES

On completion of this unit you should :

- Understand the principles of parallel and serial data transfer systems and be able to compare their inherent strengths and weaknesses.
- Understand the basic principles of analogue input-output systems.
- Understand the main functional elements and operating principles for common peripheral interfaces - Sound, Video, Graphics and Network.
- Know where to locate and apply Web based resources that inform system developers about current and future developments on a variety peripheral interface technologies.

ADDITIONAL MATERIAL

There are many excellent websites with application notes and papers on topic of bus systems technology. Also Wikipedia provides a wealth of information and links on these topics:

SOUNDCARD TECHNOLOGY AVAILABLE ON THE WEB

Source	Comment
/techreport.com/gpu/	Reviews of Graphics processors and cards
pdf1.alldatasheet.com/datasheet-pdf/view/109644/ETC/ES1371.html	Soundcard datasheet
www.alldatasheet.com/datasheet-pdf/pdf/122231/ETC/ES1938.html	Soundcard datasheet
/www.digit-life.com/articles2/multimedia/creative-x-fi.html	Creative soundcard information
http://developer.nvidia.com/what-cuda	CUDA parallel processing on NVIDIA graphics cards.

TUTORIAL : PERIPHERAL INTERFACES

LEARNING OUTCOME

On completion of this tutorial you should understand the basic architecture of computer system I/O and interface. In particular you will understand the main requirements for sound and video interfaces and be able to describe the role of the main functional components.

1. A general purpose Analogue I/O Interface Card is required for a PC microcomputer. Sketch a block diagram showing the main components of a this type of interface card and describe the function of each block.

2. A number of analogue signals are to be measured by a PC-based data acquisition system. This will use a plug-in data acquisition board.

   i. Describe how the sampling rate and resolution affect the measurements that are being made.

   ii. What other components, apart from the Analogue to Digital Converter will be needed to accurately measure multiple signals? Briefly explain their function.

   iii. Give a block diagram to show how the components are connected together to make the data acquisition board.

3. (a) Identify and describe the main design parameters that influence the choice of Analogue-to-Digital Converter for a given application.

   (b) For the following applications select a suitable type of ADC and specify an appropriate sampling rate and resolution :
   i. Hi-fi audio processing.
   ii. Video digitisation.

4. Use a system block diagram to identify the architecture a typical PC Soundcard. Give an overview of the Soundcard operation paying particular attention to the function of the Soundprocessor and the sound CODEC. For the CODEC clearly identify the type of digitiser, stating appropriate values for device sampling rate and resolution.

5. (a) Use a system block diagram to identify the architecture of a typical PC video I/O card. Give an overview of the video card operation, and explain the operation of the RAMDACS.

   (b) The video card may be one of a number of interface cards plugged into a PC bus, such as the PCI bus. Explain how interrupts from these cards are connected to the processor, and how the source of the interrupt is identified.

6. Use a system block diagram to identify the architecture a typical PC Graphics Adapter. Give a overview of the Adapter card operation paying particular attention to the function of the Graphics processor, video signal generation and PC bus interface.

7. Use a system block diagram to identify the architecture a Ethernet Network Interface Controller and give a overview of the operation of the controller.

FURTHER READING AND PRACTICAL WORK : PERIPHERAL INTERFACES

LEARNING OUTCOME

These topics are intended to broaden your reading and improve you approach to academic research and investigation. On completion of this exercise you should be able locate and use Web based support resources for allocated to the peripheral interface devices found on a typical PC microcomputer.

PRACTICAL

Using the resources found at the Websites listed below as your starting point answer the following :

1. Use a search engine to locate Web data for the sound and graphics cards in your own PC system. A good starting point is the PC motherboard technical manual but if you have dedicated Sound and Graphics cards then you may need to look more widely. Identify the following parameters:

   i. The available sampling rates and the resolution of sampling.
   ii. The name/ part number of the sound processor on the card – this may require further searching. Use this information to

   If the sound and video cards are unknown, more information can be gained by accessing the system hardware list in the Windows control panel.

2. You have been asked to buy a graphics card for 50 PC's that will be used in a multimedia laboratory. Note down the criteria that you will use to make your choice and suggest a suitable card. Note that minimum cost is a key factor in your decision making.

3. As a keen gamer you are about to invest in a new buy a graphics card. Note down the criteria that you will use to make your choice and suggest a suitable card. Note that as a student your budget is limited and the trade-off between performance and cost is a key factor in your decision making.

4. Graphics card architectures are massively parallel processing systems containing many processor cores. Development environments, such as CUDA and OpenGL are seeking to take advantage of these powerful architectures by using them to solve large software problems on PC's. Go to the CUDA or OpenGL websites and explore the range potential applications that graphics cards are being used to solve. 'The future is Parallel' Prof Jon Kerridge, 2008.

ANALOGUE SIGNAL INTERFACING

INTRODUCTION

Analogue signal I/O interfaces appear in many forms, for example, they can be seen in computer systems as general purpose I/O cards, or as dedicated sound and video cards. They can also be interfaced with the computer system bus via any of the standard peripheral interface buses, for example, PCI bus, USB, Firewire, Parallel Port or Serial Comport. Irrespective of the form they take, the fundamental role of the analogue signal acquisition system is to provide the computer system with an accurate digital representation of the analogue signals generated by its sensor inputs. Conversely, the role of an analogue signal generation (or signal reconstruction) system is to accurately reproduce an analogue representation of a digitised signal which is usually derived from a stored (or algorithmically generated) data.

In a typical multimedia desktop PC environment the obvious sensors are microphones, touch panels and cameras which provide signals inputs for audio, mouse and video processing. Less obvious sensors, now found on PC motherboards, are the temperature sensors which monitor the processor temperature and control the CPU fan speed.

In the broader context of computer based scientific and industrial instrumentation systems, signal acquisition systems gather data from sensors such as temperature, touch, proximity, pressure, flow, humidity, acoustic, optical and ultrasonic detectors. Also, the range of sensors interfaced to computer systems is rapidly expanding as gesture sensing, virtual reality and wearable computing applications are developed. This type of system is shown in Figure 33. At present there is considerable research in this area.

ANALOGUE SIGNAL INPUT – DATA ACQUISITION

Before discussing specialist interface functions it is worth considering the architecture and components of a generic DATA ACQUISITION SYSTEM (DAS). As shown in Figure 33 the DAS processes analogue signals from their raw state, as generated by the sensor inputs, into a stream of digital values which are available for processing by a computer. The digital bit-stream represents the current value of some physical quantity, for example temperature, touch, proximity, pressure, displacement, light etc.

