MATH0025: Machine Architectures, Assemblers and Low-Level Programming.

Russell Bradford

1 Introduction

The aim of this course is to introduce you to some of the deeper aspects of programming machines. To do this we need to get quite close to the actual hardware, and see what we can exploit, and what limitations we are bound by. Typically to program at this level requires lot of tiny actions (moving data in and out of registers, adding or multiplying pairs of numbers, comparing two numbers and making a single decision from that ...) whose cumulative effect is the process we want.

At the other end of the scale we have "high level" languages such as Scheme or C: such languages allow us to express more complicated ideas in easier ways. For example, loops, function calls, complicated `if` statements.

There is a gap between the two approaches: given an expression in a high level language, what would we have to write in a low level (assembler) language. This is called the "semantic gap."

This course will help you to understand how low level machine code relates to high level (e.g., C) code.

1.1 Practical Work

This course has 25% coursework. This will be essentially writing assembler programs.

1.2 Books

Useful books include any with the number "68000" in the title. Perhaps

1. "The Motorola MC68000: an introduction to processor, memory and interfacing." J Bacon, Prentice Hall.
2. "Assembly language programming for the 68000." A Gill, E Corwin, A Logar, Prentice-Hall.
3. "MC68000 assembly language and systems programming." W Ford and W Topp, DC Heath.
4. "The 68000 Microprocessor: Hardware and Software Principles and Applications." JL Antonakos, Prentice-Hall.

1.3 Why 68000?

The astute amongst you will wonder why we are going to look specifically at the Motorola MC68000 assembly language. After all, nobody much uses the 68000 these days (and perhaps not for many years); it has been obsoleted by the 68010, 68020 and so on; there are other processors about that are far faster, and far more powerful (various Pentiums, the PowerPC, Sparc, Alpha, MIPS,...).

The reason is that the 68000 is an especially *nice* processor, it has a uniform language with few quirks, and no special features that serve to complicate matters. It is good to *start* with 68000, and then use it as a solid base to go on and understand the more modern type of processor. The 68000 was reasonably ground-breaking for its time—it was certainly miles ahead of its predecessors 8080, Z80, 6800, 6502 (all 8 bit processors)—and was used in several computers, most notibly the Macintosh, Atari and Amiga.

> Aside: it was allegedly named the "68000" as its implementation required 68000 transistors.

We may even have time at the end of the course to look at some of the quirks and complications of some more fashionable machines.

1.4 Course Outline

We will roughly follow this path:

1. Basic 68000. This is to get you started on your practical work.
2. Bits, Bytes and Words. Arranging memory for fun and profit.
3. Conditions and Jumps. How to make decisions, and where to go to do them.
4. Index Modes in 68000. Amusing ways of addressing memory.
5. Subroutines and Functions. How to package up useful stuff in a handy reusable way.
6. Parameter conventions. How to do recursions and interrupts.
7. Relationships between high and low level. What does a C construct look like if expressed in assembler?
8. More modern processors. The CISC/RISC divide.
9. Whacky stuff. Pipelines; multiple issues; prediction
10. The Future. Where's it going?
11. The Real World. Examples of real assembler.

2 Basic 68000

I'll assume that you have a fair idea of what a processor looks like (registers, arithmetic unit, and so on), that you have a vague idea of the kind of machine instructions you might find (load, store, add), and how these might be packed into a machine instruction.

2.1 Inside the 68000

The 68000 has several registers: principally the *data registers*, and the *address registers*. The first set are used mainly for the data you are manipulating, the second set for talking about addresses of memory in the machine. These days there is no real need to have separate kinds of registers for data and addresses, but when the 68000 was designed it was found helpful to building the hardware to do it like this (and they weren't so sure that addresses can be data, too).

The data registers are named `D0` to `D7` and are generally interchangeable. Each is 32 bits long.

The address registers are `A0` to `A6`, with a special `A7`. Again, these are 32 bits long. The registers `A0` to `A6` are interchangeable, but the last address register `A7` has a special rôle: it is also called `SP`, the *stack pointer*. This is used heavily in functions. (This description of `SP` is simplified to avoid extra confusion at this point.)

There are two more registers, though these are generally not manipulated directly by the user. First, the *program counter* `PC`. This points to the next machine instruction to be executed. The `PC` is 32 bits long, only the bottom 24 are used. Later processors in the 68000 series used all 32 bits.

This restriction means that a program can only span an address range of 2^{24} bytes. This is 16777216 bytes, or 16Mbytes. Many programs these days are larger than that (e.g., Netscape). The full 32 bit processors can address 4Gbytes: some datasets are larger, e.g., detailed geological maps. Some processors like the Pentium Xeon can address 2^{36} bytes, or 64Gbytes. New processors these days are moving to 64 bit registers, addressing 16 exabytes (1.8×10^{19}, or 17 billion Gbytes). This should be enough for a while!

> Aside: kilo (1024^1); mega (1024^2); giga (1024^3); tera (1024^4); peta (1024^5); exa (1024^6); zetta (1024^7); yotta (1024^8).

The last register in the 68000 is `SR`, the *status register*. This contains various flags that indicate the state of the system, e.g., interrupts. This happens to be 16 bits long, but not all the bits in it are actually used.

(Basic) 68000 instructions are 2 bytes long; however many take extra bytes as arguments, and you can find instructions taking a total of 6 bytes.

Typical 68000s run at 12.5 MHz, that is 12.5 million cycles per second (recall that it might take several cycles to complete one instruction). Compare this with processors running at 500 MHz today, and a promised 1 GHz soon! (In fact, things are even more so than they seem: a 68000 may take many cycles to complete one instruction, whereas modern processors can complete several instructions per cycle!)

2.2 Take a byte

The data registers in a 68000 are 32 bits—4 bytes—wide: however a lot of data is smaller. For example ASCII characters are 8 bits (one byte), Unicode characters are 16 bits (two bytes). When we move data between registers, we may only want to move a byte, not 4 bytes. There are three versions of the basic *move data* instruction `MOVE`:

1. `MOVE.B` to move a byte.
2. `MOVE.W` to move 2 bytes. In 68000-speak this is a *word*, where the rest of the world generally reserve the word "word" for 32 bits. This would more normally be called a *short*.
3. `MOVE.L` to move 4 bytes, called in 68000 circles a *longword* (more normally, a *word*).

Aside: other assemblers may call these `MOVEB`, `MOVEW` and `MOVEL`.

A byte occupies the lowest 8 bits of a register, a word the lowest 16 bits.

```
31                              15              7             0
x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
                                               |  <- byte ->
                                |          <- word ->
                         <- longword ->
```

Bit 0 is the *least significant bit*, while bit 31 is the *most significant bit*.

2.3 Addressing the Problem

So how so we refer to the various bytes, words and longwords in memory? Let's assume that memory is nice and uniformly arranged, from byte 0 up to byte whatever. You should note that not all machines in real life are like this, the memory might be chunked up awkwardly: the early Intel chips are an example. The 68000 *does* have uniform memory.

An address always refers to a byte: so if an address is 123, that means the 124th byte (as we count from 0).

```
            n  n+1
x  x  x  x  AB CD EF 89 x  x  x  x  x
```

Byte n would be loaded to a register as `x x x AB`. Bit 7 is the top bit of `A`. Acting on bytes (and similarly words) does not affect the upper part of the register.

What happens if we want to get a word? We still use a byte address, but now it must be even. The word we want is byte n and byte $n+1$. The upper 8 bits of the word is byte n, the lower 8 bits is byte $n+1$.

```
            n n+1
x  x  x  x  AB CD EF 89 x  x  x  x  x
```

This will be held in a register as `x x AB CD`. Bit 15 is the top bit of `A`.

Next, a longword. Now the address must be a multiple of 4. The upper word is n and $n+1$, the lower word is $n+2$ and $n+3$.

```
            n        n+3
x  x  x  x  AB CD EF 89 x  x  x  x  x
```

This will be held in a register as `AB CD EF 89`.

If we try to load or store a word on an odd address, or a longword on an address not divisible by 4, an *exception* happens. This is a bit like an interrupt: the normal flow of the program is interrupted and the processor starts running a special subroutine. This subroutine usually tries to patch up the error, or signals to the user that their program has gone awry. The famous Unix error "bus error" is often a sign that this has happened. We say words are "2 byte aligned," longwords are "4 byte aligned."

Aside: other processors may not have this kind of restriction on addressing. Or may claim not to, but take many times longer to do the load or store (maybe even 100 times longer) without telling you. An Intel 386 takes 5 clock ticks to load an aligned longword, and 10 clock ticks to load an *un*aligned longword. Many modern processors have this kind of alignment, with some data types being 8 byte aligned, e.g., double floats.

2.4 Assembler Directives

We treat briefly a few instructions in an assembly language program that are directed at the *assembler* itself. These are not instructions that become part of the code of the final program, but instead tell the assembler things like where to put the program, or to allocate space at some place for data.

There are five *directives* (also known as *pseudo-opcodes*) that we will consider.

1. `ORG` *origin*
2. `DC` define constant
3. `DS` *define storage*
4. `EQU` *equal*
5. `END`

Aside: we shall talk about the ASM68K assembler that you are using for the practical classes. Other assemblers for the 68000 may have different names for opcodes and directives, e.g., `.origin` instead of `ORG`, but you will always find equivalents.

A memory map

```
+----------------+  High addresses
|                |
|                |
+----------------+
|                |
|   your code    |
|    and data    |
|                |
+----------------+  ORG
|                |
|                |
+----------------+
|  interrupt     |
|   tables       |
+----------------+
|                |
+----------------+
|  boot address  |
+----------------+  Low addresses
```

The `ORG` directive tells the assembler where to start putting the code or data that follows. Thus

```
      ORG $8000
START MOVE.B     #$1, D1
      ...
```

The `MOVE.B` will be located at machine address hex 8000.

With the ASM68K assembler you should normally set the start of your program to be (say) hex 8000. There is nothing very special about this number, it is just nice and round. You *should* avoid starting a program at a small address (e.g., 0), as typically machines reserve the area of memory from 0 to n for special purposes. For example, often the word at address zero is used to give the address of the boot code for the machine, i.e., when you first switch the machine on the processor fetches an address from memory location 0, and then starts executing from that address. Similarly, there might be tables of addresses for code to deal with exceptions and interrupts. Thus if you have a processor that allows 32 different interrupts, there is a table somewhere of 32 addresses of the code to deal with those interrupts.

The `ORG` can be used more than once, allowing you to put different parts of program or data in different areas of memory.

```
      ORG $8000
START MOVE.B    #$1, D1
      ...
      ORG $1000
SUB   MOVE.L    #$1234, D5
      ...
```

After an `ORG` the data or program that follows is stored in consecutive addresses.

To allocate a space for data, use `DC` and `DS`.

```
ANS   DS.B      16             ; room for 16 bytes
```

`DS` (*declare storage*) reserves an area of 16 bytes of memory, whose address starts at `ANS`. Note that `ANS` is an *assembler label*. This is just a convenient shorthand for the number that is the address of the data area. It is *not* a variable.

This area is allocated wherever in the flow the assembler happens to be at the time. You can put it in the middle of your program, but probably you don't want that:

```
      MOVE.B    #$56, D1      ; address n
ANS   DS.B      16            ; address n+4
      MOVE.B    #$42, D2      ; address n+20
```

when the program is running, it will try to execute whatever is in the data field. Probable result: invalid instruction exception. Certain result: a non-working program.

Also we have,

```
ANS   DS.W      16             ; room for 16 words
BIG   DS.L      32             ; room for 32 longwords
```

saves an area of 16 words, or 32 bytes, followed by 32 longwords, or 128 bytes.

The directive `DC` (*declare constant*) finds space for a value:

```
SUM   DC.W     0
```

makes `SUM` (shorthand for) the address of a word of memory that is initialised to the value 0. Note that we *can* update the value stored in address `SUM`, calling it a constant is a little misleading. We also have `DC.B` and `DC.L`.

Next we have `EQU`. This is another useful shorthand device, allowing us to give names to constant values. With

```
LF      EQU   $A
CR      EQU   $D
```

we may use the names `LF` and `CR` to stand for the hex values A and D. Note that we do not need `.B` or whatever variants, as (a) we are not allocating space anywhere; (b) the symbols are simply textually replaced with their values wherever they appear.

The last directive is `END`. This is used for two things:

1. to inform the assembler that this is the end of the source file
2. to indicate where to enter the program

Ending a program

```
     ORG $8000
SUM  DC.B     0
     ...
HERE MOVE.B   SUM, D1
     ...
     END      HERE
```

indicates that when you run this program (using the EMU68K), you should start running it from address `HERE`, not from the first address you come across—maybe `SUM` in this case, which would not be the desired effect.

Exercise: what are the values of addresses `BB` and `WW` in

```
     ORG $8000
BB   DS.B          1
WW   DS.W          1
```

What implication does this have on alignment?

2.5 Some Assembler Mnemonics

The basic assembler instruction for addition is `ADD`. We can add bytes:

```
ADD.B     D4, D5
```

This will add the value of `D4` into the value of `D5`, and leave the result in `D5`.

> Aside: assemblers for other processors given an instruction like `add a, b` might leave the result in `a`; or they might want *three* arguments, `add a, b, c`, where `c` gets `a` plus `b` or `a` gets `b` plus `c`. This gets even more confusing with non-commutative operators like subtraction!

The instruction above adds the lowest bytes of `D4` and `D5`. The other bytes of the registers are unaffected. There does remain the question of what happens when there is a overflow (carry) from the addition, but we will talk about that later.

Example: if `D1` has value 012345FE, and `D2` has value 10000005 (numbers in hexadecimal, so it is easy to see what is happening), then

```
ADD.B     D1, D2
```

results in `D2` containing 01234503. Notice that the addition did not carry into the upper bytes of the register.

Adding words and longwords

```
ADD.W     D4, D5
ADD.L     D4, D5
```

This affects, respectively, the lower two bytes, and all the bytes of the registers.

With `D1` and `D2` as above,

```
ADD.W     D1, D2
```

produces 01234603, while

```
ADD.L     D1, D2
```

produces 11234603.

There is also:

```
ADDA.x    D4, A3
```

to add the value of `D4` into the address kept in `A3`. The `x` can only be `W` or `L`. (Note this is a departure from a completely *orthogonal* instruction set: that is one where all instructions come in all variants that make sense.) You might want to do this sort of thing if, say, `A3` contained the address of the start of a table of values, and `A4` was the index into the table.

> Aside: I'll emphasise again that there is no real difference between data and addresses; this is just a distinction made by the 68000 for its own purposes. The physical bits of wire coming out of the chip for memory addressing and data movement are connected to different gubbins inside. Such data and address distinctions are rare these days.

Next

```
ADDI.x    #32, D4
```

for `x` being `B`, `W`, or `L`, to add *immediate* data, viz., the (decimal) number 32 into the destination register. The sharp sign (`#`) indicates an immediate value. Incidentally, we can notate hexadecimal numbers by using a `$`:

```
ADDI.x    #$20, D4
```

does the same as above.

A slight variant of the above is the *add quick*

```
ADDQ.x    #n, D4
```

`x` is `BWL`, and `n` is 1 or 2 or 3 or 4 or 5 or 6 or 7 or 8. The advantage of this instruction is that the `n` can be encoded as part of the `ADDQ` instruction. There are 3 bits inside the instruction itself which give `n`: thus there are effectively 8 different add quick instructions. On the other hand, the `ADDI` instruction takes 2 or 4 bytes extra as arguments to specify the immediate value.

```
+-----+-----+-----+-----+
|    ADD    |    arg    |               B or W
+-----+-----+-----+-----+

+-----+-----+-----+-----+-----+-----+
|    ADD    |          arg          |   L
+-----+-----+-----+-----+-----+-----+

+-----+-----+
| 111 ADDQ  |
+-----+-----+
   ^
3 arg bits
```

This saves a lot of space, as it turns out that when we add immediate data (e.g., increment a variable in C) we almost always add 1. This kind of idea—having special case instructions for particular values of arguments—is something that is strongly avoided by modern processors.

> Aside: the 68000 was designed when memory was expensive, and at a premium, so saving a few bytes here and there was worthwhile. These days there is no such pressure.

Another is

```
ADDX.x    D3, D4
```

the *add extended* operation. This adds D3 into D4 and adds in the value of the X bit. This is one of the bits in the status register: it would be set (or not set) by an earlier arithmetic operation, thus allowing us to to do arithmetic on integers bigger than can fit into 32 bits. For example, to add together two 64 bit integers held in registers (D1, D2), (D3, D4) is just

```
ADD.L    D2, D4          ; this sets X bit if needed
ADDX.L   D1, D3
```

where the add extended does the carry for us.

There another variants on addition, including ABCD to add BCD (binary coded decimal) numbers. For details, see a book.

The above are just examples of what we can do: more generally instead of simple registers as sources and destinations of operations, many instructions allow the use of *effective addresses*. These we will describe later, generally allow us to refer to registers or memory in quite complex ways.

slide of p.26 of Antonakos.

2.6 Adding it all up

The other arithmetic operators include:

1. SUB, SUBA, SUBI, SUBQ, SUBX for subtract, subtract from address, subtract immediate, subtract quick, subtract extended. These are much as you would expect.

   ```
   SUB.L    D1, D2
   ```

   puts the result of D2 − D1 in D2.

2. NEG, NEGX to negate a number. Negate with extend subtracts off the X bit, too.

3. MULS, MULU for multiply signed integers, multiply unsigned numbers. We will expand on signed and unsigned in just a moment.

4. DIVS, DIVU for division of signed and unsigned integers.

5. CLR sets the destination to be 0.

These come in the usual flavours of BWL. Notice that CLR.B will clear the bottom 8 bits to 0, and CLR.W will clear the bottom 16 bits to 0.

The MUL instructions only come in the W size. They multiply together two 16 bit arguments to produce a 32 bit result. The MULU treats the arguments as unsigned integers; and the MULS treats them as signed (2's complement), and produces an answer of the appropriate sign.

To see the difference, multiply the hex value 8001 by 2. Unsigned this is decimal 32769 times 2, with result hex 10002; signed this is decimal -32767 times 2, with result hex FFFF0002.

Aside: the processor does not "know" whether we want to regard a number as signed or unsigned; is purely *how we interpret* the number. Usually (as with addition) the mechanics of bit pushing give us the correct result regardless; multiplication doesn't. So for multiplication (and division) we have to specify. There is a more general point that data only has meaning in a context, i.e., of how you wish to interpret it. Thus, hex byte E1 may be a decimal integer (value 225); or a negative integer (-30); or the ISO character "á"; or a colour value in a picture; or a volume in a sound; or anything else whatever.

Similarly, `DIVS.W` and `DIVU.W` are signed and unsigned quotient with remainder operations. They take a 32 bit and a 16 bit pair of arguments, and produce a 16 bit quotient in lower word of the destination, and a 16 bit remainder in the upper word of the destination. There are two untoward things that might happen with division:

1. division by zero. An exception is called.
2. quotient overflow. When the answer won't fit into 16 bits. If this happens the `V` flag in the status register is set, allowing the user to take some appropriate action. The operands remain unchanged.

An example:

```
MOVE.L     #$8001, D1
MOVE.L     #$2, D2
DIVU.W     D1, D2        ; D2 / D1
```

leaves 00014000 in register `D2`. This is quotient 4000, and remainder 0001.

Aside: modern processors typically provide 32 bit (or larger) arguments giving a 64 bit product, and 64 bit arguments giving a 32 bit quotient and 32 bit remainder.

2.7 The Status Register

It is time to talk a little about the status register. For the moment we will concentrate on just five bits in the register. These are various flags that are set or cleared according to the results of each operation. For example, and `ADD` might cause the carry bit to be set. These bits are called *condition codes*. Later we will see how we can test the values and make decisions using them.

1. `N` the *negative* bit. This is set if the MSB of the result is set, cleared otherwise. Note that the MSB of a byte operation is bit 7, and of a word operation is bit 15. This allows us to tell if value is negative.
2. `Z` the *zero* flag. This is set if the result is zero, cleared otherwise.
3. `V` the *overflow* bit. This is set if there is a possibility that the sign bit of the result is wrong, e.g., adding two large positive numbers using 2's complement might trigger the top bit, apparently producing a negative answer. For example, using byte arithmetic, adding positive numbers 01111111 (127) and 00000001 (1) results in (10000000), which is -128. As unsigned numbers, this is fine, but as signed numbers, the flag is set to warn us that we may have a problem. (Again, the processor doesn't "know" which we want, so flags are provided to cover all eventualities.)

4. `C` the *carry* bit. This is set if there is a carry out of the MSB during an addition, or a borrow during a subtraction.

5. `X` the *extend* bit. Most operations do not affect this flag, and those that do usually set it the same as `C`. Used in `ADDX` and similar operations.

To determine the exact effect an instruction has on these flags you need to look at the documentation for the individual operation. For example, the documentation for `ADD` says

> `X` Set the same as the carry bit
> `N` Set if the result is negative, cleared otherwise
> `Z` Set if the result is zero, cleared otherwise
> `V` Set if an overflow is generated, cleared otherwise
> `C` Set if a carry is generated, cleared otherwise.

And `CLR` has

> `X` not affected
> `N` always cleared
> `Z` always set
> `V` always cleared
> `C` always cleared.

Whereas `MULU` has

> `X` not affected
> `N` Set if the result is negative, cleared otherwise
> `Z` Set if the result is zero, cleared otherwise
> `V` always cleared
> `C` always cleared.

There is a curiosity that the `N` bit can be set during an operation on unsigned numbers: that's probably an indication that the `N` bit is implemented as a hard wire to the top bit.

There is one instruction included in the 68000 set specifically to affect the condition codes.

```
CMP.x    D4, D5
```

This is the *compare* instruction. It subtracts the source value from the destination value (`D5` − `D4`) and sets the flags according to the result. The destination register is left unchanged, i.e., the result value is discarded.

