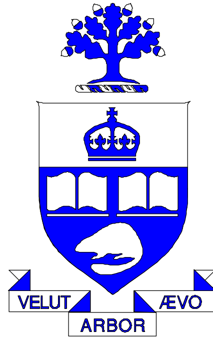
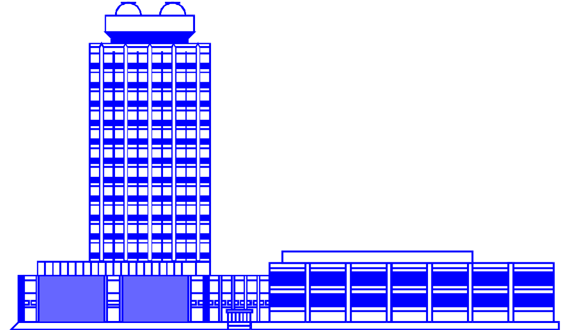




UNIVERSITY OF TORONTO
DEPARTMENT OF PHYSICS

PHY406F/PHY1406F

MICROPROCESSOR INTERFACING TECHNIQUES

Fall 1997

J.R. Drummond

Introduction to Microprocessor Hardware

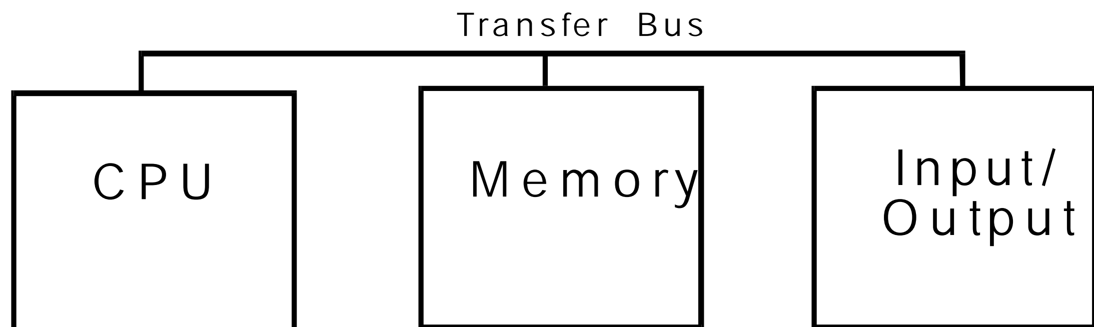
It is evidently impossible to attempt to review in detail the very many different types of processor and processor hardware available today and therefore this course will necessarily deal with either (and mainly) general principles which can be quickly particularised to most systems, or with some specific examples mainly aimed at our specific system. I do hope however to discuss some other systems as they are relevant to our lectures and for variety.

Definitions

Before we go further, a word on terminology - a "bit" (binary digit) is a single logical entity and may either be TRUE or FALSE, or 0 or 1 for short. A "byte" is conventionally 8 bits and we designate for this course bit 0 to be the least-significant bit and bit 7 to be the most significant. Please note that this is not universal and some people do it the other way round. A "word" is a collection of bits and may or may not be an integer number of bytes long. It needs a qualifier such as "16-bit word". In this course we will take the term "word" without a qualifier as a "16-bit word". There is a term for 4 bits called a "nibble" which is sometimes useful but not everybody recognises it.

Simple Systems

A computer in general consists of a set of objects in some sort of network, each of which may itself be a computer. However microprocessor systems tend to be a little simpler in that they have one Central Processor Unit (CPU), which bosses the operation and other units which are its slaves. Slaves have, however, been known to revolt. The simplest diagram of a system is as follows:



Conceptual Microcomputer System

Please note that this diagram has been drawn for simplicity, not for truth as we shall see later. The CPU fetches an entity from the memory by providing it with an address and receiving "data" back. Since the busses over which communication occurs are logical networks we can describe both the

address and the "data" as either collections of logical signals or group them together as numbers. For the computer the most logical number system is binary so that the computer fetches 10010001 from 1110000010100000 (assuming 8-bit data and 16-bit address - see below). However this is not very convenient for us as we would tend to say that it fetches 145 from address 57504. In order to get the two systems (human and computer) to discuss things in a reasonably compact form, we compromise on hexadecimal notation where each group of 4 bits from the binary representation is represented by a single alphanumeric according to the value of the 4-bit number. 0-9 are the digits 0-9 and 10-15 are the letters A-F. Thus our fetch now becomes 91(h) from E0A0(h), using the (h) notation to eliminate any ambiguity. (There are many other ways of denoting hexadecimal notation - Turbo Pascal uses a "$" sign, eg $E0A0, and the C language uses "0x" as a leader, eg 0xE0A0) The advantages of hex notation are compactness, reasonable ease of arithmetic and easy expansion to the binary where necessary (but hopefully not often). (There is also the octal system of notation which groups things together in threes but we shall not use that here)