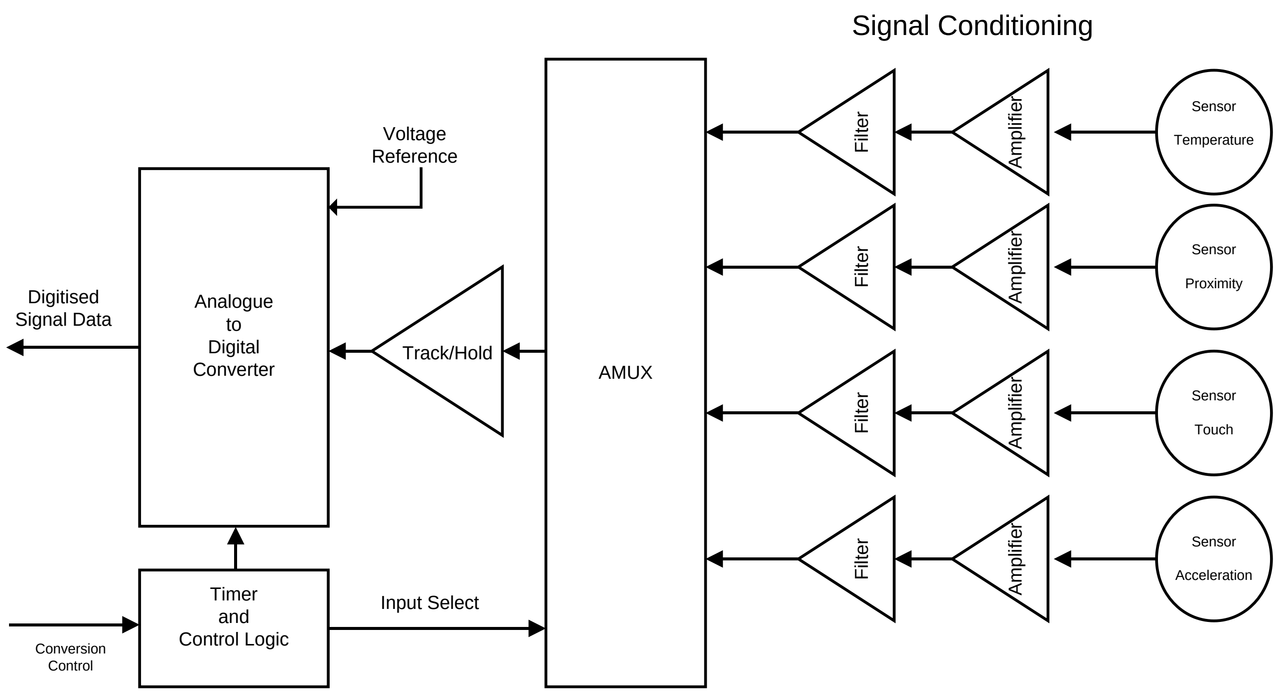


Figure 33. A Basic Data Acquisition System

Typically, the DAS is operated under the control of a microcomputer, embedded processor, or it may be implemented by a ‘system-on-a-chip’, single-chip microcontroller. It is also quite common for the DAS to be operated as a remote unit. This is common in industrial systems where the DAS is controlled remotely over a network, which can be implemented in wire, fibre optic or it may even be a radio or infra-red communications link. Many of these features are explored in the case studies that follow.

As will be discussed later, the basic DAS shown in Figure 33, has many features in common with PC sound and video interfaces.

TRANSDUCERS AND SENSORS

As previously stated, a sensor simply provides an electrical signal output that is proportional to some physical quantity, for example, temperature, capacitance, pressure or light. A thermocouple, for example, develops an output voltage that is proportional to temperature, a peizoelectric crystal produces and output voltage that is proportional to pressure and a flow sensor produces a signal frequency that is proportional to flow rate. A detailed operation of sensors is beyond the scope of this course but a working knowledge of sensor technologies opens-up the potential for many novel applications. Fortunately, from the systems perspective, a sensor can be viewed as a signal source that produces a measurable electrical output signal change in response to a change in some physical quantity. The change in electrical signal output can be voltage, current, resistance, charge or frequency and the measured physical quantity can cover a vast , and increasing range of possibilities - temperature, pressure, touch, proximity, flow, displacement and acceleration are common but more exotic sensors are constantly appearing Salinity, pH, gas, infra-red, touchless, gesture etc. Sensor technology is also a highly active area research.

The reality of sensors and transducers are that they are far from ideal. Typically, they suffer from any of the following imperfections:

- Generally, the signal output level is low-level, microvolts to millivolts.
- The signal output is non-linear i.e. Vout <> K x Sensed Input Value.
- Component ageing degrades the repeatability of the measurement over time. Recalibration is required.
- Hysteresis is common during measurement cycling. This means that the output signal is different depending on the direction in which the measurement point is approached. For example, a temperature reading of 20°C produces slightly different output voltage depending on whether the temperature is increasing or decreasing.

This list is not exhaustive but it clearly shows the potential for measurement errors and given that any measurement system has to account for these limitations and imperfections then it follows that the DAS must accurately condition the 'raw' sensor signal in order to produce useful measurements. Hardware and/or software can be used to implement the signal conditioning functions, for example, filtering and linearisation can be implemented in hardware or software and the choice of method requires the usual compromises between performance, flexibility and cost

SIGNAL CONDITIONING

Given that sensors generally produce low-level signals, microvolts to millivolts, and that the signal input range required by the ADC is typically ±1 to ±5 volts then signal **amplification** is necessary in order to match the sensor signal level to the ADC input. Additionally, the sensed signal usually contains undesirable signal components such as 'noise and interference'. These undesirables cause errors in the digitised signal if they are not removed, so signal **filtering** is required to remove any unwanted signal frequencies from the input signal. Specialised techniques such as differential, instrumentation and isolation amplification are often used to minimise the effects of mains frequency and common-mode signal interference. Signal filtering is also used to 'band-limit' the input frequency spectrum so that signal aliasing does not occur.

MULTI-CHANNEL OPERATION - AMUX

Multiple sensor inputs are supported by incorporating an Analogue Multiplexer (AMUX) into the system. The AMUX acts as a signal router which selects an individual sensor signal and connects it to the input of the Analogue-to-Digital Converter (ADC). An AMUX typically supports 8 or 16 input channels which can be single-ended or differential. In essence an AMUX operates like a large rotary switch.