Variants are

1. `CMPA` compare address

2. `CMPI` compare immediate

3. `TST` test for zero

`TST` is like `CMPI #0`, but shorter. Again, it turns out that a huge percentage of tests made are, in fact, tests for zero, so having this specific instruction saves (a) space, and (b) time, in that we do not have to read in the argument to the `CMPI` instruction.

> Aside: apart from the fact that (theoretically) all we need to write a program are operations (1) to add 1, and (2) to test against zero, in practice these two operations are by far the most popular.

Also there are the more sophisticated `CMPM` (compare memory) and `TAS` (test and set). There is no compare quick in the 68000 (apart from `TST`, which is a "quick" test against zero).

3 Bits, Bytes and Words

Next we are going to look in more detail at moving bytes about. But first we have to consider the way things are laid out in memory in more detail.

3.1 Big and Little Endian

The arrangement of bytes and words within longwords in the 68000 is by no means the only way of doing things: and by some counts it is in the minority.

Recall that the number ABCDEF89 is held in memory as

```
            n        n+3
x   x   x   x   AB  CD  EF  89  x   x   x   x   x
```

This arrangement is called *big endian*. The "big" end of the number is at the earliest address. The principal opponent is *little endian*. Little endian is used in Intel processors, and so is fairly widespread across the world. Roughly speaking, most processor designs these days use big endian, while most processors out there in the world use little endian.

The picture for little endian is this:

```
            n        n+3
x   x   x   x   AB  CD  EF  89  x   x   x   x   x
```

will correspond to the number 89EFCDAB. The "little" end of the number is at the earliest address. A better way of viewing this is to think that the number ABCDEF89 is held in memory as

```
            n        n+3
x   x   x   x   89  EF  CD  AB  x   x   x   x   x
```

The choice between the two is really based on hardware considerations, and there's not a lot to choose between the two: however we should note that the "network byte order," (the standard order chosen on the Internet) is big endian. Thus Intel machines have to do a lot of byte swapping whenever it talks to the 'net.

Big endian processors include 68000, Sparc (Sun), MIPS (Silicon Graphics). Little endian include VAX (DEC), Pentium (Intel), and MIPS again. Some processors can be configured to go either way: a MIPS makes the choice whenever it is reset, and an i860 has a software instruction to swap endianess while running.

Generally, the endianness of a process does not make a difference to a program. However, there are circumstances when it does, most when you try to pick apart a number into its constituent bytes: you will have to know which is the least and most significant bytes to be consistent.

3.2 More Moving

We have seen the most basic instruction to move data from one place to another is the `MOVE` instruction.

```
MOVE.x    D4, D5
```

This moves the value of `D4` overwriting the value of `D5`. It also sets the condition codes according to the sign of the value moved.

Variants:

```
MOVEA.x   D4, A5
```

for `x` in `WL`. This is how to get data into an address register.

```
MOVEQ.L   #n, D5
```

move quick, where `n` is an 8 bit value. Notice that the whole longword destination is affected. Further, the 8 bit value is *sign extended* to the 32 bit value.

Sign extension is where the top bit of a shorter value is extended out across a longer value: if the byte F0 is sign extended to a word, it becomes FFF0. The reason for this is to preserve the meaning of the value as a 2's complement number. Both F0 as a byte and FFF0 as a word mean -16, whereas 00F0 as a word means +240.

In fact there is an explicit instruction to do sign extension:

```
EXT.W     D4
EXT.L     D5
```

The first extends a byte to a word by copying bit 7 into bits 8–15; the second extends a word to a longword by copying bit 15 into bits 16–31.

To load a register with more than an 8 bit value, the basic `MOVE` will work:

```
MOVE.L    #$12345678, D6
```

More exotic data movements are

```
EXG.L   R2, R3
```

where R can be D or A. This exchanges the values in R1 and R2. Also

```
SWAP.W   D3
```

swap the high and low words of a register. This means the value 1234ABCD becomes ABCD1234. This is not quite enough to convert from big to little endian, unfortunately, but is useful in getting at the upper word; for example, the remainder of a divide.

There are also instructions to move data in and out of the stack pointer and the status register; MOVEM *move multiple*, to move collections of registers to and from memory; MOVEP *move peripheral*, for moving data to and from peripherals.

4 Conditions and Jumps

4.1 Feeling Jumpy

Not all programs run in a single straight line from start to finish. Sometimes we need to jump about a bit, say skip a bit of code, or go back and do something again. HLLs give us do or while loops, and if statements to allow repetition of code and conditional execution of code. In assembler we don't have such luxuries: instead we have to jump around code. But notice there are two classes of jump: the jump at the end of a while loop is *unconditional*, we should always do it. On the other hand the jump in an if is conditional, sometimes we want to do it, and sometimes not.

The unconditional jump instructions in 68000 are JMP (*jump*) and BRA (*branch always*). There are two unconditional jump instructions for a very good reason. They are used like this

```
JMP     #$123456    ; jump to address $123456
```

and

```
BRA     #$12        ; branch to address $12 bytes away from here
```

The first, JMP, gives an *absolute address*. After executing this instruction, the program continues from precisely that address.

> Aside: in fact all that this instruction does is to stuff the given address into PC, the program counter.

The second kind of jump is more subtle: it gives a *relative address*, counting from the present address. This relative value may be positive or negative.

> Aside: to do this, the given value is *added* into the PC.

Why provide two different ways? Firstly, `BRA.x` is limited to `x` in `BW`, that is, a maximum offset of -32768 to 32767, which is a limited part of the entire address space (this is often enough, though). On the other hand, the `JMP` instruction can reach *all* of the 24 bit address space.

Secondly, absolute addresses tend not to be quite so convenient in machines with virtual memory. This is because a program may be loaded into any part of the physical address space, so that an absolute address in a program is usually meaningless. Much better is always to give *relative* offsets, so that a jump will be correct wherever the program is loaded. Such a program is called *relocatable* or has *position independent code.*

> Aside: relocatable code is absolutely essential these days so that the same piece of code can be shared amongst many programs all running simultaneously.

> Aside: the 68000 also provides a *PC displacement* `JMP` variant; but it's still more convenient to use the branch instruction.

Note: in practice we never use numbers in jumps and branches, but use labels instead.

```
HERE MOVE ...
     ...
     BRA HERE
```

The assembler does the hard work of counting the number of bytes offset required.

4.2 Decisions, decisions

In programs we often need to make decisions. Such as "is this number bigger than that number?" Or "is this number zero?" And after having made a decision we can choose to follow different courses of action. Thus we may wish to process a value differently if it is negative rather than positive or zero (e.g., the discriminant of a quadratic equation). In a high level language we have constructs like `if` that neatly structures our decisions, but in assembler we must use *conditional* branches: these are jumps that only happen if certain conditions (the flags) are met.

> Aside: it is very easy to get into a great muddle with jumps going here there and everywhere. Experience over the years has shown that the constructs used in modern high level languages are much more suitable to comprehensible and maintainable programming than simple jumps. Early HLLs had jumps (gotos), and it was found that it was easy to write very bad programs using them. However, processors only provide jumps, so we are stuck with them. Thus we have to be careful when we do so, and model our use of jumps on what we might have done in a HLL.

To do this within a 68000 program, we will need to look again at the status word, and most importantly the `N`, `Z`, `V` and `C` flags.

4.3 A good condition

Certain operations affect the ZNVC flags: MOVE will set Z if the result is zero; it will clear it if non-zero. It will set N is the result is negative; it will clear it if non-negative. The instructions ADD, SUB, MULS all do similarly. And DIVS sets the flags according to the value of the quotient.

If we have values presented to us from an uncertain source, we can always ensure the flags are set to the appropriate values by using CMP.

So let us assume that we have a condition, and we need to branch on it. We are provided with the following:

1. BHI branch if high
2. BLS branch if lower or same
3. BCC branch if carry clear
4. BCS branch if carry set
5. BNE branch if not equal
6. BEQ branch if equal
7. BVC branch if overflow clear
8. BVS branch if overflow set
9. BPL branch if plus
10. BMI branch if minus
11. BGE branch if greater than or equal
12. BLT branch if less than
13. BGT branch if greater than
14. BLE branch if less than or equal

The are used like

```
Bcc     #$1234
```

or more likely

```
Bcc     label
```

where cc is whatever you require, and label marks the destination. If the relevant condition is true, we branch to the specified address; if not we continue with the next instruction after the branch.

```
CMPI.L      #$2A, D1            ; ASCII '*'
BNE         #$AB
ADD.W       ...
...
SUB.W       ...                 ; this instruction is $AB later in code
```

If the value in D1 is not '*', then we branch to the SUB instruction. Otherwise, we continue with the ADD.

More realistically,

```
AGAIN  MULU     D2, D3   <--+
       SUBQ     #1, D2      |
       BNE      AGAIN    >--+
       ...
```

The symbolic *label* AGAIN is used rather than an explicit number. The assembler does the hard work of counting how many bytes there are between branch and destination. If D3 started with 1, and D2 with 5, then D3 will end with $5 \times 4 \times 3 \times 2 \times 1 = 120$ when the loop finishes.

The technical meaning of these branches is as follows:

1.	HI	C = 0 and Z = 0	$\overline{C} \wedge \overline{Z}$
2.	LS	C = 1 or Z = 1	$C \vee Z$
3.	CC	C = 0	$\overline{C}$
4.	CS	C = 1	C
5.	NE	Z = 0	$\overline{Z}$
6.	EQ	Z = 1	Z
7.	VC	V = 0	$\overline{V}$
8.	VS	V = 1	V
9.	PL	N = 0	$\overline{N}$
10.	MI	N = 1	N
11.	GE	N = V	$(N \wedge V) \vee (\overline{N} \wedge \overline{V})$
12.	LT	N $\neq$ V	$(N \wedge \overline{V}) \vee (\overline{N} \wedge V)$
13.	GT	N = V, and Z = 0	$(N \wedge V \wedge \overline{Z}) \vee (\overline{N} \wedge \overline{V} \wedge \overline{Z})$
14.	LE	N $\neq$ V, or Z = 1	$Z \vee (N \wedge \overline{V}) \vee (\overline{N} \wedge V)$

Thus BPL will also branch on zero, as zero has the flag N cleared.

4.4 It's a Sign

Comparison of signed and unsigned numbers is slightly different: suppose register D0 contains byte 80, and D1 the byte 7F, respectively. The byte numbers 80 and 7F mean decimal 128 and 127 as unsigned integers, but -128 and 127 as signed integers. Thus we should expect different answers to a comparison of D0 and D1 dependent on whether they are to be regarded as signed or unsigned.

D0 < D1	signed
D0 > D1	unsigned

What we do is choose our branch conditions carefully. After

```
     CMP.B    D1, D0
```

(recall this sets flags according to the value of D0 − D1) for signed arithmetic:

$\texttt{D0} > \texttt{D1}$	use BGT
$\texttt{D0} \geq \texttt{D1}$	use BGE
$\texttt{D0} = \texttt{D1}$	use BEQ
$\texttt{D0} \neq \texttt{D1}$	use BNE
$\texttt{D0} \leq \texttt{D1}$	use BLE
$\texttt{D0} < \texttt{D1}$	use BLT

For unsigned arithmetic:

$\texttt{D0} > \texttt{D1}$	use BHI
$\texttt{D0} \geq \texttt{D1}$	use BCC
$\texttt{D0} = \texttt{D1}$	use BEQ
$\texttt{D0} \neq \texttt{D1}$	use BNE
$\texttt{D0} \leq \texttt{D1}$	use BLS
$\texttt{D0} < \texttt{D1}$	use BCS

If you work though the various combinations, you will see why these work.

4.5 More Branches

There is another set of branch instructions: `DBcc`, for `cc` as before. These are the *decrement and branch* instructions. These simultaneously do the test, and also decrement a value. This is useful for counting in loops.

Another related set is the *set according to condition* instructions, `Scc`. These test flags, and then set a value to all ones or all zeros if the condition is true or false respectively.

4.6 Nothing to do

The `BRA` instruction always jumps: it also has an opposite that *never* jumps!

The `NOP` instruction does nothing. No state is affected. Strictly speaking, that is not true: the program counter moves on, and some time passes (4 cycles). This instruction is useful for stuffing into timed loops.

> Aside: modern processors have a more essential use for `NOP`. We might call `NOP` an "unstruction"!

4.7 Caught in a trap

Special conditions like interrupts and exceptions allow us to deal with special circumstances. An interrupt is called when someone in the outside world is trying to get our attention (e.g., a slow peripheral is ready); while an exception is raised when there is a problem within our program (e.g., division by zero, or misaligned load). When one of these happens special system code is called (the processor also changes into a special state, called *supervisor*, where certain extra instructions are enabled) which will deal with the problem. Sometimes we want to trigger the same sort of thing ourselves, which is what the `TRAP` instruction does. The processor allows for 16 different `TRAP`s, and there is a table of 16 addresses indicating where the code for each lies. In fact, the table must lie in memory locations 000080 to 0000BF. Calling

```
TRAP    #0
```

will look up the address stored in 000080,81,82,83 and continue executing from that address.

For now, all we need to think about is the 68000 emulator, and this has a selection of `TRAP`s defined, including:

`#0` get a character (into `D1`) from keyboard

`#1` put a character (from `D1`) to screen

`#3` put a zero-terminated string (starting address in `A3`)

`#9` return to emulator

So to write a character you can use

```
MOVE.B   #65, D1          ; ASCII 'A'
TRAP     #1
...                       ; after output, continues from here
```

5 Subroutines and Functions

We are taught these days to program using lots of small functions: this aids program maintainability, and means you can reuse the same function in many places. The same sort of thing can be done in assembler, using `JSR` *jump to subroutine* or `BSR` *branch to subroutine*; and `RTS` *return from subroutine*. As with branches and jumps above, the main difference between `JSR` and `BSR` is that the former takes an absolute address, and the latter a relative address.

5.1 Stacking it up

We first need to recall details of the stack. The stack is just an area of memory that is used in a special way. There is a value, the *stack pointer*, generally a register, that points at the area, and as values are pushed and popped off the stack the SP is increased or decreased as appropriate, so always to point at the top of the stack.

> Aside: when we say something "points at" a memory location, this just means that the value of the something is the address of that memory location.

In term of function calls, the stack is used to save *return addresses*: if a subroutine is called, the current PC is pushed onto the stack, then the program jumps to the location of the subroutine. When the subroutine is done, the old value of the PC is popped off the stack, and execution continues from the old address. The beauty of this is that we can have nested subroutine calls, and the first-in-last-out nature of the stack ensure that the return addresses always come out in the right order.

Suppose we have a register `SP` for the stack pointer. Calling a subroutine has this effect:

1. $\texttt{SP} - 4 \to \texttt{SP}$
2. $\texttt{PC} \to (\texttt{SP})$ save return address
3. new destination address $\to$ `PC`

The "4" is the size of an address in bytes, i.e., 32 bits (although only 24 bits are actually used in a 68000).

A picture:

```
       aa    high address            aa    high address
       aa                            aa
       aa                            aa
SP --> aa                            aa
       xx                            bb
       xx                            bb
       xx                            bb
       xx                     SP --> bb
       xx                            xx
       xx                            xx
       ..    low address             ..    low address

     before                  after, where bbbbbbbb was in PC
```

This means

1. the stack (in a 68000, and many other machines) actually goes downwards in memory, and the `SP` points at the last stored value. So we decrement `SP` by the length of an address to point at a new empty slot in the stack.
2. we store the `PC` in the address the `SP` refers to.
3. we continue executing from the new address

Returning from a subroutine:

1. $(\mathtt{SP}) \to \mathtt{PC}$ jump back to saved address
2. $\mathtt{SP} + 4 \to \mathtt{SP}$

or

1. we take the value in memory referred to by `SP`, and plonk it in the `PC`, so we will continue to execute from this restored address, after
2. having incremented the `SP` to point at the previous value in the stack.

The picture

```
       aa    high address            aa    high address
       aa                            aa
       aa                            aa
       aa                     SP --> aa
       bb                            bb
       bb                            bb
       bb                            bb
SP --> bb                            bb
```

```
   xx                              xx
   xx                              xx
   ..    low address               ..    low address

 before                  after: note old address still on stack!
```

This may seem complicated, but it is worthwhile spending some time understanding what is going on.

The 68000 uses the `A7` address register as the stack pointer, and there are several instructions that implicitly use it, in particular `JSR`, `BSR`, and `RTS`.

The `BSR/JSR` instructions essentially do all of the first set of instructions; while the `RTS` does the second set, using `A7` where we had `SP`. This is an especially convenient way of doing things:

```
     BSR     SUM  ---+
     MOVE.B  ...  <--+--+
     ...             |  |
SUM  MOVE.W  ...  <--+  |
     ...                |
     RTS          ------+
```

This code will branch to the subroutine `ADD`, saving the return address (of the instruction after the `BSR`) on the stack (as indexed by `A7`); and then the `RTS` will pull that address back off the stack, restoring both `PC` and `A7` to their previous values.

More specifically we have

```
        ORG $8000
START   MOVE.L  A7, D1          ; 8000
        BSR     FOO             ; 8002 ---+
        TRAP    #9              ; 8006 <--+--+
FOO     MOVE.L  A7, D2          ; 8008 <--+  |
        BSR     BAR             ; 800A ---+  |
        RTS                     ; 800E <==+==+
BAR     MOVE.L  A7, D3          ; 8010 <--+  |
        RTS                     ; 8012 ------+
        END     START
```

The address stored in `D1` is 000FF0, `D2` has `000FEC`, and `D3` gets `000FE8`. The stack in the emulator starts at FF0, and goes down 4 bytes for each subroutine call.

> Aside: this limits us to a depth of about decimal 1000 nested subroutines.

Before `BSR FOO`

```
SP --> FF0  xxxx
       FEC  xxxx
       FE8  xxxx
```

and after

```
       FF0  xxxx
SP --> FEC  008006
       FE8  xxxx
```

after `BSR BAR`

```
       FF0  xxxx
       FEC  008006
SP --> FE8  00800E
```

Then after the `RTS`

```
       FF0  xxxx
SP --> FEC  008006
       FE8  00800E
```

and after the second `RTS`

```
SP --> FF0  xxxx
       FEC  008006
       FE8  00800E
```

> Aside: use `d F70` to see the stack. Remember that the top of the stack is at the end of the block.
>
> Aside: this is another reason why it is good to start your programs at, say, 8000 in the emulator. Then there is no chance that the stack growing down will overwrite the program. The emulator typically has this layout:
>
> | | |
> |---|---|
> | $\uparrow$ | high address |
> | data | |
> | | |
> | $\uparrow$ | |
> | program | |
> | | |
> | stack | |
> | $\downarrow$ | |
> | | |
> | system info | low address |
>
> though it actually is up to you how you arrange the program and data.
>
> A typical layout of memory in a real machine is
>
> | | |
> |---|---|
> | stack | high address |
> | $\downarrow$ | |
> | | |
> | $\uparrow$ | |
> | data | |
> | | |
> | $\uparrow$ | |
> | program | |
> | | |
> | system info | low address |

with arrows indicating which direction they expand as they grow. The top of the stack is typically many mega- or gigabytes (in virtual address terms) away from the program and data, which are often fairly close together.

5.2 Subroutine Reuse

A major point of subroutines is that you can call them many times from many places. Careful use of subroutines means that you have to have some particular convention on calling them: in particular how to pass data to them, and return data from them.

Typically, there is a convention of passing arguments in registers, e.g., in the exercise `ex0.asm` the subroutine `OUT` expects a character in register `D1`. And `TRAP 1` expects a character in the same register. Similarly, the `IN` routine returns a character in `D1`.

We will cover such conventions in detail later, but for the moment you should pick something reasonable, and (a) document it in the subroutine code, and (b) stick to the convention.

There are other instructions that perform similar, but more sophisticated, operations to the ones already presented, namely `LINK` and `UNLK`. We will cover these later when we get to parameter conventions.

6 Index Modes in 68000

So far we have only mentioned operations on registers

```
ADD.L   D1, D2
```

immediate data

```
ADDQ.B  #4, D5
```

and hinted at

```
 JMP     #$123456
```

for an absolute address. That was to keep things simple. in fact there is a rich variety of different ways of operating on memory, immediate data, and registers. These are the different *addressing modes* of the 68000.

> Aside: the 68000 has a particularly large number of addressing modes, some more useful that others. Modern processors tend to stick to those few that are most heavily used.

The modes are named

1. data register direct
2. address register direct
3. address register indirect

4. address register indirect with postincrement
5. address register indirect with predecrement
6. address register indirect with displacement
7. address register indirect with index
8. absolute short address
9. absolute long address
10. program counter with displacement
11. program counter with index
12. immediate data
13. quick immediate data
14. implied addressing

These come under the collective umbrella of *effective address*.

> Aside: the 68020 added even *more* addressing modes.

In documentation you will find the specification of, say, `ADD` to be

```
ADD <ea>, Dn
```

or

```
ADD Dn, <ea>
```

together with a list of the allowable types of effective address in each circumstance. For example, as a source operand, any of the above modes are allowed, while as a destination, only modes that alter memory are allowed: this excludes, for example, a destination of immediate data.

```
ADD    D1, #$1234
```

is meaningless, and is disallowed.

To find the modes allowable for any particular operator, there is no recourse other than to look at the documentation. A fully orthogonal instruction set would aloow all modes that are meaningful: the reality is that the 68000 cuts corners here and there.

6.1 Data Register Direct

This is a mode we are familiar with already: simply a data register `D0` to `D7`.

Example

```
ADD.L    D4, D5
```

both source and destination are DRD mode.

6.2 Address Register Direct

The same as DRD, but now with an address register.