The entity which a computer fetches is a bit pattern, however it may be interpreted by the system in various ways depending upon the "context", or in other words, what the CPU thought it was getting. Thus if a 6809 system fetches 4F(h), it may interpret that as CLRA (clear register A) if it was expecting an instruction, or as data if it expected data, or as another part of an instruction if it is halfway through a multi-byte instruction. The most common way in which you expect "context" is as follows: Sitting at a computer thinking about a program you might want to look at the set of instructions you gave the compiler - that might be "prog.cpp" - so you "type prog.cpp" and expect to see text. The computer expects to get executable instructions for a program and so it's "version" of the program is in "prog.exe". Now if you try "type prog.exe" you will be confused and if the computer tries to execute "prog.cpp" it will be confused. Both files are just collections of bits and bytes - they can even be said to represent that same thing - but you and the computer were going to interpret them in a particular context.

All bit patterns are interpreted according to the context. We shall see later that even common numbers have at least three different representations depending upon the context (character, integer or real).

In a normal machine there is a differentiation in the designer's mind between code (that is, the program) and data which is the stuff the program acts on. However as we have seen above the distinction is in the designer's mind, not the computer's which will happily attempt to execute data as code and vice versa.

© *James R. Drummond - September 1997*

ROM, RAM and Other Memory

In order to prevent a run-away machine from modifying its program and in order to avoid reloading the program every time the thing is turned on some memory is designed as Read-Only Memory (ROM) in the sense that the computer is unable to change it. (There must be some ultimate way of changing it otherwise things get difficult.) Some read-only memory is sufficiently standard to warrant manufacturing "en masse", e.g. the dot patterns for an alpha-numeric display, but mostly the memory may be written by special equipment (Programmable, Read-Only Memory (PROM)) and may further be erasable by similarly specialised equipment (EPROM). Newer developments allow the machine to change the memory with difficulty (EEPROM). Thus a simple microprocessor system dedicated to a single task may have its program stored in EPROM and some RAM (Random Access Memory or more correctly Read-Write Memory) for use by the program.

Languages and Other Issues

Armed with our knowledge of hardware and our software "hooks", or bridges, we can conquer the world in a high-level language. Why not assembler or machine code? - Well the answer to that is that in a high-level language all the useful tools are available and the code is much more readable. Which is better?

```
CLRA
STA $0007
```

or

```
A = 0
```

The rule-of-thumb I was taught was that a good programmer should produce about 100 lines of documented, tested code/day on average[1]. Now 100 lines of C does much more than 100 lines of assembler so that productivity is higher. I gratefully dropped machine-code some years ago when I made this discovery. I have more recently been informed that the bench-mark is now ten lines of code which has implications I don't even want to think about!!

Of course, the advent of grahpical languages such as LabVIEW makes the computation of "lines of code/day" an extremely tenuous thing, but the principle still remains that the higher the level of the language, the more you can get done in a finite time.

High-level languages isolate you from the machine architecture which makes the program somewhat transportable - not wholly transportable, but better than nothing.

---

[1] Note that I specified <u>documented, tested code</u> not just lines of characters which happen to pass through the compiler!

Having said all that, I admit that there are a number of occasions when assembler code is important. These are:

When the language won't support what you want to do Some languages can't support some operations. Interrupt handling routines are a case in point which can be handled effectively by very few languages.

When there is a special instruction which is very useful A case in point is an instruction to reverse the bits in a word. This turns out to be handy in a fourier transform routine, is lousy (and slow) to do in high-level language but is dandy if there is a machine instruction to do it. In this case write a short (I said SHORT) piece of assembler code as a subroutine to do the job and flee back to high-level as soon as possible.

When speed is of the essence and you can't go any faster The first defence for getting speed is to check out the algorithm for effectiveness. The second defence is to check out the coding for efficiency and the third defence is to buy the fastest machine possible. Only if all the above fails should you re-code the speed-critical portion of your code in assembler again by putting it in a short (remember I said SHORT) subroutine.

When doing DSP programming This is an exotic field, but one which we get into in the course. Since DSP programming is tricky and extremely speed-critical, but generally short - it is a candidate for assembler programming

© James R. Drummond - September 1997