TRACK AND HOLD (SAMPLE/HOLD)

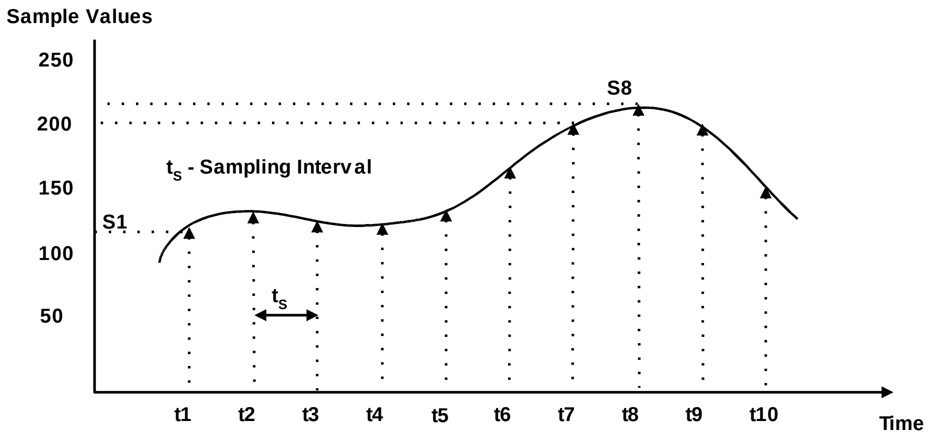
A fundamental operating condition for the ADC is that the input voltage should be held constant during the conversion process. If this requirement is not met, significant conversion errors appear in the digitised signal values. The Track and Hold circuit which immediately precedes the ADC in Figure 33 is used for this purpose. In ***tracking*** mode the input voltage to the ADC simply follows, or tracks, the analogue input value but when the circuit is switched into ***hold*** mode the ADC input is held constant while the ADC digitises the input signal. The hold signal is applied for the time it takes for the ADC to complete the digitisation process and is dependent of the type of ADC used. For example, Successive Approximation ADCs require a number of clock cycles to fully acquire a signal sample, a sample-hold circuit is therefore essential here. Alternatively, a Flash ADC acquires the signal in a single clock cycle and therefore does not require a sample-hold circuit. These ADC types are briefly describes in Table 4.

TIMER AND CONTROL LOGIC

The inclusion of a timer and control logic block into the DAS allows flexibility in sampling and triggering. For example, programmable sample rates and sequential channel scanning are allowable under software control. Transient capture and delayed triggering are also possible acquisition functions.

ANALOGUE-TO-DIGITAL CONVERSION

An analogue signal is characterised by the fact that it is time continuous. In Figure 34 the analogue signal has a value at all points along the graph; it has a value at every instant in time. The ***sampling*** process takes a snapshot of the analogue signal at equally spaced time intervals, t1,t2,t3 etc, and ***quantises*** the signal at each point in time. The quantisation process assigns a binary value that is proportional to the height (amplitude) of the signal. For example, the fourth sample point (t4) in Figure 34 is assigned the value 120.



110	128	121	120	125	162	201	214	207	146

Figure 34 Analogue Signal Sampling and Quantisation.

The ***DIGITISED*** signal that results from the sampling and quantisation processes is represented by an array of values also shown in Figure 34. An important point to note is that the original signal can be reconstructed from the stored array if the original sampling interval, ts in Figure 34, is known. Also, this assumes that the interval between sampling instants is short and that no signal features in the original signal are lost. In Figure 34, for example, it is possible to draw a completely different signal profile that passes through the same sample points - see SAMPLING RATE for further discussion on this point.

QUANTISATION AND RESOLUTION

Quantisation is the process of banding the analogue input value signal and assigning it to a binary code. Resolution, is the number of bits (N) in the binary word that represents the input value. If the Full Scale (FS) input signal range is V_{REF} then quantisation interval is determined as:

$$q = V_{REF} /(2^N - 1)$$ where N is the resolution, or wordlength, in bits.

Assuming an ADC with an 8-bit resolution (N =8) , and a reference voltage V_{REF}= 2.55V :

Then $$\mathbf{q = 2.55/(2^8 - 1) = 2.55/(256-1) = 10mV}$$

The quantisation interval can also be specified as the change in input voltage required to produce a change of 1bit in binary output value. In the above example, this would be 10mv/bit.

Obviously, the accuracy of the quantised signal is determined by the number of bits of resolution, for example, a 10-bit ADC provides 2^{10} = 1024 quantisation levels, and a 16-bit ADC offers 2^{16} = 65536 quantisation levels. Given again a V_{REF} = 2.55 then this would correspond to quantisation intervals of approximately 1mv/bit and15.2µv/bit.

Resolution is often expressed as a percentage of full-scale (FS) range. For example, a 12-bit converter can resolve an input signal with an accuracy of 1/4096 or 0.024% and a 16-bit converter resolves to 1/65,636 or 0.015%. Assuming a 10-bit ADC with an input range of 1V FS then the ADC can resolve input changes down to 1V/1024, approximately 1mV or 0.1%.

QUANTISATION ERROR

The transition between quantisation levels is offset by 0.5q (0.5 bit) to ensure that the maximum quantisation error is the range ±0.5q; with no offset applied the quantisation error ranges from 0-1q. The only downside in applying this offset is that the full scale range (FSR) is reduced by 1-bit, however, this is not such a heavy price to pay for the resulting improvement in accuracy which now ranges over ±0.5 bit.

SAMPLING RATE

It should be clear from Figure 34 that the process of digitising an analogue signal results in a loss of signal information. In Figure 34 the digitised signal contains no information about the behaviour of the signal between sampling instants. Given that this loss of information is an inevitable consequence of the sampling process then the question arises of just how fast should the input signal be sampled in order to preserve its original information. The original work by Nyquist and Shannon linked the theoretical, or ideal, relationship between the minimum required sampling rate and the maximum analogue input frequency by following simple rule:

An analogue signal containing frequency components up to a maximum frequency, f_a can be represented by regularly-spaced samples, provided the sampling rate (f_s) is at least twice the maximum input signal frequency range ($2f_a$) samples per second, or $f_{smin} = 2f_a$

In an *ideal system* a speech signal with a frequency range, '**bandwidth**', 0-3kHz would be sampled at 6ksps. Similarly ,a video signal with a bandwidth of 0-5MHz requires a minimum sampling frequency of 10Msps. Note that these criteria assume ideal conditions but for good practical reasons, notably **signal aliasing** and imperfect filters, the sampling rate is normally set higher than the, ideal, Nyquist rate. In speech applications, for example, the sampling rate is sampling rate is set around 8kHz.