Example

```
MOVEA.L    D4, A5
```

the source is DRD, and the destination is ARD.

> Aside: the documentation for `MOVE` says
>
> ```
> MOVE <ea>, <ea>
> ```
>
> and allows an address register as destination. However assemblers may require you to use
>
> ```
> MOVEA <ea>, An
> ```
>
> The same code is produced in either case (i.e., the bits in the compiled program are the same).

6.3 Address Register Indirect

This is the first mode that manipulates memory, rather than registers.

```
MOVE.L    D4, (A5)
```

stores the value in `D4` into the memory location indicated by `A5`. And

```
MOVE.L    (A2), D1
```

loads the value in the memory location indicated by `A2` into the register `D1`.

Similarly,

```
ADD.L    D4, (A6)
```

adds the value of `D4` into the value stored in memory at `A6`.

Suppose `A3` contains hex 00AABBCC, while in memory location 00AABBCC there is the longword value hex 42424242. Note the difference between

```
MOVE.L   A3, D1
```

and

```
MOVE.L   (A3), D1
```

The first moves the value 00AABBCC into `D1`. The second fetches 42424242 from memory, and puts that in `D1`. This distinction is *very important*, and you must get it clear in your mind what is happening.

6.4 Address Register Indirect with Postincrement

This is a clever mode, designed for traversing data in memory, commonly used in loops.

Suppose we have a table of 16 bytes in memory, starting at address `TAB`, and we wish to add all of them together. We might do this

```
     MOVEA.L    #TAB, A0           ; pointer into table
     MOVE.B     #16, D1            ; init loop counter
     CLR.W      D0                 ; clear sum
LOOP ADD.B      (A0)+, D0          ; add it in
     SUBQ.B     #1, D1             ; decrement loop counter
     BNE        LOOP               ; loop if non-zero
     ...
TAB  DS.B       16
```

Then `D0` ends with the sum. The important line is

```
     ADD.B      (A0)+, D0
```

The source is ARIwPI mode: the effect is to retrieve from memory the value pointed to by `A0`, and then increment the value of `A0` by 1. Thus `A0` is ready pointing at the next element in the table.

Exercise: find and fix the bug in the above program.

The variants `.W` and `.L` increment by 2 and 4 respectively, so we are always pointing at the correct address for the next data object.

If `A0` contains 00001234, and memory location 001234 contains 23232323, then after

```
     ADDI.L    #1, (A0)+
```

the register `A0` contains 00001238, and memory location 001234 contains 23232324.

6.5 Address Register Indirect with Predecrement

This is similar to ARIwPI mode, but instead the address register is decremented (by 1, 2, or 4 as appropriate) *before* the operation. This is usually what is required when traversing memory in the other direction.

If `A0` contains 00001238, and memory location 001234 contains 23232323, then after

```
     ADDI.L    #1, -(A0)
```

the register `A0` contains 00001234, and memory location 001234 contains 23232324.

These modes, ARIwPI and ARIwPD, are useful for maintaining stacks: if you have a stack pointed to by `A3`, use

```
     MOVE.L    D0, -(A3)
```

to push the value of D0 onto the (downwards growing) stack; and

```
    MOVE.L    (A3)+, D0
```

to pop the top value off the stack into D0.

6.6 Address Register Indirect with Displacement

This is an alternate way of indexing into memory that is not restricted to sequential access.

Suppose we have a table of 128 bytes

```
TAB  DS.B     128
```

then we can access the 7th byte in the table by

```
    MOVEA.L  #TAB, A0
    MOVE.B   7(A0), D1
```

which will put the byte in address $(\mathtt{A0}) + 7$ into D1. More generally, we have $d_{16}(\mathtt{An})$, meaning:

1. get the value of An
2. add in the number d_{16}
3. access the data at this resulting address

The contents of An remain unchanged.

> Aside: the syntax $d_{16}(\mathtt{An})$ can vary. Some assemblers require $(d_{16}, \mathtt{An})$.

The *displacement* is a 16 bit signed value, so it can vary from -32768 to 32767. This is a limitation on the size of arrays we can use with this method. Also the displacement d_{16} is a literal constant, i.e., is a number written into the code. For a generalised version of this instruction, we have address register indirect with index, below.

Note that the displacement is in bytes, regardless of the type of operand

```
    MOVE.W   3(A0), D1
```

would fail with an address exception (assuming the value in A0 is even).

Example. Suppose TAB contains the byte values

```
TAB  DC.B      0,1,4,9,16,25
```

Then

```
     MOVEA.L   #TAB, A0
     MOVE.B    3(A0), D1
     MOVE.W    2(A0), D2
```

finishes with 9 in D1 and hex 0409 in D2.

Example. If, now, we have a table of *words*

```
TAB  DC.W      0,1,4,9,16,25
```

Then we want

```
     MOVEA.L   #TAB, A0
     MOVE.B    6(A0), D1
     MOVE.W    4(A0), D2
```

as table entry 3 is 6 bytes offset and table entry 2 is 4 bytes offset.

6.7 Address Register Indirect with Index

What can we do if we want more than 16 bits of range, or a non-constant offset (i.e., computed in the program)? We have *address register indirect with index*. This has syntax d_8(An, Xn).

> Aside: or (d_8, An, Xn).

Here d_8 is an 8 bit signed displacement, and Xn is either an address or a data register. The (signed) value of Xn is added into the value of An, and the signed 8 bit displacement is added into the result. This gives the target address. Using a register to offset the address allows us a full 32 bit displacement.

With TAB as above, we find

```
     MOVEA.L   #TAB, A0
     MOVE.B    2(A0), D1
     MOVE.B    1(A0, D1.L), D2
```

loads D2 with decimal 25. We can use D1.W or D1.B, too, with the expected meanings.

We would use ARIwD for constant (small) offsets known at assemble/compile time, e.g.,

```
   x = table[7];
```

and we would use ARIwI for non-constant offsets *not* known at assemble time, e.g.,

```
   x = table[i];
```

Later we shall discuss this in the context of the semantic gap.

6.8 Program Counter with Displacement

This is the same as ARIwD, except that it uses the program counter rather than an address register. Syntax is d_{16}(`PC`).

> Aside: or (d_{16},`PC`).

Thus we have a 16 bit signed displacement off the PC. This can be used in the jump instructions `JMP` and `JSR`.

```
JMP    44(PC)
```

which is effectively the same as

```
BRA    #44
```

This mode can also be used to implement relocatable data (position independent code).

6.9 Program Counter with Index

This is a variation of PCwD, syntax d_8(`PC,Xn`). Again, `Xn` is an address or data register, a signed 32 bit value, and d_8 is a signed 8 bit value.

Again for PIC.

6.10 Absolute Short and Absolute Long Address

This is the mode used to directly address memory, and it comes in two flavours. To load a byte from address hex ABCD, use

```
MOVE.B    $ABCD, D1
```

This is actually of mode *absolute short*. Because the address fits into two bytes, the `MOVE.B` instruction is encoded in 4 bytes: 2 for the instruction plus 2 for the address. For a longer address

```
MOVE.B    $ABCDEF, D1
```

the mode *absolute long* is used, where 6 bytes are needed: 2 for the instruction plus 4 for the address.

Normally you do not need to think about short or long modes as it is clear from the context, and the assembler copes. However, there is a wrinkle in the absolute short mode: absolute short addresses are sign extended to 24 bits. Thus the absolute short address range of 0000 to FFFF actually refers to two 32K blocks, namely 000000–007FFF and FF8000–FFFFFF.

AS addresses in the range 0000–7FFF map directly to full addresses 000000–007FFF, while AS addresses in the range 8000–FFFF map to full addresses FF8000–FFFFFF. So the 16 bit words with values 0000–7FFF have top bit zero, and this is extended to 24 bits as 000000–007FFF. On the other hand, the 16 bit words with values 8000–FFFF have top bit one, and this is extended to 24 bits as FF8000–FFFFFF.

6.11 Immediate Data

This we have seen before. Rather than referring to the contents of a register or a piece of memory, the data is explicit. Such *immediate* data is recognised by a # mark.

```
MOVE.L   #$8000, D1
```

stored the value hex 8000 in `D1`. Contrast this with

```
MOVE.L   $8000, D1
```

which stores the value *in memory location 8000* into `D1`.

Generally, immediate data cannot be a destination argument of an instruction:

```
ADD.L   D1, #$8000
```

fails to assemble.

Quick immediate is really a special case of immediate. As mentioned before, using a quick mode results in shorter and usually faster code. The only limitation is the small range of possible values.

```
ADDQ.B    n, D1
```

and `n` can be 1–8;

```
MOVEQ.B   n, D1
```

and can be 0–255. The range allowed depends on how much free space there is in the instruction that can be used to specify the value. The `ADDQ` has just 3 bits to spare, while `MOVEQ` has 8 bits.

Byte values are usually sign extended

```
MOVEQ.L   #$23, D1
MOVEQ.L   #$8F, D2
```

results in `D1` having 00000023, and `D2` having FFFFFF8F.

6.12 Implied Addressing

The last mode is one where there is no explicit address or data at all! In the *implied addressing* mode, the source or destination is actually implied by the instruction. An example is `RTS`, which operates on register `A7`.

6.13 Summary of Addressing modes

This wraps up the different kinds of effective address that are available. There are a couple of instructions that deal directly with the computation of effective addresses: `LEA` and `PEA`.

Instruction `LEA` is *load effective address*.

```
LEA    <ea>, An
```

computes the effective address, and stores it in an address register. This instruction allows an ea of: ARI, ARIwD, ARIwI, ASA, ALA, PCwD, PCwI. Other modes either do not make sense, or are actually a `MOVE`.

The `PEA` instruction computes an ea like `LEA`, but then pushes it onto the stack (as pointed by `A7`). You may need this if you are doing advanced subroutines.

In the end, the best thing you can do is look at the documentation! For any particular instruction this is the only way to discover what modes are allowed, and in what combination for source and destination.

7 Miscellaneous Instructions

There are a whole load of other instructions. We shall briefly mention them. They include

1. logical
2. shift and rotate
3. bit manipulation
4. binary coded decimal
5. system control

7.1 Logical

1. `AND` bitwise and
2. `ANDI` and immediate
3. `OR` bitwise or
4. `ORI` or immediate
5. `EOR` exclusive or
6. `EORI` exclusive or immediate
7. `NOT` complement

These come in the expected `BWL` variations, and with the expected semantics. One operand must be a data register, neither operand can be an address register.

Example.

```
MOVEI.L   #$1234, D1
ANDI.B    #$F, D1
```

leaves hex 1204 in D1.

7.2 Shift and Rotate

1. ASL arithmetic shift left
2. ASR arithmetic shift right
3. LSL logical shift left
4. LSR logical shift right
5. ROL rotate left
6. ROR rotate right
7. ROXL rotate left with extend
8. ROXR rotate right with extend

These shift or rotate the bits up (L) or down (R) within a byte, word or longword. A shift of 1 to 8 can be specified as immediate, but any larger and you must use a register.

```
MOVE.L    #$10, D1
ASL.L     #4, D1
MOVE.L    #$10, D2
ASL.B     #4, D2
```

returns hex 100 in D1, but 0 in D2.

The difference between *arithmetic* and *logical* shifts is only apparent for right shifts (ASL and LSL are identical). A logical right shift fills in the top bits with zeros as it goes; an arithmetic right shift copies the old top bit to be the new top bit (thus preserving the sign of a number). We can think of a logical shift as an *unsigned* arithmetic shift.

```
MOVE.L    #$8000, D1
ASR.W     #3, D1
MOVE.L    #$8000, D2
LSR.W     #3, D2
```

ends with D1 containing 0000F000 and D2 containing 00001000. Think of the bit patterns:

8000 hex	1000 0000 0000 0000	becomes	1111 0000 0000 0000	logically
			0001 0000 0000 0000	arithmetically

As an added bonus, the last bit shifted out of the value is saved in the C status flag.

The *rotate* and *rotate with extend* operations do just that: rotate the bits in a value. The extended version treats the X flag as one more bit attached to the end of the value.

```
C <-- x x x x x x x x <-- 0
            ASL

  --> x x x x x x x x --> C
  |___|     ASR

C <-- x x x x x x x x <-- 0
            LSL

0 --> x x x x x x x x --> C
            LSR
      ------------------
      ^                 |
      |                 |
C <-- x x x x x x x x <--
            ROL
    -----------------
   |                 ^
   |                 |
   --> x x x x x x x x --> C
            ROR
      ------------------------
      ^                       |
      |                       |
C <-- x x x x x x x x <-- X <--
            ROXL
    -----------------------
   |                       ^
   |                       |
   --> X --> x x x x x x x x --> C
            ROXR
```

Bit shifts are particularly useful for multiplying and dividing by powers of 2. To multiply `D0` by 4 use

```
    ASL.L    #2, D0
```

This happens quite often when compiling a statment like `table[i]` in C, where `table` is a collection of 4-byte integers. The index `i` needs to be multiplied by 4 to get a byte offset.

Other multiplies can be done, too. Multiplying by 10 can be

```
    MULS.W   #10, D0
```

or

```
    ASL.L    #1, D0          ; multiply by 2
    MOVE.L   D0, D1
    ASL.L    #2, D0          ; and by 4 again = 8 times in total
    ADD.L    D1, D0          ; 2+8 = 10 times
```

which is much faster than the `MUL` (and is a longword multiply, too). A `MULS` can take about 70 cycles, while the above takes about 30 cycles.

7.3 Bit Manipulation

These instructions alter the bits within a value.

1. `BCLR` bit clear
2. `BSET` bit set
3. `BTST` bit test
4. `BCHG` bit change

These operators, like most others, affect the status flags. However, unlike the others, the test is done *before* the bit is updated, and only flag `Z` is affected. They only come in `BL` variants, and immediate operands are restricted to 0–7.

```
BCLR.B   #2, D1
```

this looks at the 2nd (counting from 0 at from the least significant bit) bit of `D1`, sets `Z` if it is zero, clears `Z` otherwise. It then sets the bit to 0.

The others:

- `BSET` tests, then sets the bit to 1
- `BTST` tests, then leaves the bit alone
- `BCHG` tests, then inverts the bit (i.e., 0 to 1, and 1 to 0)

Example

```
MOVE.L    #10, D1
MOVE.L    #$ABCD, D2
BSET.L    D1, D2
```

leaves `Z` at 1, and D2 at AFCD (ABCD is 1010 1011 1100 1101, and AFCD is 1010 1111 1100 1101).

7.4 Binary Coded Decimal

BCD is is a way of encoding integers within a byte that allows easy decoding to decimal. It uses 4 bits per decimal digit 0–9. The values 10-15 are unused. The coding for decimal 49 would be 01001001. Adding 1 into this produces 50, encoded 01010000.

> Aside: the track number on a CD is stored in BCD.

> Aside: this again emphasises the point that the machine only sees patterns of bits, and it is up to us to put an interpretation on them. The bit pattern `01000011` could mean 67 (normal binary), or 43 (in BCD), or the character 'C', or anything else.

1. `ABCD` add decimal with extend (uses `X` bit)
2. `NBCD` negate decimal with extend
3. `SBCD` subtract decimal with extend

These instructions cope with the adjustment of nibbles with values over 9, and always return a BCD number. We can compute a 9's complement of a number using `NBCD` with $\texttt{X} = 1$; or a 10's complement using `NBCD` with $\texttt{X} = 0$.

Only the `B` variant is implemented, i.e., only a single byte BCD can be operated on at a time (i.e., two decimal digits), thus limiting the utility of these instructions. However, the `X` bit allows us to do multiprecision arithmetic.

> Aside: some processors (e.g., 80x86) have BCD *adjust* instructions that take the result of a normal add, subtract or whatever on BCD data and jiggles the bits appropriately to produce the correct BCD value. Most modern processors, however, don't bother supporting BCD as its utility is limited.

7.5 System Control

This is a collection of instructions that perform privileged operations such as altering the status register, and resetting or stopping the processor. Most can only be used when the processor is in the `supervisor state`. The 68000 has two states: the *user state* and the `supervisor state`. The former state is for normal programming, and certain system control instructions are disabled. The latter state is entered under certain conditions, e.g., during an interrupt or exception, when these system control instructions are enabled. The reason is to prevent the normal user from accidentally (or otherwise!) altering essential system state.

Here are just a few:

1. `STOP` stop the processor (priviledged)
2. `RESET` reset the processor (priviledged)
3. `TRAP #n` call trap `n`
4. `ILLEGAL` an illegal instruction; cause an illegal instruction exception
5. `CHK` check register against bounds; if the destination register holds a value that is < 0 or $>$ the value of source, then cause an exception.

Using a privileged instruction while in user state causes a privilege violation exception.

7.6 Floating Point

So far all the operations have been on integer-like data. We have not mentioned any floating point things. This is because the 68000 does not have any! So what do you do if you require floating point on a 68000? There are a few ways you can go.

1. Emulate floating point. For example, the 32 bits in a longword could be regarded as 24 bits of mantissa plus 8 bits of exponent. Build a library of arithmetic operations that perform the required computations. This is quite likely to be very slow in comparison with integer arithmetic, but involves no extra hardware cost.

2. Use a floating point coprocessor. This is an additional piece of hardware that is dedicated to floating point arithmetic, and does nothing else. This is a more expensive option.

The 68881 floating point coprocessor was designed to run alongside the 68000, and tightly integrates with it. New instructions like `FSUB` actually pass data to the coprocessor, wait for the result, retrieve it, and store it in the correct place. The illusion is of an on-chip floating point multiply. The 68881 has eight (`FP0`–`FP7`) 80-bit floating point registers, and provides instructions including trigonometric (sin, cos, tan, and inverses); hyperbolic (sinh, cosh, tanh, and inverses); exponential and logarithm (bases 10, 2 and e); square root; as well as the usual arithmetics.

This is all we are going to say about 68000 assembler for now. The only real way of learning all the variants on all the instructions is to write some programs, and discover for yourself!

8 Parameter conventions

In a HLL calling a function is easy: just write the function name, then arguments, a few brackets in the right places, and it's done. In assembler things are much harder. How do we pass things from one routine to a subroutine? And how do we pass answers back?

Here's the difficulty: supposed we have worked hard on writing a piece of assembler code to raise an integer to an integer power. We would like to reuse that code in several places in a program. Clearly, we need to put this code in a subroutine, label it with a convenient name, say `POW`, so that we can elsewhere do `BSR POW`, and the program leaps to our carefully written routine, computes the power, and presents us with the answer.

We need some sort of convention on how arguments are passed, and values are returned. This is important, as we might want to use somebody else's code, so we will have to use their conventions (which may or may not clash with *our* conventions). Ideally, there would be a world-wide convention for parameters in 68000, so that code would be totally reusable: quite often this is in fact the case. A particular machine running a particular operating system will quite often have a specified convention, and every compiler sticks to it. And every assembly programmer, too, if they want their code to interface cleanly.

Here are three popular ways of passing parameters, though there are many more:

1. in "well known" memory locations (c.f. FORTRAN)

2. in registers

3. on the stack

8.1 A Memory Block

The first way is for the program and subroutine to agree on the address of a particular block of memory, and to write and read arguments and results from this block.

```
        MOVE.L     #42, FNARG1
        MOVE.L     #23, FNARG2
        BSR        FN
        MOVE.L     FNRES, D5
        ...
FN      MOVE.L     FNARG1, D1
        ADD.L      FNARG2, D1
        MOVE.L     D1, FNRES
        RTS
FNARG1 DS.L        1
FNARG2 DS.L        1
FNRES  DS.L        1
```

This leaves decimal 65 in register `D5`. But note

1. the subroutine overwrites a register (`D1`).
2. every function written using this conventions needs an associated block of memory
3. there is no scope for recursive functions

The second point is not really a problem, and we deal with the first point below, but it is the third that really puts most people off this parameter convention. Most people really do want recursive functions.

> Aside: passing parameters in memory blocks was the style used by FORTRAN until really quite recently. Only in the latest versions of FORTRAN can you have recursive functions.

8.2 Saving Registers

The first point is awkward, but can be overcome in a couple of ways.

The first way of overcoming the problem is to save the old value of a register before using it; then restoring the old value when you are done.

```
FN      MOVE.L     D1, FND1
        MOVE.L     FNARG1, D1
        ADD.L      FNARG2, D1
        MOVE.L     D1, FNRES
        MOVE.L     FND1, D1
        RTS
FNARG1 DS.L        1
FNARG2 DS.L        1
FNRES  DS.L        1
FND1   DS.L        1
```

The subroutine `FN` is now *transparent* in its use of registers: there is no net change in any register value. Now `FN` can be used anywhere without fear of values being trashed.

> Aside: though `FN` is not *reentrant*.

An alternative for the subroutine is to use the stack to save the registers:

```
FN      MOVE.L     D1, -(A7)          ; push D1 on to stack
        MOVE.L     FNARG1, D1
        ADD.L      FNARG2, D1
        MOVE.L     D1, FNRES
        MOVE.L     (A7)+, D1          ; pop D1 from stack
        RTS
FNARG1 DS.L        1
FNARG2 DS.L        1
FNRES  DS.L        1
```

> Aside: the stack can be used for any data, not only saving the return address of a subroutine. You must be sure, however, that whenever you come to a `RTS` that the data on the top of the stack is indeed a return address: otherwise chaos will ensue when the processor jumps to wherever the data on the stack is telling it to go! Again, the processor cannot know whether a number is an address or not: addresses are merely bunches of bits, just like any other data.

The second way is for the *calling* function to save those registers it knows it wants to keep safe:

```
        MOVE.L     #42, FNARG1
        MOVE.L     #23, FNARG2
        MOVE.L     D1, -(A7)          ; I want to ensure D1
        MOVE.L     D2, -(A7)          ; and D2 are not overwritten
        BSR        FN
        MOVE.L     (A7)+, D2          ; restore
        MOVE.L     (A7)+, D1          ; them
        MOVE.L     FNRES, D5
        ...
FN      MOVE.L     FNARG1, D1
        ADD.L      FNARG2, D1
        MOVE.L     D1, FNRES
        RTS
FNARG1 DS.L        1
FNARG2 DS.L        1
FNRES  DS.L        1
```

Or, again, you could save to memory.

When the subroutine saves registers, it is called *callee register saving*. When the calling code saves registers, it is called *caller register saving*. Either is fine, but often callee saving is used, as this tends to require less saving and restoring of registers, as you only move those registers you know you need, rather than (with caller save) you have to save all the registers that might possibly be overwritten.

The third way of approaching this problem is to have a convention that there are two classes of registers: temporary and preserved, say `D0` to `D4` are preserved across subroutine call, while `D5` to `D7` are temporary (*scratch*) registers whose values you are happy to have overwritten. Whenever you call a subroutine you must make sure any data you want safe is in the preserved registers. The subroutine will limit itself to using the temporary registers, and if that is not enough, it will save the preserved registers before using them.

As long as all concerned stick to this, you will never have a case of a subroutine trashing a value in a register. This convention is a little difficult on the 68000, since there are only 8 registers. You might want to partition up the temporary and preserved registers differently, but it is always going to be tight. This convention works much better when there are a large number of registers, say 32 or more.

Note that scratch registers are effectively caller saved, while preserved registers are effectively callee saved.

8.3 Registers

Next we look at passing arguments in registers. You decide that, say, D0, D1,... will be used for the first, second,... argument, and that the result will be returned in D0. In some ways this is very much like passing arguments in memory, except that you are using registers rather than memory locations. Things are better, though, as you now do not have the overhead of moving data in and out of (relatively) slow memory. The arguments are already there waiting to be processed. You may need a few register-register moves, but these are fast, and if you are clever, you can arrange the data to end up in the correct register as the result of a previous operation.

The problem of few registers and of recursion remain, though. The typical solution is the same as previously: to use the stack to free up registers. This also solves the recursion problem:

```
     MOVE.L     #6, D0
     BSR        FACT        ; result in D0
     ...
     ;; FACT takes a positive integer
FACT CMPI.L     #1, D0
     BNE        CONT
     RTS                    ; answer is 1
CONT MOVE.L     D1, -(A7)   ; save D1
     MOVE.L     D0, D1
     SUBQ.L     #1, D0
     BSR        FACT        ; (n-1)!
     MULU.W     D1, D0      ; n*(n-1)!
     MOVE.L     (A7)+, D1   ; restore D1
     RTS
```

This computes the factorial of 6 to be 720. It is worthwhile to sit and think of how the values of D0 and D1 change while this is running (and the values saved on the stack).

8.4 Stacking thing up

Given that we are using the stack so much, why not use it to pass arguments, too? We can push arguments on to the stack before calling the subroutine, which can easily access the values, just skipping over the return address also saved on the stack.

```
     MOVE.L     #$12345678, -(A7) ; push arg 1
     MOVE.L     #$9ABCDEF0, -(A7) ; push arg 2
     BSR        ADD2
     ...
ADD2 MOVE.L     4(A7),  D1        ; retrieve arg 2
     MOVE.L     8(A7),  D0        ; retrieve arg 1
```

The second argument is 4 off `A7`, as we have to offset past the subroutine return address.

The stack looks like this:

```
xx   <-- old A7        high address
78       arg 1
56
34
12                     + 8
F0       arg 2
DE
BC
9A                     + 4
ra       return address
ra
ra
ra   <-- current A7    low address
```

So arg 2 starts at `4(A7)`, and arg 1 at `8(A7)`. Suppose that `ADD2` does this

```
ADD2 MOVE.L     4(A7),  D1        ; retrieve arg 2
     MOVE.L     8(A7),  D0        ; retrieve arg 1
     ADD.L      D0, D1
     MOVE.L     D1, 4(A7)         ; put back on stack
     RTS
```

Then the result is returned via the stack. The main routine will be

```
     MOVE.L     #$12345678, -(A7)  ; push arg 1
     MOVE.L     #$9ABCDEF0, -(A7)  ; push arg 2
     BSR        ADD2
     MOVE.L     (A7), D0           ; get result off stack
     ADDA.L     #8, A7             ; adjust stack
```

Aside: or

```
     MOVE.L     (A7)+, D0          ; get result off stack
     ADDA.L     #4, A7             ; adjust stack
```

Once the return address has been removed off the stack the result is at the top. The final `ADDA` is to restore the stack to its original position (dropping the slots where we saved the arguments and result).

The computation of offsets in this system is vital: if we are using callee saving of registers, we might use

```
ADD2 MOVE.L     D0, -(A7)         ; save D0
     MOVE.L     D1, -(A7)         ; save D1
     MOVE.L     12(A7),  D1       ; retrieve arg 2
     MOVE.L     16(A7),  D0       ; retrieve arg 1
     ADD.L      D0, D1
     MOVE.L     D1, 12(A7)        ; put back on stack
```

```
MOVE.L     (A7)+, D1         ; restore D1
MOVE.L     (A7)+, D0         ; restore D0
RTS
```

Aside: we actually need to save only one register, as we can use ARIwD mode in `ADD`. And there is a 68000 instruction for moving multiple registers to memory: `MOVEM`.

The stack now looks like this:

```
xx   <-- old A7        high address
78       arg 1
56
34
12                     + 16
F0       arg 2
DE
BC
9A                     + 12
ra       return address
ra
ra
ra                     + 8
d0       saved D0
d0
d0
d0                     + 4
d1       saved D1
d1
d1
d1   <-- current A7    low address
```

The difficulty is now that we can't really use the stack with impunity as we did before: using the stack will change the offset required to fetch arguments. Note that we are using the stack for several purposes

1. passing arguments
2. saving registers
3. saving the subroutine return address

so we need to be a little careful.

However, this kind of programming is so popular that a specific 68000 instruction exists to assist: the `LINK` operation. The area of the stack containing the various saved registers and other general workspace is called a *stack frame*, and we use a register, the *frame pointer* to refer to it. Note that the frame pointer points into the stack as well as the stack pointer. Having two pointers gives extra flexibilty.

Here is a typical use, where we employ `A6` as the frame pointer:

```
     MOVE.L     #$12345678, -(A7)  ; push arg 1
     MOVE.L     #$9ABCDEF0, -(A7)  ; push arg 2
     BSR        ADD2
     MOVE.L     -12(A7), D0        ; get answer
     ADDA.L     #8, A7             ; pop args
     ...
ADD2 LINK       A6, #-4            ; allocate a stack frame of 4 bytes
     MOVE.L     D0, -(A7)          ; save D0
     MOVE.L     D1, -(A7)          ; save D1
     MOVE.L     8(A6), D0          ; get arg 2
     MOVE.L     12(A6), D1         ; get arg 1
     ADD.L      D1, D0
     MOVE.L     D0, -4(A6)         ; store answer
     MOVE.L     (A7)+, D1          ; restore D1
     MOVE.L     (A7)+, D0          ; restore D0
     UNLK       A6                 ; deallocate frame
     RTS
```

The `LINK` does this

1. push the current value of `A6` on to the stack
2. copies the current SP (`A7`) into `A6`
3. adds the signed displacement into the SP

The stack:

```
xx   <-- old A7         high address
78       arg 1
56
34
12                      12(A6)
F0       arg 2
DE
BC
9A                      8(A6)
ra       return address
ra
ra
ra
a6       old A6
a6
a6
a6   <-- current A6      frame pointer
xx       space for result
xx
xx
xx                       -4(A6)
d0       saved D0
d0
```

```
    d0
    d0                          subroutine stack frame
    d1       saved D1
    d1
    d1
    d1   <-- current A7         low address
```

Doing it like this allows us to separate the workspace used by the subroutine from the space required to pass arguments (at the expense of using an extra register). We can modify the SP without having to worry about offsets of the arguments changing as the arguments are always the same offset from `A6` (but not always the same offset from the stack pointer).

The `UNLK` unwraps all of the above:

1. copies `A6` into the SP
2. pops the longword off the stack into `A6`

This leaves the SP pointing at the return address, ready for the `RTS`, and restores `A6` to its previous value.

After the `RTS`, the saved value is -12 off the SP: 4 bytes over the return address, 4 bytes over the saved copy of `A6`, then 4 bytes to the start of the data.

All you need to remember is: within the subroutine, arg n is $4(n + 1)$ off the FP (`A6`), and the result is to be stored -4 off FP. When back in the caller, the result is -12 off SP (`A7`). And always remember to pop the args off the stack upon return.

An alternative to the callee creating the frame is to have it done by the caller.

```
     LINK      A6, #-12                ; allocate frame (*)
     MOVE.L    #$12345678, -8(A6)      ; store arg 1 in frame
     MOVE.L    #$9ABCDEF0, -12(A6)     ; store arg 2 in frame
     BSR       ADD2
     MOVE.L    -4(A6), D0              ; get answer
     UNLK      A6                      ; deallocate frame
     ...
ADD2 MOVE.L    D0, -(A7)               ; save D0
     MOVE.L    D1, -(A7)               ; save D1
     MOVE.L    -8(A6), D0              ; get arg 1
     MOVE.L    -12(A6), D1             ; get arg 2
     ADD.L     D1, D0
     MOVE.L    D0, -4(A6)              ; store answer
     MOVE.L    (A7)+, D1               ; restore D1
     MOVE.L    (A7)+, D0               ; restore D0
     RTS
```

Memory layout:

```
    xx   <-- old A7          high address
    a6       old A6
    a6
    a6
```

```
a6  <-- current A6       frame pointer
xx       space for result
xx
xx
xx
78       arg 1
56
34
12
F0       arg 2
DE
BC
9A   <-- A7 after (*)
ra       return address
ra
ra
ra
d0       saved D0
d0
d0
d0
d1       saved D1
d1
d1
d1   <-- current A7      low address
```

This is neater than callee linking: the arguments and result areas are kept together, and offsets off `A6` are the same inside and outside the subroutine. The function result is at `-4(A6)` always, and argument n is at $-4(n+1)$(`A6`) always; in the `LINK` you allocate space for arguments plus results.

NB. In the above we have assumed all arguments are longword sized. You can have word and byte arguments and results, in which case you need to keep track of offsets (and alignments).

Aside: some processors have a *branch and link* instruction.

Factorial with callee linking:

```
     MOVE.L     #6, -(A7)               ; push arg
     BSR        FACT
     MOVE.L     -12(A7), D0             ; get result
     ADDA.L     #4, A7                  ; pop arg
     ...
FACT LINK       A6, #-4                 ; frame
     CMPI.L     #1, 8(A6)
     BNE        CONT
     MOVE.L     #1, -4(A6)              ; save result
     UNLK       A6
     RTS
CONT MOVE.L     D0, -(A7)               ; save D0
     MOVE.L     D1, -(A7)               ; save D1
```

```
        MOVE.L   8(A6), D0           ; get n
        SUBQ.L   #1, D0
        MOVE.L   D0, -(A7)           ; push n-1
        BSR      FACT
        MOVE.L   -12(A7), D0         ; (n-1)!
        ADDA.L   #4, A7              ; pop arg
        MOVE.L   8(A6), D1           ; get n
        MULU.W   D1, D0              ; n*(n-1)!
        MOVE.L   D0, -4(A6)          ; save result
        MOVE.L   (A7)+, D1           ; restore D1
        MOVE.L   (A7)+, D0           ; restore D0
        UNLK     A6
        RTS
```

Aside: an earlier version of this program saved only one register (`D0`), and did `MULU.W 8(A6), D0`. Can you see why this doesn't work?

Factorial with caller linking:

```
        LINK     A6, #-8        ; allocate frame
        MOVE.L   #6, -8(A6)     ; put argument
        BSR      FACT
        MOVE.L   -4(A6), D0     ; get result
        UNLK     A6             ; deallocate frame
        ...
FACT    CMPI.L   #1, -8(A6)
        BNE      CONT
        MOVE.L   #1, -4(A6)     ; put result
        RTS
CONT    MOVE.L   D0, -(A7)      ; save D0
        MOVE.L   D1, -(A7)      ; save D1
        MOVE.L   -8(A6), D0     ; get argument
        MOVE.L   D0, D1
        SUBQ.L   #1, D0
        LINK     A6, #-8        ; allocate frame
        MOVE.L   D0, -8(A6)     ; put argument
        BSR      FACT           ; (n-1)!
        MOVE.L   -4(A6), D0     ; get result
        UNLK     A6             ; deallocate frame
        MULU.W   D1, D0         ; n*(n-1)!
        MOVE.L   D0, -4(A6)     ; put result
        MOVE.L   (A7)+, D1      ; restore D1
        MOVE.L   (A7)+, D0      ; restore D0
        RTS
```

8.5 More variants

There are many other ways of passing arguments. Here is a selection.

8.5.1 Address Registers

When using data blocks, pass the addresses of the arguments in registers.

```
       MOVE.L     #1, FNARG1
       MOVE.L     #2, FNARG2
       MOVEA.L    #FNARG1, A1        ; address of arg1
       MOVEA.L    #FNARG2, A2        ; address of arg2
       BSR        ADD2
       ...
       MOVEA.L    #FNARG3, A1        ; can use different locations
       MOVEA.L    #FNARG4, A2        ; in different calls
       BSR
       ...
ADD2   MOVE.L     (A1), D1
       ADD.L      (A2), D1
       ...
FNARG1 DS.L       1
FNARG2 DS.L       1
       ...
FNARG3 DS.L       1
FNARG4 DS.L       1
```

This is slightly different from pre-agreed memory blocks, as each call of the subroutine can refer to different part of memory. Similarly, we can pass the address of a memory location indicating where to store the result. Limitations of this method include the scarcity of address registers.

8.5.2 Address registers: own parameters

Here we allocate the parameter memory directly following the subroutine call. Thus we are implicitly passing the addresses of memory blocks, as above.

```
       MOVE.L     #1, FNARG1
       MOVE.L     #2, FNARG2
       BSR        ADD2
FNARG1 DS.L       1
FNARG2 DS.L       1
FNRES  DS.L       1
       MOVE.L     ...                ; continue here
       ...
ADD2   MOVEA.L    (A7), A5
       MOVE.L     (A5)+, D1          ; FNARG1
       ADD.L      (A5)+, D1          ; FNARG2
       MOVE.L     D1, (A5)+          ; FNRES
       MOVEA.L    A5, (A7)           ; update return address on SP
       RTS
```

The fiddling with `A7` means it now points to the instruction beyond `FNRES`, thus the `RTS` will jump back to the right place to continue.

Note that we can store registers on the stack as before (after the first `MOVEA`), and that this doesn't interfere with the argument passing.

This allows us a limited kind of subroutine recursion: as long as we call a particular subroutine *from different part of the textual program*. The factorial program is not like this, though. Also, this requires that the program area of memory is writable: this is not so, especially if you have your program in, say, ROM.

8.5.3 Argument registers: own parameters addresses

As above, but pass the *addresses* of the memory locations in a block following the `BSR`.

8.5.4 Parameters in an array

Here we assume all are parameters are contiguous in memory, and simply pass the address of the start of the block.

```
     MOVE.L     #1, ARGS
     MOVE.L     #2, ARGS+4          ; simple arithmetic OK
     MOVEA.L    #ARGS, A5
     BSR        ADD2
     ...
ADD2 MOVE.L     0(A5), D1
     ADD.L      4(A5), D1
     MOVE.L     D1, 8(A5)
     RTS
     ...
ARGS DS.L       3                   ; space for two args plus result
```

This allows us to pass an large number of parameters: we are not limited by the number of registers, etc. Problems arise with data that are not uniformly sized (e.g., a mixture of bytes and words), as offset calculation is tricky. Also, indexed addressing is much slower than using registers.

8.5.5 Parameters addresses in an array

Rather than having a block of arguments, we pass the address of a block of addresses of the arguments. With this we can scatter the actual arguments at will across memory, and there is no problem with variable sized data.

```
     MOVE.L     #1,    ARG1
     MOVE.L     #2,    ARG2
     MOVEA.L    #ARGS, A5
     BSR        ADD2
     ...
ANS  DS.L       1
ADD2 MOVEA.L    0(A5), A1           ; address of arg 1
     MOVE.L     (A1), D1            ; value of arg 1
     MOVEA.L    4(A5), A1           ; address of arg 2
```

```
     ADD.L      (A1), D1
     MOVEA.L    8(A5), A1         ; address of result
     MOVE.L     D2, (A1)          ; put result
     RTS
ARG1 DS.L       1
ARG2 DS.L       1
ARGS DC.L       ARG1, ARG2, ANS
```

This is very flexible, but problems include the cost of the extra indirection.

8.6 Which is best?

Of these three main parameter conventions which should we use? As always, it really depends on the circumstances. If you are compiling FORTRAN, memory storage is fine: in fact, the language standard once forced the use of this technique. Otherwise, it is probably better to use registers because

1. it is faster: reading and writing from the stack or other memory is slower than using registers
2. there is less messing about

Though, of course, the case for registers is much better when there are lots of registers to play with.

These days, programmers tend to use a mixture of registers and stack. The first few parameters are passed in registers, the rest on the stack (and also arguments of lengths that don't fit into registers). Almost all functions use less than 3 or 4 args, so the stack is used relatively rarely.

Typically, also, there are a few spare registers that can be used as scratch registers. This means we do not have to save and restore them, saving more time and space at the cost of having fewer register to use. If we run out of registers, of course, we can always stack a few values as before.

On the 68000, we can use the address registers for arguments that are addresses of objects, and the data registers for values: as long as we know whether an argument is supposed to be an address or a value this will give us a little more breathing space on registers.

One final note: in the above we have assumed that functions returned a single result. Some functions return *multiple values*, for example a quotient-and-remainder function. It should be clear that there is no problem extending these techniques to allow multiple value returns.

9 Relationships between high and low level

We now turn to higher-level concepts. In a HLL there are many features that have evolved over the years as being "good" programming style. This includes structured `if` statements (rather than unstructured jumps), loops, functions, datastructures, and so on. However, at the assembler level we have none of that: the only control is a branch operator.

The big question is: how does the clever stuff in a HLL relate to the simple stuff in assembler? This is called the *semantic gap*, and is what we are going to address next.

We have seen how functions and arguments in a HLL can be translated in a large number of ways to subroutines, but what of the other high level constructs?

HLL	asm
functions	subroutines
`if` conditionals	?
`for` loops	?
`struct` datastructures	?

9.1 Decisions

In Scheme we have the `if` expression:

```
(if condition expression1 expression1)
```

where the `condition` is evaluated, and if this returns true, the `expression1` is evaluated; otherwise the `expression2` is evaluated.

In C we have

```
if (condition) {
   statements1;
}
else {
   statements2;
}
```

when `statements1` are executed if `condition` is true; otherwise `statements2` are executed.

FORTRAN has

```
IF (condition) THEN
   block1
ELSE
   block2
END IF
```

Clearly, these are the same construct (modulo a few details). Most HLLs have an analogous construct: evaluate a condition, and based on the result, either execute one bunch of stuff or another. And then (and this is sometimes overlooked), we continue executing from directly after the `if` statement.

The `expressions` or `statements` can themselves be complicated collections of further statements, but overall the idea is clear.

Suppose we have the following:

```
if (x > 1) {
  x = x + 1;
}
else {
  x = x - 2;
}
```

In assembler the only conditional operations are branches (`BEQ`, `BGT` etc.), so we must translate this `if` to a branch. One way is

```
     CMPI.L     #1, X               ; X names an address in memory
     BGT        TRU                 ; X > 1
     SUBQ.L     #2, X
     BRA        CONT
TRU  ADDQ.L     #1, X
CONT ...                            ; next thing after if
```

Another way is

```
     CMPI.L     #1, X
     BLE        FAL                 ; X <= 1
     ADDQ.L     #1, X
     BRA        CONT
FAL  SUBQ.L     #2, X
CONT ...
```

by inverting the sense of the test, and swapping the two subsequents. The choice is yours: sometimes it is more convenient one way than the other.

> Aside: it would be faster and better to load `X` into a register and use that, but that would confuse the point.

Things get more fun when we have nested `if`s. However, by following the above pattern, we can cope with anything.

```
if (x > 1) {
  if (x < 10) {
    x = x + 5;
  }
  else {
    x = x + 6;
  }
}
else {
  if (x < 0) {
    x = x - 3;
  }
  else {
    x = x - 4;
  }
}
```

It is easiest to work from inside outwards. The top inner `if`

```
      CMPI.L     #10, X
      BLT        TRU1                    ; X < 10
      ADDQ.L     #6, X
      BRA        CONT1
TRU1  ADDQ.L     #5, X
CONT1 ...
```

and the lower inner `if`

```
      CMPI.L     #0, X
      BLT        TRU2                    ; X < 0
      SUBQ.L     #4, X
      BRA        CONT2
TRU2  SUBQ.L     #3, X
CONT2 ...
```

Notice the generation of labels for the statements: in assembler you will find many more labels as targets of branches since these arise in precisely the above way.