The sampling process simply takes a 'snapshot' of the analogue signal input voltage at fixed instants in time to produce a digital value. A key feature of the sampling process is that signal information is lost. Firstly, the digitised signal only contains information about the input signal at the sampling instants, any information about the signal between sampling instants is lost. Secondly, the quantisation process 'bands' the input voltage into a quantisation intervals and the number of quantisation intervals directly affects the accuracy digital signal representation. The implication of these potential sampling errors is that the DAS should operate at a sampling rate that ensures that no signal detail is lost. Also the resolution of the digitiser (i.e. the number of bits) should be sufficiently high to minimise the size of quantisation errors.

From the foregoing discussion it should be clear that the choice of sampling rate is an important factor for accurately representing an analogue signal in the digital domain. If the sampling rate is too low then important signal characteristics are missed. Conversely, if the sampling rate is too high, high-speed ADCs, large amounts of memory and faster processors are required to digitise and process the acquired data. Simply put this states that **oversampling**, if it is not required, increases system costs unnecessarily.

TYPES OF ADC

From the earlier discussion on signal quantisation and sampling it follows that the two main performance selection criteria for ADCs are :

- **RESOLUTION (N-Bits)** because with more bits the ADC will more accurately quantise the input signal.
- **SAMPLING -** higher sampling frequencies allow higher frequency signal features to be captured.

In practice resolution and sampling rate are often conflicting requirements in ADC designs, therefore, high resolution devices are usually only available at lower sampling rates, therefore, a number of different types of ADC technologies have been developed to meet the wide range of potential applications for analogue signal acquisition These are identified in the table below alongside their typical range of operating parameters, their relative cost, and their intended areas of application :

ADC TYPE	Resolution N bits	Sampling Rate Samples/s	Relative Cost	Comments and Applications
Successive Approximation	12-16	100k – 10M	Lowest	This is the oldest type of ADC in common use. It is widely used in general purpose data acquisition and measurement systems.
Flash (Full-Flash)	6-10	10M – 500M	Highest	This is the most accurate type of converter for high-speed applications such as broadcast quality video acquisition and radio frequency measurement applications.
Subranging Flash	8-14	1M – 75M	Moderate	This flavour of ADC uses a modified flash type of architecture to provide low-cost solutions for video digitisation and ultrasonic applications.
Sigma-Delta	16-24	40k – 100k	Moderate - Low	This is the preferred device for audio applications. High resolution means that the requirements of high-fidelity audio applications are easily met. This type of converter is also seeing increased use in general purpose measurement systems – simply because of the high accuracy it provides.

Table 4. Common Analogue-to-Digital Converter Technologies

DAS Implementations

The DAS shown in Figure 33 can be viewed as a generic DAS architecture and the elements described are common to the majority DAS implementations. Actual DAS implementations are often embedded into systems as sound, video or data logging interface cards and the interfaces very often combine signal acquisition with signal reconstruction. At this point, therefore, it is appropriate to introduce analogue signal reconstruction.

Analogue Signal Output – Signal Reconstruction

Simply put, analogue signal reconstruction is the signal acquisition process in reverse. Figure 35 shows the signal path for an analogue signal generation interface. The output signal originates as a stream of binary values stored in the computer system memory. The DAC, signal filter and amplifier, signal chain, is then used to produce an analogue output signal .

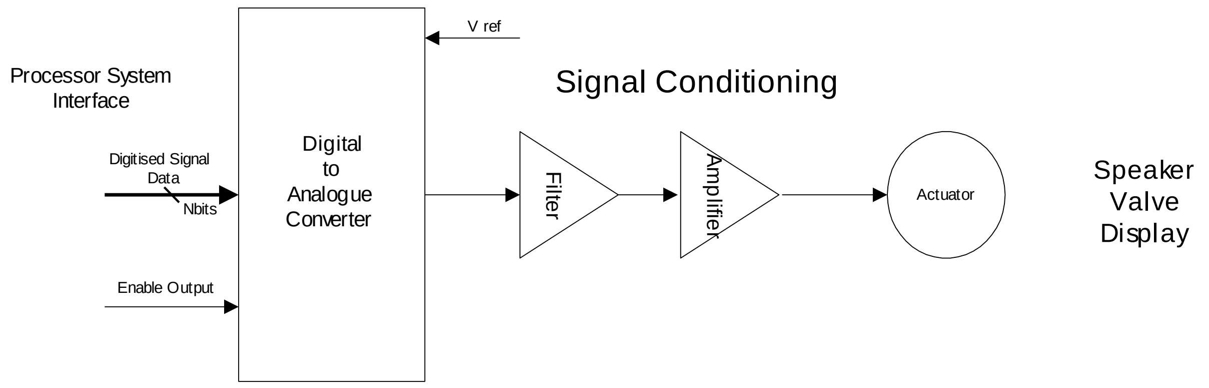


Figure 35. A generic signal generation interface.

Digital-to-Analogue Conversion

The DAC generates an output voltage that is proportional to the digital value presented at its input. To achieve this accurately, the DAC ratios the digital input against a fixed reference voltage, Vref. The output value is then derived as a fraction of the full-scale using the following relationship –

$$Vo = Vref * N_{DAC}/N_{FS}$$

Where N_{DAC} is the digital input value and N_{FS} is the maximum value (Full-Scale) of the DAC. For example, an 8-bit DAC N_{FS} would be 11111111 (0xff) and a typical Vref would be 2.5 (or5) volts. If N_{DAC} = 0x56 then the output voltage is 2.5x 86/255 = 0.843 volts.

Sample values written to the DAC are stored in an internal register which connects directly to the voltage conversion circuit. Since this value is held constant between sample updates an output track-and-hold circuit is unnecessary.

The range of signal frequencies generated by the DAC is determined by the rate at which the processor can provide samples to the DAC. High-speed interface techniques, such as DMA and memory buffering, are often used to maximise the signal frequencies generated.

Output Filter

The purpose of the signal filter in this remove the very-high frequency components arising from the stepped nature of the DAC output signal. The filter removes these high frequency components to provide a smoothed signal output characteristic. These filters can be quite complex if high quality outputs signals are required.

Output Amplifier

There is a vast range of output devices that can be driven by the output amplifier: sounders, headphones, loudspeakers, cooling fans, valve actuators and motors to name but a few. The amplifier output stage simply boosts the output signal so that it matches the DAC signal to the output device.

Two excellent practical examples of analogue output interfaces, found in the PC and sound cards that generate audio signals and graphics cards generate video signals. Both of these interfaces are consider in more detail later.

CASE STUDY 1 – PC GENERAL PURPOSE ANALOGUE I/O CARDS

Many DAS systems, particularly those used in laboratory environments, are based on a PC microcomputer with plug-in DAS interface cards. These are available from many vendors – National Instruments are Agilent notable examples. Figure 36 provides a block schematic of an 8 input DAS card.