The skeleton of the outer `if`

```
      CMPI.L    #1, X
      BGT       TRU3                    ; X > 1
      something
      BRA       CONT3
TRU3  something
CONT3 ...
```

Just plug in the code from above

```
      CMPI.L     #1, X
      BGT        TRU3                    ; X > 1
      CMPI.L     #10, X
      BLT        TRU1                    ; X < 10
      ADDQ.L     #6, X
      BRA        CONT1
TRU1  ADDQ.L     #5, X
CONT1 BRA        CONT3
TRU3  CMPI.L     #0, X
      BLT        TRU2                    ; X < 0
      SUBQ.L     #4, X
      BRA        CONT2
TRU2  SUBQ.L     #3, X
CONT2 CONT3 ...
```

There are two labels on the continuation: we merge them, and relabel

```
      CMPI.L     #1, X
      BGT        TRU3                    ; X > 1
      CMPI.L     #10, X
```

```
      BLT        TRU1                 ; X < 10
      ADDQ.L     #6, X
      BRA        CONT1
TRU1  ADDQ.L     #5, X
CONT1 BRA        CONT3
TRU3  CMPI.L     #0, X
      BLT        TRU2                 ; X < 0
      SUBQ.L     #4, X
      BRA        CONT3
TRU2  SUBQ.L     #3, X
CONT3 ...
```

This is the code that corresponds to our original C. We can improve this a little by noting that rather than jumping to `CONT1` we could do better by jumping directly to `CONT3`

```
      CMPI.L     #1, X
      BGT        TRU3                 ; X > 1
      CMPI.L     #10, X
      BLT        TRU1                 ; X < 10
      ADDQ.L     #6, X
      BRA        CONT3                ; patch jump
TRU1  ADDQ.L     #5, X
      BRA        CONT3
TRU3  CMPI.L     #0, X
      BLT        TRU2                 ; X < 0
      SUBQ.L     #4, X
      BRA        CONT3
TRU2  SUBQ.L     #3, X
CONT3 ...
```

This action of working on a little bit of code to make local improvements is called *peephole optimisation*.

> Aside: in a final year course on compilation you will see how to do all this properly.

Notice how a clear(ish) piece of C translates to a complicated mess of tests and branches. But this is inevitable, due to the nature of machine code.

> Aside: but then again there are some machines that are different…

This becomes slightly harder if the condition is more complicated

```
if (0 <= x && x <= 10) {
  x = x + 7;
}
```

becomes

```
     CMPI.L     #0, X
     BLT        CONT
     CMPI.L     #10, X
     BGT        CONT
     ADDQ.L     #7, X
CONT ...
```

(if any condition fails, jump to continuation), and

```
if (x <= 0 || x >= 10) {
  x = x + 5;
}
```

becomes

```
    CMPI.L     #0, X
    BLE        OK
    CMPI.L     #10, X
    BLT        CONT
OK  ADDQ.L     #5, X
CONT ...
```

(if any condition succeeds, jump to statement).

In C, any non-zero value is regarded as true, while 0 is false.

```
if (odd(n)) {
  n = 2*n + 1;
}
else {
  n = n/2;
}
```

can be

```
     MOVE.L   N, D0
     BSR      ODD               ; arg and result in D0
     CMPI.L   #0, D0
     BEQ      ELS
     MOVE.L   N, D0
     ASL.L    #1, D0            ; 2*n
     ADDQ.L   #1, D0            ; 2*n+1
     MOVE.L   D0, N
     BRA      CONT
ELS  MOVE.L   N, D0
     ASR.L    #1, D0            ; n/2
     MOVE.L   D0, N
CONT ...
```

The code for `ODD` could be

```
ODD  AND.L    #1, D0
     RTS
```

More ambitious control structures, such as `cond` in Scheme, or `switch` in C are quite similar to the above.

9.2 Loops

Loops are pretty much central to most programs: it is the ability of computers to repeat operations many times that make them so useful. There are a variety of looping structures in C, including `for`, `while`, and `do`, but actually these are all the same:

```
while (condition) {
  statements;
}
```

is the same as

```
for (; condition; ) {
  statements;
}
```

and

```
do {
  statements;
} while (condition);
```

is the same as

```
statements;
for (; !condition; ) {
  statements;
}
```

In essence a loop is

1. perhaps an initialisation
2. a test of some condition
3. the body, executed while some condition is true
4. perhaps some update of state

and other varieties of loop can be transformed back to this. In this form, it is clear how to implement a loop in assembler.

Suppose we have

```
for (i = 61; i < 67; i++) {
  putchar(i);
}
```

We can translate this as

```
      MOVE.B    #61, D2
LOOP  CMPI.B    #67, D2
      BGE       CONT
      MOVE.B    D2, D1
      BSR       PUTCHAR            ; putchar arg in D1
      ADDQ.B    #1, D2
      BRA       LOOP
CONT  ...
```

Often a loop variable never needs an allocation of memory: they can live entirely within a register.

Using the equivalence above, while loops are even easier (since there is no initialisation step).

```
while (x > 0) {
  y = y + x;
  x = x/2;
}
```

becomes

```
LOOP  CMPI.L   #0, X
      BLE      CONT
      MOVE.L   Y, D0
      ADD.L    X, D0
      MOVE.L   D0, Y
      MOVE.L   X, D0
      ASR.L    #1, D0              ; shift right 1 bit
      MOVE.L   D0, X
      BRA      LOOP
CONT  ...
```

A better way of doing this is

```
      MOVE.L   X, D0
      MOVE.L   Y, D1
LOOP  CMPI.L   #0, D0
      BLE      CONT
      ADD.L    D0, D1
      ASR.L    #1, D0              ; shift right 1 bit
      BRA      LOOP
CONT  MOVE.L   D0, X
      MOVE.L   D1, Y
      ...
```

as this involves accessing memory much less. Or even by a little optimisation:

```
      MOVE.L   Y, D1
      MOVE.L   X, D0
LOOP  BLE      CONT
      ADD.L    D0, D1
      ASR.L    #1, D0                ; shift right 1 bit
      BRA      LOOP
CONT  MOVE.L   D0, X
      MOVE.L   D1, Y
      ...
```

Recall that a conditional branch depends on the condition codes, and these are set by `MOVE` and `ASR` (amongst others) and are unaffected by `BRA`.

9.3 Functions

Functions we have more-or-less covered: the real difficulty lies in parameter conventions. Once that is sorted, the rest is easy.

The factorial

```
int fact(int n)
{
  if (n == 1) {
    return 1;
  }
  return n*fact(n-1);
}
```

translates into any of the assembler programs described earlier.

```
int foo(int n)
{
  if (n == 1) {
    return(1);
  }
  if (odd(n)) {
    return foo(2*n+1);
  }
  return foo(n/2);
}
```

```
FOO   CMPI.L   #1, D0
      BNE      L1
      RTS
L1    MOVE.L   D0, -(A7)
      BSR      ODD
      BEQ      L2
```

```
      MOVE.L   (A7)+, D0
      ASR.L    #1, D0
      BSR      FOO
      RTS
L2    MOVE.L   (A7)+, D0
      ASL.L    #1, D0
      ADDQ.L   #1, D0
      BSR      FOO
      RTS
```

Aside: we could do

```
FOO   CMPI.L   #1, D0
      BNE      L1
      RTS
L1    MOVE.L   D0, -(A7)
      BSR      ODD
      BEQ      L2
      MOVE.L   (A7)+, D0
      ASR.L    #1, D0
      BRA      FOO
L2    MOVE.L   (A7)+, D0
      ASL.L    #1, D0
      ADDQ.L   #1, D0
      BRA      FOO
```

This is *tail recursion*: we can convert a `BSR` just before a `RTS` into a jump (as long as the function arguments are correctly placed). This is faster: no pushing of return addresses on to the stack, and no unwinding down the stack when done. And because no stack is being used, we can recurse as deep as we like.

9.4 Data Structures

We will now cover fairly briefly how HLL data structures can be represented at the assembler level.

9.4.1 Arrays/Vectors

Scheme and C both have concepts of vectors: a sequence of similar objects somewhere in memory that can be accessed quickly by an index.

```
(let ((table (make-vector 128)))    int table[128];     INTEGER TABLE(128)
...                                 ...                 ...
   (vector-set table 4 7)           table[4] = 7;       TABLE(4) = 7
```

declares a table of 128 integers (512 bytes). The assembler equivalent of this is easy

```
TABLE DS.L    128
...
      MOVE.L  #7, TABLE+16
```

since a constant index into the table (of 16 bytes) can be precomputed. If we do not have a constant index

```
  table[i] = n;
```

we use ARIwI mode

```
      MOVEA.L #TABLE, A0
      MOVE.L  I, D0
      ASL.L   #2, D0            ; convert to byte offset
      MOVE.L  N, 0(A0, D0.L)
```

The only subtlety with vectors is to ensure you have the correct offset for the size of datum you are operating on.

9.4.2 C structures

A standard C structure can look like this

```
struct stuff {
  int a;
  char b;
  int c;
  short d;
};

struct stuff eg;
```

which declares `eg` to refer to an area of memory big enough to hold two integers (4 bytes each), a character (1 byte), and a short (2 bytes).

Aside: these are typical sizes, and are by no means the only possible.

FORTRAN has

```
TYPE STUFF
  INTEGER A
  ...
END TYPE STUFF

TYPE(STUFF) EG
```

Thus `eg` will occupy an area of memory of at least 11 bytes. The C standard requires that the components `a`, `b`, `c` and `d` are laid out in memory, *in that order*. That is, the address of the location of the `a` component *must be* less than the address of the location of the `b` component, which must be less than the address of the location of the `c` component, and so on.

Memory could look like this:

```
dd                high address
dd       9
cc
cc
cc
cc       5
bb       4
aa
aa
aa
aa       0        low address
```

but more probably looks like this

```
dd                high address
dd       12
cc
cc
cc
cc        8
xx      pad
xx      pad
xx      pad
bb        4
aa
aa
aa
aa        0       low address
```

The reason is alignment: in most machines integers must lie on 4 byte boundaries, shorts must lie on 2 byte boundaries. Thus the need for unused pad bytes between `b` and `c`. This structure will occupy 14 bytes. Furthermore, there may well be some padding at the end to ensure the entire struct is a multiple of 4 bytes long, thus ensuring a vector of these structs will always have the `int`s well aligned.

```
xx      pad       high address
xx      pad
dd
dd       12
cc
cc
cc
cc        8
xx      pad
xx      pad
xx      pad
bb        4
aa
aa
aa
aa        0       low address
```

This is 16 bytes long.

We can use the C operator `sizeof` to determine the actual size of a structure in bytes.

```
#include <stdio.h>

struct stuff {
  int a;
  char b;
  int c;
  short d;
};

main()
{
  printf("size %d bytes\n", sizeof(struct stuff));
}
```

Generally, this inflation is no real hardship. If you are worried about memory usage, you can rearrange the components of the struct (as long as your program does not rely on their specific order). For most purposes

```
struct stuff {
  int a;
  int c;
  short d;
  char b;
};
```

will be fine, and this requires no internal padding. It would probably have a single byte of padding at the end.

Typical alignments for 32 bit and 64 bit machines are like this:

32 bit machine (ILP32 model)		
type	size	alignment
`char`	1	1
`short`	2	2
`int`	4	4
`long int`	4	4
`long long int`	8	8
`float`	4	4
`double`	8	8
`long double`	16	8 or 16
`void*`	4	4

64 bit machine (LP64 model)		
type	size	alignment
`char`	1	1
`short`	2	2
`int`	4	4
`long int`	8	8
`long long int`	8	8
`float`	4	4
`double`	8	8
`long double`	16	8 or 16
`void*`	8	8

Size and alignment in bytes. Some 32 bit machines have a `long long` integer type that is 8 bytes, and a `long double` floating-point type that is 16 bytes.

The size of a `long` on a 64 bit machine is matter for debate: some have 4 bytes for backward compatiblity with 32 bit programs, other have 8 bytes. Possibitities include:

	LP64	ILP64	LLP64	ILP32	LP32
`char`	1	1	1	1	1
`short`	2	2	2	2	2
`int`	4	8	4	4	2
`long int`	8	8	4	4	4
`long long int`	8	8	8	8	
`void*`	8	8	8	4	4

The LP64 model seems to be gaining popularity for 64 bit machines. The LP32 model was used in the 16-bit Windows API, and the Apple Macintosh.

> Aside: note that this means on a 64 bit machine we cannot cast a pointer to an integer without losing bits. Many bad programs do this: such programs will not port easily to a 64 bit architecture. Binaries on Alphas are often designed to dump core if you try this kind of trick: you can arrange for the entire address range of 0000000000000000 to 00000000FFFFFFFF to be inaccessible to user programs. This may sound weird as it cuts off 4 Gbytes of address space: that's the entire addressable space of a 32 bit machine. But it's only about one trillionth of the available address space in a 64 bit machine.

In C, we access a component by name

```
  n = eg.c;
  m = eg.d;
```

FORTRAN uses `N = EG%C`.

In assembler there are no names: this translates to (using the original definition for `stuff`)

```
     MOVE.L    EG+8, N
     MOVE.W    EG+12, M
```

The conversion is not difficult: references to components by name directly become constant offsets from the start of the structure, and these are computed at assembly time.

More often, we have pointers to structures:

```
struct stuff *egp;
...
  n = egp->c;
  m = egp->d;
```

This time we do not know the address of the structure at assembly time, and the code becomes

```
     MOVEA.L   EGP, A0        ; EGP contains the address of the structure
     MOVE.L    8(A0), N
     MOVE.W    12(A0), M
```

using ARIwD mode.

A memory map

```
  xx                  high address
  xx
  dd
  dd         12
  cc
  cc
  cc
  cc         8
  xx
  xx
  xx
  bb         4
  aa
  aa
  aa
  aa  <--    0
  xx    |
  xx    |
  ..    |
  ..    |
  xx    |
  xx    |
  ss    |
  ss    |
  ss    |
  ss  ---    EGP
  xx
  xx                  low address
```

9.4.3 Lisp lists

Structures in C are of a fixed size, while lists in Scheme can grow and shrink dynamically. Ignoring the memory allocation/deallocation problem, we can sketch an outline of how lists can be implemented.

We need a more ambitious structure:

```
struct intlist {
  int val;
  struct intlist *next;
};

struct intlist p;
```

Then `p` will refer to (probably) 8 bytes of memory:

```
  nn                  high address
  nn
  nn
  nn
  vv
```

```
  vv
  vv
  vv                 low address
```

where vv refers to the integer value, and nn is the address of the next element of the list.

> Aside: modern processors are gravitating towards 64 bit (8 byte) addresses, with 8 byte alignment, so the intlist structure could be 4+4 padding+8 = 16 bytes long. By reversing the components we might save 4 bytes; but then again, there would probably be padding at the end to ensure the whole structure is a mutiple of 8 bytes long.

The car accessor

```
int car(struct intlist *p)
{
  return p->val;
}
```

or (argument in A0 and result in D0)

```
CAR   MOVE.L   (A0), D0
      RTS
```

The cdr accessor

```
struct intlist *car(struct intlist *p)
{
  return p->next;
}
```

or (argument and result in A0)

```
CDR   MOVEA.L   4(A0), A0
      RTS
```

In fact, as car and cdr are so simple, we would probably not bother calling subroutines to do them, but simply code them *in line*, which would be faster and more compact (but harder to find and change if we decide to change the representation of a list).

We will not talk about cons as it requires dynamic memory allocation, which is too complex for now.

10 More Modern Processors

The 68000 was one of the early leaders in processor design. With several 32 bit registers and complex addressing modes it was to be a useful stepping stone in processor design. Modern processors have much in their design in common with the 68000, but also much that is fundamentally different. To see why this came about, we have to look at the development of HLLs.

10.1 HLLs and the Semantic Gap

As HLLs developed over the years, certain ideas became prominent and were decreed to be the correct approach to programming to aid comprehensibility, maintainability and so on. These included

1. use of high level control structures (`if`, etc.)
2. use of many small functions
3. use of data structures

HLLs have gained constructs to aid the programmer to do complicated operations in a straightforward manner: this has tended to widen the *semantic gap* between HLL and machine code.

To address this problem, designers of processors have looked at HLLs and determined what operations at the machine level are appropriate to assist the efficient implementation of these complicated constructs. Essentially, this means processor support in addressing modes to support vectors and structures; opcodes to support efficient call and return from functions; efficient ways of computing decisions and branching on the results.

One approach to this is to make the individual machine instructions more complex: this is the CISC (*complex instruction set computer*) approach. We try to make the processor more like the HLL, thus closing the semantic gap. The CISC idea was taken up by the 68000, VAX, and most notably by Intel with the 80n86 series of processors (including 386, 486, Pentium and beyond).

10.2 CISC and RISC

This idea ran for quite a while, but then the engineers found it more and more difficult to design better and faster processors. The problem is that complex machine instructions take (a) a lot of time to execute, and (b) a lot of space physically to implement in the processor. Thus they are expensive in time to use, and expensive in money to build. What's more, compiler found these complicated instructions quite hard to use as they were never quite flexible enough for what C or FORTRAN or whatever wanted. For example, there is an instruction in the 68030 specifically to manipulate doubly linked lists. However, it is rare that a compiler is clever enough to realise that the particular piece of C (or other HLL) is is considering is actually a doubly linked list (think of all the different ways you could rearrange a piece of C code...). Thus the clever instruction rarely gets generated. Quite often the compiler writer found it more space and time efficient to use the basic instructions to achieve what they needed, ignoring the complicated instructions completely.

Designers went back and studied what *actually* happened in programs in terms of

1. the operations performed
2. the operands they used
3. in what order they happened

They collected both *static* and *dynamic* statistics. The static count you get by looking at the program text; the dynamic count by running the program several times and counting the number of times each thing happened. The results can be quite different as, say, an instruction within a loop is counted once statically, but many times (once per execution) dynamically.

Aside: a famous study by Knuth in the 60's (?) estimated that you go round a loops 8 times on average.

The results are very interesting:

1. 66% of statements are assignments, of which a third are simple `a = b`. Thus most execution is actually data movement.
2. There are a large number of conditional branches (e.g., decisions and loops): the mean distance between branches is only a few instructions.
3. The call/return mechanism for functions generate the largest number of machine instructions per HLL statement. So you spend much of your time in function call overhead, not actually working on the program (maybe 25%)
4. Most operands are scalars (simple values, not vectors, etc.), and most are local to functions.
5. Operands of functions are referred to very frequently.
6. 98% of functions take 6 or less parameters; 92% use 6 or less local variables. Rarely is there more than 6 calls to a subroutine in a row; or more than 6 returns in a row.

Thus to optimise the running of a program we need to concentrate some some specific things. We need fast

1. data movement
2. conditional branches
3. call/return
4. access to local variables
5. access to operands
6. access to nearby stack

Building on this information, an alternative strategy to CISC arose: design processors that are optimised for a small number of operations (those above), and ignore the complicated stuff. Dedicating processor area to making some of the above go faster will have greater overall effect than using it to implement a complicated instruction that is rarely used. Complicated operations can be composed from a sequence of the simple ones, so that they too will benefit from the simple instructions' optimisation.

For example, rather than have a indirect with increment (`(A0)+`), replace this with two instructions, one for the indirect and one for the increment. For example, replace `MOVE.L D0, (A0)+` by `MOVE.L D0, (A0)` and `ADDQ.L #1, A0`. This may seem to be worse as it is longer, but may actually run faster.

Now making each of these simpler things faster (which is easier to do than speeding up a complex instruction) will make the whole thing faster. Furthermore, a side effect that (the expanded version of) `-(A0)` will *also* benefit by dint of making the indirect faster.

This is the RISC (*reduced instruction set computer*) approach, and is typified by

1. a simple instruction set
2. simple and few instruction formats

3. a large number of registers
4. only load and store refer to memory, everything else is register-register

Note that it is the *instructions* that are reduced, not the *set*. There can be a large number of instructions, but each tends to be simple or have a limited number of variants.

As the instructions are simpler they can be optimised in hardware more easily, and there are less difficulties of having faster and faster cycle times.

RISC and CISC have grown side-by-side, and ideas have flowed from RISC to CISC: the modern Pentiums, though CISC in heritage, have taken up many RISC ideas.

> Aside: CISC processors include 68000 family, 80n86 family; RISC includes SPARC, MIPS, Alpha. The Pentium Pro and Pentium II are CISC on the outside, but are based on a RISC core. The classification of the next generation processor from Intel (Merced) is moot, but will certainly incorporate RISC philosophy.

10.3 Instruction decoding and pipelines

A big feature of a RISC instruction set is the simplicity of the machine instruction set. The 68000 instruction set contains many formats

```
         15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
         ---------------------------------------------------------------
ADD      |1   1   0   1 |  register |   opmode  |   effective address  |
         ---------------------------------------------------------------

         15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
         ---------------------------------------------------------------
AND      |1   1   0   0 |  register |   opmode  |   effective address  |
         ---------------------------------------------------------------

         15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
         ---------------------------------------------------------------
ADDI     |0   0   0   0   0   1   1   0 |  size |   effective address  |
         ---------------------------------------------------------------
         |           word data           |         byte data           |
         ---------------------------------------------------------------
         |                         longword data                       |
         ---------------------------------------------------------------

         15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
         ---------------------------------------------------------------
ANDI     |0   0   0   0   0   0   1   0 |  size |   effective address  |
         ---------------------------------------------------------------
         |           word data           |         byte data           |
         ---------------------------------------------------------------
         |                         longword data                       |
         ---------------------------------------------------------------
```

```
         15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
         ---------------------------------------------------------------
ADDQ     |0   1   0   1  |  data    | 0 |  size  |  effective address  |
         ---------------------------------------------------------------

         15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
         ---------------------------------------------------------------
ASL/R    |1   1   1   0  | count or |L/R|  size |I/R| 0   0 | register |
         |               | register |   |       |   |       |          |
         ---------------------------------------------------------------
                                                  immediate/
                                                   register

         15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
         ---------------------------------------------------------------
Bcc      |0   1   1   0  |  condition   |       8 bit displacement     |
         ---------------------------------------------------------------
         |       16 bit displacement if 8 bit displacement = 0         |
         ---------------------------------------------------------------

         15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
         ---------------------------------------------------------------
BRA      |0   1   1   0   0   0   0   0 |       8 bit displacement     |
         ---------------------------------------------------------------
         |       16 bit displacement if 8 bit displacement = 0         |
         ---------------------------------------------------------------
```

and so on. The first four bits of the machine instruction give you some idea of what the instruction means; but sometimes you need more. In some instructions the meaning of one field depends on the value of another (e.g., `ASL/R`). Some instructions have variable length.

Decoding an instruction can take a long time: and sometimes a variable amount of time. After that, the execution of the instruction takes a variable amount of time, giving a total from 4 cycles for a simple register-to-register add, to nearly 160 cycles for a signed divide from an indexed memory location on a 68000.

This is all very difficult to enhance. The throughput of a processor in terms of instructions depends very much on the instructions, and irregularities in accesses to memory prevent the effective use of hardware acceleration.

> Aside: you sometimes hear of processors *retiring instructions*. This is when the execution of an instruction is finished.

Another step taken by RISC is to ensure:

1. all instructions are the same size, with very few variations on format
2. all instructions take the same amount of time to execute (more or less)

The first allows us to decode an instruction much faster: we don't have to make decisions and sub-decisions as we go.

The MIPS (Microprocessor without Interlocked Pipeline Stages) processor has only 3 different instruction formats:

```
          |31      26|25      21|20      16|15      11|10      6|5      0|
          |----------|----------|----------|----------------------------|
I-type    |    op    |    rs    |    rt    |         immediate          |
          |---------------------------------------------------------------|
J-type    |    op    |                     target                       |
          |---------------------------------------------------------------|
R-type    |    op    |    rs    |    rt    |    rd    |  shift  | funct  |
          ---------------------------------------------------------------
```

The I-type (*immediate*), has `rs` source register, `rt` target register, and a 16 bit immediate data field.

The J-type (*jump*) has a 26 bit immediate address.

The R-type (*register*) has source, target, and destination registers; and shift and function fields, which are used by certain instructions.

Such uniformity of formats means there is little work to be done to decode an instruction. This in itself saves much time. Thus we can get `rs` and `rt` even before we know what the operation is: we can decode in parallel. Just throw the results away if it turns out to be a J-type instruction. Compare this with the 68000, where we have to determine the operation *before* we can determine registers: this is sequential and inherently slower.

Secondly, because instructions take roughly equal times to retire, MIPS can use *pipelining*. Consider the action of a 68000: to complete an instruction it must

1. fetch the instruction from memory
2. decode it
3. fetch arguments from memory if appropriate
4. execute it
5. save the result to memory if appropriate

So repeatedly, it has to

```
FI DI FA EX SR FI DI FA EX SR FI DI FA EX SR FI DI FA EX SR ...
             5             10             15             20
```

We have simplified immensely, and are assuming each stage takes one cycle to do. This is certainly not true in the 68000.

RISC observes that, if the instruction stream is nice and simple, we can overlap these operations:

```
FI DI FA EX SR
   FI DI FA EX SR
      FI DI FA EX SR
         FI DI FA EX SR
            ...
 1  2  3  4  5  6  7  8
```

That is, while we are decoding the first instruction we can be simultaneously fetching the next. And while we are fetching the arguments for the first we can decode the second, and fetch the third. And so on. Thus we can (theoretically, for this example), complete four instructions in 8 cycles, rather than the 20 above. An even better: after the *pipeline* is full, we retire an instruction every cycle, not every 5.

> Aside: 5 is not a magic figure, it is just a number we get from splitting an operation up into these 5 parts. Other ways can get larger or smaller values.

Thus we have a 5-fold speedup. The two processors may cycle at the same rate, but one has a 5 times larger throughput. And the finer the subdivisions you can make the the better the throughput. The cost, of course, is the hardware has to be so much more sophisticated than a non-pipelined system.

However, there are several problems with pipelines.

10.3.1 Jumps

Whenever we get to a jump instruction, we will already have partially done some other instructions, but we don't want those instruction next. Rather we actually want the instructions that are the target of the jump.

One approach is just to throw that work away, *drain the pipeline*, and start again. It will take us 5 cycles to fill the pipeline and get back up to full speed again. This works, but we lose the benefit of the pipeline at every jump: and jumps come quite often.

Another approach is to use *branch delay slots*. Here we *don't* throw away the all partial work, but allow one or two partial instructions to complete *before* doing the jump. With a single delay slot

```
     load   r1, 4
     jmp    lab
     add    r1, r1, 1         ; delay slot
     add    r1, r1, 1
     ...
lab  ...
```

`r1` will have value 5 at `lab`. The processor can start fetching and decoding instructions from the jump destination while the `add` carries on, thus keeping the pipeline filled.

This is confusing to read, but reasonably straightforward for compilers to produce. There are several ways to fill the delay slot.

1. Use `NOP`. Just stuff a `NOP` after each jump and you will be safe.
2. Move the instruction that was immediately before the jump to after it. E.g., function call

   ```
   load r1, N
   load r2, M
   jsr  fun
   nop
   ...
   ```

   becomes

```
     load r1, N
     jsr  fun
     load r2, M
     ...
```

3. If we know the code at the destination we can replace

```
     jmp lab
     nop
     ...
lab  instr1
     instr2
```

by

```
     jmp lab2
     instr1
     ...
lab  instr1
lab2 instr2
```

Similarly for conditional jumps and returns from subroutines. Notice that the return address stored by a jump subroutine instruction will not be be the next instruction, but the next but one instruction.

There are other ways to deal with jump delay slots that we shall look at later.

10.3.2 Instruction conflicts

Another problem with a pipeline is that the overlapping instructions can sometimes interfere with each other.

```
add    r1, r2, r3       ; sets r1
add    r5, r1, r4       ; uses r1
```

The second instruction uses the result from the first, but in the pipeline

```
FI DI FA EX SR
   FI DI FA EX SR
```

the FA of the second is supposed to happen at the same time as the EX of the first. Extra mechanisms are used to *forward* the value of the EX of the first directly into the EX of the second

```
FI DI FA EX SR
           \
   FI DI FA EX SR
```

to keep the pipeline going.

The idea of pipelines is equally applicable to CISC machines, but is very much harder to implement. This is because of the varying sizes for instructions (so the IF takes a variable amount of time); complicated and variable instruction formats (decoding takes a variable amount of time); complicated instruction executions (taking a variable amount of time). Nevertheless, pipelining appears in modern CISC machines (in particular the Pentium).

10.3.3 Load delays

Another feature of many RISC systems is the use of *load delay slots*. This is something else that arises out of the target of one cycle per instruction.

The time delay between a processor asking for a value from memory and the value arriving can be large. For example, the 386 takes 4 cycles for data to arrive. Similarly, it takes 2 cycles to write data (it doesn't have to wait to see the write finish). As processors get faster, memory lags behind. A processor can be running at 500MHz, while the memory bus might be running at 100MHz. We will talk about hardware fixes (cache) later, but there is a big disparity between processor and memory speeds.

> Aside: the speed of the memory cache can make a huge difference to performance as witnessed by the Intel Celeron chips.

Thus we need to wait a period after requesting data before we can act on it. Some of this delay is hidden in the pipeline, but still for many RISC machines we cannot write

```
load   r2, N
add    r3, r2, r1
```

and hope it will complete in 2 cycles, as the value of `N` will not be ready by the time the `add` requires it. If we were to do this, interlock mechanisms would cause the pipeline to *stall*, i.e., the processor would have to stop and wait until the data arrived. Some processors don't bother with interlocks and just give you the wrong answer!

We can use `NOP`

```
load   r2, N
nop
add    r3, r2, r1
```

to give the system time for the memory reference, but better is *instruction scheduling*:

```
load   r1, N
load   r2, N
add    r3, r2, r1    <- stall
load   r5, N
load   r6, N
add    r7, r6, r5    <- stall
```

is replaced by

```
load   r1, N
load   r2, N
load   r5, N
load   r6, N
add    r3, r2, r1
add    r7, r6, r5
```

10.3.4 One Instruction per Cycle

Another aim of RISC machines is to retire (at least) one instruction per cycle. This is a lot more feasible for RISC than CISC as each instruction requires roughly the same amount of work, and by the use of pipelines. A good RISC machine will *average* about one instruction per cycle using these techniques, but if jumps or slow loads or whatever result in a broken pipeline, this average can fall quickly.

There are two important arithmetic operations that pose a big problem for this aim, namely multiply and divide. These inherently take a while to do, so how can we hope for one instruction per cycle when we have them? There are several ideas.

1. Do not provide multiply or divide instructions. This is the simplest route, but it is not terribly satisfactory. Typically, library subroutines are provided instead.
2. Break up the multiply operation into several steps. The early SPARC provided a *multiply step* instruction that does a 1 bit step of a full multiply. The code writer has to issue the appropriate of steps to complete the multiply. A full 32 bit multiply takes 32 multiply steps, but if we are multiplying by a known constant we can do it in less: e.g., to multiply by 9 takes four steps. The early SPARC did not provide a divide.
3. Use a coprocessor. Have specialised extra hardware in your computer to deal specifically with multiply and divide. When you get a multiply instruction just ship the arguments out to the hardware, and retrieve the result an appropriate time later. You can continue processing the pipeline in the meantime.
4. Take the operation out of the pipeline. This is essentially the same as the preceeding case, but the operation is performed on-chip. The MIPS processor has a 32 by 32 bit giving 64 bit multiply instruction, and a 32 by 32 bit giving 32 plus 32 bit quotient and remainder instruction. In each case the operation is taken out of the pipeline and continues in parallel with it. The result(s) are placed in special 32 bit registers `HI` and `LO`. If you try to access these before the result is ready, an interlock mechanism halts the pipeline until the results are ready. Thus, with careful instruction scheduling, we get a fast multiply and divide.
5. Do it in one cycle. This is the preferred choice, but it can be very expensive in terms of processor area.

10.4 CISC or RISC?

The dividing line between RISC and CISC is getting fuzzier all the time. Many of the good ideas of RISC have been taken up by CISC (pipelines, large register sets, one cycle per instruction, etc.), so the technologies appear to be converging, and there is decreasingly little to choose between them.

Some take the best parts from each philosophy and try to tread a middle ground between the two. For example, the Coldfire processor from Motorola claims to be a "variable length RISC" processor. It is designed to be used in embedded systems (washing machines, cars, etc.) where memory is a premium, so they use CISC ideas to get compact code, but they use RISC ideas for speed. In fact, the instruction set of the Coldfire is based on the 68000 with the more complicated and rarely used features removed.

Many Modern CISC machines (e.g., Pentium II) are "really RISC" underneath: the simple instructions are executed directly in hardware, while the more complicated ones are implemented as sequences of the simple ones. This kind of idea has been around for a long time, in particular the implementation of instructions using *microcode*. This is a language much closer to the hardware: at the level of the elements of the pipeline or lower (fetching, decoding, adding, etc.).

A complicated instruction might be implemented as a sequence of microcode instructions rather than having specific hardware dedicated to it. You can do this in many ways:

1. an instruction is decoded into the relevant series of microcode instructions as it is executed (e.g., Pentium);
2. an instruction is expanded at assembly time into the microcode;
3. the hardware itself recognises that it doesn't implement a certain instruction, and does a trap to a subroutine that contains the microcode (e.g., Clipper).

It was a microcode bug that caused the famous Pentium FDIV problem. The instructions in hardware will be faster than the microcoded ones, but you can choose to microcode those instructions that are less often used. Thus the overall impact is lessened.

> Aside: microcode has been around for a long time. The IBM 360 range of machines allowed a consistent instruction set across a widely varying performance hardware series. The cheap end of the price range would have a lot of instructions done in microcode, while the expensive end would do most instructions in hardware. Thus all machines could run the same machine code, but at greatly varying speeds.

The expectation is that a CISC machine will produce shorter code than RISC, as a single instruction does more. There is some truth in this, but the difference is smaller than you would expect, and the fact that there is a difference is less important in these days of cheap memory. The reason is that the complicated instructions are hard for compilers to generate: there may be a special instruction designed for doubly linked lists, but most compilers will have great difficulty in spotting that the user's code actually *is* describing a linked list. So, on the whole, both CISC and RISC machines end up executing the same kinds of instructions (which, after all, is the RISC philosophy's thesis). Furthermore, CISC instructions can actually vary greatly in length with some being very long: it takes more space to specify the more complicated options of some instructions.

Another expectation is that CISC machines will go faster (as more gets done in a single operation), but this too is not necessarily borne out in practice. For the real estate on the processor that is dedicated to implementing a whizzy new instruction that never gets used could be better devoted to making the basic instructions go faster. And then the sequence of basic instructions that does the same as the single whizzy instructions could well be faster overall.

Having said all that, there is currently a race between Intel (Pentium, CISC) and DEC (Alpha, RISC) to produce the fastest chip: and they keep overtaking each other. Much of the case for CISC really seems to be Intel's requirement for backwards compatibility with the old 80n86 architecture. It is interesting to note that next year Intel will end development of the 80n86 and Pentium family, and will introduce an entirely new architecture.

> Aside: you can take RISCiness too far. For a while Alphas did not have any byte operations, instead you had to load and mask off 32 or 64 bit words. Thus byte operations on Alphas used to be *slower* than word operations. The designers did some more analysis, and later included byte instructions.

11 Other Architectures

We now look briefly at other ways in which the architectures of computers can vary, in particular we move away from the classical von Neumann model of a sequential computer to look at parallel machines. That is, computers with more than one processing unit.

11.1 Parallelism

The next (seemingly) logical thing to do to make your programs run faster is to use multiple processors. This is the "many hands make light work" theory. The ideal is that you somehow split your problem up and use n processors, then your answer should appear n times sooner. After all, this is a common way for humans to work: many people cooperating on a single project (sometimes) brings the result faster.

There is another reason for parallelism that sometimes get lost in the quest for speed. Some problems are so large that a single person or processor can never hope to finish it. Some jobs simply need more than one set of hands (e.g., building a skyscraper).

Much research has gone into hardware and software to support this kind of approach. Unfortunately, this is very far from reality. Perhaps a more suitable aphorism is "many cooks spoil the broth". There are many instances of a project falling behind schedule, and the management have added in extra workers only to find the project running even slower! Sometimes extra people just get in each other's way.

There are a few simple measures of the "goodness" of a parallel system, the most straightforward being *speedup*. The definition of the speedup of some process on a parallel system is easy:

$$\text{speedup } S_n = \frac{\text{time taken on 1 processor}}{\text{time taken on } n \text{ processors}}.$$

Warning: this definition is *too* simplistic, and can produce misleading values, but will do for now. See a final year course on parallel processing.

The ideal speedup of $S_n = n$ for n processors is quite rare, due to many factors.

1. Not all programs split into n independent parts. Some parts of a project may depend on other parts begin finished earlier: you cannot test a product until it has been built. Thus (generally speaking) you cannot run testing in parallel with construction. Similarly, many algorithms have inherently sequential parts. In computing $\sin(x + y)$ you cannot calculate the sin until after you have finished the addition.

2. Contention on shared resources. Some resources are limited, and can only be used by one agent at a time. If you have 10 people all wishing to print out their project reports simultaneously on a single printer, some of them will have to wait. If they are organised, maybe they can get on with some other work while waiting, but this is not always possible. Idle time means that all people (or processes) are not working all the time, so reducing speedup.

   In terms of processors, there are several limited resources, including memory. If there are 10 simultaneous requests for values from memory, some processes will have to wait.

3. Communications costs. If there is more than one processor working on a problem, there must be some transfer of data between processors. There are some very rare exceptions to this, but this is generally true. The time a processor spends communicating a result to another processor is time not spent on useful computation. Sometimes the cost of communication is low, sometimes high (see below), but always you have to leave off computation to talk (sometimes it's faster to do something yourself than to spend time telling someone else to do it). This, too, reduces the speedup you get.

Amdahl made a simple observation (that many people overlook) that places a natural limit on the maximum speedup you can expect from a system. Suppose you have a program that has 10% serial code. That is to say this code cannot be made parallel, e.g., reading data or printing the results. Then even if we have an infinite number of processors at out disposal, and can make the rest of the code run infinitely fast, the maximum speedup we can get is

$$S_n = \frac{100}{10} = 10.$$

Think about that: even with as much hardware as we like, we can never make this problem go more than 10 times faster. For we are limited by the sequential part to a certain overall time for the program to run.

Different programs will have different natural limits, but there will always be a limit if you want to see the results! Some problems will only go faster if you get a faster processor.

11.2 Classification

Having cast a cloud of gloom over parallel systems, we now proceed to look at the various ways such systems can be arranged. After all, many of these systems are designed and sold, so there must be a positive side to them!

There is a simple classification of sequential and parallel systems that was proposed by Flynn.

	one data stream	many data streams
single instruction stream	SISD	SIMD
multiple instruction stream	MISD	MIMD

Aside: more complicated classifications exist.

11.2.1 SISD

One processor acting on a single datastream. This is the normal von Neuman system.

11.2.2 SIMD

Many (upwards of 1000) processors all executing same instruction stream on different data. Single instruction decoder, multiple processing elements. Very good for regular problems, e.g., weather prediction that requires same computation on a large number of weather measurement data samples. Good for regular data communications between elements. Poor for irregular problems.

```
CPU--CPU--CPU--CPU--CPU--CPU--CPU--CPU
 |    |    |    |    |    |    |    |
CPU--CPU--CPU--CPU--CPU--CPU--CPU--CPU
 |    |    |    |    |    |    |    |
```

```
CPU--CPU--CPU--CPU--CPU--CPU--CPU--CPU          <--
 |    |    |    |    |    |    |    |              \
CPU--CPU--CPU--CPU--CPU--CPU--CPU--CPU               \
 |    |    |    |    |    |    |    |          <---- Instruction
CPU--CPU--CPU--CPU--CPU--CPU--CPU--CPU               /   stream
 |    |    |    |    |    |    |    |              /
CPU--CPU--CPU--CPU--CPU--CPU--CPU--CPU          <--
 |    |    |    |    |    |    |    |
CPU--CPU--CPU--CPU--CPU--CPU--CPU--CPU
 |    |    |    |    |    |    |    |
CPU--CPU--CPU--CPU--CPU--CPU--CPU--CPU
```

All processors are executing the *same* instruction but on different data. For example, we might do 1024 additions simultaneously. Such a machine is called an *array processor*. Closely related are the *vector processors*, which consist of a single row of processors but obey the same principles.

Here is some example C-like code, called *MPL*, the MasPar programming language.

```
int x;           /* a single copy of x, shared by all processors */
plural int y;    /* each processor gets its own y */

y = y + 2;       /* increment each copy of y simultaneously */

if (y == 1) {
  y = y - 1;     /* only do this on processors that had y = 1 */
}
else {
  y = y * 2;     /* only do this on processors that had y != 1 */
}
```

If all processors are executing the same instruction, how do decisions ever get made, e.g., in the `if` above?

Decision making is done on a per-processor process. A processor either takes part in the communal operation, or the processor is *inhibited*, where the processor just sits idle and does nothing for a while. So in the `if` expression above, some processors will subtract 1 from their `y`s while others are idle; next the others have their turn and double their `y`s while the fist set are idle. So, in a sense, both branches of an `if` are *always* executed, but just at different times on different processors: and `if` will take time that is the sum of the times for the two branches.

In

```
while (y > 2) {
  y = y - 1;
}
```

the loop will continue going round, each active processor decrementing its `y`, and becoming inhibited when it drops to 0. Only when all values of `y` fail the test does the `while` end, and the program continue. A `while` loop will execute in the time taken for the last processor to finish.

Data can move between processors:

```
y = xnetN[2].y;
```

The *Xnet* is a way of neighbouring processors to communicate. This example sets the new `y` value on each (active) processor to be the existing `y` on processor 2 to the north (i.e., 2 up in array). This all happens simultaneously: potentially a great deal of data moving in a single statement. There is also `xnetW`, `xnetSW`, etc.

A more complicated contruct, the `router`, exists, which moves data in much more sophisticated ways: each processor can reference an arbitrary other processor.

This seems a strange way of programming at first, but soon you get used to the different perspective. The major point to note is that the data are the important things to concentrate on, while the processors are secondary: it doesn't matter if you have some idling processors (or even processor doing useless work) as long as the data is in the right place.

Array processors have been around for a long time. An early version was the ICL DAP which had an array of 1-bit processors. Each processors had a few bytes of local memory. The MasPar MP-1 advanced to 4-bit processors with 16k bytes per processor. Next, the MasPar MP-2 had full 32-bit processors, and 64k bytes per node. Currently there are some large machines built from standard Sparc or Alpha chips, with a few megabytes per node. These can be very expensive when you have an array of 1000s of nodes!

11.2.3 MISD

Multiple processors doing different things to a single datastream. Not very popular, maybe includes systolic.

11.2.4 MIMD

This is the most popular view of what "a parallel computer" is: multiple independent processors running their own instruction streams on different data.

Possibly sharing one memory space. Good for problems that split neatly into independent parts. Degrades when these parts are interrelated, or need communication between them. More time is spent waiting on another processor, or another piece of data, than is spent computing. Problems with resource contention: e.g., simultaneous reading and updating same variable. For example, if processor P_1 executes `x=1` and processor P_2 executes `x=2`, what is the resulting value of `x`? It depends on the order that they did it, and this can vary from run to run of the same program (other activity in the system can slow a processor down).

11.3 Typical MIMD Architectures

MIMD machines come in many shapes and sizes.

11.3.1 Shared Memory

The standard von Neumann model has a cpu and memory connected by a bus:

```
                  memory
                    |
    --------------------------------- bus
       |
      cpu
```

with other bits and pieces (such as I/O) relying on the bus, too. The simplest multiple processor architecture expands on this by having more than one cpu:

```
                  memory
                    |
--------------------------------- bus
    |      |      |      |
   cpu    cpu    cpu    cpu
```

So many processors share a single memory. This allows the programmer to write fairly simple parallel programs, in the knowledge that the value of variable `x` that cpu 1 sees is the same that cpu 2 sees, as they both refer to the same area in memory where `x` is stored. If cpu 1 updates `x`, then cpu 2 will see the change.