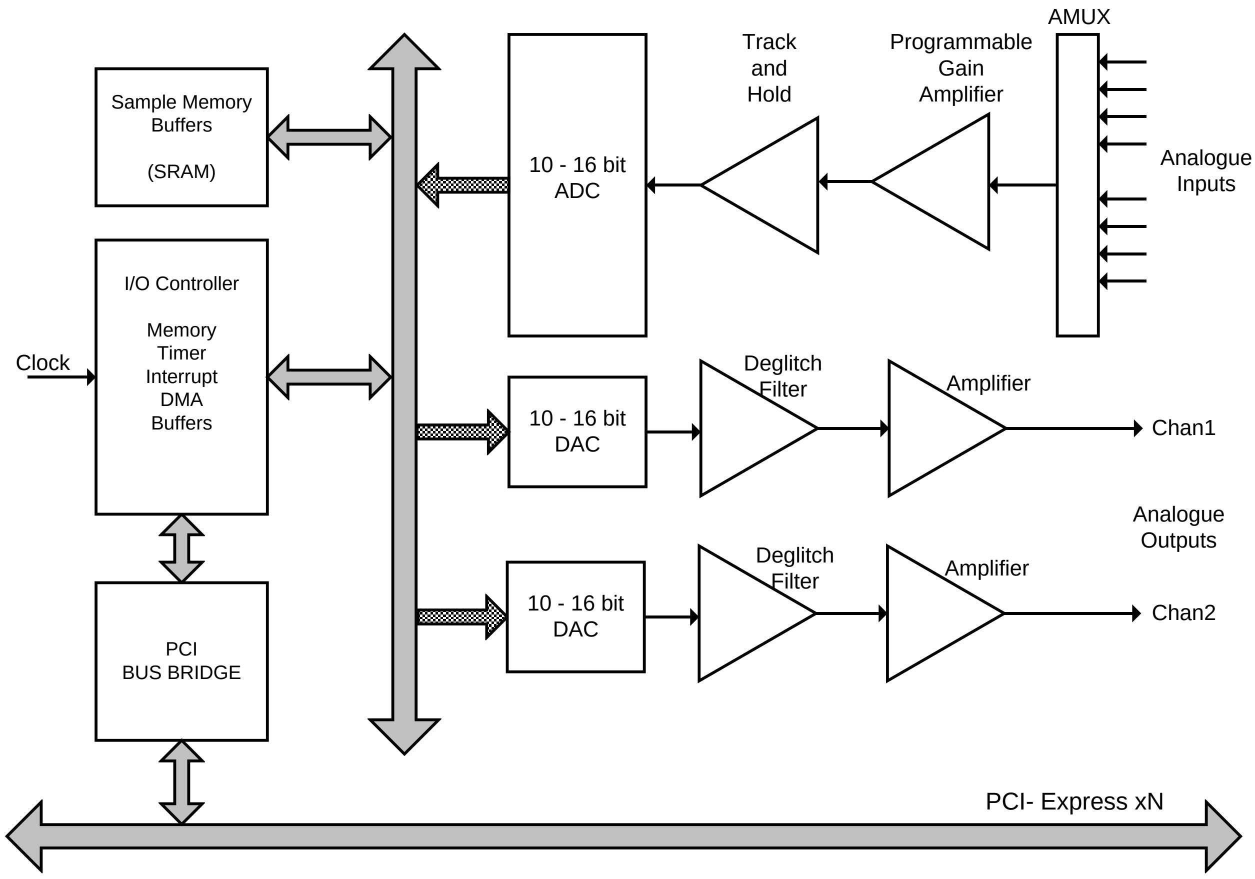


Figure 36. A PC Data Acquisition Card

TYPICAL DAS FEATURES

More complex, and hence more expensive, DAS systems provide many enhancements to the basic features described in Figure 33, for example:

- Programmable gain amplifiers – allows the optimisation of input signal levels under software control.
- On-board RAM - allows data steaming and block sampling for transient recording applications.
- Multiple ADCs* – allows simultaneous and interleaved sampling.
- Programmable Filtering – allowing the effects of noise, pickup and interference to be minimised.
- Autocalibration - allows on-board test signals to assist with calibration and test procedures.

* The simple DAS structure shown in Figure 33 can only sample a single channel at any sampling instant. It cannot, for example, sample two inputs simultaneously. If a measurement is required to provide the precise time interval between two signals then simultaneous sampling is required and additional ADCs, or Track/Hold circuits are necessary.

A key development in the application of PC based data acquisition systems is the availability of support software. Device/ Instrument drivers certainly ease the basics of controlling and acquiring data but there remains a lot of software development work to be done in order to implement a usable measurement and instrumentation systems. VIRTUAL INTRUMENTATION packages such as VEE from Agilent and LabView from National Instruments do much to speed up the process by providing a graphical user interface environment in which to rapidly develop instrumentation systems. Complex instrumentation can be developed efficiently and reliably without the need to get 'bogged down' in the complexity of programming detail. For example, statistical data analysis, data filtering and the application of linearisation polynomials can be built into systems by connecting predefined signal processing blocks into the data flow. Similarly, operator panels showing mimic diagrams and current process details can be developed in a fast, efficient and reliable manner. VI tools are certainly the way ahead for the software aspects instrumentation systems development; there are considerable savings achievable in development time, maintenance and productivity.

CASE STUDY 2 - PC AUDIO I/O CARDS

In the PC environment the explosion of multimedia technology has produced a vast range of sound and video interface cards. The similarity between a sound card interface and a basic DAS is obvious, a sound card is simply a DAS optimised to handle audio signal frequencies (20-20kHz) and audio sound levels. There is however one notable difference between the two, sound cards normally include an on-board processing which allows signal processing to be carried out on the card. This added processing power allows signal processing tasks such as compression, decompression and filtering to be carried out on the card under the control of the host processor system.

A typical PC sound card interface is shown below.