The problem is the bus: it soon becomes a bottleneck for data flowing to and from the memory. Even on a single processor system, the bus is typically slower than the cpu, and if you have 4 or 8 or more cpus all trying to share this limited resource you soon become unstuck. It is very much like a road with too much traffic: soon we get traffic jams with lots of cars waiting about. Similarly, the cpus have to sit and wait for access to memory.

11.3.2 Alternate Topologies

Thus the *shared memory* above architecture has limitations: you can't connect very many cpus together, perhaps 4 or 8 or so with current bus technology.

Some engineers tried to address the bus problem by connecting the cpus together in different ways, e.g., butterflies, torus, hypercubes.

```
                                         n-----n
                                        /|    /|
n----n              n-----n            n-----n |
                    |     |            | n---|-n
                    |     |            |/    |/
                    n-----n            n-----n
```

And so on for higher dimensions. In the latter cases there are many paths between nodes, so data can flow alongs different routes. This should relieve the bottleneck problem.

Unfortunately such designs were expensive to make, and programs for them were not particularly portable between machines of different sizes, so this idea languished for a while. Lately, however, the SGI S2MP (Scaleable Shared-memory Multi-Processing) architecture has revived this architecture. Nodes contain both processor and memory, and as you add nodes, you automatically add extra bandwidth. Clever hardware and software makes it easy to use (see virtual shared memory, below).

11.3.3 Cacheing

Caching, snoopy.

```
                memory
                  |
--------------------------------- bus
    |       |       |       |
  cache   cache   cache   cache
   cpu     cpu     cpu     cpu
```

Local copy kept in per-processor caches. Reduces traffic on bus, as we read more often than write. Need cache *snooping* to ensure consistency: when one processor writes a value, others snoop on the write and update their own copy if they have one.

11.3.4 Distributed Memory

The problem is the shared address space (the single global chunk of memory): how to engineer the machine so that `x` on processor is the same as `x` on another.

What happens if we dispense with this requirement?

Distributed memory. Message passing.

Individuals working on a shared project. Each has their own mindset, and communicate by talking (sending messages). So `x` on one processor is not the same as `x` on another: they have separate address spaces.

```
--------------------------------- network
    |       |       |       |
   mem     mem     mem     mem
   cpu     cpu     cpu     cpu
```

Now speed of network is the bottleneck. Good for programs with smaller communications.

11.3.5 Virtual Shared Memory

The single memory address space is so useful to program that people were loath to lose it, even though it is impractical for more than a few processors. Thus the standard computer science trick of virtualising is used. Instead of having *real* shared memory, just arrange things in the operating system (and the hardware to support it) so that you *appear* to have a single shared address space. Thus memory can be distributed, and some mechanism behind the scenes maps addresses to access the relevant chunk of memory, wherever it lives. This is called *Virtual shared memory*, analogous to virtual memory (i.e., swapping).

There is, of course, overhead associated with using virtual shared memory, just as there is overhead with using swapping, but the benefits are (claimed) to outweigh the losses. This leads to a style of machine called *NUMA*, for *non-uniform memory access*. Thus finding the value of a variable may take a different amount of time on different processors: if it's in local memory it will be much faster to get than if it's remote. The NUMA style of programming tends to try to keep often-used data local, and lets lesser used data be remote. Some systems have hardware to do this for you: they migrate data from processor to processor to optimise locality.

> Aside: The SGI Origin is a commmercial NUMA architecture. It has a remote access time of about 600ns, compared to a local access time of probably around 10ns. Clever cacheing helps cuts down this disparity. They have shipped 128 processor systems, and claim expandability to 1000s of processors.

SPMD.

11.3.6 Clustering

Another way of joining machines together that is popular because it is cheap. Joining standard workstations together using a network. Cheapest end of the market is using a standard network, e.g., Ethernet, but this is quote slow: talking to a remote machine can take milliseconds. Faster interconnects include PAPERS (a cheap synchronisation method for PCs that can get microsecond responses), and more expensive networks like ATM, MyraNet, and the emergent gigabit and 10 gigabit Ethernets.

A cluster need not be composed of identical machines, they can be different sizes or speeds. At a pinch they could be different processors: e.g., a cluster of Alphas and Pentiums is possible, but this is tricky as you have to provide code for each processor architecture. Also, they need not be uniprocessors: a cluster of shared memory machines is called a *clump*.

11.4 VLIW

Here is another attempt at speeding up computions, this time based around a single processor.

In the 1980's there were a few experimental machines based on the VLIW (very long instruction word) concept. The idea is this: doing two instructions at once will be twice as fast as one at a time. Three at once will be three times as fast, and so on, always assuming that we have enough replication in our hardware to be able to execute these simultaneous adds, subtracts, etc. Taking this to its conclusion, we might try to do n instructions simultaneously. The "long instruction" is simply the concatenation of all n individual instructions. There is a limitation, of course, on how big n can be.

1. Sequentiality of code. Some instructions *must* be done later than other ones. For example, we cannot print a result until we have computed it.
2. Branches. Code is rarely a straight line, there will always be decisions.

These two are fundamental restrictions on the length of a long instruction. There are clever ways of collecting together individual instructions from here and there and placing them together in a long instruction (similar to instruction scheduling in RISC architectures), but there will always be a limit.

This idea was pioneered by Cydrome and Multiflow among others, but did not get much acceptance. Essentially, it failed due to insufficiently clever compilers, and the then current state of VLSI. A VLIW machine needs a lot of hardware to execute the long instructions, and a clever compiler to be able to sort out those instructions to put together. At the time, compiler technology was inadequate, though today we could have a good go at it. In fact, some of the ideas of VLIW are being revived in the EPIC architecture, as we shall see later.

12 Whacky stuff

We now look at the more strange aspect of modern processors. We talk mostly about the MIPS processor, as it is fairly typical of RISC machines.

12.1 More RISCy stuff

Looking back at the instruction formats for MIPS, there is much we can deduce:

1. 32 bit instructions
2. the top 6 bits identify the instruction uniquely
3. there are 32 registers (5 bit fields)
4. MIPS is a three register architecture (register, register to register)
5. immediate values can range from -32768 to 32767, uniformly over instructions

It would appear that immediate jumps have a range of 26 bits. In fact MIPS takes advantage of the fact that all machine instructions are 4 byte aligned, and appends 00 to the `target` address, giving a 28 bit range. This is normally enough (a range of plus or minus 128Mbytes), but if you need more there is an jump-to-address-in-register instruction too.

> Aside: the Sparc jump instruction is identified by the top two bits, and the 30 others have 00 appended as in the MIPS. Thus the Sparc has a full range.

There is no means to load an immediate 32 bit value: instead the value must be loaded in two 16 bit halves. There is `lui` *load upper immediate* to load 16 bits into the upper half of a register (and clear the lower half). There is no *load lower immediate* as you can use `ori` to achieve the same effect:

```
lui    $4, 0x1234
ori    $4, $4, 0xABCD
```

to load register `$4` with hex 1234ABCD.

The first argument is the target. In fact many MIPS assemblers allow the `li` *load immediate* instruction.

```
li     $4, 0x1234ABCD
```

This is a *pseudo opcode* or *pseudo instruction* that is expanded by the assembler into the two instruction sequence above.

> Aside: pseudo opcodes are quite popular, and can lead to an assembler program looking significantly different from the machine code it produces. Intelligent assemblers can do a lot, including reordering code, replacing code by other faster equivalent code (e.g., `mul a0, a0, 4` becomes a shift), and sundry other optimisations. Sometimes it is useful to use a *dissassembler* that takes machine code and produces assembler to see what is *really* going on. IRIX and Solaris have the `dis` command.

This two-step load is not as bad as it seems: it takes 8 bytes instruction space, and 2 cycles to execute. A typical CISC equivalent could take 8 bytes space (4 for the instruction plus 4 for the immediate data), but take more cycles to execute.

Similarly for loading from memory. The pseudo opcode

```
lw    $4, 0x12345678
```

is really

```
lui   $4, 0x1234
lw    $4, 0x5678($4)    ; real opcode
```

taking 8 bytes, 2 cycles. This is comparable to CISC in time and space, too.

This feature of not providing an instruction when another will do just as well is a big feature of MIPS, and of RISC in general. For example, there is no move register-register, and instead of

```
move   $4, $5
```

we can use

```
or     $4, $5, $0
```

as register `$0` always contains 0.

> Aside: the 68000 contains many cases of equivalent instructions. For example, `MOVEA` and `LEA`; `CLR` and `MOVE #0`; `MOVE #1` and `MOVEQ #1`; and so on.

The MIPS processor has no condition codes. Instead there are instructions that take two registers and a 16 bit displacement. MIPS noted that comparisons are almost always followed immediately by jumps, and so they conflated the two.

```
beq    $4, $5, label
```

will compare the values in `$4` and `$5` and branch to `label` if they are equal. Similarly for `bne`. Notice that by replacing (say) `$5` by `$0` in the above we have a test for zero (as `$0` is always 0).

Branching on comparisons is somewhat different in MIPS. They employ the `slt` *set on less than* instruction.

```
slt    $d, $s, $t
```

will set `$d` to 1 if `$s` < `$t`, else to 0. This, together with `beq` and `bne` is enough to code any comparison.

Suppose we want to branch to `label`:

when $a0 < a1$		`slt at,a0,a1`
		`bne at,zero,label`
$a0 \le a1$	i.e., $a1 \not< a0$	`slt at,a1,a0`
		`beq at,zero,label`
$a0 > a1$	i.e., $a1 < a0$	`slt at,a1,a0`
		`bne at,zero,label`
$a0 \ge a1$	i.e., $a0 \not< a1$	`slt at,a0,a1`
		`beq at,zero,label`

`at` is a temporary register (actually `$1`); `zero` is another name for `$0`. Doing it like this allows us to separate a test from its use: it effectively allows us to use any register as a condition register. Thus we can make a test and use its result later in the knowledge that some other operation has not overwritten the status flag.

Assember pseudo opcodes `bgt`, `bge`, `blt`, `ble` compile appropriately into `slt` and `beq` or `bne`.

Unusually for a RISC, there are also specialised instructions

1. `bgez` branch if greater than or equal to 0
2. `bgtz` branch if greater than 0
3. `bltz` branch if less than 0
4. `blez` branch if less than or equal to 0

The designers for MIPS clearly believed these comparisons occur often enough to warrant being encoded as single instructions, rather than the two-step above.

The 32 registers of MIPS are conventionally allocated in this way:

register number	**symbolic name**	
0	zero	always contains 0
1	at	used by the assembler for 2 part loads, `slt`, etc.
2–3	v0–v1	integer function results
4–7	a0–a3	integer arguments, not preserved across function calls
8–15	t0–t7	scratch registers, not preserved across function calls
16–23	s0–s7	saved registers, must be preserved across function calls
24–25	t8–t9	scratch registers, not preserved across function calls
26–27	k0–k1	used by the operating system
28	gp	global pointer
29	sp	stack pointer
30	fp	frame pointer
31	ra	subroutine return address

Registers `t0`–`t9` are caller saved, and `s0`–`s9` are callee saved. The *global pointer* points at the data area of a program, and is used as an offset for loads and stores (rather than doing a two step `lw`).

12.2 Floating Point

We haven't really mentioned floating point operations so far, even though many people think that number crunching is what computing is all about. This is mostly not the case, as much more time is spent on integer or non-numeric (e.g., character) processing that appears at first. A lot more people run word processors than run big data crunchers. On the other hand, pure floating point ability is something that is easy to measure, so much time is spent on optimising this ability.

Many modern processors have on-chip floating point arithmetic: also many use dedicated floating point coprocessors. The MIPS uses a coprocessor with 32 of 32 bit registers that can be addressed individually (`f0` to `f31`), and in 16 of 64 bit pairs (by means of an even `f`). There are also various instructions to move data to and fro.

```
lui     at,0x3ff0                 # fp representation of 1.0
mtc1    at,$f5                    # at -> f5
mtc1    zero,$f4
add.d   $f12,$f12,$f4             # 1.0 into f12 (r4,r5) -> (r12,r13)
mul.d   $f0,$f12,$f14             # multiply
...
swc1    $f6,4(at)                 # store both halves of a 64
swc1    $f7,0(at)                 # bit fp value
```

The `.d` indicates a *double* value (as in C). There are also `.s` single-precision floats. Further instructions do floating point compare and branch; convertions between single, double and integer formats.

Aside: Arguments are conventionally passed in `f12` and up, values returned in `f0`.

The usual caveats about delay slots apply.

12.3 All You Need is Cache

Processors may munch through data at over 500 million operations per second, but raw speed is no good unless you can get the data to the processor fast enough. A PC these days might have a 66Mz bus: that is the pathway that leads from memory to processor only delivers at 66 million words per second.

This is a massive bottleneck, and can produce a huge slowdown as the processor repeatedly has to wait for data to arrive.

Already there are 100MHz buses, and soon 133MHz buses, but there is still a gap in speeds to be overcome.

And memory itself is not so speedy, either. Current fast memory cycles at about 100MHz, so you might have to wait up to 5 cycles on your 500Mhz processor. New technologies (such as Rambus) are emerging to increase speeds to 600MHz and soon 800MHz.

The solution generally used is to *prefetch* data to local very fast memory. Such a *cache* is close to the processor and can deliver data to the processor at rates the processor needs. Cache memory is very expensive, so it tends to be limited to a few kilo- or megabytes. Some processors have on-chip cache, that is the cache is part of the processor itself. Additionally such machines can have a larger external *level 2* cache, for a two-stage caching system.

```
CPU <-> On-chip <------> Off-chip L2 <-----------------------> main memory
         cache               cache
```

The big question is "which data to prefetch?" Fortunately, most programs have good *locality* of data. This means that, with large probability, if you are using byte n now, you will require a byte near to n in just a moment. In other words, programs tend to use data from certain areas of memory, rather than at random. Of course, it is easy to write programs that *do* use memory at random. Such programs usually perform noticeably poorly on modern architectures.

Caches work by loading big chunks of memory at a time: whenever a piece of data is required the data surrounding it is loaded at the same time. Locality means that nearby data is in the cache already, and so can be accessed very quickly.

> Aside: in fact, there are certain classes of irregular programs that run faster when cacheing is removed. But these are relatively rare.

> Aside: all the above applies to program, as well as data. That is we can prefetch program instructions to cache, and program instructions tend to have good locality, too.

12.4 Superscalar

Hardware in cpu notices that in

```
add    r1, r2, r3
add    r4, r5, r6
```

there is no dependency between the statements, and so executes them simultaneously. Thus get more than one instruction per cycle, at the cost of multiple functional units. That is, one than one adder, multiplier, etc. This is *superscalar* processing. It is now fairly common, but it has a large price: the extra hardware needed in a chip to recognise the independence is complicated and takes up a large area of the chip (the "real estate").

The UltraSPARC-II has two integer units, five floating point units (two for floating-point addition, two for floating-point multiplication, and one for floating-point division and square roots), a branch-processing unit, and a load/store unit. It is described as "four-way superscalar", meaning that up to four instructions can be executed in a single clock cycle: two integer, and two floating point. This is at peak performance: it may not always be possible to get four such instructions together, so actual performance might be lower.

12.5 Instruction Reordering

A superscalar processor needs a mixture of different types of instructions to gain most benfit from the multiple functional units. However, instructions may not always arrive in the most suitable order. If the processor could rearrange instructions it could retire tham faster.

Hardware in cpu notices that in

```
load   r1, foo
add    r1, r2, r3
add    r4, r5, r6
```

we can reorder the last two statements (i.e., swap them) to allow the `load` to complete before we need its value. Thus we avoid a pause in the pipeline. The cost is the extra hardware needed to make this kind of analysis, and to untangle the results as they arrive out of order. This turns out to be a very valuable advance in speed, but at a huge cost in processor real estate.

The MIPS R10000 takes instructions through the first three steps of the pipeline before sending them to one of three queues: an integer queue; a floating point queue, and a load/store queue. The queues are dealt with in whatever order depending on when the relevant resources become free. Program order is ensured by the R10000 always retiring the oldest instruction first.

The Hewlett-Packard PA-8000 keeps a 56-instruction-deep instruction reorder buffer, which looks ahead to the next 56 instructions in the stream to find four that can execute in parallel.

12.6 Speculative Evaluation and Register Renaming

In

```
     add    r4, r5, r6
     bne    r4, r2, foo
     sub    r1, r2, r3
...
foo  mul    r1, r2, r3
```

The `sub` can be evaluated in parallel with the `add` as long as we don't update `r1` until we know the result of the branch. If we take the branch, we just discard the result of the subtraction. If we don't take the branch, we have come out ahead of the game. This is *speculative evaluation* of instructions, and is very difficult to get right. If we *do* get it right, there is quite a lot to be gained.

Further, the processor might also speculatively evaluate the `mul` at the destination of the jump, keeping or discarding the result appropriately.

The MIPS R10000 has four-way speculative execution (branches after branches), using *register renaming* to ensure that no results are committed to the real registers until the branch is resolved. With register renaming the association between names and hardware registers can be changed dynamically. You may store a value in any old register that happens to be free, and only later name that register to be the register where the programmer wanted the result to be. If this is a speculative execution we can ignore the result. In the above example this means that both the `mul` and the `sub` can execute simultaneously and store their results in two different registers. Then, when the result of the branch is known, one of those registers can be named `r1`.

Similarly, we can have speculative loads: the hardware notices that a value from memory might be needed in a short while, so it issues a load instruction in advance. If, later, we didn't need the value we can throw it away, and not much is lost. If we *do* require the value, we don't need to wait for it.

12.7 Register Windows

Registers are a valuable resource in a processor: they are much faster than memory, and are extremely useful for passing arguments to functions and keeping partial results. The difficulty is that there never seems to be enough registers, so that there is an overhead of saving and restoring in each function call. The designers of the SPARC processor noted this and wondered if they could somehow expand the register set so that there would always be enough to play with.

One way to do this would be to have a huge number of registers, but that was a too costly option at the time SPARC was being designed. So they had to think of something else while keeping the flexibility to run the same program regardless of the actual number of registers a particular chip could afford to implement. They came up with a solution that has a window on a larger set of registers called a *register file*.

Aside: Intel's next processor has 128 integer registers.

At any one point in time, a SPARC subroutine can refer to one of 32 registers. The first 8 are global registers, that can always be referenced by any subroutine. The other 24 are divided in to three groups: *in registers*, *out register*, and *local registers*. The *out*s are numbers 8–15 (`o0`–`o7`), the *local* are numbers 16–23 (`l0`–`l7`), and the *ins* are 24–31 (`i0`–`i7`). These are collectively the *register window*.

```
 in   local out
24-31 16-23 8-15
             in   local out
            24-31 16-23 8-15
                         in   local out
                        24-31 16-23 8-15
                                ...
                                                    global
                                                      0-7
```

When a subroutine is called, the argument passing convention deems that a `SAVE` instruction is called. This instruction moves a pointer so that the current register window is now 16 registers further down the register file. The registers are renumbered accordingly. Thus the out registers of the previous subroutine are now the in registers of the current subroutine. Therefore we have passed parameters without any copying. We have a fresh set of local registers: no need to save and restore. And registers ready to pass arguments to the next subroutine.

At the end of the subroutine, we do a `RESTORE` to shift the window back to its previous position.

There is an additional optimisation: if you are a *leaf subroutine*, that is you do not call a subroutine yourself, you need not do a `SAVE`, as you can use the out registers instead of the locals. This turns out to cover a fairly significant number of cases, as most programs have more leaves than internal nodes.

The suggested convention for SPARC uses only six registers to pass arguments. Register `o6` doubles up as the stack pointer, while register `o7` is a temporary. Register `i6` is used as a frame pointer. The `SAVE` and `RESTORE` instructions do the right thing regarding saving and restoring stack and frame pointers. Or, rather, the moving window does the right thing.

Of course, we only have a finite number of registers, so there must be a case of when the window runs over the end of the file. What happens then? The SPARC specification says you can have anything from 2 to 32 register windows. This means there can be anything from 40 to 520 registers. The first SPARC chip had seven register windows, or 120 registers.

Aside: number of registers = 8 global + 16 $\times$ number of register windows.

The register file is actually kept as a circular list: the last register set overlaps the first. When you run out of windows a *window overflow* trap occurs, and a supervisor routine saves the contents of one or more of the old windows (a *register spill*). When the windows unwind back again, another trap is called to restore the old windows.

There is a choice to be made in how many windows to save: too many and you waste time saving; too few and you will get too many traps. A study reckoned on saving two windows if you have less than eight windows, and saving one window otherwise.

The great beauty of this scheme is that a particular processor can have a register file of 40 registers, or 56 registers or 72 registers or whatever, and the same program works regardless. There will be more swapping of register sets on the smaller processors, so your program will run more slowly, but it will run.

There are drawbacks, too:

1. deeply recursive procedures are almost guaranteed to cause multiple spills, thus distorting programmers' style when they try to avoid writing recursive programs;
2. context switches are expensive: in a multiuser system the operating system switches process every few 100s or 1000s of procedure calls, and it may well be that *every* register in the register file will have to be saved. This means that process switching is very time consuming;
3. when you run out of register windows a trap occurs, this is another large overhead. You may end up saving and restoring the same number of registers overall as in a conventional system—you do it in blocks rather than piecemeal—but you also pay the price of the trap;
4. because many subroutines are leaves, the window rolling mechanism is not as useful as it might be. Thus register windows fall prey to the RISC philosophy: why waste processor real estate on them?

It's expensive when you use it: it's a waste when you don't.

The problems with windows means that only SPARC uses them. All other processors have stuck with the single set of registers as they have found they can optimise for that better. Indeed, even new SPARC chips have only eight windows (136 registers in total), not the 32 that was originally envisaged.

Of course, you don't *have* to use register windows, and in practice, many operating systems don't if they can avoid it. They prefer to reserve the use of windows for relatively rare events, such as traps, where windows do turn out to be useful.

12.8 Extended Instruction Sets

The way people are computing has changed over the last few years. Less emphasis is made on pure floating point crunching, and more is made on integer processing (including word processing) and multimedia activities (graphics and sound). Therefore it makes sense that the kinds of instructions found in a processor have evolved to meet the new requirements.

The most visible example of this is Intel's addition of MMX (*MultiMedia eXtensions*) instructions to the 80x86 architecture. This is simply a new set of instructions that are specifically aimed at the type of data movement and manipulation that graphics and sound want. Similarly, Sun has added VIS (*Visual Instruction Set*) instructions to its latest UltraSPARC, MIPS has MDMX, pronounced "Mad Max" (*MIPS Digital Media eXtension*), Alpha has MAX (*MultimediA eXtensions*), Hewlett Packard has MAX (*Multimedia Acceleration eXtensions*), and Motorola developed AltiVec for the PowerPC.

Intel	MMX	MultiMedia eXtensions
Sun	VIS	Visual Instruction Set
MIPS	MDMX	Mad Max
DEC	Alpha	MultimediA eXtensions
Motorola	AltiVec	

Aside: the prize for the best name goes to MIPS, I think.

Notice this is *not* a departure from RISC: the individual instruction may be quite complicated, but as long as it completes in one cycle, it will be compatible with the rest of the architecture. We repeat: it's a reduced instruction *set*, not a reduced instruction. The designers look at the benefit to cost ratio, and if an instruction would be used often enough, it is included in the instruction set.

As a fortuitous side-effect, some of these instructions are also useful in other kinds of computing: Sun claims to have seen improvements in areas such as numerical analysis and computational fluid dynamics, and Motorola's AltiVec is claimed to be good for networking.

Examples of new instructions include:

1. Multiply four 16 bit values by four 8 bit values per cycle. This is used in pixel interpolation.
2. Compare four 16 bit or two 32 bit fixed point values. This is used in edge detection and 3D rendering.
3. The *align* operation allows the processor to access pixels that are in the middle of a 64-bit word.
4. The *expand* instruction unpacks a 32 bit pixel value (8 bits each of red, green, blue, and alpha) into a 64 bit register.

And so on. These are, in essence, SIMD instructions, where the multiple data is packed into single registers. This is a restricted kind of *vector processor*: another idea that has been around for a long time (like an $n \times 1$ array processor).

Aside: using MMX does have a cost, though. It turns out that Intel had to make compromises in the implementation that have the effect that switching between using using the FPU and using MMX incurs a large overhead. This means that, say, a computer game that uses a lot of floating point computation and a lot of MMX operations doesn't gain as much as you would hope.

And now the latest processors have additional instructions, this time to do SIMD floating point. The Pentium III processor has the KNI (Katamai New Instructions) and AMD has 3DNow!. The PIII can handle four floating point operations in once cycle.

Aside: The AltiVec likens itself to a "short vector processor," in comparison to the long vector processors of the 80s: thus they have a few parallel elements rather than 1000s. An AltiVec has 128 bit registers that can be used as

1. 16-way parallelism for 8-bit integers and characters
2. 8-way parallelism for 16-bit integers
3. 4-way parallelism for 32-bit integers and IEEE floating point numbers

12.9 64 bits

The tendency is for new processor designs to be 64 bits wide. This is to allow a greater range of addressable memory, and to swallow more data at a single gulp. The 68000 may have had 32bit registers, but it had 16 bit data pathways, i.e., only 16 wires for values to flow along from memory to register or wherever. This means that it took two passes to get a 32 bit value. Most current processors have 32 bit or more pathways.

machine	register sizes	address range	data pathways	
68000	32 bits	24 bits	16 bits	
68008	32 bits	24 bits	8 bits	backwards compatibility with 8 bit machines
68020	32 bits	32 bits	32 bits	
80386	32 bits	32 bits	32 bits	
80386SX	32 bits	32 bits	16 bits	backwards compatibility with 16 bit buses (IBM PC AT)
R3000	32 bits	32 bits	32 bits	
R4000	64 bits	64 bits	64 bits	often used in 32 bit mode
UltraSPARC-II	64 bits	64 bits	64 bits	
Alpha	64 bits	64 bits	64 bits	

Alphas still have 32 bit instructions, though two at a time are loaded in a single 64 bit fetch, and superscalar decoding and execution ensues.

> Aside: Cray were delivering 64 bit machines in the late 1970s! MIPS produced their first 64 bit chips in 1991, but it took them until 1994 to provide an operating system that could make full use of it.

> Aside: the games consoles that were proudly announcing "32 bits" were in fact several years behind the times when they came out! Now the newest games machines have started using 128 bit custom processors.

13 The Future

The normal development of processors is not set to stop for a long while yet. There are plans for versions of the Pentium that run at 600Mhz or more, capable of executing 5 instructions per cycle at peak rate. The Alpha is promised to be running at 1000Mhz very soon (Intel have demonstrated an experimental 1GHz processor), peaking at 4 instructions per cycle. These have roughly 3 times the performance of their current analogues.

> Aside: A DEC Alpha running at 500Mz can emulate a 200Mz PentiumPro in a sort of "high level microcoding" way. Also: Alphas may have a fast clock cycle, but they tend to have a smaller throughput per cycle than, say, a MIPS. It is said that DEC would probably have to achieve frequencies in excess of 800 MHz to match the system performance of the 300 MHz MIPS R12000.

Before 1993, about 50% of performance enhancements came from boosting clock speeds while the rest came from architectural changes. Since then, however, the chip industry has relied disproportionately on changes to raw clock speed.

The change from CISC to RISC was essentially a shift of complexity in the hardware to complexity in the compiler. Instead of the compiler simply outputing complicated instructions, it had to work harder and output sequences of simpler instructions, all the while worrying about pipelines, instruction scheduling and so on. In the years since RISC was introduced, processors have become steadily more complex again, implementing speculative, superscalar and other ideas. A modern RISC processor is actually quite complex.

As computers have become more and more powerful, it has become feasible to do more and more complex analysis in the compilation, and there are currently a couple of CPU companies that want to repeat RISC's success by again moving complexity in the hardware to complexity in the compiler. But this time the complexity is that of the superscalar and other techniques.

One company, Intel, is making big noises about their new IA-64 architecture. Another, Transmeta, is keeping quiet, but rumours suggest a related approach.

IA-64 (Intel Architecture 64 bit) is a completely new 64 bit instruction set. The design principle is named *EPIC*, (explicitly parallel instruction computing). The first IA-64 implementation, the Merced processor, will also support the old 32 bit 80x86 instruction set (probably microcoded, but still faster than old 80x86 processors). This allows the new processor to run legacy code. Of course, it sould be better to recompile for the new instruction set, but this is not always easy or cheap to do. In fact, many PC programs spend from 30% to 70% of their time in the OS anyway (which will be running IA-64), and so they will benefit that way, too.

> Aside: those figures are more likely an indication of how inefficient the OS is... Though a lot of cycles these days are spent on pretty interface graphics rather than useful computation.

> Aside: note that the philosophy is EPIC (compared to RISC or CISC), the instruction set is IA-64 (compared to 68000 or 80386), and a particular implementation is Merced (compared to Deschutes or Xeon for Pentiums).

It will run at 1000Mz or more, providing maybe 5 to 10 times the performance of current processors, and has the potential of much greater speeds and architectural improvements.

A lot of processor real estate these days is devoted to the clever stuff that decides how to reorder instructions and run them in parallel using multiple units. The big idea behind IA-64 is to move that decison making from the hardware up into the compiler, thus freeing real estate for other optimisations, such as more functional units. There will be no need for interlocking: the processor will just execute the instructions as fast as it can in the knowledge that it is safe, because the compiler said so.

The compiler will generate instructions in "bundles" that have associated tag bits that inform the processor of the instructions' interdependencies, and when they can run in parallel. A bundle of 6 instructions might take 2 cycles to run on one processor, but the exact same code could run in 1 cycle on a more advanced chip implementation with more functional units.

> Aside: Rumours are that at least one version of the chip will have four floating point units, though there will be other variants to suit differerent price ranges.

There is a strong resemblance between this and an idea from the 1980s: the *very long instruction word*, or VLIW. But many of the RISC concepts are present, too.

A single 128 bit bundle contains

```
-----------------------------------------------------------------
|  instruction 2  |  instruction 2  |  instruction 2  | template |
-----------------------------------------------------------------
```

Each instruction is

1. opcode

2. predicate register (6 bits)
3. source register 1 (7 bits)
4. source register 2 (7 bits)
5. destination register (7 bits)
6. opcode extension/branch destination/miscellaneous.

The template is 5 bits long, while each instruction occupies 41 bits. The template contains (amongst possibly other information) the details on grouping of instructions this and other bundles.

> Aside: 3 instructions in 16 bytes (5.3 bytes per instruction) compares reasonably with 4 bytes per instruction of current machines.

There are 64 *predicate register*, each 1 bit long. An instruction is executed if the register selected by the predicate register field is true. This is very useful in this kind of situation:

```
if (r1 == r2) {
  r2 = r2 - 1;
}
else {
  r1 = r1 + 1;
}
r3 = r1 * r2;
```

or

```
     beq    r1, r2, lab1
     add    r1, 1
     bra    lab2
lab1 sub    r2, 1
lab2 mul    r3, r1, r2
```

Rather than executing the jump, with all the costs of pipeline disruption, it can be cheaper to clear the predicate register bit for the `add` and `sub` and simply continue straight on through.

```
     cmp.eq p6, p7 = r1, r2 ;;   # set predicate registers p6 and p7
(p6) sub    r2 = r2, 1           # only exec if p6 = 1
(p7) add    r1 = r1, 1 ;;        # only exec if p7 = 1
     mul    r3 = r1, r2
```

This will suit many cases of `if–then` when a small amount of code needs to be skipped.

The predicate bits come in complemented pairs: `p4/p5`, `p6/p7` and so on, so that `p6` is 1 exactly when `p7` is 0.

The double semicolon (`;;`) indicates that the following instruction cannot be excuted until the current one has finished. Remember we have to give *explicit* instruction parallelism. Notice, though, that we *can* speculatively evaluate the `sub` and the `add` in parallel.

The 7 bit register fields mean there are 128 registers each 64 bits long. There are also 128 floating point registers, each 80 bits long (the IEEE *double extended* format). That's over 2K of register space. This should reduce much of the register shuffling that goes on in current code.

Details of IA64 are available from `http://developer.intel.com/design/ia64/architecture.htm`

There is less hard fact about how Transmeta are approaching the task of producing faster processors (or, strictly, faster process*ing*). However, Transmeta-held patents have been found that talk of "code morphing," that can run multiple instructions sets on a single processor. This is possibly done by taking complex (say 80x86) instructions, decomposing them into simpler parts, and packing them into VLIW-like instructions. Such a chip could be much simpler (there are claims of one quarter the size of a Pentium) and run faster than a conventional chip, quite possibly emulating 80x86 faster than an real Pentium can run. The morphing is done partly by software and partly by hardware.

Another company, TeraGen, also appear to be taking the code morphing approach. They have claims of being able to take multiple instruction streams and to merge them into composite VLIW instructions, thus giving the potential of more (cross-stream) optimisations (e.g., floating point operations from one stream could execute simultaneously with integer operations from another). This is time-sharing in the extreme!

Similarly, Elbrus (Russia) are developing an EPIC processor that employs code morphing. They claim to have built an EPIC machine in 1985.

14 The Real World

The assembler you have used for the 68000 is much simplified, and real-world assembler are much more complex. As we have mentioned already, they can expand pseudo instructions into sequences of real instructions, reorder instructions, replace instructions by other equivalent (you hope) instructions, and so on.

In this section we have a brief look at examples of actual assembler.

14.1 Function Call

Here are some simple function calls

```
int foo(int n)
{
  return(n+1);
}

int bar(int n)
{
  return(foo(n) + 2);
}
```

On an SGI, using `cc -S` this becomes

```
    .verstamp   3 19                    # 7    int bar(int n)
    .option     pic2                    # 8    {
    .text                                   .ent        bar 2
```

```
    .align      2
    .file       2 "eg.c"
    .globl      foo
    .loc        2 3
 # 1
 # 2    int foo(int n)
 # 3    {
    .ent        foo 2
foo:
    .option     01
    .set         noreorder
    .cpload     $25
    .set         reorder
    .frame      $sp, 0, $31
    .loc        2 3
    .loc        2 4
 # 4      return(n+1);
    addu        $2, $4, 1
    .livereg    0x2000FF0E,0x00000FFF
    j   $31
    .end        foo
    .text
    .align      2
    .file       2 "eg.c"
    .globl      bar
    .loc        2 8
 # 5    }
 # 6
```

```
bar:
    .option     01
    .set         noreorder
    .cpload     $25
    .set         reorder
    subu        $sp, 32
    sw  $31, 28($sp)
    .cprestore  24
   sw  $4, 32($sp)
    .mask       0x90000000, -4
    .frame      $sp, 32, $31
    .loc        2 8
    .loc        2 9
 # 9      return(foo(n) + 2);
    lw  $4, 32($sp)
    .livereg    0x0800000E,0x00000000
    jal foo
    addu        $2, $2, 2
    .livereg    0x2000FF0E,0x00000FFF
    lw  $31, 28($sp)
    addu        $sp, 32
    j   $31
    .end        bar
```

Notes

1. various informative parts `.verstamp`, `.file`
2. `.text` informs the assembler that the following is to be placed in the *text* (i.e., program instruction) part of memory. Other areas might be `.data` for data.
3. `.align` to ensure that the following is aligned properly.
4. `.ent` marks the entry point of the code.
5. `reorder` and `noreorder` to inform the assembler that it can and can't shuffle instructions about (e.g., into delay slots)
6. `.frame` This is expanded into a sequence of instructions to handle argument passing.
7. Parts of the oringinal code are scattered about as comments.
8. `$4` is an argument register, and `$2` is a result register.
9. `.livereg` indicates which registers have been used, and is informatin available for extra optimisation.
10. At the end of a subroutine we jump back to the address stored in register `$31` (the subroutine return register).

11. In `bar`, register `$31` is saved on the stack.
12. `jal` does a *jump and link*, saving the return address in `$31`.
13. The value of `$31` is restored before we end `bar`.
14. There is unncessary shuffling about: saving and reloading `$4`.

If we assemble this code, and look at the result (using `dis`) we find

```
      foo:
 [eg.c:   3] 0x0:      3c1c0000        lui     gp,0
 [eg.c:   3] 0x4:      279c0000        addiu   gp,gp,0
 [eg.c:   3] 0x8:      0399e021        addu    gp,gp,t9
 [eg.c:   4] 0xc:      03e00008        jr      ra
 [eg.c:   4] 0x10:     24820001        addiu   v0,a0,1
      bar:
 [eg.c:   8] 0x14:     3c1c0000        lui     gp,0
 [eg.c:   8] 0x18:     279c0000        addiu   gp,gp,0
 [eg.c:   8] 0x1c:     0399e021        addu    gp,gp,t9
 [eg.c:   8] 0x20:     27bdffe0        addiu   sp,sp,-32
 [eg.c:   9] 0x24:     8f990000        lw      t9,0(gp)
 [eg.c:   8] 0x28:     afa40020        sw      a0,32(sp)
 [eg.c:   8] 0x2c:     afbf001c        sw      ra,28(sp)
 [eg.c:   9] 0x30:     8fa40020        lw      a0,32(sp)
 [eg.c:   9] 0x34:     0320f809        jalr    ra,t9
 [eg.c:   8] 0x38:     afbc0018        sw      gp,24(sp)
 [eg.c:   9] 0x3c:     8fbf001c        lw      ra,28(sp)
 [eg.c:   9] 0x40:     8fbc0018        lw      gp,24(sp)
 [eg.c:   9] 0x44:     27bd0020        addiu   sp,sp,32
 [eg.c:   9] 0x48:     03e00008        jr      ra
 [eg.c:   9] 0x4c:     24420002        addiu   v0,v0,2
```

The `.frame`s have become manipulations of the global data register, and the adds have been reordered into the delay slots. The two argument `subu` in the assembler becomes a three argument `add`. The `lw t9` is part of the frame sequence.

With more optimisation (`-O2`) the assembled code for `foo` becomes

```
      foo:
 [eg.c:   4] 0x0:      03e00008        jr      ra
 [eg.c:   4] 0x4:      24820001        addiu   v0,a0,1
```

Compare the above with the output of another assember, `gcc -S -O2` (with moderate optimisation)

```
        .file   1 "eg.c"
        .option pic2

 # GNU C 2.7.2.2 [AL 1.1, MM 40] SGI running IRIX 5.x compiled by GNU C

 # Cc1 defaults:
```

```
 # -mabicalls

 # Cc1 arguments (-G value = 0, Cpu = 3000, ISA = 1):
 # -quiet -dumpbase -O2 -o

gcc2_compiled.:
__gnu_compiled_c:
        .text
        .align  2
        .globl  foo
        .ent    foo
foo:
        .frame  $sp,0,$31          # vars= 0, regs= 0/0, args= 0, extra= 0
        .mask   0x00000000,0
        .fmask  0x00000000,0
        .set    noreorder
        .cpload $25
        .set    reorder
        .set    noreorder
        .set    nomacro
        j       $31
        addu    $2,$4,1
        .set    macro
        .set    reorder

        .end    foo
        .align  2
        .globl  bar
        .ent    bar
bar:
        .frame  $sp,32,$31         # vars= 0, regs= 2/0, args= 16, extra= 8
        .mask   0x90000000,-4
        .fmask  0x00000000,0
        .set    noreorder
        .cpload $25
        .set    reorder
        subu    $sp,$sp,32
        .cprestore 16
        sw      $31,28($sp)
        sw      $28,24($sp)
        jal     foo
        addu    $2,$2,2
        lw      $31,28($sp)
        addu    $sp,$sp,32
        j       $31
        .end    bar
```

The `addu` instruction in `foo` is placed *after* the jump returns: this is using a delay slot.

> Aside: both `cc` and `gcc` on full optimisation notice that `foo(n) + 2` is really just `n + 3`, and replace the call to `foo` by `addu $2,$4,3`, and dispense with the now unneeded frame.

Here is the same for a different instruction set, the Pentium:

```
        .file   "eg.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl foo
        .type    foo,@function
foo:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%edx
        incl %edx
        movl %edx,%eax
        jmp .L1
        .align 4
.L1:
        leave
        ret
.Lfe1:
        .size    foo,.Lfe1-foo
        .align 4
```

```
.globl bar
        .type    bar,@function
bar:
        pushl %ebp
         movl %esp,%ebp
        movl 8(%ebp),%eax
        pushl %eax
        call foo
        addl $4,%esp
        movl %eax,%eax
        leal 2(%eax),%edx
        movl %edx,%eax
        jmp .L2
        .align 4
.L2:
        leave
        ret
.Lfe2:
        .size    bar,.Lfe2-bar
        .ident  "GCC: (GNU) 2.7.2.1"
```

And the 64 bit Alpha

```
        .set noat
        .set noreorder
        .text
        .align 4
        .file 1 "eg.c"
        .loc 1 2
 #      2 int foo(int n)
        .globl  foo
        .ent    foo
        .loc 1 2
foo:
   # 000002
        .frame  $sp, 0, $26
        .save_ra $0
        .prologue 0
        .loc 1 3
 #      3 {
        sextl   $16, $16
   # 000003
        .loc 1 4
 #      4   return(n+1);
        addl    $16, 1, $0
   # 000004
        .loc 1 5
```

```
 #      5 }
        ret     ($26)
   # 000005
        .end    foo
        nop
        .loc 1 7
 #      6
 #      7 int bar(int n)
        .globl  bar
        .ent    bar
        .loc 1 7
bar:
   # 000007
        mov     $26, $1
        .frame  $sp, 0, $26
        .save_ra $1
        .prologue 0
        .loc 1 8
 #      8 {
        sextl   $16, $16
   # 000008
        .loc 1 9
 #      9   return(foo(n) + 2);
        bsr     $26, foo
   # 000009
        addl    $0, 2, $0
        .loc 1 10
 #     10 }
        mov     $1, $26
   # 000010
        ret     ($1)
        .end    bar
```

Which, on disassembly, is

```
        foo:
  0x0:  43f00010        addl    zero, a0, a0
  0x4:  42003000        addl    a0, 0x1, v0
  0x8:  6bfa8001        ret     zero, (ra), 1
  0xc:  47ff041f        bis     zero, zero, zero
        bar:
  0x10: 27bb0001        ldah    gp, 1(t12)
  0x14: 23bd8050        lda     gp, -32688(gp)
  0x18: 23defff0        lda     sp, -16(sp)
  0x1c: a77d8010        ldq     t12, -32752(gp)
  0x20: b75e0000        stq     ra, 0(sp)
  0x24: 43f00010        addl    zero, a0, a0
  0x28: 6b5b4000        jsr     ra, (t12), foo
  0x2c: 27ba0001        ldah    gp, 1(ra)
  0x30: a75e0000        ldq     ra, 0(sp)
  0x34: 23bd8034        lda     gp, -32716(gp)
```

```
0x38: 40005000        addl    v0, 0x2, v0
0x3c: 23de0010        lda     sp, 16(sp)
0x40: 6bfa8001        ret     zero, (ra), 1
```

Aside: It has been noted that if a 500Mhz machine that has to wait for a (fast!) 10msec disc, it is the equivalent to a human waiting 100 years for a bus.