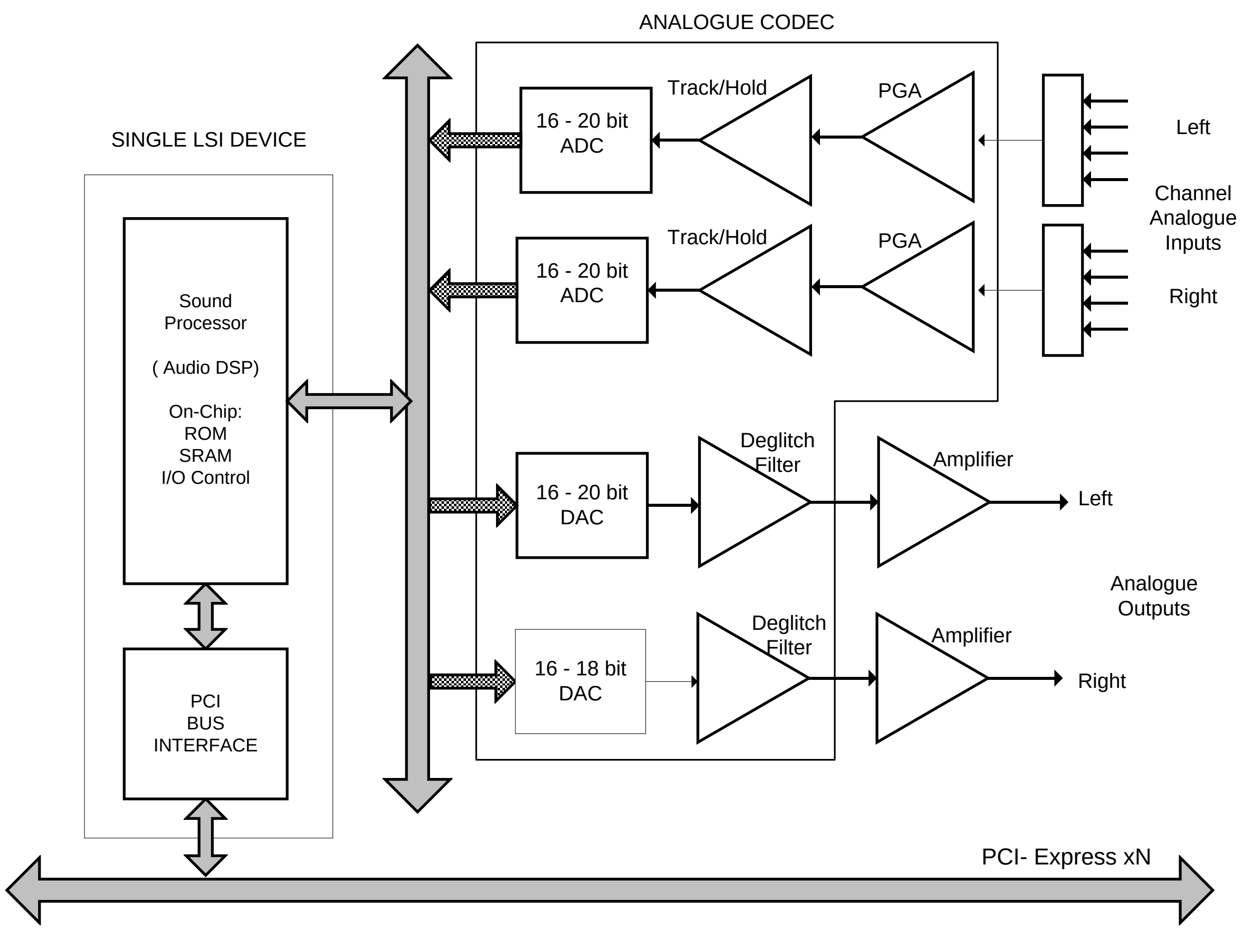


Figure 37. PC Sound Card Architecture

Typical audio card features are :

INPUTS
Two channels (stereo - left and right) - 16-18-bit ADCs variable sampling rate 4k-80kHz
I/Ps for MIC and LINE - (1 - 500mV) Automatic Gain Control.

OUTPUTS
Two channels (stereo - left and right) - 16 -18bit DACs variable sampling rate 4k-120kHz
O/Ps AUX (500mV) or Amplified signal for direct headphone or speakers.

PROCESSING - On-board audio signal processor for WAVETABLE and FM synthesis.

BUS INTERFACE
Sound cards use the PCI bus for interfacing to the system processor. This is essential for the high data transfers rates demanded by high-performance sound cards.

CASE STUDY 3 - PC Video Frame Grabber

This particular interface has been chosen because computer based image processing systems are now being used in an increasing number of applications, for example, industrial control (robotics), surveillance and security. The architecture presented in Figure 38 is a low-cost solution, it simply grabs video frames from 4 out-off 8 video input sources, usually cameras, buffers them and then sends them off to a host PC where the image processing is carried out. There is no image processing carried out locally on the grabber card and costs are kept down. More expensive cards include a powerful Digital Signal Processor(DSP) to perform feature extraction and recognition.

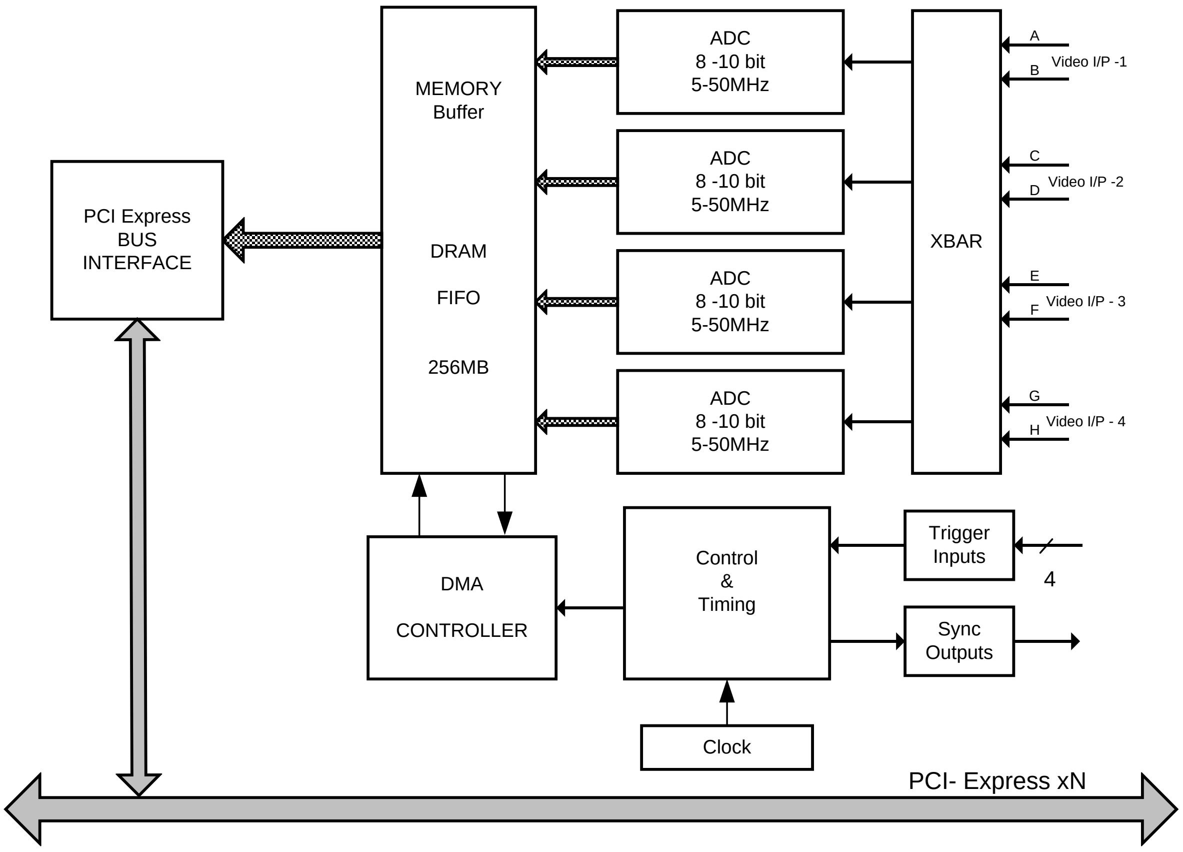


Figure 38. Video Capture Card Architecture

Typical features are :

<u>INPUTS</u> - 8x selectable video input channels. The crossbar switch (XBAR) is used to select one-off-two video inputs. Signals are digitised with an 8-10 bit resolution at a user programmable sampling rate between 5-50MHz. At this sampling rate and resolution a subranging flash type ADC is used – a full-flash ADC would be too expensive and power hungry. The four trigger inputs can be used to trigger frame captures from external video synchronisation pulses or l event triggers, such as, motion detectors and alarm signals.

<u>OUPUTS</u> – Synchronisation outputs are used for triggering external devices, such as, lighting and alarms.

<u>ON-BOARD VIDEO MEMORY</u> – A 256MB First-In-First-Out (FIFO) memory buffer stores captured image frames. The width of the FIFO depends upon the resolution of the ADC, assuming 8-bit ADCs then a 32-bit wide FIFO is required. Filling and emptying the FIFO is synchronised by the DMA controller and timing logic.

<u>PROCESSING</u> - This architecture is targeted at low-cost applications, so there is no on-board image processing. The acquisition system simply fills the FIFO with video samples and uploads them to the host PC. Maximum efficiency is ensured using a Direct Memory Access Controller to manage the data transfers.

<u>BUS INTERFACE -</u> PCI Express bus is ideal for this type of interface. A single lane operating at 250MBps is sufficient to match the real-time data transfer requirements of the FIFO stored images.

SIMILAR VIDEO BOARDS

PC based Personal Video Recorders (PVR) and PC-TV cards are extensions of this basic architecture. PVRs typically include audio capture while TV cards include an on-board tuner.

CASE STUDY 4 - PC Graphics Card

Graphics cards are high-performance parallel processor systems in their own right. They include on-board processing and memory that allows high-speed pixel processing to be carried out on the card. A basic PC graphics card interface is shown in Figure 39.

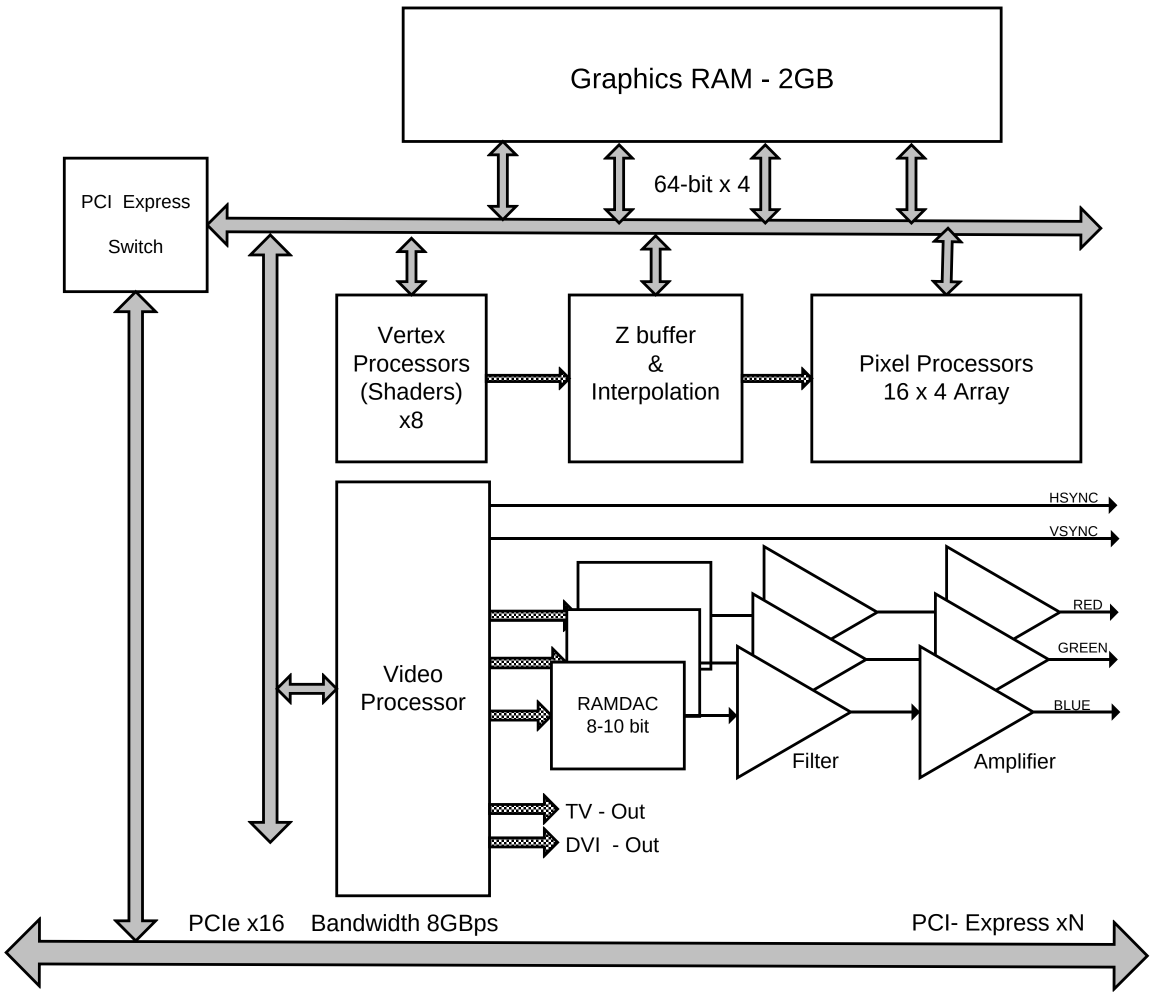


Figure 39. Graphics Card Architecture

Typical features of the graphics card are :

OUTPUTS - Single Video output channel - 8-10bit RAMDAC x3 for RGB channels, variable sampling rate 10-80MHz. The RAMDAC is a combination of FIFO and DAC. It converts the digital video data stream written into the FIFO by the processor to an analogue signal that drives the PC monitor/display/. For colour display three primary colours Red, Green, Blue are required so three channels are necessary. The horizontal and vertical (HSYNC & VSYNC) synchronisation signals provide pulses for monitor line and frame synchronisation. Additonal TV and DVI out signal provide interfaces for TV display and video recording.

ON-BOARD VIDEO MEMORY – 128 - 512MB for image buffering. This is fast DRAM, typically DDR-3 type memory. The amount of memory determines the achievable screen resolution and available colour palette.

PROCESSING - Graphics processing is a numerically intensive requiring massive amounts of real-time pixel translations and calculations. In order to maximise performance and produce realistic moving images the graphics processor executes many tasks in parallel and the graphics processor is therefore a complex device comprising several smaller numerical processing engines operating in parallel. The graphics processor in Figure 39 actually comprises several smaller processing engines operating in parallel, for example, 24 pixel processors and 8 Vertex processors. The main tasks performed by these processors are : Scaling, Translation, Perspective Correction (the road narrows in the distance), Z-buffering (not drawing objects behind other objects), Anti-Aliasing (smoothing lines to removed jagged edges), texturing and shading.

BUS INTERFACE
Video cards use the PCI Express (x16) or AGP for interfacing to the system processor.

CASE STUDY 5 - Network Interface

The Ethernet controller provides a single chip solution for network connections and is now found as a standard component in all PC's. This particular interface has been chosen because it is a digital interface with a larger number of on-chip programmable registers, typically 60-100, and on-chip memory buffers for transmit and receive data synchronisation.

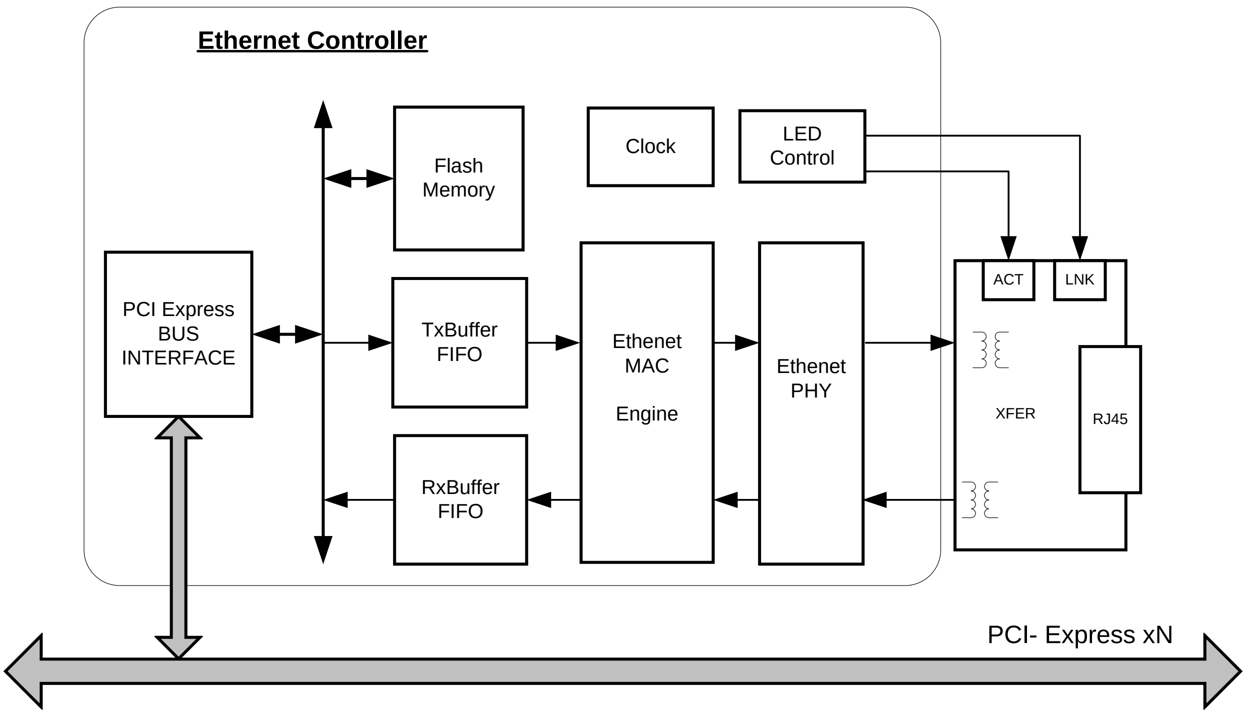


Figure 40. Network Interface Card

Typical features of the NIC are :

ETHERNET CONTROL
The Ethernet Physical block provides signal encoding and meets the signal characteristics and detects collisions. The MAC engine is responsible for managing Ethernet Frames, including CRC generation and padding.

MEMORY BUFFERS
As discussed previously these allow data transfers to be synchronised with the host controller. These are First-in-First-Out (FIFO) memory buffers which are typically 4 -8 kBytes long.

ON-CHIP FLASH
Typically around 8kB of memory which can be used either as general purpose memory or for network bios routines.
The Flash is supplied pre-programmed with a factory assigned MAC address.

LED CONTROL
The LED control block, is in reality just a couple of programmable I/O lines that can be used to control external LEDs. In this context they are used to indicate that the Link is up (LNK) and that there is activity on the network (ACT). These outputs are accessible through an internal programmable I/O registers

XFER
In order to connect to Ethernet a transformer (XFER) and a connector are required. These are conveniently supplied in a single metal package provides in a separate package.

CLOCK
An external clock XTAL (around 20MHz) is required to generate all signal timings all

BUS INTERFACE
PCI Express bus is ideal for this type of interface. A single lane operating at 250MBps is sufficient to match the real-time data transfer requirements of 1GBps Ethernet

END OF UNIT - SUMMARY OF ACHIEVEMENTS

On completion of this unit you should :

- Understand the requirements principles of analogue signal acquisition.
- Understand the requirements architecture and components of basic Data Acquisition systems (DAS).
- Understand the main functional elements and operating principles for common peripheral interfaces for Sound, Video, Graphics and Network.
- Know where to locate and apply Web based resources that inform system developers about current and future developments on a variety peripheral interface technologies.

• Frank Greig • School of Computing • Edinburgh Napier University • Edinburgh •